Improving the Write Performance of an NFS Server

Chet Juszczak - Digital Equipment Corporation

ABSTRACT

The Network File System (NFS) utilizes a stateless protocol between clients and servers; the major advantage of this statelessness is that NFS crash recovery is very easy. However, the protocol requires that data modification operations such as *write* be fully committed to stable storage before replying to the client. The cost of this is significant in terms of response latency and server CPU and I/O loading. This paper describes a *write gathering* technique that exploits the fact that there are often several write requests for the same file presented to the server at about the same time. With this technique the data portions of these writes are combined and a single metadata update is done that applies to them all. No replies are sent to the client until after this metadata update has been fully committed, thus the NFS crash recovery design is not violated. This technique can be used in most NFS server implementations and requires no client modifications.

1. Introduction

The Network File System (NFS) remains a de facto standard in the UNIX industry. NFS utilizes a stateless protocol between clients and servers. The major advantage of this statelessness is that NFS crash recovery is very easy. Neither client nor server must detect the other's crashes. Since a server has no state information to maintain, there is nothing for it to throw away after a client crashes. Likewise, there is no state information to re-build when the server returns after a crash. However, this protocol requires that data modification operations, e.g. write, be fully committed to stable storage before replying to the client [SAND85].

The write operation is usually the most costly of all NFS server operations (see *NFS Writes*, below). A heavy write load typically yields poorer server performance than loads of other types. It is not unusual, e.g., to see the write operation portion (15%) of the SPEC SFS 1.0 NFS server benchmark (SPEC/LADDIS) account for two thirds or more of the total latency and contribute more than its share to server CPU and I/O loading [WITT93]. Also, the large latencies of NFS write operations can lead to more serious problems on the server when it is faced with retransmissions from impatient clients [JUSZ89].

This paper describes the implementation of a *write gathering* technique that exploits the fact that there are often several write requests for the same file presented to the server at about the same time. With this technique the data portions of these writes are combined and a single metadata update is done that applies to them all. No replies are sent to the client until after this metadata update has been fully committed, thus the NFS crash recovery design is not violated. This technique can be used in most NFS server implementations. It requires no client modifications; it exploits behavior that is common among existing workstation clients. The implementation of this technique has resulted in a significant increase in server write bandwidth and an increase in overall server capacity and responsiveness (see *Results*).

2. Related Work

I do not claim to have originated the notion of write gathering as a performance optimization for NFS writes. Prior to 1990, while at Epoch Systems, Inc., Dave Noveck did some similar work within the filesystem below the NFS server layer [BROW90]. Dave is now at the Open Software Foundation, Cambridge, MA. Drew Perkins, while at Interstream, did an NFS server layer implementation for a SunOS after-market product; a demo of this product at the 1991 NFS Connectathon event got me interested in the problem. Drew is now at Fore Systems, Inc., Pittsburgh, PA. Sun has an NFS server layer implementation in current versions of Solaris that may make its way into a future *reference port* of NFS; it is rumored that they chose from one of several in-house implementations. I suspect other NFS server vendors also have implementations. Suresh Sivaprakasam describes an implementation for SunOS that clusters NFS writes in [SIVA93]; I make some comparisons to my implementation later (see *Write Gathering*).

I do claim to have an original implementation that has proven to be useful. I found no discussion of this topic in the literature when I did the work in 1991/92; to date, I can find only [SIVA93]. Thanks to some nudging by Jeff Mogul, I am sharing my implementation details via this paper in the interest of sparking some discussion. Hopefully the level of NFS service provided by the vendor community will benefit.

3. Goals

My goal in originating this work in 1991 was to increase NFS write speed from the perspective of a single client writing a large file, and to do so without consuming an inordinate amount of server capacity. We were in the process of modifying our local filesystem to cluster reads and writes for higher disk file throughputs (similar to [MCVO91]). I wanted remote NFS clients to achieve throughputs similar to those of local processes modulo network limitations.

4. Background

This section contains background information on various NFS topics; it is provided to help understand the environment in which write gathering operates. References to "typical" NFS clients and servers refer to implementations of NFS derived from the 4.3BSD based kernel implementation available from Sun Microsystems, Inc. Similarly, the term *reference port* will refer to this implementation. My work was done with version 4.2 of the reference port. The following characterizations should largely apply to other NFS implementations as well.

4.1. NFS Clients

NFS client systems range in size from single-threaded (dumb) PCs to small workstations to large multi-processor timesharing systems with hundreds of users.

Typical NFS clients (and servers) communicate via UDP messages with an effective maximum size (excluding protocol headers) of 8K. A typical client will retransmit a request if it has not received a response from the server for that request within an interval of time that defaults to a starting value of 1.1 seconds. The retransmission interval is dynamically adjusted based on past server performance. Server write performance is an important part of the client backoff algorithm. Since the write operation is typically the most expensize for the server to perform, and with the highest latency, write performance is used as an indicator of server performance for heavyweight operation types. Poor write performance will affect client behavior with respect to other types of requests. (The other two indicators are performance of read (middleweight) and lookup (lightweight) operations.)

A client system can have multiple outstanding read and/or write requests. A client process blocks whenever a read or write request cannot be satisfied locally and must be processed by the server. When it blocks, another process can run; that process may also generate a read or write request. A single process can have multiple outstanding read and/or write requests if the client system is running NFS block I/O (*biod(8)* or *nfsiod(8)*) daemons, referred to simply as *biods* from here on. Biods perform client read-ahead and write-behind functions asynchronously, allowing the client process to continue execution in parallel with client/server communication.

