vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O

Ming Chen, ***Stony Brook University;*** **Dean Hildebrand,** ***IBM Research-Almaden;***
Henry Nelson, ***Ward Melville High School;*** **Jasmit Saluja,**
Ashok Sankar Harihara Subramony, and Erez Zadok, ***Stony Brook University***

https://www.usenix.org/conference/fast17/technical-sessions/presentation/chen

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Open access to the Proceedings of the 15th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O

Ming Chen, Dean Hildebrand*, Henry Nelson+, Jasmit Saluja,
Ashok Sankar Harihara Subramony, and Erez Zadok
*Stony Brook University, *IBM Research - Almaden, +Ward Melville High School*

Abstract

Modern systems use networks extensively, accessing both services and storage across local and remote networks. Latency is a key performance challenge, and packing multiple small operations into fewer large ones is an effective way to amortize that cost, especially after years of significant improvement in bandwidth but not latency. To this end, the NFSv4 protocol supports a *compounding* feature to combine multiple operations. Yet compounding has been underused since its conception because the synchronous POSIX file-system API issues only one (small) request at a time.

We propose *vNFS*, an NFSv4.1-compliant client that exposes a vectorized high-level API and leverages NFS *compound procedures* to maximize performance. We designed and implemented vNFS as a user-space RPC library that supports an assortment of bulk operations on multiple files and directories. We found it easy to modify several UNIX utilities, an HTTP/2 server, and Filebench to use vNFS. We evaluated vNFS under a wide range of workloads and network latency conditions, showing that vNFS improves performance even for low-latency networks. On high-latency networks, vNFS can improve performance by as much as two orders of magnitude.

1 Introduction and Background

Modern computer hardware supports high parallelism: a smartphone can have eight cores and a NIC can have 256 queues. Although parallelism can improve throughput, many standard software protocols and interfaces are unable to leverage it and are becoming bottlenecks due to serialization of calls [8, 16]. Two notable examples are HTTP/1.x and the POSIX file-system API, both of which support only one synchronous request at a time (per TCP connection or per call). As Moore's Law fades [44], it is increasingly important to make these protocols and interfaces parallelism-friendly. For example, HTTP/2 [5] added support for sending multiple requests per connection. However, to the best of our knowledge little progress has been made on the file-system API.

In this paper we similarly propose to batch multiple file-system operations. We focus particularly on the Network File System (NFS), and study how much performance can be improved by using a file-system API friendly to NFSv4 [34, 35]; this latest version of NFS supports *compound procedures* that pack multiple operations into a single RPC so that only one round trip is needed to process them. Unfortunately, although NFS compounds have been designed, standardized, and implemented in most NFS clients and servers, they are underused—mainly because of the limitations of the low-level POSIX file-system interface [8].

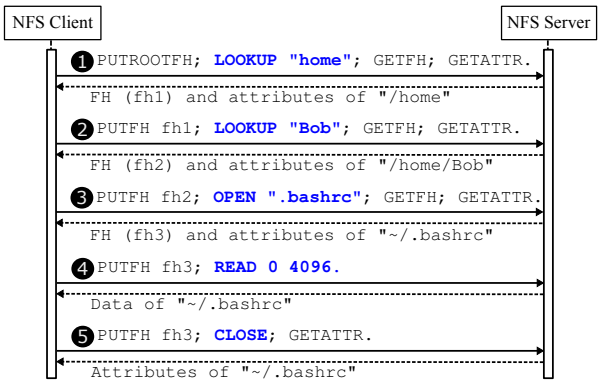


Figure 1: NFS compounds used by the in-kernel NFS client to read a small file. Each numbered request is one compound, with its operations separated by semicolons. The operations use an NFSv4 server-side state, the current filehandle (CFH). PUTROOTFH *sets the CFH to the FH of the root directory;* PUTFH *and* GETFH *set or retrieve the CFH;* LOOKUP *and* OPEN *assume that the CFH is a directory, find or open the specified name inside, and set it as the CFH;* GETATTR, READ, *and* CLOSE *all operate on the file indicated by the CFH.*

To explain the operations and premise of NFS4's compound procedures, we discuss them using several instructive figures. We start with Figure 1, which shows how reading a small file is limited by the POSIX API. This simple task involves four syscalls (`stat`, `open`, `read`, and `close`) that generate five compounds, each incurring a round trip to the server. Because compounds are initiated by low-level POSIX calls, each compound contains only one significant operation (in bold blue), with the rest being trivial operations such as PUTFH and GETFH. Compounds reduced the number of round trips slightly by combining the syscall operations (LOOKUP, OPEN, READ) with NFSv4 state-management operations (PUTFH, GETFH) and attribute retrieval (GETATTR), but the syscall operations themselves could not be combined due to the serialized nature of the POSIX API.

Ideally, a small file should be read using only one NFS compound (and one round trip), as shown in Figure 2. This would reduce the read latency by 80% (by removing four of the five round trips). We can even read multiple files using a single compound, as shown in Figure 3. All these examples use the standard (unmodified) NFSv4 protocol. SAVEFH and RESTOREFH operate on

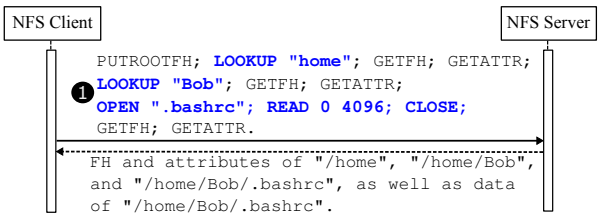


Figure 2: Reading `/home/Bob/.bashrc` using only one compound. This single compound is functionally the same as the five in Figure 1, but uses only one network round trip.

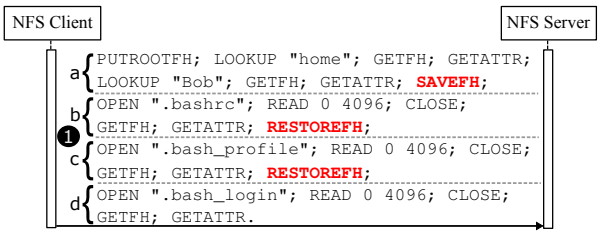


Figure 3: One NFS compound that reads three files. The operations can be divided into four groups: (a) sets the current and saved filehandle to `/home/Bob`; (b), (c), and (d) read the files `.bashrc`, `.bash_profile`, and `.bash_login`, respectively. SAVEFH *and* RESTOREFH *(in red) ensure that the CFH is `/home/Bob` when opening files. The reply is omitted.*

the *saved filehandle* (SFH), an NFSv4 state similar to the *current filehandle* (CFH). SAVEFH copies the CFH to the SFH; RESTOREFH restores the CFH from the SFH.

For compounds to reach their full potential, we need a file-system API that can convey high-level semantics and batch multiple operations. We designed and developed *vNFS*, an NFSv4 client that exposes a high-level vectorized API. vNFS complies with the NFSv4.1 standard, requiring no changes to NFS servers. Its API is easy to use and flexible enough to serve as a building block for new higher-level functions. vNFS is implemented entirely in user space, and thus easy to extend.

vNFS is especially efficient and convenient for applications that manipulate large amounts of metadata or do small I/Os. For example, vNFS lets `tar` read many small files using a single RPC instead of using multiple RPCs for each; it also lets `untar` set the attributes of many extracted files at once instead of making separate system calls for each attribute type (owner, time, etc.).

We implemented vNFS using the standard NFSv4.1 protocol, and added two small protocol extensions to support file appending and copying. We ported GNU's Coreutils package (`ls`, `cp`, and `rm`), `bsdtar`, `nghttp2` (an HTTP/2 server), and Filebench [15, 40] to vNFS. In general, we found it easy to modify applications to use vNFS. We ran a range of micro- and macro-benchmarks on networks with varying latencies, showing that vNFS can speed such applications by 3–133× with small network latencies (≤5.2ms), and by up to 263× with a 30.2ms latency.

The rest of this paper is organized as follows. Section 2 summarizes vNFS's design. Section 3 details the vectorized high-level API. Section 4 describes the implementation of our prototype. Section 5 evaluates the performance and usability of vNFS by benchmarking applications we ported. Section 6 discusses related work and Section 7 concludes.

