Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server

Vijay Vasudevan
Carnegie Mellon University
vrv@cs.cmu.edu

Michael Kaminsky
Intel Labs
michael.e.kaminsky@intel.com

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

ABSTRACT

The performance of non-volatile memories (NVM) has grown by a factor of 100 during the last several years: Flash devices today are capable of over 1 million I/Os per second. Unfortunately, this incredible growth has put strain on software storage systems looking to extract their full potential.

To address this increasing software-I/O gap, we propose using *vector interfaces* in high-performance networked systems. Vector interfaces organize requests and computation in a distributed system into collections of similar but independent units of work, thereby providing opportunities to amortize and eliminate the redundant work common in many high-performance systems. By integrating vector interfaces into storage and RPC components, we demonstrate that a single key-value storage server can provide 1.6 million requests per second with a median latency below one millisecond, over fourteen times greater than the same software absent the use of vector interfaces. We show that pervasively applying vector interfaces is necessary to achieve this potential and describe how to compose these interfaces together to ensure that vectors of work are propagated throughout a distributed system.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design–*Distributed Systems*; D.4.8 [**Operating Systems**]: Performance–*Measurements*

General Terms: Design, Measurement, Performance

Keywords: Performance, Measurement, Non-volatile Memory, Key-value Storage

1. INTRODUCTION

Fast non-volatile memories (NVMs) are changing the landscape of data-intensive computing. Flash technology today can deliver 6GB/s sequential throughput and just over 1 million I/Os per second from a single device [11]. But the emergence of these fast NVMs has created a painful gap between the performance the devices can deliver and the performance that application developers can extract from them.

As we show in this paper, an implementation of a networked key-value storage server designed for the prior generation of Flash storage only sustains 112,000 key-value queries per second (QPS) on a server with a storage device capable of 1.8 million QPS. This large performance gap exists for a number of reasons, including Ethernet and storage interrupt overhead, system call overhead, poor cache behavior, and so on. Unfortunately, the bottlenecks that create this gap span application code, middleware, the kernel, and the storage device interface. Solutions that do not address all of these bottlenecks will not substantially narrow this performance gap.

To address such stack-spanning bottlenecks, this paper advocates for pervasive use of "vector interfaces" in networked systems. At a high level, vector interfaces express work in *classes* of independent units whose computation can be shared or amortized across the vector of work. Vector interfaces to storage submit multiple reads or writes (but not both) in one large batch; vector interfaces to RPC coalesce a class of RPCs (e.g., `get()`) into one large RPC to reduce per-message overhead and send fewer packets over the network. We use the term *vector* to differentiate from the more general *batching*: The same operation is applied to all entries in a vector, whereas batching does not necessarily place the same constraint. Vector interfaces are therefore similar in spirit to SIMD instructions, where a processor executes the same set of instructions over independent data.

We focus our attention on one compelling use case, distributed key-value storage backed by fast NVM (e.g., solid state drives), to illustrate how applications can best use vector interfaces. We demonstrate that using vector interfaces throughout the entire stack improves throughput by an order-of-magnitude for our networked key-value storage system, allowing a single server to handle 1.6 million queries per second at a median latency below one millisecond, providing roughly 90% of the operation throughput of the underlying NVM device. We show that failure to use vector interfaces at *both* the RPC and storage layer limits throughput to approximately 20% of device capability. We also describe the latency versus throughput tradeoffs introduced by vector interfaces and provide guidance to system designers as to when vector interfaces are useful and when they are not effective.

2. THE SHRINKING CPU-I/O GAP

Fast non-volatile memories provide several benefits over traditional magnetic storage and DRAM systems. In contrast to magnetic disks, they provide several orders of magnitude lower access latency and higher small random I/O performance. They are therefore well suited to key-value storage workloads, which typically exhibit small random access patterns across a relatively small dataset. In response, researchers have developed several key-value storage systems specifically for flash [5, 4, 7, 8].

Solid state drive (SSD) performance has increased dramatically over the last few years. SATA-based Flash SSDs have been capable

SOCC'12, October 14-17, 2012, San Jose, CA USA.