Let’s assume a typical workstation-class client system that is running biods, and a client process, C, that is generating write requests. When C generates a write that "needs to go to the wire" (a client kernel decision, typically decided by the application writing to the end of an 8K cache block), the task of communicating the write request is handed off to a biod and C continues. If no biod is available for hand-off, then C will block until *that*

particular request has received a response. This is independent of responses received for earlier writes handed off to biods. The blocking of C provides a simple client/server flow control.

Most NFS clients impose a *sync on close* semantic where the close(2) call blocks until all outstanding writes have received responses. Mostly this is to capture an ENOSPACE server response to asynchronous write requests and communicate it to the client application process.

4.2. NFS Servers

NFS server systems range in size from workstation-class systems with several disks to large multi-processor systems with disk farms.

A typical NFS server system simply waits for work to appear on an incoming request queue. This queue is the socket buffer allocated for the NFS socket. Incoming requests are converted into a form understandable by the local filesystem routines that actually perform the work of getting data to/from a disk. The incoming request queue is typically of fixed size. If the queue fills (requests coming in faster than they can be processed) then some incoming requests may be lost and client backoff/retransmission comes into play. The server depends upon its clients to attenuate their request loads as it becomes heavily loaded (i.e. the aggregate load is coming in faster than can be processed). Write requests are typically large and time consuming; processing them more quickly and efficiently can help keep a server ahead of its clients.

The amount of work that a server can perform is called server bandwidth or capacity. It is usually limited by exhaustion of one of the following three:

- CPU capacity,
- network interface capacity, or
- disk subsystem capacity.

Server capacity is sometimes measured in a general manner, e.g. NFS operations/second over some sort of operation mix (e.g. SPEC/LADDIS), and sometimes specifically, e.g. read or write speed in Kbytes/second. When seeking maximum capacity, a benchmarker typically adds network and I/O capacity until CPU is exhausted.

A typical server does not assign priorities to incoming requests based on type of request or originating client. It processes incoming requests within the context of several *nfsd* daemons. The number of nfsds controls the number of NFS requests that a server can work on concurrently.

4.3. NFS Server Writes and Stable Storage

An NFS server should commit any modified data to stable storage before responding to the client that the request is complete [SAND85]. If a server is not following this rule, then it is not living up to its part of the agreement implicit in the NFS crash recovery design. An asynchronous operation carries with it the promise to fully complete that operation at some later time. The NFS protocol contains no provisions for recalling past promises (which is precisely why crash recovery is so easy). Without a way to recall past unkept promises, a server should not make them. Traditionally, this meant that the server had to fully commit all data and modified metadata associated with the write operation to disk before responding.

The cost, in latency and server loading, of meeting this requirement is significant, and has led to varying vendor reactions and developments:

- The appearance of filesystem accelerators consisting of s/w and non-volatile RAM (NVRAM) h/w, e.g. Prestoserve [MORA90], [PRES93], that address the issues of latency and, to some degree, I/O loading (but not server CPU loading) while operating within the stable storage paradigm.
- Some vendors have chosen to make async NFS writes, a.k.a "dangerous mode", an administrative option. In this scenario, the server responds after data is committed to volatile storage, whether main memory or disk controller, an option. Some vendors have made this mode the default behavior and supply an uninterruptible power supply (UPS) with their servers. Some vendors simply make it the default behavior without UPS.

After much discussion, SPEC has arrived at the following requirement for reporting SPEC/LADDIS *baseline* results [SPEC93]. A baseline-conforming NFS server must:

- commit all write data and associated metadata to disk,
- or to other stable storage (e.g. NVRAM) that is recovered and flushed to disk after server failure,
- or to volatile RAM (main memory) if powered by UPS *and* if the OS supports data recovery following power and h/w failures.

This paper assumes that the server conforms to the SPEC baseline reporting requirements with respect to stable storage guarantees by using one of the first two operational techniques listed above.

4.4. Local Filesystem Operations

This paper assumes that the filesystem being served is of BSD FFS vintage [MCKU84], as is the case with the reference port. My work was done using a BSD 4.3 filesystem (UFS) with extensions that cluster reads and writes into larger device request sizes (up to 64K) in a manner similar to that described in [MCVO91].

For each remote write request, at least one, and possibly two or three synchronous disk operations must be performed by the server before a response can be sent to the client indicating that the request has been completed. At the very least, the data block in question must be written. If the write increased the size of the file, or on-disk structures have changed (e.g. adding a direct block to fill a "hole" in the file), then the block containing the inode must be written. Finally, if an indirect block was modified, then it too must be written before responding.

The reference port makes a special case for the file modify time in the inode. If modify time is the only item changed in the inode as a result of a write operation, i.e. a write to a previously allocated block, then the inode update to disk is performed asynchronously. This (the file modification time) is one promise that the server may not keep; this risk is taken for the benefits of better performance.

5. A Case Study

Let’s assume we are using a network monitoring tool (like tcpdump(8)) to observe the network while client process C, described earlier, writes a reasonably large new file, and for the sake of simplicity assume that the client is doing nothing else, and that this is a private network. Finally, let’s assume that the client and a typical server, S, are well matched enough so that we don’t have any lost requests or responses, timeouts, etc.

As C begins to run, network traffic will resemble a freight train of 8K (actually a little larger due to protocol headers, etc.) datagrams fragmented into transport units directed toward S. The total length of the train is dependent upon the number of biods available for use by C. Many biods mean a very long train. The time needed for S to process the first write request (which was handed off to a biod) is typically larger than the time needed for C to generate more requests. Several, and perhaps many, writes can build up on the server side before the first response is sent. Process C blocks when the biods all become busy. The traffic direction now changes to one of responses directed toward C at intervals of time needed by S to process the buffered write requests. No further requests are generated by C until a response is received for the last write request; the traffic direction now switches again as C resumes processing. A cycle of these uni-directional traffic shifts continues as described above until the entire file has been transferred.