2 Design Overview

In this section we summarize vNFS's design, including our goals, choices we made, and the architecture.

2.1 Design Goals

Our design has four goals, in order of importance:

- **High performance**: vNFS should considerably outperform existing NFS clients and improve both latency and throughput, especially for workloads that emphasize metadata and small I/Os. Performance for other workloads should be comparable.
- **Standards compliance**: vNFS should be fully compliant with the NFSv4.1 protocol so that it can be used with any compliant NFS server.
- **Easy adoption**: vNFS should provide a general API that is easy for programmers to use. It should be familiar to developers of POSIX-compliant code to enable smooth and incremental adoption.
- **Extensibility**: vNFS should make it easy to add functions to support new features and performance improvements. For example, it should be simple to add support for Server Side Copy (a feature in the current NFSv4.2 draft [17]) or create new application-specific high-level APIs.

2.2 Design Choices

The core idea of vNFS is to improve performance by using the compounding feature of standard NFS. We discuss the choices we faced and justify those we selected to meet the goals listed in Section 2.1.

Overt vs. covert coalescing. To leverage NFS compounds, vNFS uses a high-level API to overtly express the intention of compound operations. An alternative would be to covertly coalesce operations under the hood while still using the POSIX API. Covert coalescing is a common technique in storage and networking; for example, disk I/O schedulers combine many small requests into a few larger ones to minimize seeks [3]; and Nagle's TCP algorithm coalesces small outbound packets to amortize overhead for better network utilization [21].

Although overt compounding changes the API, we feel it is superior to covert coalescing in four important respects: **(1)** By using a high-level API, overt compounding can batch dependent operations, which are impossible to coalesce covertly. For example, using the

POSIX API, we cannot issue a `read` until we receive the reply from the preceding `open`. **(2)** Overt compounding can use a new API to express high-level semantics that cannot be efficiently conveyed in low-level primitives. NFSv4.2's Server Side Copy is one such example [17]. **(3)** Overt compounding improves both throughput and latency, whereas covert coalescing improves throughput at the cost of latency, since accumulating calls to batch together inherently requires waiting. Covert coalescing is thus detrimental to metadata operations and small I/Os that are limited by latency. This is important in modern systems with faster SSDs and 40GbE NICs, where latency has been improving much slower than raw network and storage bandwidth [33]. **(4)** Overt compounding allows implementations to use all possible information to maximize performance; covert coalescing depends on heuristics, such as timing and I/O sizes, that can be suboptimal or wrong. For example, Nagle's algorithm can interact badly with Delayed ACK [10].

Vectorized vs. start/end-based API. Two types of APIs can express overt compounding: a vectorized one that compounds many desired low-level NFS operations into a single high-level call, or an API that uses calls like `start_compound` and `end_compound` to combine all low-level calls in between [32]. We chose the vectorized API for two reasons: **(1)** A vectorized API is easier to implement than a start/end-based one. Users of a start/end-based API might mix I/Os with other code (such as looping and testing of file-system states), which NFS compounds cannot support. **(2)** A vectorized API logically resides at a high level and is more convenient to use, whereas using a low-level start/end-based API is more tedious for high-level tasks (similar to C++ programming vs. assembly).

User-space vs. in-kernel implementation. A kernel-space implementation of vNFS would allow it to take advantage of the kernel's page and metadata caches and use the existing NFS code base. However, we chose to design and implement vNFS in user space for two reasons: **(1)** Adding a user-space API is much easier than adding system calls to the kernel and simplifies future extensions; and **(2)** User-space development and debugging is faster and easier. Although an in-kernel implementation might be faster, prior work indicates that the performance impact can be minimal [39], and the results in this paper demonstrate substantial performance improvements even with our user-space approach.

2.3 Architecture

Figure 4 shows the architecture of vNFS, which consists of a library and a client. Instead of using the POSIX API, applications call the high-level vectorized API provided by the vNFS library, which talks directly to the vNFS client. The vNFS library facilitates application adoption,

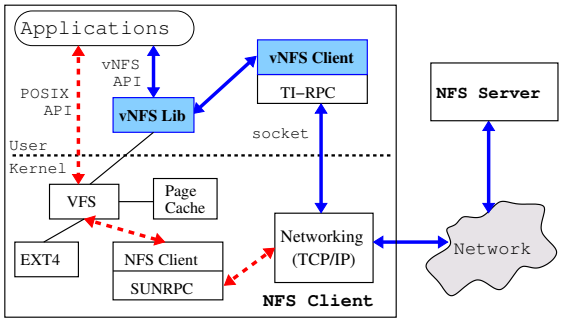


Figure 4: vNFS Architecture. The blue arrows show vNFS's data path, and the dashed red arrows show the in-kernel NFS client's data path. The vNFS library and client (blue shaded boxes) are new components we added; the rest already existed.

since most modern applications are developed using libraries and frameworks instead of OS system calls [2]. To provide generic support and encourage incremental adoption, the library detects when compound operations are unsupported, and in that case converts vNFS operations into standard POSIX primitives. Thus, the vNFS library can also be used with file systems that do not support compounding, e.g., as a utility library for batching file-system operations.

The vNFS client accepts vectorized operations from the library, puts as many of them into each compound as possible, sends them to the NFS server using Transport-Independent RPC (TI-RPC), and finally processes the reply. Note that existing NFSv4 servers already support compounds and can be used with vNFS without change. TI-RPC is a generic RPC library without the limitations of Linux's in-kernel SUNRPC (e.g., supporting only a single data buffer per call); TI-RPC can also run on top of TCP, UDP, and RDMA. Like the in-kernel NFS client, the vNFS client also manages NFSv4's client-side states such as sessions, etc.

3 vNFS API

This section details vNFS's vectorized API (listed in Table 1). Each API function expands its POSIX counterparts to operate on a vector of file-system objects (e.g., files, directories, symbolic links). vNFS functions handle errors in a standard manner: return results for successful operations, report the index of the first failed operation in a compound (if any), and ignore any remaining operations that were not executed by the server. Figure 5 demonstrates the use of vNFS API to read three small files in one NFS compound. To simplify programming, vNFS also provides utility functions for common tasks such as recursively removing a whole directory, etc.

`vread/vwrite`. These functions can read or write multiple files using a single compound, with automatic on-demand file opening and closing. These calls boost throughput, reduce latency, and simplify programming. Both accept a vector of I/O structures, each containing

Function	Description
vopen vclose	Open/close many files.
vread vwrite	Read/write/create/append files with automatic file opening and closing.
vgetattrs vsetattrs	Get/set multiple attributes of file-system objects.
vsscopy vcopy	Copy files in whole or in part with/out Server Side Copy.
vmkdir	Create directories.
vlistdir	List (recursively) objects and their attributes in directories.
vsymlink	Create many symbolic links.
vreadlink	Read many symbolic links.
vhardlink	Create many hard links.
vremove	Remove many objects.
vrename	Rename many objects.

Table 1: vNFS vectorized API functions. Each function has two return values: an error code and a count of successful operations. NFS servers stop processing the remaining operations in a compound once any operation inside failed. To facilitate gradual adoption, vNFS also provides POSIX-like scalar API functions, omitted here for brevity. Each vNFS function has a version that does not follow symbolic links, also omitted.

a `vfile` structure (Figure 5), offset, length, buffer, and flags. Our vectorized operations are more flexible than the `readv` and `writev` system calls, and can operate at many (discontinuous) offsets of multiple files in one call. When generating compound requests, vNFS adds OPENs and CLOSEs for files represented by paths; files represented by descriptors do not need that since they are already open. OPENs and CLOSEs are coalesced when possible, e.g. when reading twice from one file.

The length field in the I/O structure also serves as an output, returning the number of bytes read or written. The structure has several flags that map to NFS's internal Boolean arguments and replies. For example, the flag `is_creation` corresponds to the NFS `OPEN4_CREATE` flag, telling `vwrite` to create the target file if necessary. `is_write_stable` corresponds to NFS's WRITE `DATA_SYNC4` flag, causing the server to save the data to stable storage, avoiding a separate NFS COMMIT. Thus, a single `vwrite` can achieve the effect of multiple writes and a following `fsync`, which is a common I/O pattern (e.g., in logging or journaling).