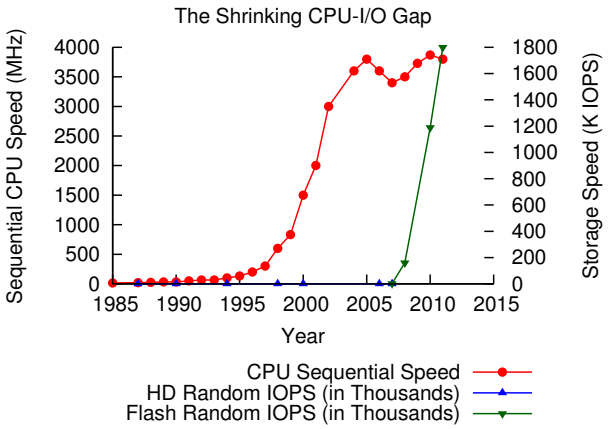

Figure 1: The Shrinking I/O Gap: CPU frequency vs storage I/O operations per second.

of up to 100,000 small (512-byte) random reads per second since 2009 [31], and enterprise-level PCIe-based SSDs claim 1.2 million random I/Os per second [11]. Prototype PCIe NVM platforms achieve similar throughput with latencies of 40 microseconds for phase-change memory [3] and 10 microseconds for NVM emulators [6]. Meanwhile, CPU core sequential speeds have plateaued in recent years, with aggregate system performance achieved through using multiple cores in parallel. Figure 1 depicts this shrinking I/O gap using historical data over the past 25 years.[1] While aggregate CPU performance might continue to improve due to increases in core count, I/O latency and efficiency would remain stagnant if bottlenecked by single-core performance.

Given this dramatic improvement in throughput and latency, what changes are needed to allow key-value storage systems to take advantage of these improvements?

We answer this question in this section by measuring the performance of an open-source distributed key-value storage system developed for flash storage called FAWN-KV [5, 1] running on a prototype NVM emulator described in more detail in the next section. The PCIe-based NVM prototype is capable of 1.8 million IOPS, reflecting a modest but not implausible advance over the 1.2 million IOPS available off-the-shelf today from companies such as FusionIO; thus, our prototype represents a reasonable performance target for future NVM software storage systems. FAWN-KV was designed for a hardware architecture combining flash devices from a prior generation of CompactFlash or SATA-based SSDs with low-power processors such as the Intel Atom. The software has therefore already been optimized to minimize memory consumption and CPU overhead to take advantage of SSDs capable of 80,000+ IOPS [31].

2.1 NVM platform and baseline

We evaluate three different system configurations to understand where the performance is limited (Figure 2). In the *networked system evaluation*, client machines send requests at high rate to a "backend" storage node. The backend node translates these requests into reads to the PCIe NVM emulator, waits for the results, and

[1] CPU numbers collected from `http://cpudb.stanford.edu`, showing the fastest clock rate for an Intel processor released every year. Hard drive seek time numbers come from an IBM GPFS Whitepaper [10]. Flash IOPS are derived from public specification sheets.

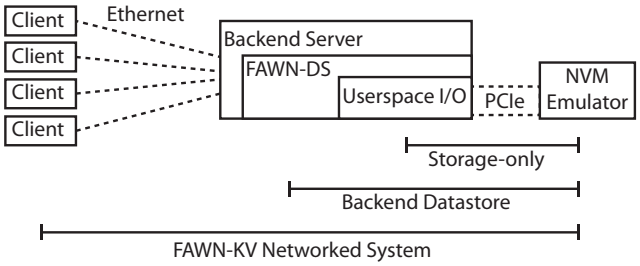

Figure 2: Benchmark scenarios for NVM platform.

sends replies back to the clients. The *backend datastore evaluation* measures only the backend storage node and local datastore software. Finally, the *storage-only evaluation* omits the datastore software and sends storage queries from a stripped-down benchmark application designed to elicit the highest possible performance from the NVM platform.

The backend storage node is a typical fast server machine with two 4-core Intel Xeon X5570 CPUs operating at 2.933 GHz with hyperthreading disabled. It uses an Intel X58 chipset, contains 12GB of DRAM, and is attached to the network using an Intel 82575EB 1Gbps on-board network interface.

The NVM prototype is a PCIe-based device that plugs into the backend; it offers both a traditional system call interface as well as a userspace software interface that provides `read()`- and `write()`-like calls. The availability of a userspace interface allows us to efficiently and easily implement vector storage interfaces that are currently unavailable in off-the-shelf PCIe SSDs. The userspace interfaces translate reads and writes into memory copies and commands issued directly to the device over PCIe. The device uses four independent DMA engines for read and write commands, requiring that at least 4 different threads simultaneously access the device for full performance.