The left half of Figure 1 (the communication between Client and Standard Server) depicts a portion of this traffic flow for the 4 biod case, after the client has written about 100K of data.

If the file in question was newly created and of size N*8K, and the server filesystem was of blocksize 8K, then the total number of server disk operations was roughly 3N:

- N data writes,
- N inode block updates,
- N indirect block updates (minus n direct blocks)

6. Write Gathering

The write gathering technique described here attempts to reduce the 3N disk operations described in the previous section (*A Case Study*) to as close to N as it can. With an underlying filesystem that supports clustering [MCVO91], and a sequential write pattern, the number of disk operations can be reduced far below N. Optimal

write gathering occurs when the minimal number of disk transactions is generated for a particular load with a particular underlying filesystem. The description of write gathering that follows assumes an underlying server filesystem (e.g. UFS) that supports write clustering.

The object of this technique, from the perspective of an nfsd, is to avoid doing the metadata update. An nfsd, D, tries to assign this task to some other nfsd which will send D's response as well as its own.

If we use the network monitoring example from the previous section with a write gathering server instead, the change we see is in the timing of the replies from server to client. The server "digests" all the write requests and sends replies in first in, first out order; all the replies have the same file modify time in the returned file attributes. The total elapsed time from first request to last reply is less due to disk efficiencies (fewer, larger writes, fewer seeks).

The right half of Figure 1 (the communication between Client and Gathering Server) depicts a portion of this traffic flow for the 4 biod case, after the client has written about 100K of data. In this case, optimal write gathering of 3 disk transactions has been achieved.

6.1. NFS/RPC Architectural Changes

In a reference port server, each nfsd process makes a call into kernel level RPC (the svc_run() routine) with a transport handle (used to store client and request information) and the address of an NFS layer routine (rfs_dispatch()) that dispatches incoming requests to appropriate server layer action routines (e.g. rfs_write() for incoming write requests). Svc_run() does not return until the nfsd process dies. Information needed to send a response is stored in the transport handle which is tied to the nfsd process that started work on the request. When the action routine returns, rfs_dispatch() sends the reply to the client and returns to svc_run().

This architecture was modified so that one nfsd can process a write request to a certain point and then arrange for another nfsd to send the reply. The first nfsd is then free to look for other work. It returns an indication to rfs_dispatch() that the reply is delayed, who conveys this back into the RPC layer; another transport handle is taken from a cache of free handles and the nfsd process is ready to process other work, possibly another write to the same file. This architecture allows optimal write gathering to take place with as few as one nfsd available on the server; this is an architecture that should scale well for large servers with many active client writers.

6.2. New Data Structures

A global array of nfsd state was created so that one nfsd can ascertain the state of others. Most notably, whether another nfsd is processing a write, and to which file, and to which offset and length, and at what stage the nfsd is in the processing of a write. With this information an nfsd can decide whether it can leave the task of metadata update to another, "following" nfsd.

As mentioned above, a cache of free transport handles was implemented, along with data structures that package up active write requests for handoff and a queue of these active requests.

OSF/1 provides a vnode spin lock, but not a sleep lock. I added a vnode sleep lock for nfsd serialization and synchronization.

6.3. Prestoserve/NVRAM Acceleration vs. Disks

NVRAM accelerated disks have radically different latency properties from non-accelerated ones. With a non-accelerated disk, the best policy is to cache/coalesce/cluster writes within UFS as long as possible in the hope of doing fewer, larger disk writes. With a Prestoserve accelerated disk, the best policy is to get individual writes down to the Prestoserve driver (Presto) as soon as possible. Presto does its own clustering. With Presto there is no benefit to holding writes within UFS as when using a non-accelerated disk; in fact it is less efficient because Presto can drive disks asynchronously and in parallel with NFS write and reply processing. Also, due to the relatively small size of the NVRAM cache (typically one or more MB), Presto may decline to accept requests above a certain size (typically 8K), resulting in performance that degrades to underlying disk speed.

The observations above lead to a duality within the server write layer; it was modified to query Presto as to acceleration state of a filesystem (on/off) and operate in different ways depending upon state.

6.4. Filesystem Hints Through VFS

The VFS (GFS for ULTRIX) layer was modified so that the server layer could send hints to the underlying filesystem.

If the filesystem is accelerated, the VOP_WRITE routine is called with the IO_SYNC and the (new) IO_DATAONLY flags, delivering the data to Presto but delaying any metadata copies (and consumption of CPU cycles). If the filesystem is not accelerated, the VOP_WRITE routine is called with the (new) IO_DELAYDATA flag, freeing UFS to choose its own clustering policy (and perhaps starting an asynchronous write). Metadata is flushed via a call to VOP_FSYNC with the FWRITE and the (new) FWRITE_METADATA flags to ensure that only the inode and indirect blocks are flushed.

For non-accelerated disks, when write gathering terminates (described below) and metadata is flushed, data blocks are flushed via a call to the (new) VOP_SYNCDATA routine with beginning and ending offsets as hints.

6.5. The Socket Buffer

With Prestoserve acceleration, there is often no I/O event associated with a VOP_WRITE, and the nfsd process D does not block. If it does not block there is no opportunity for write requests to be delivered to other nfsds even if they have been placed on the socket buffer. A routine (the mbuf hunter) was written (hacked) to scan the socket buffer searching for NFS writes for a given file and returning true/false. The mbuf hunter is a gross violation of kernel layering, but with a fast server this technique is often a win (and thus the hack has redeeming virtue).