■ **State management** NFSv4 is stateful, and OPEN is a state-mutating operation. The NFSv4 protocol requires a client to open a file before reading or writing it. Moreover, READ and WRITE must provide the *stateid* (an ID uniquely identifying a server's state [34]) returned by the preceding OPEN. Thus, state management is a key challenge when `vread` or `vwrite` adds OPEN and READ/WRITE calls into a single compound. vNFS solves this by using the NFS *current stateid*, which is a server-side state similar to the current filehandle. To ensure that the NFS server always uses the correct state, `vread` and `vwrite` take advantage of NFSv4's special support for using the current stateid [34, Section 8.2.3].

```
struct vfile {
  enum VFILETYPE type;    // PATH or DESCRIPTOR
  union {
    const char *path;  // When "type" is PATH,
    int fd;       // or (vNFS file) DESCRIPTOR.
  };
};
// The "vio" I/O structure contains a vfile.
struct vio ios[3] = {
  { .vfile = { .type = PATH,
               .path = "/home/Bob/.bashrc" },
    .offset = 0,
    .length = 64 * 1024,
    .data = buf1, // pre-allocated 64KB buffer
    .flags = 0,  // contains an output EOF bit
  }, ...   // two other I/O structures omitted
};
struct vres r = vread(ios, 3); // read 3 files
```

Figure 5: A simplified C code sample of reading three files at once using the vectorized API.

■ **Appending** `vwrite` also adds an optional small extension to the NFSv4.1 protocol to better support appends. As noted in the Linux manual page for `open(2)` [28], "O_APPEND may lead to corrupted files on NFS filesystems if more than one process appends data to a file at once." The base NFSv4 protocol does not support appending, so the kernel NFS client appends by writing to an offset equal to the current known file size. This behavior is inefficient as the file size must first be read separately, and it is vulnerable to TOCTTOU (time-of-check-to-time-of-use) attacks. Our extension uses a special offset value (`UINT64_MAX`) in the I/O structure to indicate appending, making appending reliable with a tiny (5 LoC) change to the NFS server.

vopen/vclose. Using `vread` and `vwrite`, applications can access files without explicit opens and closes. Our API still supports `vopen` and `vclose` operations, which add efficiency for large files that are read or written many times. `vopen` and `vclose` are also important for maintaining NFS's close-to-open cache consistency [25]. `vopen` opens multiple files (specified by paths) in one RPC, including LOOKUPs needed to locate their parent directories, as shown in Figure 3. Each file has its own open flags (read, write, create, etc.), which is useful when reading and writing are intermixed, such as external merge sorting. We also offer `vopen_simple`, which uses a common set of flags and mode (in case of creation) for all files. Once opened, a file is represented by a file descriptor, which is an integer index into an internal table that keeps states (file cursor, NFSv4 stateid and sequenceid [34], etc.) of open files. `vclose` closes multiple opened files and releases their resources.

vgetattrs/vsetattrs. These two functions manipulate several attributes of many files at once, combining multiple system calls (`chmod`, `chown`, `utimes`, and `truncate`, etc.) into a single compound, which is especially useful for tools like `tar` and `rsync`. The aging POSIX API is the only restriction on setting many attributes at once: the Linux kernel VFS already supports multi-attribute operations using the `setattr` method of `inode_operations`, and the NFSv4 protocol has similar SETATTRs support. `vgetattrs` and `vsetattrs` use an array of attribute structures as both inputs and outputs. Each structure contains a `vfile` structure, all attributes (mode, size, etc.), and a bitmap showing which attributes are in use.

vsscopy/vcopy. File copying is so common that Linux has added the `sendfile` and `splice` system calls to support it. Unfortunately, NFS does not yet support copying and clients must use READs and WRITEs instead, wasting time and bandwidth because data has to be read over the network and immediately written back. It is more efficient to ask the NFS server to copy the files directly on its side. This *Server Side Copy* (SSC) has already been proposed for the upcoming NFSv4.2 [17]. Being forward-looking, we included `vsscopy` in vNFS to copy many files (in whole or in part) using SSC; however, SSC requires server enhancements.

`vsscopy` accepts an array of copy structures, each containing the source file and offset, the destination file and offset, and the length. The destination files are created by `vsscopy` if necessary. The length can be `UINT64_MAX`, in which case the effective length is the distance between the source offset and the end of the source file. `vsscopy` can use a single RPC to copy many files in their entirety. The copy structures return the number of copied bytes in the length fields.

`vcopy` has the same effect but does not use SSC. `vcopy` is useful when the NFS server does not support SSC; `vcopy` can copy N small files using three RPCs (a compound for each of `vgetattrs`, `vread`, and `vwrite`) instead of $7 \times N$ RPCs (2 OPENs, 2 CLOSEs, 1 GETATTR, 1 READ, and 1 WRITE for each file). A future API could provide only `vcopy` and silently switch to `vsscopy` when SSC is available; we include `vsscopy` separately in this paper for comparison with `vcopy`.

vmkdir. vNFS provides `vmkdir` to create multiple directories at once (such as directory trees), which is common in tools such as `untar`, `cmake`, and recursive `cp`. vNFS's utility function `ensure_directory` uses `vmkdir` to ensure a deep directory and all its ancestors exist. Consider `"/a/b/c/d"` for example: the utility function first uses `vgetattrs` with arguments `["/a"; "/a/b"; ...]` to find out which ancestors exist and then creates the missing directories using `vmkdir`. Note that simply calling `vmkdir` with vector arguments `["/a"; "/a/b"; ...]` does not work: the NFS server will fail (with `EEXIST`) when trying to recreate the first existing ancestor and stop processing all remaining operations.

vlistdir. This function speeds up directory listing with four improvements to `readdir`: (1) `vlistdir` lists multiple directories at once; (2) a prior `opendir` is not necessary for listing; (3) `vlistdir` retrieves attributes along with directory entries, saving subsequent `stats`; (4) `vlistdir` can work recursively. It can be viewed as a fast vectorized `ftw(3)` that reads NFS directory contents using as few RPCs as possible.

`vlistdir` takes five arguments: an array of directories to list, a bitmap indicating desired attributes, a flag to select recursive listing, a user-defined callback function (similar to `ftw`'s second argument [27]), and a user-provided opaque pointer that is passed to the callback. `vlistdir` processes directories in the order given; recursion is breadth-first. However, directories at the same level in the tree are listed in an arbitrary order.

vsymlink/vreadlink/vhardlink. These three vNFS operations allow many links to be created or read at once. Together with `vlistdir`, `vsymlink` can optimize operations like `"cp -sr"` and `"lndir"`. All three functions accept a vector of paths and a vector of buffers containing the target paths.

vremove. `vremove` removes multiple files and directories at once. Although `vremove` does not support recursive removal, a program can achieve this effect with a recursive `vlistdir` followed by properly ordered `vremoves`; vNFS provides a utility function `rm_recursive` for this purpose.

vrename. Renaming many files and directories is common, for example when organizing media collections. Many tools [1, 22, 24, 45] have been developed just for this purpose. vNFS provides `vrename` to facilitate and speed up bulk renaming. `vrename` renames a vector of source paths to a vector of destination paths.

4 Implementation

We have implemented a prototype of vNFS in C/C++ on Linux. As shown in Figure 4, vNFS has a library and a client, both running in user space. The vNFS library implements the vNFS API. Applications use the library by including the API header file and linking to it. For NFS files, the library redirects API function calls to the vNFS client, which builds large compound requests and sends them to the server via the TI-RPC library. For non-NFS files, the library translates the API functions into POSIX calls, and therefore can also be used as a utility library. (Our current prototype considers a file to be on NFS if it is under any exported directory specified in vNFS's configuration file.) The vNFS client builds on NFS-Ganesha [12, 30], an open-source user-space NFS

server. NFS-Ganesha can export files stored in many backends, such as XFS and GPFS. Our vNFS prototype uses an NFS-Ganesha backend called PROXY, which exports files from *another* NFS server and can be repurposed as a user-space NFS client. The original PROXY backend used NFSv4.0; we added NFSv4.1 support including session management [34]. Our prototype implementation added 10,632 lines of C/C++ code and deleted 1,407. vNFS is thread-safe; we have tested it thoroughly.