The NVM device is backed by DRAM but looks like a PCIe SSD to the backend's OS, allowing us to focus on the higher levels of the software stack to understand how applications and networked system software must adapt to support the throughput capability of a future class of fast storage devices. We do not use the device to model a specific device technology with particular latencies, wear-out, or access peculiarities (such as erase blocks); we instead use FAWN-KV's storage layer, FAWN-DS, which has already been optimized to write sequentially using a log-structured layout. We believe that the performance-related aspects of our work will generalize to future NVMs such as phase-change memory (PCM), but device-specific optimizations beyond the scope of this work will likely be necessary to make the best use of an individual technology.

The NVM device provides 1.8 million I/O operations per second (IOPS) when accessed directly via a simple test program (left-most bar in Figure 3). Microbenchmarks using FAWN-DS, our log-structured, local key-value datastore application, achieve similar performance. We measure the throughput of looking up random key-value pairs with 512B values. Each lookup requires a hash computation, a memory index lookup, and a single read from the underlying storage device. These results show that a *local* key-value datastore can saturate the NVM device, despite requiring hashing and index lookups. A single key-value pair retrieval takes 10 microseconds, on par with the fastest NVM emulator platforms available today [6].

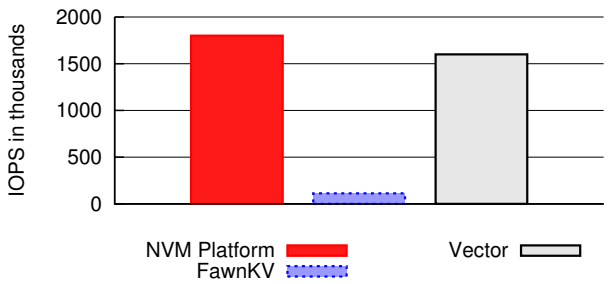

Figure 3: Networked key-value systems without vector interfaces may significantly underutilize future NVMs. Using vector interfaces can improve throughput to 90% of device capability.

2.2 FAWN-KV networked system benchmark

The FAWN-KV benchmark is an end-to-end benchmark. The server communicates with 25 open-loop, rate-limited client load generators (enough to ensure that the clients are not the bottleneck). The middle bar in Figure 3 shows that the networked system is an order of magnitude slower than the purely local datastore, achieving only 112,000 IOPS. Even a highly optimized networked key-value storage system designed for current SSDs cannot take advantage of the performance capabilities of next generation SSD systems.

Understanding this large performance gap requires breaking down the components of FAWN-KV. FAWN-KV builds three layers on top of FAWN-DS: 1) networking, 2) RPC, and 3) the associated queues and threads to make parallel use of flash using a staged execution model similar to SEDA [34]. These additional components are responsible for the significantly reduced throughput since FAWN-DS alone can saturate the device capability; we further tested FAWN-KV using a "null" storage backend that returned a dummy value immediately, which only modestly improved throughput.

FAWN-KV uses Apache Thrift [2] for cross-language serialization and RPC. Each key-value request from the client requires protocol serialization and packet transmission; the backend receives the request and incurs a network interrupt, kernel IP and TCP processing, and protocol deserialization before it reaches the application layer; these steps must be repeated for each response as well. These per-request computations are one source of additional overhead.

To amortize the cost of these per-request computations, the rest of this paper describes the implementation, evaluation and tradeoffs of using vector interfaces to storage and RPC systems. As we build up to in this paper, pervasively using vector interfaces improves networked key-value storage throughput from 112,000 IOPS to 1.6M IOPS, shown as the rightmost bar in Figure 3, a factor of 14 improvement in performance that achieves roughly 90% of the capability of the NVM platform.

3. VECTOR INTERFACES

Given that a state-of-the-art cluster key-value store cannot provide millions of key-value lookups per second (despite underlying storage hardware that can), there are three mostly-orthogonal paths to improving its throughput: Improving sequential code speed; embracing parallelism by doling out requests to the increasing number of cores available in modern CPUs; and identifying and eliminating redundant work that can be shared across requests.

In this work, we focus on only the last approach. Sequential core speeds show no signs of leaping forward as they once did, and sequential code optimization is ultimately limited by Amdahl's law [6]—and, from our own experience, optimizing code that spans the entire height of the kernel I/O stack is a painful task. Although improving parallelism is a ripe area of study today, we avoid this path as well for two reasons: First, we are interested in improving system *efficiency* as well as raw throughput; one of the most important metrics we examine is IOPS per core, which parallelism alone does not address. Second, and more importantly, the vector-based interfaces we propose can degrade into parallel execution, whereas a parallel approach may not necessarily yield an efficient vector execution.