6.6. Procrastinate

(pro-kras'-ti-nate) *verb.* To put off, esp. habitually, doing something until a future time. --pro-cras'ti-na'tor.
This is probably the most controversial aspect of write gathering. The technique injects a small amount of latency into the processing cycle, hoping to give another write request an opportunity to arrive at the server. This latency is approx. (modulo h/w clock accuracy) 8 msec for Ethernet or multi-segment requests and 5 msec for FDDI based requests. These values were derived via empirical lab experiments. Private networks were used to ascertain values that allowed for optimal gathering, and a tick or two was added for conservatism. Also, systems in more general use have been monitored for gathering success rates (but conclusions are sometimes difficult to draw without detailed knowledge of request patterns).

I wish I could say I know how to calculate the "right" number, but I don't. Clearly there is room for more work here. When write gathering "fails" because the server didn't wait long enough, it falls back to typical, standard, server processing behavior.

The implementation described in [SIVA93] takes the first write encountered and sends it to disk, using this operation as "the latency device" which gives more write requests time to arrive at the server. I considered this approach early on and abandoned it for two reasons:

> First, running spindles with a request pattern of anything other than a pure stream of large requests is suboptimal in both drive throughput and CPU utilization. It's difficult to approach raw device speeds with 8K requests in the device queue without dangerous mode operation (controller write caching). Ignoring protocol requirements, still leaves too many trips through the driver.
>
> Second, it just won't work with NVRAM acceleration where the first write is done faster than other writes can arrive.

6.7. Order of Replies

At first it seemed a good idea to order replies in LIFO order so as to wake up a blocked client process C (see *A Case Study*, above). In fact, this yielded dismal results for the common file transfer case because C would generate fewer (and maybe none for a fast client) writes before blocking the next time if there were not enough biods available. LIFO was abandoned for FIFO; this optimized the case of a single sequential file writer. It also seems reasonable to free up biods on the client for other work (by other processes) sooner, in the multiprocessing client case. Note that FIFO is the order used by typical (standard) existing servers, and this is not a change in behavior.

6.8. The Write Gathering Algorithm

D is an nfsd handed a write request:

```
Hand off data to UFS via VOP_WRITE (as described above).
Do
    Look for another nfsd blocked on the same vnode.
    If one is,
        Add write descriptor to the active write queue.
        Return to rfs_dispatch() with a reply-pending code.
    Else search the socket buffer for another write request to the same file.
    If there is,
        Add write descriptor to the active write queue.
        Return to rfs_dispatch() with a reply-pending code.
    Sleep (procrastinate) for a transport dependent interval.
While not procrastinating more than once.
Become the metadata writer and assume responsibility for this file:
    Flush this and other data for active writes via VOP_SYNCDATA.
    Flush the metadata via VOP_FSYNC.
    Send all pending replies for the file to the client.
    Return to rfs_dispatch() with a reply-done code.
```

6.9. Duplicate Requests, Stale File Handles, Etc.

Stale file handles are client references to files that no longer exist. See [JUSZ89] for a discussion of duplicate NFS requests.

The existence of these requests, in the socket buffer e.g., could have caused nfsds to delay their replies. The implementor must not to be too hasty discarding duplicates, etc. (meaning I was at first!); this could result in orphaned writes on the active write queue with no meta data writer to send replies.

6.10. What About Dumb PCs?

Single threaded PCs (or clients with no biods, or clients that emit a single write every once in a while) are the worst case for write gathering. There is added processing and latency for no gain. The actual measured loss from the client's perspective (easily simulated by killing all biods) is about 15% in throughput (over Ethernet) with a reasonably quick server and a quick single threaded client (see *Results*). This loss decreases in significance as slower clients are used.

6.11. What About Random Access?

The write gathering algorithm does not assume an ordering on the delivery of writes. A grouping of random access writes will accrue the same benefits of metadata amortization as a grouping of sequential access writes. The clustering of data blocks, and the resultant number of disk transactions for them, is an underlying filesystem issue.

7. Results

7.1. NFS Sequential Write Bandwidth

This section contains the results of experiments where a 10MB file is written over private Ethernet and FDDI networks with and without write gathering in effect and while varying the number of client biods. The server used 8 nfsds. For Ethernet, the client and server are DEC 3400s, the server is using an RZ26 (1GB SCSI) disk. For FDDI, I used somewhat faster systems (for no better reason than that is the way my lab is set up), the client is a DEC 3500, the server is a DEC 3800, using either one, or a stripe set of three RZ26 disk(s), as labeled.

Table 1 shows the experiment, without write gathering, being limited by spindle speeds for 8K transfers. Table 1 also shows the cost of write gathering in the worst case (no biods) as 15%, and the gain in the best case (15 biods) as 228%. For the 7 biod case the gain is 145%.

Table 1. NFS 10MB file copy: Ethernet

# of Client Biods	0	3	7	11	15
Without Write Gathering					
client write speed (KB/sec.)	165	194	201	203	205
server cpu util. (%)	9	11	11	12	12
server disk (KB/sec)	480	570	590	590	590
server disk (trans/sec)	61	71	72	73	74
With Write Gathering					
client write speed (KB/sec.)	140	375	493	575	674
server cpu util. (%)	7	14	16	19	21
server disk (KB/sec)	415	550	610	660	750
server disk (trans/sec)	52	47	24	31	21

What is not obvious in Table 1 is that write gathering is conserving server CPU (by saved UFS and driver trips); this is hidden by the greater throughput. Table 2 shows the same disk under Prestoserve acceleration, where the latencies involved are NVRAM copies instead of moving head disk operations. With Prestoserve (where Presto is clustering and handling the underlying disk efficiently), write gathering increases CPU efficiency at the expense of some client throughput. The 7 biod case shows a decrease of 26% in server utilization at the cost of 15% in client throughput. The bulk of this savings is the reduction of metadata operations through UFS and Presto.