RPC size limit. The vNFS API functions (shown in Table 1) do not impose a limit on the number of operations per call. However, each RPC has a configurable memory size limit, defaulting to 1MB. We ensure that vNFS does not generate RPC requests larger than that limit no matter how many operations an API call contains. Therefore, we split long arguments into chunks and send one compound request for each chunk. We also merge RPC replies upon return, to hide any splitting.

Our splitting avoids generating small compounds. For data operations (`vread` and `vwrite`), we can easily estimate the sizes of requests and replies based on buffer lengths, so we split a compound only when its size becomes close to 1MB. (The in-kernel NFS client similarly splits large READs and WRITEs according to the `rsize` and `wsize` mount options, which also default to 1MB.) For metadata operations, it is more difficult to estimate the reply sizes, especially for READDIR and GETATTR. We chose to be conservative and simply split a compound of metadata operations whenever it contains more than k NFS operations. We chose a default of 256 for k, which enables efficient concurrent processing by the NFS server, and yet is unlikely to exceed the size limit. For example, when listing the Linux source tree, the average reply size of READDIR—the largest metadata operation—is around 3,800 bytes. If k is still too large (e.g., when listing large directories), the server will return partial results and use cookies to indicate where to resume the call for follow-up requests.

Protocol extensions. vNFS contains two extensions to the NFSv4.1 protocol to support file appending (see Section 3 [¶`vread`/`vwrite`]) and Server Side Copy (see Section 3 [¶`vsscopy`/`vcopy`]). Both extensions require changes to the protocol and the NFS server. We have implemented these changes in our server, which is based on NFS-Ganesha [11, 12, 30]. The file-appending extension was easy to implement, adding only an `if` statement with 5 lines of C code. In the NFS server, we only need to use the file size as the effective offset whenever the write offset is `UINT64_MAX`.

Our implementation of Server Side Copy follows the design proposed in the NFSv4.2 draft [17]. We added the new COPY operation to our vNFS client and the NFS-Ganesha server. On the server side, we copy data using `splice(2)`, which avoids unnecessarily moving data across the kernel/user boundary. This extension added 944 lines of C code to the NFS-Ganesha server.

Path compression. We created an optimization that reduces the number of LOOKUPs when a compound's file paths have locality. The idea is to shorten paths that have redundancy by making them relative to preceding ones in the same compound. For example, when listing the directories `"/1/2/3/4/5/6/7/a"` and `"/1/2/3/4/5/6/7/b"`, a naïve implementation would generate eight LOOKUPs per directory (one per component). In such cases, we replace the path of the second directory with `"../b"` and use only one LOOKUPP and one LOOKUP; LOOKUPP sets the current filehandle to its parent directory. This simple technique saves as many as six NFS operations for this example.

Note that LOOKUPP produces an error if the current filehandle is not a directory, because most file systems have metadata recording parents of directories, but not parents of files. In that case, we use SAVEFH to remember the deepest common ancestor in the file-system tree (i.e., `"/1/2/3/4/5/6/7"` in the above example) of two adjacent files, and then generate a RESTOREFH and LOOKUPs. (However, this approach cannot be used for LINK, RENAME, and COPY, which already use the saved filehandle for other purposes.) We use this optimization only when it saves NFS operations: for example, using `"../../c/d"` does not save anything for paths `"/1/a/b"` and `"/1/c/d"`.

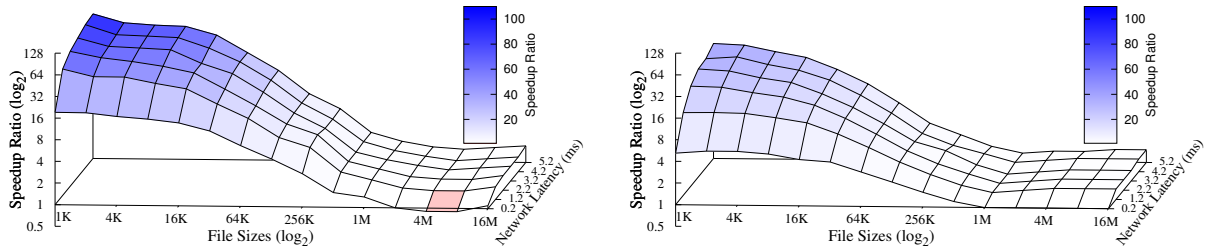
Client-side caching. Our vNFS prototype does not yet have a client-side cache, which would be useful for rereading recent data and metadata, streaming reads, and asynchronous writes. We plan to add it in the future. Compared to traditional NFS clients, vNFS does not complicate failure handling in the presence of a dirty client-side cache: cached dirty pages (not backed by persistent storage) are simply dropped upon a client crash; dirty data in a persistent cache (e.g., FS-Cache [19]), which may be used by a client holding write delegations, can be written to the server even faster during client crash recovery. Note that a client-side cache does not hold dirty metadata because all metadata changes are performed synchronously in NFS (except with directory delegations, which Linux has not yet implemented).

5 Evaluation

To evaluate vNFS, we ran micro-benchmarks and also ported applications to use it. We now discuss our porting experience and evaluate the resulting performance.

5.1 Experimental Testbed Setup

Our testbed consists of two identical Dell PowerEdge R710 machines running CentOS 7.0 with a 3.14 Linux kernel. Each machine has a six-core Intel Xeon X5650 CPU, 64GB of RAM, and an Intel 10GbE NIC. One ma-



(a) Reading whole files

(b) Writing whole files

Figure 6: vNFS's speedup ratio (the vertical Z-axis, in logarithmic scale) relative to the baseline when reading and writing 1,000 equally-sized files, whose sizes (the X-axis) varied from 1KB to 16MB. vNFS is faster than (blue), equal to (white), or slower than (red) the baseline when the speedup ratio is larger than, equal to, or smaller than 1.0, respectively. The network latency (Y-axis) starts from 0.2ms (instead of zero) because that is the measured base latency of our testbed (see Section 5.1).

chine acts as the NFS server and runs NFS-Ganesha with our file-appending and Server Side Copy extensions; the other machine acts as a client and runs vNFS. The NFS server exports to the client an Ext4 file system, stored on an Intel DC S3700 200GB SSD. The two machines are directly connected to a Dell PowerConnect 8024F 10GbE switch, and we measured an average RTT of 0.2ms between them. To emulate different LAN and WAN conditions, we injected delays of 1–30ms into the outbound link of the server using `netem`.

To evaluate vNFS's performance, we compared it with the in-kernel NFSv4.1 client (called *baseline*), which mounts the exported directory using the default options: the attribute cache (`ac` option) is enabled and the maximum read/write size (`rsize`/`wsize` options) is 1MB. Our vNFS prototype does not use `mount`, but instead reads the exported directory from a configuration file. We ran each experiment at least three times and plotted the average value. We show the standard deviation as error bars, which are invisible in most figures because of their tiny values. Before each run, we flushed the page and dentry caches of the in-kernel client by unmounting and re-mounting the NFS directory. vNFS has no cache. The NFS-Ganesha server uses an internal cache, plus the OS's page and dentry caches. To quantify the effort of porting applications, we report the LoC change for each application including the error-handling code.

5.2 Micro-workloads

Small vs. big files. vNFS's goal is to improve performance for workloads with many small NFS operations, while staying competitive for data-intensive workloads. To test this, we compared the time used by vNFS and the baseline to read and write 1,000 equally-sized files in their entirety while varying the file size from 1KB to 16MB. We repeated the experiment in networks with 0.2ms to 5.2ms latencies, and packed as many operations as possible into each vNFS compound. The results are shown (in logarithmic scale) in Figure 6, where *speedup ratio* is the ratio of the baseline's completion time to vNFS's completion time. Speedup ratios greater than one mean that vNFS performed better than the baseline; ratios less than one mean vNFS performed worse.