Vector interfaces group together multiple requests with independent data, but the same operation (e.g., "read these 10 disk locations"), whose execution can be shared across the vector of work. Doing so allows a system to eliminate redundant work found across similar requests and amortize the per-request overheads found throughout the stack. By eliminating redundant work and amortizing costs of request execution, we can significantly improve throughput as measured by IOPS as well as efficiency as measured by IOPS/core.

Vector interfaces also provide an easy way to trade latency for improved throughput and efficiency by varying the width of the vector. Moving to storage devices that support higher throughput can be as simple as increasing the vector width at the cost of higher latency.

We focus on two types of explicit vector interfaces in this work: 1) **Vector RPC interfaces**, aggregating multiple individual, similar RPCs into one request, and 2) **Vector storage interfaces**, or vector-batched issuing of I/O to storage devices.

3.1 Vector RPC interfaces

Vector RPC interfaces batch individual RPCs of the same type into one large RPC request. For example, `memcached` provides programmers a multiget interface and a multiget network RPC. A single application client can use the multiget *interface* to issue several requests in parallel to the `memcached` cluster, improving performance and reducing overall latency. The multiget *RPC* packs multiple key-value get requests into one RPC; reducing the number of RPCs between an application client and a single `memcached` server means fewer network interrupts and system calls, and reduced RPC processing overhead.

3.2 Vector storage interfaces

Vector storage interfaces specify vectors of file descriptors, buffers, lengths, and offsets to traditional interfaces such as `read()` or `write()`. They differ from the current "scatter gather I/O" interfaces `readv()` and `writev()`, which read or write only sequentially from or to a single file descriptor into several different buffers; vector storage interfaces, in contrast, can read or write from random locations in multiple file descriptors.

Our proposed vector storage interfaces resemble the `readx()`/`writex()` POSIX extension interfaces, which were designed to improve the efficiency of distributed storage clusters in high-performance computing. These interfaces take multiple file descriptors, buffers, and offsets as arguments and execute the vector of requests typically on a distributed, parallel file system. The differences are twofold: First, gaining the efficiency we seek requires pushing the vector grouping as far down the storage stack

as possible—in our case, to the storage device itself—requiring both software and hardware support currently not available in off-the-shelf PCIe SSDs. Second, we emphasize and evaluate the synergistic benefits of comprehensive vectorization: as we show in our evaluation, combining network and storage vectorization provides a large boost in throughput without imposing extra latency for batching requests together (the price of batching, once paid, is paid for all subsequent vector interfaces).

Storage devices today read and write individual sectors at a time. A future device supporting multiread or multiwrite takes a vector of I/O requests as one command. In addition to saving memory (by avoiding duplicated request headers and structure allocation) and CPU time (to initialize those structures, put more individual items on lists, etc.), a major benefit of these vector interfaces is in drastically reducing the number of storage interrupts. A multi-I/O operation causes at most one interrupt per vector. Interrupt mitigation features found on modern network interface cards share these benefits, but using interrupt migitation features specifically for multi-I/O provides several advantages over the network case: First, there are no heuristics for how long to wait before interrupting; the device knows exactly how many requests are being worked on and interrupts exactly when they are complete. Second, it is less likely to unintentionally delay delivery of a message needed for application progress—because the application itself determined which requests to batch.

4. VECTOR INTERFACES TO STORAGE

We begin by describing the implementation and benefit of vector interfaces to storage, showing that they can help systems match the potential throughput of high-speed SSDs for both asynchronous and synchronous access.

4.1 Vector interface to device

A storage device supporting a vector interface must implement a single I/O storage command containing multiple, individual and independent requests to storage.

Implementation Details. Current I/O stacks do not contain commands to submit multiple-operand I/O operations to a device. Although hosts can send up to 31 outstanding I/Os to SATA devices using Native Command Queuing (NCQ), these devices process each I/O independently and generate a separate interrupt for each submitted I/O. The SATA AHCI specification details a compatible design called Command Completion Coalescing, but we could not find sufficient Linux and device support for this feature (Furthermore, these commands are not vectors—they can have different operations, which, on flash, can result in a mix of low-latency reads waiting on high-latency writes for completion).