Table 2. NFS 10MB file copy: Ethernet, Presto

# of Client Biods	0	3	7	11	15
Without Write Gathering					
client write speed (KB/sec.)	809	1025	1080	1103	1112
server cpu util. (%)	30	38	41	42	43
server disk (KB/sec)	789	1004	1080	1104	1080
server disk (trans/sec)	7	8	9	9	9
With Write Gathering					
client write speed (KB/sec.)	439	787	915	959	991
server cpu util. (%)	18	26	30	32	34
server disk (KB/sec)	430	770	885	949	985
server disk (trans/sec)	4	7	7	9	8

Now we change the configuration by moving to an FDDI network, reducing network latencies, server CPU overhead due to packet reassembly, etc. In Table 3, for the 15 biod case, we see the server processing NFS writes at about 1MB/sec. without Presto acceleration.

Table 3. NFS 10MB file copy: FDDI

# of Client Biods	0	3	7	11	15
Without Write Gathering					
client write speed (KB/sec.)	207	209	207	209	208
server cpu util. (%)	6	6	6	6	6
server disk (KB/sec)	605	610	605	615	615
server disk (trans/sec)	76	77	76	75	77
With Write Gathering					
client write speed (KB/sec.)	177	534	846	876	1085
server cpu util. (%)	6	9	10	11	12
server disk (KB/sec)	520	780	975	1000	1175
server disk (trans/sec)	66	65	38	45	33

Table 4 shows the RZ26 disk being driven at the raw device write bandwidth limit for 64K transfers under Presto acceleration.

Table 4. NFS 10MB file copy: FDDI, Presto

# of Client Biods	0	3	7	11	15
Without Write Gathering					
client write speed (KB/sec.)	1883	1898	1863	1900	1918
server cpu util. (%)	33	34	35	35	34
server disk (KB/sec)	1833	1848	1844	1844	1900
server disk (trans/sec)	16	16	15	15	16
With Write Gathering					
client write speed (KB/sec.)	927	1850	1888	1895	1894
server cpu util. (%)	13	24	28	27	27
server disk (KB/sec)	910	1745	1889	1882	1867
server disk (trans/sec)	8	17	16	16	16

This experiment shown by Table 4. was limited by spindle speeds again similar to Table 1., but the client is operating at near maximal device bandwidth. Tables 5. and 6., where a 3 drive stripe set is used, shows the effect upon the configuration of increasing disk bandwidth. In Table 5., for the 15 biod case, write speed has increased with write gathering by 262%; CPU utilization has increased 36%. In Table 6., for the 15 biod case, write speed has decreased with write gathering by 20%, but CPU utilization has also decreased by 40% (relative to Table 5.).

Table 5. NFS 10MB file copy: FDDI, 3 striped drives

# of Client Biods	0	3	7	11	15	19	23
Without Write Gathering							
client write speed (KB/sec.)	200	275	299	304	308	308	313
server cpu util. (%)	7	10	11	11	11	11	12
server disks (KB/sec)	560	827	865	895	879	921	927
server disks (trans/sec)	72	104	110	112	111	115	117
With Write Gathering							
client write speed (KB/sec.)	187	574	814	987	1115	1287	1618
server cpu util. (%)	7	11	13	15	15	18	22
server disks (KB/sec)	560	785	984	1109	1225	1384	1695
server disks (trans/sec)	71	72	60	65	67	71	74

Table 6. NFS 10MB file copy: FDDI, Presto, 3 striped drives

# of Client Biods	0	3	7	11	15	19	23
Without Write Gathering							
client write speed (KB/sec.)	2102	3403	3394	3503	3474	3360	3342
server cpu util. (%)	40	66	69	68	70	71	70
server disks (KB/sec)	2067	3146	3515	3349	3305	3575	3445
server disks (trans/sec)	47	71	80	77	76	80	78
With Write Gathering							
client write speed (KB/sec.)	1015	2144	2649	2775	2754	3078	3048
server cpu util. (%)	6	29	42	42	42	43	46
server disks (KB/sec)	1008	2143	2644	2724	2685	2501	2627
server disks (trans/sec)	22	49	61	62	63	59	63

7.2. SPEC/LADDIS Results

Writes are a small (15%) portion of Nhfsstone [LEGA89] and SPEC/LADDIS [WITT93] workloads, but they are expensive to process. This technique yielded a positive effect on SPEC/LADDIS server throughput and latency via its server efficiency gains. Figure 2 shows an increase of 13% in server capacity along with an 11% reduction

in average latency for a DEC 3800 server using write gathering as measured by SPEC SFS 1.0 (LADDIS). Figure 3 shows more modest, but still positive, gains for the same configuration with Prestoserve in effect.

8. Future Work

The NFS Version 3 protocol supports reliable asynchronous writes in addition to the Version 2 stable storage write semantics. Many V3 clients may opt for the simpler kernel implementation of V2 write semantics. This will ensure the usefulness of write gathering in a V3 environment. It will be interesting to see if this technique applies itself in some new way in a mixed environment of V2 clients, V3 clients using V2 semantics, and V3 clients using reliable asynchronous writes.

The worst case scenario for the current write gathering algorithm is with single threaded clients, such as dumb PCs, where there is never an opportunity to gather writes, and the CPU effort and added latency is a loss. This is a tradeoff that should be considered by the implementor and/or server administrator (it's easy to turn write gathering off). Some might say that it's doubtful whether a truly dumb (spelled slow) PC can tell the difference with an otherwise fast server, but it would be very nice to say that there is no performance penalty for single threaded writers. Jeff Mogul has suggested a scheme where the server builds a small database of "learned" information about individual clients, and uses this to direct gathering behavior. Clearly there is room for improvement here.