Because vNFS combined many small read and write operations into large compounds, it performed much better than the baseline when the file size was small. With a 1KB file size and 0.2ms network latency, vNFS is 19× faster than the baseline when reading (Figure 6(a)), and 5× faster when writing (Figure 6(b)). As the network latency increased to 5.2ms, vNFS's speedup ratio improved further to 103× for reading and 40× for writing. vNFS's speedup ratio was higher for reading than for writing because once vNFS was able to eliminate most network round trips, the NFS server's own storage became the next dominant bottleneck.

As the file size (the X-axis in Figure 6) was increased to 1MB and beyond, vNFS's compounding effect faded, and the performance of vNFS and the baseline became closer. However, in networks with 1.2–5.2ms latency, vNFS was still 1.1–1.7× faster than the baseline: although data operations were too large to be combined together, vNFS could still combine them with small metadata operations such as OPEN, CLOSE, and GETATTR. Combining metadata and data operations requires vNFS to split I/Os below 1MB due to the 1MB RPC size limit (see Section 4). When a large I/O is split into pieces, the last one may be small; this phenomenon made vNFS around 10% slower when reading 4MB and 8MB files in the 0.2ms-latency network. However, this is not a problem in most cases because that last small piece is likely to be combined into later compounds. This is why vNFS performed the same as the baseline with even larger file sizes (e.g., 16MB) in the 0.2ms-latency network. This negative effect of vNFS's splitting was unnoticeable for writing because writing was bottlenecked by the NFS server's storage. Note that the baseline (the in-kernel NFS client) splits I/Os strictly at the 1MB size, although it also adds a few trivial NFS operations such as PUTFH (see Figure 1) in its compounds, meaning that the baseline's RPC size is actually larger than 1MB.

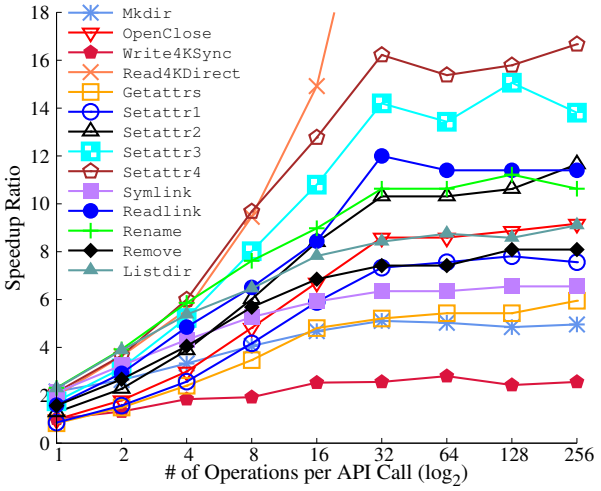


Figure 7: vNFS's speedup ratio relative to the baseline under different degrees of compounding. The X-axis is $\log_2$. The network latency is 0.2ms. Write4KSync writes 4KB data to files opened with `O_SYNC`; Read4KDirect reads 4KB data from files opened with `O_DIRECT`; `SetattrN` sets N files' attributes (mixes of mode, owner, timestamp, and size). The vector size of the baseline is actually the number of individual POSIX calls issued iteratively. The speedup ratio of Read4KDirect goes up to 46 at 256 operations per call; its curve is cut off here.

Compounding degree. The degree of compounding (i.e., the number of non-trivial NFS operations per compound) is a key factor determining how much vNFS can boost performance. The ideal is to perform a large number of file system operations at once, which is not always possible because applications may have critical paths that depend on only a single file. To study how the degree of compounding affects vNFS's performance, we compared vNFS with the baseline when calling the vNFS API functions with different numbers of operations in their vector arguments.

Figure 7 shows the speedup ratio of vNFS relative to the baseline as the number of operations per API call was increased from 1 to 256 in the 0.2ms-latency network. Even with one operation per call, vNFS outperformed the baseline for all API functions except two, because vNFS could still save round trips for single-file calls. For example, the baseline used three RPCs to rename a file: one LOOKUP for the source directory, another LOOKUP for the destination directory, and one RENAME; vNFS, however, used only one compound RPC combining all three operations. `Getattrs` and `Setattr1` are the two exceptions where vNFS performed slightly worse (17% and 14% respectively) than the baseline. This is because each of these two calls needs only a single NFS operation; so vNFS could not combine anything yet incurred the overhead of performing RPCs in user space.

When there was more than one operation per API call, compounding became effective and vNFS significantly outperformed the baseline for all calls; note that the Y axis of Figure 7 is in logarithmic scale. All calls except `Write4KSync` (bottlenecked by the server's storage stack) were more than 4× faster than the baseline when multiple operations were compounded. Note that `vsetattrs` can set multiple attributes at once, whereas the baseline sets one attribute at a time. We observe in Figure 7 that the speedup ratio of setting more attributes (e.g., `Setattr4`) at once was always higher than that of setting fewer (e.g., `Setattr3`).

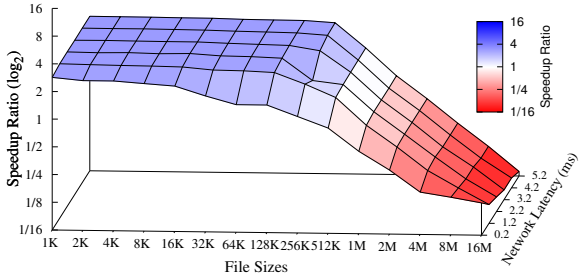


Figure 8: The speedup ratio of vNFS over the baseline (in logarithmic scale) when repeatedly opening, reading, and closing a single file, whose size is shown on the X-axis. vNFS is faster than (blue), equal to (white), and slower than (red) the baseline when the speedup ratio is larger than, equal to, and smaller than 1, respectively. Our vNFS prototype does not have a cache yet, whereas the baseline does. The Z-axis is in logarithmic scale; the higher the better.

In our experiments with slower networks (omitted for brevity), vNFS's speedups relative to the baseline were even higher than in the 0.2ms-latency network: up to two orders of magnitude faster.

Caching. Our vNFS prototype does not yet support caching. In contrast, the baseline (in-kernel NFS client) caches both metadata and data. To study the cache's performance impact, we compared vNFS and the baseline when repeatedly opening, reading, and closing a single file whose size varied from 1KB to 16MB. Figure 8 shows the results, where a speedup ratio larger than one means vNFS outperformed the baseline; and a speedup ratio less than one means vNFS performed worse.

The baseline served all reads except the first from its cache, but it was slower than vNFS (which did not cache) when the file size was 256KB or smaller. This is because three RPCs per read are still required to maintain close-to-open semantics: an OPEN, a GETATTR (for cache revalidation), and a CLOSE. In comparison, vNFS used only one compound RPC, combining the OPEN, READ (uncached), and CLOSE. The savings from compounding more than compensated for vNFS's lack of a cache. For a 512KB file size, vNFS was still faster than the baseline except in the 0.2ms-latency network. For 1MB and larger files, vNFS was worse than the baseline because read operations dominated: the baseline served all reads from its client-side cache whereas vNFS sent all reads to the server without the benefit of caching.

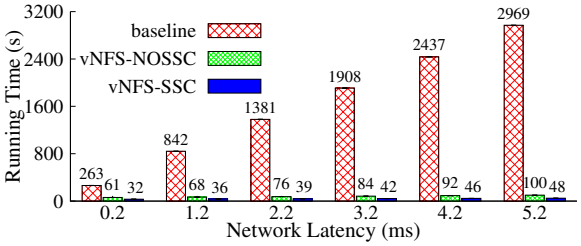


Figure 9: Running time to copy (`cp -r`) the entire Linux source tree. The lower the better. vNFS runs much faster than the baseline both with and without Server Side Copy (SSC).

5.3 Macro-workloads

To evaluate vNFS using realistic applications, we modified `cp`, `ls`, and `rm` from GNU Coreutils, Filebench [15, 40], and nghttp2 [31] to use the vNFS API; we also implemented an equivalent of GNU `tar` using vNFS.