Instead, we access the NVM device described above using a direct userspace interface. A software library provides `read()`- and `write()`-like interfaces similar to POSIX. A non-vector read or write command prepares an I/O structure containing a buffer, offset, and length as arguments, appends that I/O structure to a submission queue, and notifies the device that a request is ready to be processed using DMA commands accessing a shared data structure. The device polls the data structure, processes the request and uses DMA to transfer the result back, similarly signaling back to the host that the request is complete and available in a completion queue.

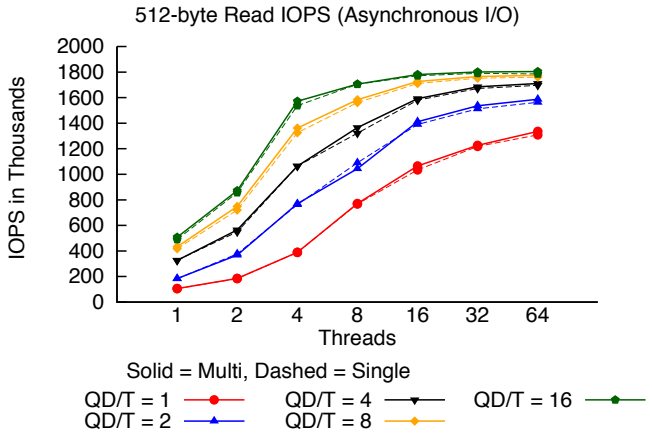

Figure 4: 512 Byte read throughput comparison between "single I/O" interface (dashed lines) and "multi-I/O" interface (solid lines).

Our proposed `read_vec()` and `write_vec()` interfaces take multiple buffers, offsets, and lengths as arguments and issue the requests to the device in one large batch. When delivered in a large batch, the NVM software running on the emulator processes each individual command in submitted order, transfers the results into the host's userspace memory using DMA, and generates a single interrupt to the host only after all of the commands in the vector are complete.

Benchmark. The benchmark in this section measures the raw capability of the NVM platform using asynchronous I/O, which allows a single thread to submit many outstanding I/Os to the device. To understand its capabilities, we vary queue depth, thread count, and vector width, whose definitions are as follows:

1. **Queue depth per thread (QD/T)**: The number of asynchronous, outstanding I/O requests sent by one thread. For single-I/O, an interrupt is generated for each individual request, and for each response, one new I/O can be issued.
2. **Thread count**: The number of independent user-level threads issuing I/Os to the device.
3. **Storage vector width**: The number of I/Os for which one interrupt notifies the completion of the entire vector. For our multi-I/O experiments in this section, the vector width is always set to half of the QD/T value to ensure the device is busy processing commands while new requests are generated. A storage vector width of one approximates `libaio`, where each I/O incurs an interrupt.

Figures 4 and 5 compare the performance of vector and single read for a variety of thread counts and queue depth/multiread settings. Error bars are omitted because the variance is under 5% across runs. The solid line shows the experiment where vector interfaces are used, and the dashed line shows when a single I/O is posted at a time (and one interrupt is returned for each).

The NVM platform contains 4 DMA engines. Peak throughput requires at least 4 independent threads—each uses one DMA channel regardless of the number of asynchronous I/Os it posts at once. Saturating the device IOPS further requires maintaining a high effective queue depth, either by having a high queue depth per thread value or by spawning many independent threads. Prior work has

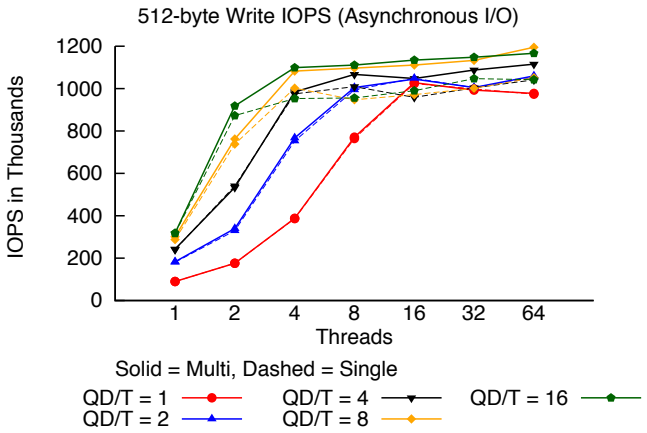

Figure 5: Throughput of 512 B single-I/O writes (dashed lines) vs. multi-I/O writes (solid lines).