9. Conclusions

Write gathering can help to improve server capacity. It takes a lot of CPU cycles to run the disk driver and field device interrupts and/or copy data to NVRAM. If the NFS server layer can avoid some disk writes it is a big win; big enough to make it worth the gamble of spending some CPU cycles trying to be clever and avoid the writes.

Write gathering improves write bandwidths. Write gathering plays well with UFS clustering; it is possible to get closer to raw device speeds with NFS writes because fewer, larger, disk writes are done and fewer seeks and missed rotations are experienced. I was able to achieve a significant increase in client write speeds while at the same time often reducing server loading.

Write gathering exploits client behavior (i.e. biods) that has been typical in workstation clients since NFS was introduced. Write gathering efficiencies increase as the number of biods increase and has led some vendors to increase their defaults. With 8K transfers and UFS 64K clustering, 7 biods result in the ideal case on the server of one 64K disk write for data and one metadata update per set of (application + 7 biods) requests. The addition of more biods on the client may increase throughput if the carrying capacity of the network/server can support it (the server socket buffer, e.g., is a limit: DEC OSF/1 currently uses a maximum of .25M for socket buffering). As a rule of thumb, I don't recommend more than 7 biods for general purpose/heavily used networks.

The work described here was done for a filesystem with BSD FFS vintage on-disk structure [MCKU84]. Log-based and log-structured local filesystems are becoming more popular. Although on-disk structures may vary, the NFS stable storage requirement imposes a relationship between disk performance and server write performance. Hopefully this description of gathering network originated write requests in a network service layer, and the passing of hints from this layer to the underlying local filesystem, will prove useful with other filesystem types.

The implementation described here was made part of the ULTRIX Version 4.3 and DEC OSF/1 Version 1.2 operating systems.

10. Acknowledgements

Thanks go to: Drew Perkins for the NFS Connectathon demo that got me going. It was similar in impact to an earlier Connectathon demo of Prestoserve. [MCVO91] for providing the UFS clustering inspiration. Paul Shaughnessy for providing/adapting our UFS implementation. Brian Nadeau and Allen Rollow for providing the disk striping driver used in *Results*. Jeff Mogul for the nudge to publish and general support. Charlie Briggs, as usual, was a sounding board and provider of clear thinking during the project; he also provided the mbuf hunter.

11. References

[BROW90] Ted Smalley Brown, "Software update speeds NFS write process on server", trade publication article, Epoch Systems Inc.'s HyperWrite product, *Digital Review*, v7, n30 (August 6, 1990), p. 17.

[JUSZ89] Chet Juszczak, "Improving the Performance and Correctness of an NFS Server", *Proceedings Winter Usenix 1989*, San Diego, CA, 53-63, January 1989.

[LEGA89] Sandberg, R., "nhfsstone" NFS load generating program, Legato Systems, Inc., Palo Alto, CA.

[MCKU84] McKusick, M. K., et. al., "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, 2(3), August 1984, 181-197.

[MCVO91] McVoy, L., Kleiman, S., "Extent-like Performance from a Unix File System", *Proceedings Winter Usenix 1991*, Dallas, TX, 33-43, January, 1991.

[MORA90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., Lyon, B., "Breaking Through the NFS Performance Barrier", *Proceedings of the 1990 Spring European Unix Users Group*, Munich, Germany, 199-206, April 1990

[PRES93] Digital Equipment Corporation, Maynard, MA, "Guide to Prestoserve", *DEC OSF/1 Prestoserve Product Documentation*, Order number AA-PQT0A-TE, March 1993.

[SAND85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., "Design and Implementation of the Sun Network Filesystem", *Summer 1985 Usenix Conference Proceedings*, Portland, OR, 119-130, June 1985.

[SIVA93] Sivaprakasam, Suresh, "Performance Enhancements in SunOS NFS", *Technical Report TR 93-18*, State University of New York, Buffalo Computer Science Dept., *May, 1993.*

[SPEC93] Standard Performance Evaluation Corporation, "SPEC Run Rules for SFS Release 1.0", *SPEC Steering Committee Memorandum*, June 15, 1993, Sec. 7.1.1.4, p. 6, Contained in the SFS 1.0 Release Materials.

[WITT93] Wittle, M, Keith, B., "LADDIS: The Next Generation in NFS File Server Benchmarking", *Summer 1993 Usenix Conference Proceedings*, Cincinnati, OH, 111-128, June, 1993.

12. Author Information

Chet Juszczak is a Consultant Engineer in the Unix Software Group at DEC where he has been working on NFS and file server performance since 1985. He was involved with the definition of the NFS Version 3 protocol. He got his M.S. in C.S. at the University of Michigan in 1983. Reach him electronically at chet@zk3.dec.com or via U.S. Mail at Digital Equipment Corp., 110 Spit Brook Rd., Nashua NH 03062.