GNU Coreutils. Porting `cp` and `rm` to vNFS was easy. For `cp`, we added 170 lines of code and deleted 16; for `rm`, we added 21 and deleted 1. Copying files can be trivially achieved using `vsscopy`, `vgetattrs`, and `vsetattrs`. Recursively copying directories requires calling `vlistdir` on the directories and then invoking `vsscopy` for plain files, `vmkdir` for directories, and `vsymlink` for symbolic links—all of which is done in `vlistdir`'s callback function. We tested our modified `cp` with `diff -qr` to ensure that the copied files and directories were exactly the same as the source. Removing files and directories recursively in `rm` was similar, except that we used `vremove` instead of `vsscopy`.

Porting `ls` was more complex because batching is difficult when listing directories recursively in a particular order (e.g., by file size). We could not use the recursive mode of `vlistdir` because the NFS READDIR operation does not follow any specific order when reading directory entries, and the whole directory tree may be too large to fit in memory. Instead, vNFS maintains a list of all directories to read in the proper order as specified by the `ls` options, and repeatedly calls `vlistdir` (not recursively) on directories at the head of the list until it is empty. Note that (1) a directory is removed from the list only after all its children have been read; and (2) sub-directories should be sorted and then inserted immediately after their parent to maintain the proper order in the list. We added 392 lines of code and deleted 203 to port `ls` to vNFS. We verified that our port is correct by comparing the outputs of our `ls` with the vanilla version.

We used the ported Coreutils programs to copy, list, and remove an entire Linux-4.6.3 source tree: it contains 53,640 files with an average size of 11.6KB, 3,604 directories with average 17 children per directory, and 23 symbolic links. The large number of files and directories thoroughly exercises vNFS and demonstrates the performance impact of compounding.

Figure 9 shows the results of copying the entire Linux source tree; vNFS outperformed the baseline in all cases. vNFS uses either `vsscopy` or `vcopy` depending on whether Server Side Copy (SSC) is enabled. However, the baseline cannot use SSC because it is not yet supported by the in-kernel NFS client. For the same workload of copying the Linux source tree, vNFS used merely 4,447 compounding RPCs whereas the baseline used as many as 506,697: two OPENs, two CLOSEs, one READ, one WRITE, and one SETATTR for each of the 53,640 files; 60,873 ACCESSes; 62,327 GETATTRs; and 8,017 other operations such as READDIR and CREATE. vNFS-NOSSC saved more than 99% of RPCs compared to the baseline, with each vNFS compounding RPC containing an average of 250 operations. Therefore, even with only a 0.2ms network latency, vNFS-NOSSC is still more than 4× faster than the baseline. The speedup ratio increases to 30× with a 5.2ms network latency.

When Server Side Copy (SSC) was enabled, vNFS ran even faster, and vNFS-SSC reduced the running time of vNFS-NOSSC by half. The further speedup of SSC is only moderate because the files are small and our network bandwidth (10GbE) is large. The speedup ratio of vNFS-SSC to the baseline is 8–60× in networks with 0.2–5.2ms latency. Even when the baseline adds SSC support in the future, vNFS would still outperform it because this workload's bottleneck is the large number of small metadata operations, not data-copying operations.

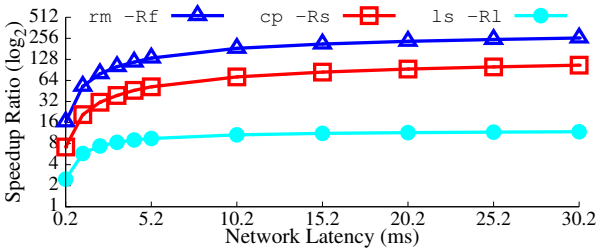


Figure 10: vNFS's speedup relative to the baseline when symbolically copying (`cp -Rs`), listing (`ls -Rl`), and removing (`rm -Rf`) the Linux source tree. The Y-axis is logarithmic.

With the `-Rs` options, `cp` copies an entire directory tree by creating symbolic links to the source directory. Figure 10 shows speedups for symlinking, recursively listing (`ls -Rl`), and removing (`rm -Rf`) the Linux source tree. When recursively listing the Linux tree, `ls`-baseline used 10,849 RPCs including 3,678 READDIRs, 3,570 ACCESSes, and 3,570 GETATTRs. Note that the in-kernel NFS client did not issue a separate GETATTR for each directory entry although the vanilla `ls` program called `stat` for each entry listed. This is because the in-kernel NFS client pre-fetches the attributes using `readdirs` and serves the `stat` calls from the local client's dentry metadata cache. This optimization enables `ls`-baseline to finish the benchmark in just 5 sec-

onds in the 0.2ms-latency network. However, with our vectorized API, `ls`-vNFS did even better and finished in 2 seconds, using only 856 RPCs. Moreover, vNFS scales much better than the baseline. When the latency increased from 0.2 to 30.2ms, vNFS's running time rose to only 28 seconds whereas the baseline increased to 336 seconds. `ls`-vNFS is 10× faster than `ls`-baseline in high-latency (>5.2ms) networks.

For symbolic copying and removing (Figure 10), vNFS was 7× and 18× faster than the baseline in the 0.2ms-latency network, respectively. This is because the baseline always operated on one file at a time, whereas vNFS could copy or remove more than 200 files at once. Compared to the baseline, vNFS improved `cp` by 52× and `rm` by 133× in the 5.2ms-latency network; with 30.2ms latency the speedup ratios became 106× for `cp`, and 263× for `rm`. For both removing and symbolic copying, vNFS ran faster in the 30.2ms-latency network (25 and 15 seconds, respectively) than the baseline did with 0.2ms latency (38s and 55s, respectively), showing that compounds can indeed help NFSv4 realize its design goal of being WAN-friendly [29].

`tar`. Because the I/O code in GNU `tar` is closely coupled to other code, we implemented a vNFS equivalent using `libarchive`, in which the I/O code is clearly separated. The `libarchive` library supports many archiving and compression algorithms; it is also used by FreeBSD `bsdtar`. Our implementation needed only 248 lines of C++ code for `tar` and 210 for `untar`, both including error-handling code.

When archiving a directory, we use the `vlistdir` API to traverse the tree and add sub-directories into the archive. We gather the listed files and symlinks into arrays, then read their contents using `vread` and `vreadlink`, and finally compress and write the contents into the archive. During extraction, we read the archive in 1MB (RPC size limit) chunks and then use `libarchive` to extract and decompress objects and their contents, which are then passed in batches to `vmkdir`, `vwrite`, or `vsymlink`. We always create parent directories before their children. We ensured that our implementation is correct by feeding our `tar`'s output into our `untar` and comparing the extracted files with the original input files. We also tested for cross-compatibility with other tar implementations including `bsdtar` and GNU `tar`.

We used our `tar` to archive and `untar` to extract a Linux 4.6.3 source tree. Archiving read 53,640 small files and wrote a large archive: 636MB uncompressed, and 86MB with the `xz` option (default compression used by `kernel.org`). Extracting reversed the process. There were also metadata operations on 23 symbolic links and 3,604 directories. Figure 11 shows the `tar`/`untar` results, compared to

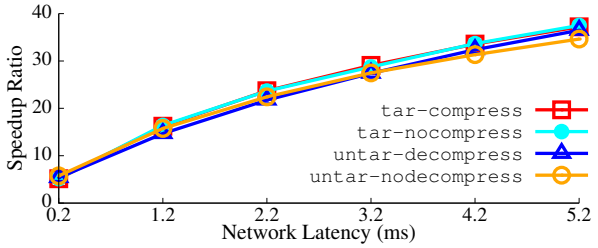


Figure 11: Speedup ratios of vNFS relative to the baseline (`bsdtar`) when archiving and extracting the Linux-4.6.3 source tree, with and without xz compression.

`bsdtar` (running on the in-kernel client) as the baseline. For `tar-nocompress` in the 0.2ms-latency network, vNFS was more than 5× faster than the baseline because the baseline used 446,965 RPCs whereas vNFS used only 2,144 due to compounding. This large reduction made vNFS 37× faster when the network latency increased to 5.2ms. In terms of running time, vNFS used 69 seconds to archive the entire Linux source tree in the 5.2ms-latency network, whereas the baseline, even in the faster 0.2ms-latency network, still used as much as 192 seconds. For `untar-nodecompress`, vNFS is also 5–36× faster, depending on the network latency.