13. Trademarks

DEC 3500 client, 3800 server, rz26 disk, FDDI network

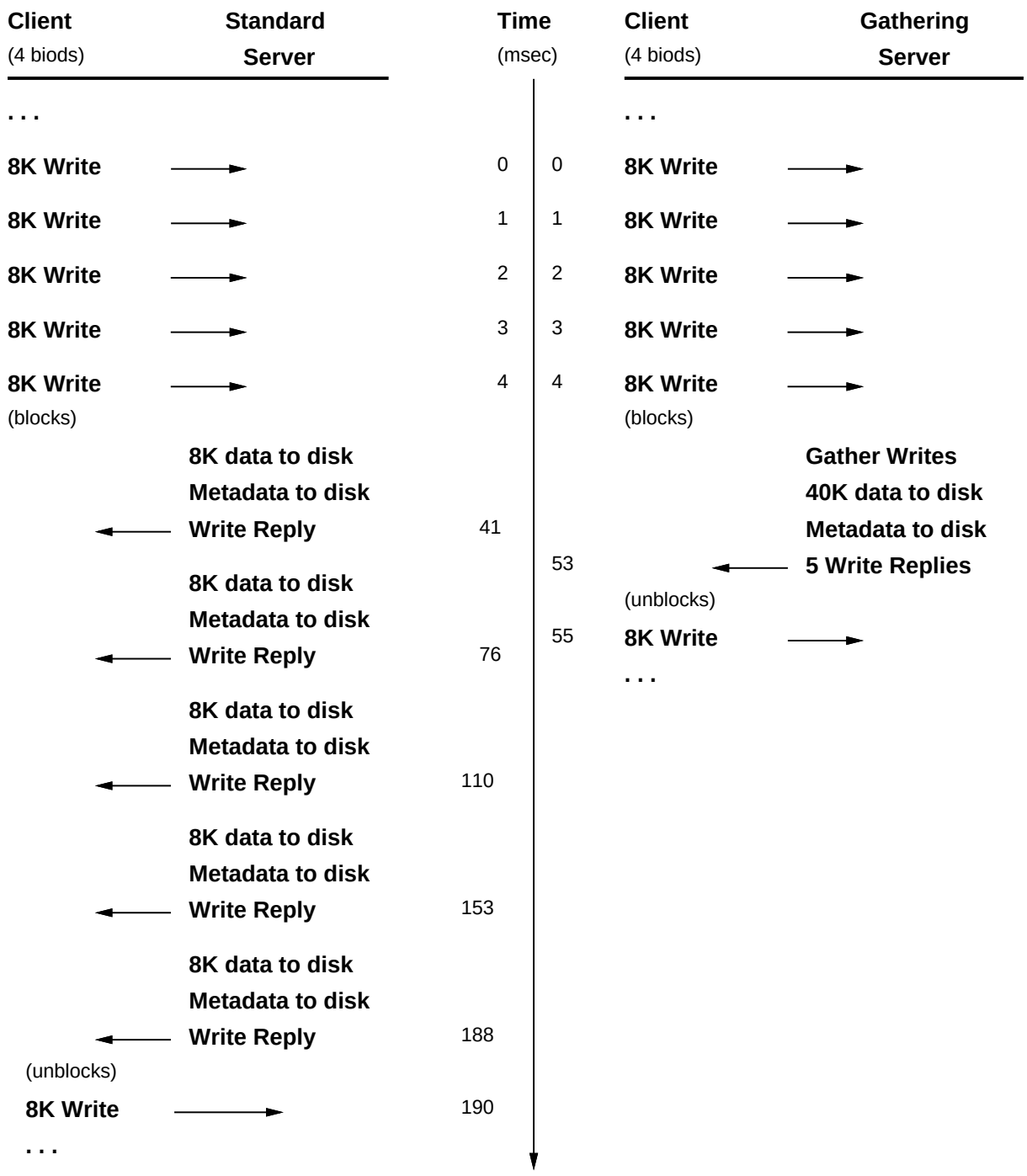

Figure 1. Write Gathering NFS Server Comparison
Sequential File Writer, >100K Into File