Figure 11 also includes the results when `xz` compression was enabled. Although compression reduced the size of the archive file by 86% (from 636MB to 86MB) and thus saved 86% of the I/Os to the archive file, it had a negligible performance impact (less than 0.5%) because the most time-consuming operations were for small I/Os, not large ones. This test shows that workloads with mixed I/O sizes are slow if there are many small I/Os, each incurring a network round trip; vNFS can significantly improve such workloads by compounding those small I/Os.

Filebench. We have ported Filebench to vNFS and added vectorized *flowops* to the Filebench workload modeling language (WML) [46]. We added 759 lines of C code to Filebench, and removed 37. We converted Filebench's File-Server, NFS-Server, and Varmail workloads to equivalent versions using the new flowops: for example, we replaced N adjacent sets of `openfile`, `readwholefile`, and `closefile` (i.e., $3 \times N$ old flowops) with a single `vreadfile` (one new flowop), which internally uses our `vread` API that can open, read, and close N files in one call.

The Filebench NFS-Server workload emulates the SPEC SFS benchmark [36]. It contains one thread performing four sets of operations: (1) open, entirely read, and close three files; (2) read a file, create a file, and delete a file; (3) append to an existing file; and (4) read a file's attributes. The File-Server workload emulates 50 users accessing their home directories and spawns one thread per user to perform operations similar to the NFS-Server workload. The Varmail workload mimics

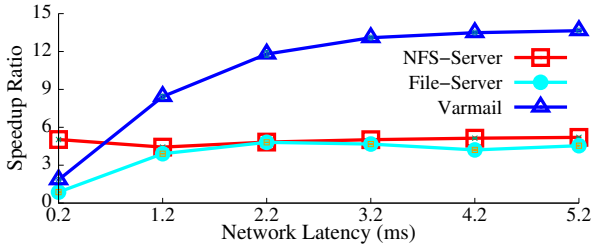


Figure 12: vNFS's speedup ratios for Filebench workloads.

a UNIX-style email server operating on a `/var/mail` directory, saving each message as a file; it has 16 threads, each performing create-append-sync, read-append-sync, read, and delete operations on 10,000 16KB files.

Figure 12 shows the results of the Filebench workloads, comparing vNFS to the baseline. For the NFS-Server workload, vNFS was 5× faster than the baseline in the 0.2ms-latency network because vNFS combined multiple small reads and their enclosing opens and closes into a single compound. vNFS was also more efficient (and more reliable) when appending files since it does not need a separate GETATTR to read the file size (see Section 3 [¶`vread/vwrite`]). This single-threaded NFS-Server workload is light, and its only bottleneck is the delay of network round trips. With compounding, vNFS can save network round trips; the amount of savings depends on the compounding degree (the number of non-trivial NFS operations per compound). This workload has a compounding degree of around 5, and thus we observed a consistent 5× speedup regardless of the network latency.

As shown in Figure 12, vNFS's speedup ratio in the File-Server workload is about the same as the NFS-Server one, except in the 0.2ms-latency network. This is because these two workloads have similar file-system operations and thus similar compounding degrees. However, in the 0.2ms-latency network, vNFS was 13% slower (i.e., a speedup ratio of 0.87) than the baseline. This is caused by two factors: (1) the File-Server workload has as many as 50 threads and generates a heavy I/O load to the NFS server's storage stack, which became the bottleneck; (2) without a cache, vNFS sent all read requests to the overloaded server whereas the in-kernel client's cache absorbed more than 99% of reads. As the network latency increased, the load on the NFS server became lighter and vNFS became faster thanks to saving round trips, which more than compensated for the lack of caching in our current prototype.

Because the Varmail workload is also multi-threaded, its speedup ratio curve in Figure 12 has a trend similar to that of the File-Server workload. However, vNFS's speedup ratio in the Varmail workload plateaued at the higher value of 14× because its compounding degree is higher than the File-Server workload.

Network Latency (ms)	0.2	1.2	2.2	3.2	4.2	5.2
Speedup Ratio	3.5	6.5	7.1	8.7	9.8	9.9

Table 2: vNFS speedup ratio relative to the baseline when requesting a set of objects with PUSH enabled in nghttp2.

HTTP/2 server. Similar to the concept of NFSv4 compounds, HTTP/2 improves on HTTP/1.*x* by transferring multiple objects in one TCP connection. HTTP/2 also added a PUSH feature that allows an HTTP/2 server to proactively push related Web objects to clients [5, Section 8.2]. For example, upon receiving an HTTP/2 request for `index.html`, the server can proactively send the client other Web objects (such as Javascript, CSS, and image files) embedded inside that `index.html` file, instead of waiting for it to request them later. PUSH can reduce a Web site's loading time for end users. It also allows Web servers to read many related files together, enabling efficient processing by vNFS.

We ported *nghttp2* [31], an HTTP/2 library and toolset containing an HTTP/2 server and client, to vNFS. Our port added 543 lines of C++ code and deleted 108.

The HTTP Archive [20] shows that, on average, an HTTP URL is 2,480KB and contains ten 5.5KB HTML files, 23 20KB Javascript files, seven 7.5KB CSS files, and 56 28KB image files. We created a set of files with those characteristics, hosted them with our modified nghttp2 server, and measured the time needed to process a PUSH-enabled request to read the file set. Table 2 shows the speedup ratio of vNFS relative to the baseline, which runs vanilla nghttp2 and the in-kernel NFS client. vNFS needed only four NFS compounds for all 96 files: one `vgetattrs` call and three `vreads`. In contrast, the baseline used 309 RPCs including one OPEN, READ, and CLOSE for each file. The reduced network round trips made vNFS 3.5× faster in the 0.2ms-latency network and 9.9× faster with the 5.2ms latency.

6 Related Work

Improving NFS performance. NFS is more than 30 years old, and has continuously evolved to improve performance. Following the initial NFSv2 [38], NFSv3 added asynchronous COMMITs to improve write performance, and READDIRPLUS to speed up directory listing [7]. NFSv4.0 [35] added more performance features including compounding procedures that batch multiple operations in one RPC, and delegations that enable the client cache to be used without lengthy revalidation. To improve performance further with more parallelism, NFSv4.1 [34] added pNFS [18] to separate data and meta-data servers so that the different request types can be served in parallel. The upcoming NFSv4.2 has yet more performance improvements such as I/O hints, *Application Data Blocks*, and Server Side Copy [17].

In addition to improvements in the protocols, other researchers also improved NFS's performance: Duchamp

found it inefficient to look up NFSv2 paths one component at a time, and reduced client latency and server load by optimistically looking up whole paths in a single RPC [13]. Juszczak improved the write performance of an NFS server by gathering many small writes into fewer larger ones [23]. Ellard and Seltzer improved read performance with read-ahead and stride-read algorithms [14]. Batsakis et al. [4] developed a holistic framework that adaptively schedules asynchronous operations to improve NFS's performance as perceived by applications. Our vNFS uses a different approach, improving performance by making NFSv4's compounding procedures easily accessible to programmers.

I/O compounding. Compounding, also called batching and coalescing, is a popular technique to improve throughput and amortize cost by combining many small I/Os into fewer larger ones. Disk I/O schedulers coalesce adjacent I/Os to reduce disk seeks [3] and boost throughput. Purohit et al. [32] proposed Compound System Calls (Cosy) to amortize the cost of context switches and to reduce data movement across the user-kernel boundary. These compounding techniques are all hidden behind the POSIX file-system API, which cannot convey the required high-level semantics [8]. The Batch-Aware Distributed File System (BAD-FS) [6] demonstrated the benefits of using high-level semantics to explicitly control the batching of I/O-intensive scientific workloads. *Dynamic sets* [37] took advantage of the fact that files can be processed in any order in many bulk file-system operations (e.g., `grep foo *.c`). Using a set-based API, distributed file system clients can pre-fetch a set of files in the optimal order and pace so that computation and I/O are overlapped and the overall latency is minimized. However, dynamic sets did not reduce the number of network round trips. SeMiNAS [9] uses NFSv4 compounds (only) in its security middleware to reduce the security overhead. To the best of our knowledge, vNFS is the first attempt to use an overt-compounding API to leverage NFSv4's compounding procedures.