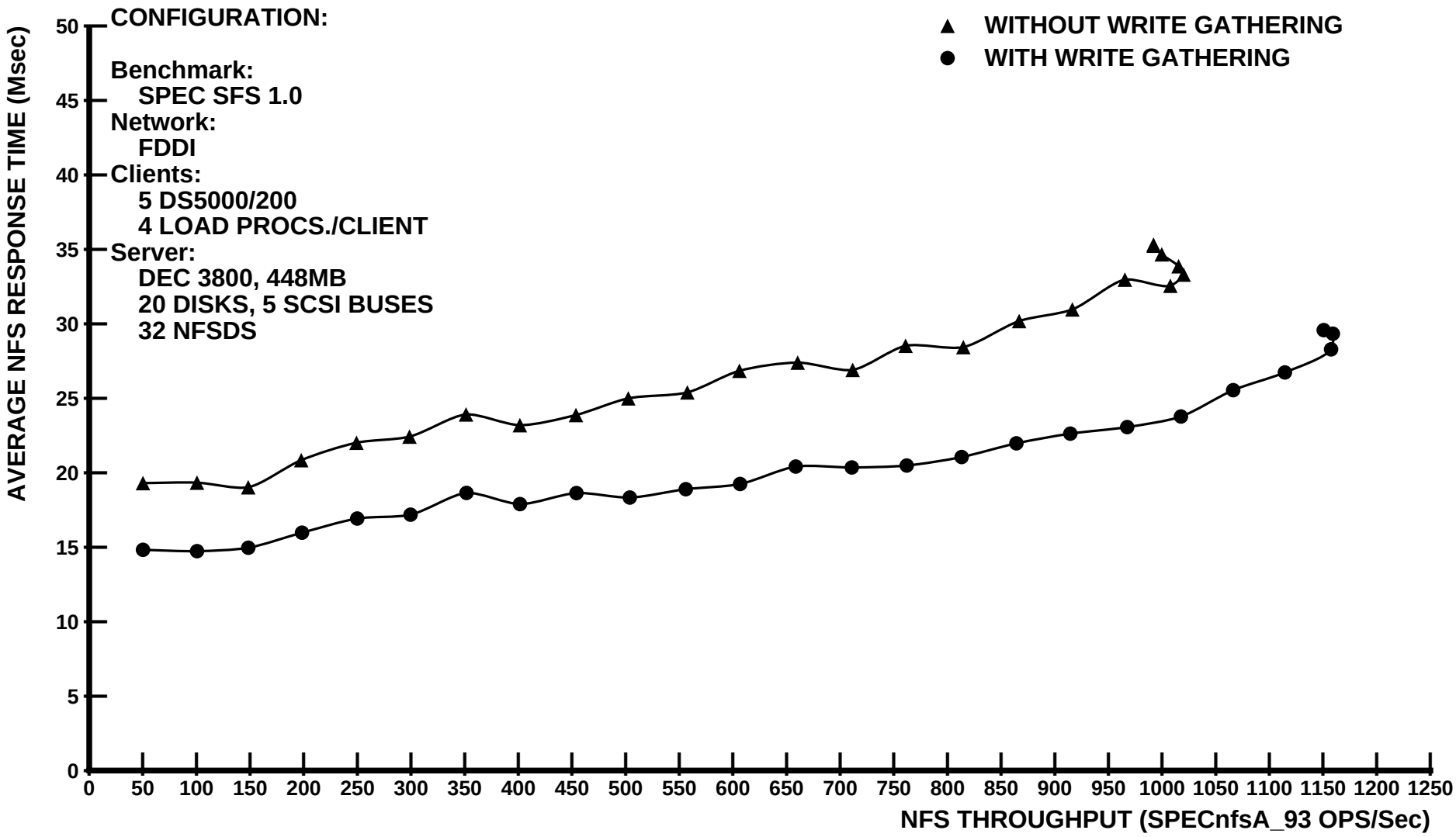

FIGURE 2. DEC 3800 SPEC SFS 1.0 BASELINE RESULTS

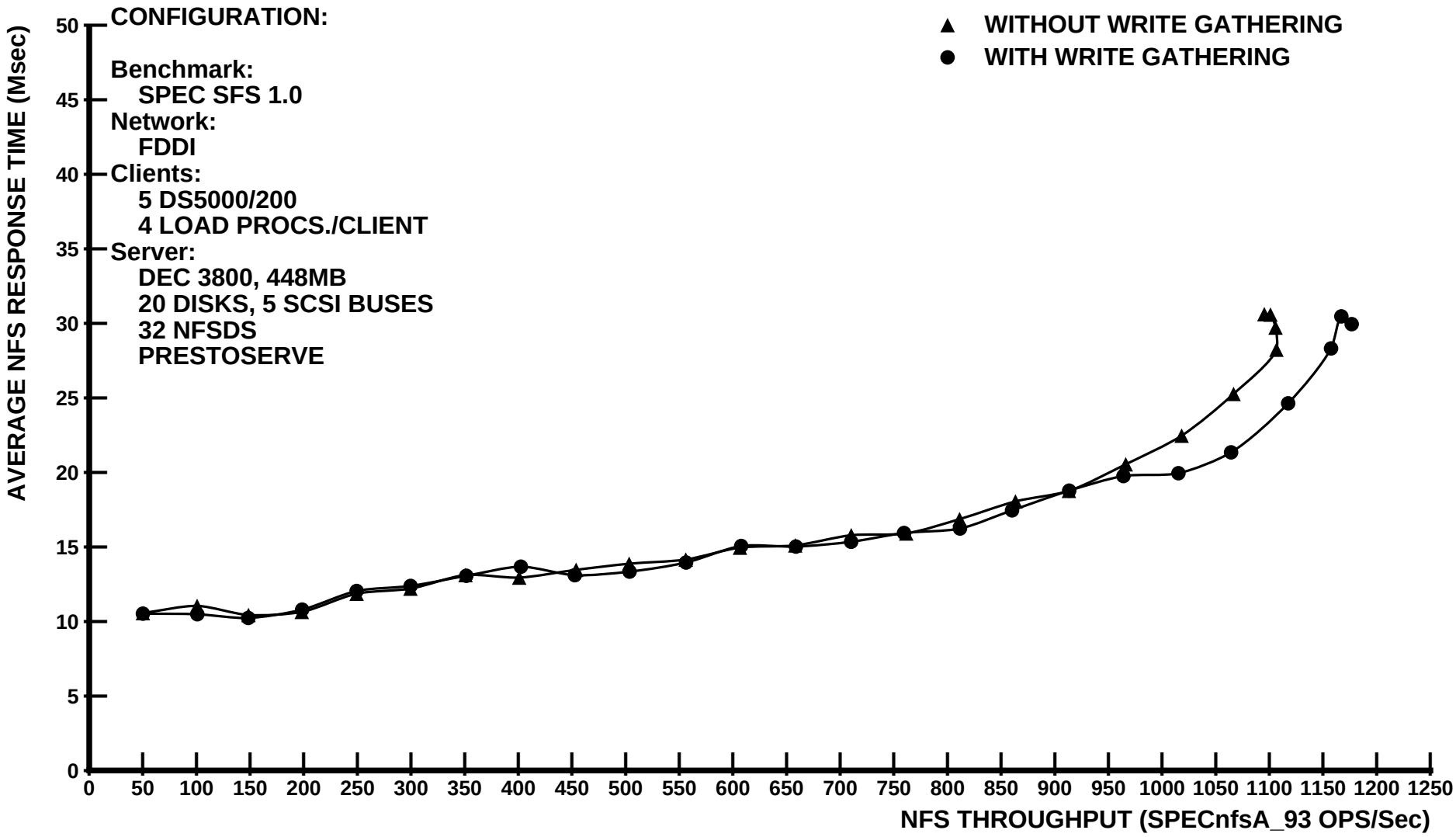

FIGURE 3. DEC 3800 PRESTOSERVE SPEC SFS 1.0 BASELINE RESULTS