Vectorized APIs. To achieve high throughput, Vilayanur et al. [43] proposed `readx` and `writex` to operate at a vector of offsets so that the I/Os could be processed in parallel. However, these operations were limited to a single file, helping only large files, whereas our `vread`/`vwrite` can access many files at once, helping with both large and small files.

Vasudevan et al. [41] envisioned the Vector OS (VOS), which offered several vectorized system calls, such as `vec_open()`, `vec_read()`, etc. While VOS is promising, it has not yet been fully implemented. In their prototype, they succeeded in delivering millions of IOPS in a distributed key-value (KV) store backed by fast NVM [42]. However, they implemented a key-value API, not a file-system API, and their vectorized KV store focuses on serving parallel I/Os on NVM, whereas vNFS focuses on saving network round trips by using NFSv4 compound procedures. The vectorized key-value store and vNFS are different but complementary.

Our vNFS API is also different from other vectorized APIs [41,43] in three aspects: (1) `vread`/`vwrite` supports automatic file opening and closing; (2) `vsscopy` takes advantage of the NFS-specific Server Side Copy feature; and (3) to remain NFSv4-compliant, vNFS's vectorized operations are executed in order, in contrast to the out-of-order execution of `lio_listio(3)` [26], `vec_read()` [41], and `readx` [43].

7 Conclusions

We designed and implemented vNFS, a file-system library that maximizes NFS performance. vNFS uses a vectorized high-level API to leverage standard NFSv4 compounds, which have the potential to reduce network round trips but were underused due to the low-level and serialized nature of the POSIX API. vNFS makes maximal use of compounds by enabling applications to operate on many file-system objects in a single RPC. vNFS complies with the NFSv4.1 protocol and has standard failure semantics. To help port applications to the vectorized API, vNFS provides a superset of POSIX file-system operations, and its library can be used for non-NFS file systems as well. We found it generally easy to port applications including `cp`, `ls` and `rm` from GNU Coreutils; `bsdtar`; Filebench; and nghttp2.

Micro-benchmarks demonstrated that—compared to the in-kernel NFS client—vNFS significantly helps workloads with many small I/Os and metadata operations even in fast networks, and performs comparably for large I/Os or with low compounding degrees. Macro-benchmarks show that vNFS sped up the ported applications by up to two orders of magnitude. Our source code is available at *https://github.com/sbu-fsl/txn-compound*.

Limitations and future work. Currently vNFS does not include a cache; an implementation is underway. To simplify error handling, we plan to support optionally executing a compound as an atomic transaction. Finally, compounded operations are processed sequentially by current NFS servers; we plan to execute them in parallel with careful interoperation with transactional semantics.

Acknowledgments

We thank the anonymous FAST reviewers and our shepherd Keith Smith for their valuable comments. We also thank Geoff Kuenning for his meticulous and insightful review comments. This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, and CNS-1622832; and ONR award 12055763.

References

[1] Antoine Potten. Ant renamer, 2016. *http://www.antp.be/software/renamer*.

[2] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 19. ACM, 2016.

[3] J. Axboe. CFQ IO scheduler, 2007. *http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.ogg*.

[4] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey. CA-NFS: A congestion-aware network file system. *ACM Transaction on Storage*, 5(4), 2009.

[5] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Internet Engineering Task Force, May 2015.

[6] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in the batch-aware distributed file system. In *NSDI*, pages 365–378, 2004.

[7] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, Network Working Group, June 1995.

[8] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, B. Singh, and E. Zadok. Newer is sometimes better: An evaluation of NFSv4.1. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*, Portland, OR, June 2015. ACM.

[9] M. Chen, A. Vasudevan, K. Wang, and E. Zadok. SeMiNAS: A secure middleware for wide-area network-attached storage. In *Proceedings of the 9th ACM International Systems and Storage Conference (ACM SYSTOR '16)*, Haifa, Israel, June 2016. ACM.

[10] Stuart Cheshire. TCP performance problems caused by interaction between Nagle's algorithm and delayed ACK, May 2005.

[11] Philippe Deniel. GANESHA, a multi-usage with large cache NFSv4 server. *www.usenix.org/events/fast07/wips/deniel.pdf*, 2007.

[12] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. GANESHA, a multi-usage with large cache NFSv4 server. In *Linux Symposium*, page 113, 2007.

[13] D. Duchamp. Optimistic lookup of whole NFS paths in a single operation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 143–170, Boston, MA, June 1994.

[14] D. Ellard and M. Seltzer. NFS tricks and benchmarking traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, San Antonio, TX, June 2003. USENIX Association.

[15] Filebench, 2016. *https://github.com/filebench/filebench/wiki*.

[16] S. Han, S. Marshall, B. Chun, and S. Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[17] T. Haynes. NFS version 4 minor version 2 protocol. RFC draft, Network Working Group, September 2015. *https://tools.ietf.org/html/draft-ietf-nfsv4-minorversion2-39*.

[18] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of MSST*, Monterey, CA, 2005. IEEE.

[19] D. Howells. FS-Cache: A Network Filesystem Caching Facility. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 427–440, Ottawa, Canada, July 2006. Linux Symposium.

[20] HTTP Archive. URL statistics, September 2016. *http://httparchive.org/trends.php*.

[21] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, Network Working Group, January 1984.

[22] Jason Fitzpatrick. Bulk rename utility, 2016. *http://www.bulkrenameutility.co.uk/Main_Intro.php*.

[23] Chet Juszczak. Improving the write performance of an NFS server. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, San Francisco, California, 1994. USENIX Association.

[24] Kim Jensen. AdvancedRenamer, 2016. *https://www.advancedrenamer.com/*.

[25] Chuck Lever. Close-to-open cache consistency in the Linux NFS client. *http://goo.gl/o9i0MM*.

[26] Linux Programmer's Manual. *lio_listio*, September 2016. *http://man7.org/linux/man-pages/man3/lio_listio.3.html*.

[27] Linux man pages. ftw(3) - file tree walk. *http://linux.die.net/man/3/ftw*.

[28] Linux man pages. open(2) - open and possibly create a file or device. *http://linux.die.net/man/2/open*.

[29] Alex McDonald. The background to NFSv4.1. *;login: The USENIX Magazine*, 37(1):28–35, February 2012.

[30] NFS-Ganesha, 2016. *http://nfs-ganesha.github.io/*.

[31] nghttp2. nghttp2: HTTP/2 C library, September 2016. *http://nghttp2.org*.

[32] A. Purohit, C. Wright, J. Spadavecchia, and E. Zadok. Cosy: Develop in user-land, run in kernel mode. In *Proceedings of the 2003 ACM Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 109–114, Lihue, Hawaii, May 2003. USENIX Association.

[33] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[34] S. Shepler and M. Eisler and D. Noveck. NFS Version 4 Minor Version 1 Protocol. RFC 5661, Network Working Group, January 2010.

[35] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3530, Network Working Group, April 2003.

[36] SPEC. SPEC SFS97_R1 V3.0. *www.spec.org/sfs97r1*, September 2001.

[37] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '97, 1997.

[38] Sun Microsystems. NFS: Network file system protocol specification. RFC 1094, Network Working Group, March 1989.

[39] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok. Terra incognita: On the practicality of user-space file systems. In *HotStorage '15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*, Santa Clara, CA, July 2015.

[40] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[41] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The case for VOS: The vector operating system. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 31–31, Berkeley, CA, USA, 2011. USENIX Association.

[42] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[43] M. Vilayannur, S. Lang, R. Ross, R. Klundt, and L. Ward. Extending the POSIX I/O interface: A parallel file system perspective. Technical Report ANL/MCS-TM-302, Argonne National Laboratory, October 2008.

[44] M. Mitchell Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.

[45] Werner Beroux. Rename-It!, 2016. *https://github.com/wernight/renameit*.

[46] Filebench Workload Model Language (WML), 2016. *https://github.com/filebench/filebench/wiki/Workload-Model-Language*.