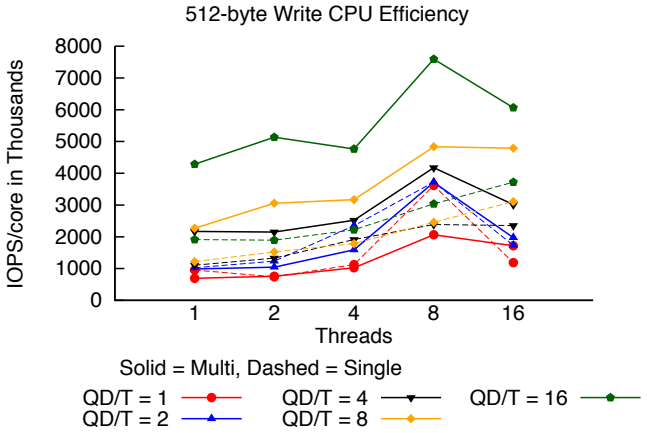

Figure 6: Measurement of I/O efficiency in IOPS per core. A multi-I/O interface can reduce overall CPU load by a factor of three.

demonstrated that maintaining an I/O queue depth of 10 is required to saturate the capabilities of current SSDs [23]; our results suggest that this number will continue to increase for next generation SSDs.

Multiread and single I/O read throughput are similar because the read throughput is limited by the hardware DMA capability, not the CPU. In contrast, multiwrite improves performance over single writes, particularly at high thread counts and high queue depth. For example, for a queue depth of 16, single I/O write performance remains between 900K and 1M IOPS, whereas multiwrite performance reaches approximately 1.2M IOPS, an improvement of 20%.

Raw throughput, however, is a potentially misleading indicator of the benefits that vector interfaces offer over `libaio`-like single I/O asynchronous interfaces. CPU efficiency, measured by IOPS/core, paints a very different picture. We calculate efficiency by dividing the IOPS performance by CPU utilization reported in `/proc/stat`.

Figure 6 shows that at high vector widths, multiread and multiwrite are between 2–3x more efficient than using single I/O. A large vector width reduces substantially the number of interrupts per I/O, allowing the CPU to devote less time to interrupt handling. As we demonstrate in the networked system evaluation in Section 5, this reduced overhead makes many more cycles available to application-level code and becomes critical to achieving high performance for networked key-value storage applications.

4.2 Vector interfaces to key-value storage

Next, we describe using vector interfaces to access a local key-value datastore on the backend node.

The log-structured FAWN-DS local key-value datastore exports a synchronous `put(string key, string value)`/`get(string key)` interface. We added `get(vector<string> key)` which returns a vector of key-value results. For each lookup, FAWN-DS hashes the key, looks up an entry in a hashtable, and `read()`s from storage using the offset from the hashtable entry.

The vector version requires taking in a vector of keys and calculating a vector of potential offsets from the hashtable in order to issue a `multiread()` for all keys. One feature of FAWN-DS complicates vectorization: To conserve memory, it stores only a portion of the key in the hashtable, so there is a small chance of retrieving the wrong item when the key fragments match but the full keys do not.

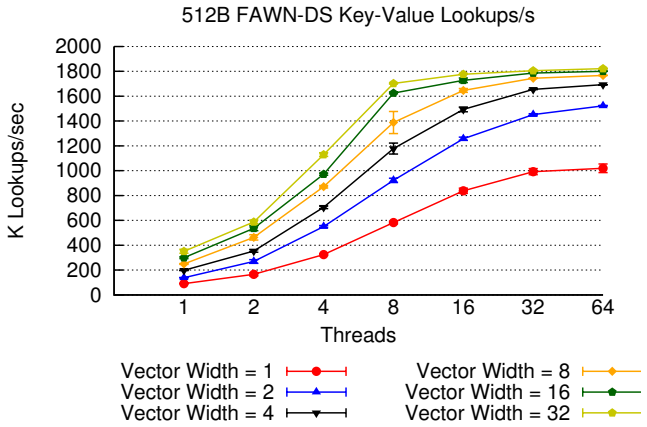

Figure 7: 512B synchronous read throughput through FAWN-DS using synchronous I/O interface.

As a result, the keys being looked up may each require a different number of `read()` calls (though most complete with only one).

The vector version of `get()` must inspect the multiread result to identify whether all entries have been retrieved; for entries that failed, vector `get()` tries again, starting from the last hash index searched. Adding vector support required changes to the code structure to manage this state, but only changed or added fewer than 100 LoC, all of which were isolated in one code module.

Benchmarking multiget in FAWN-DS: One important difference between the FAWN-DS benchmark and the earlier device microbenchmark is that the FAWN-DS API is *synchronous*, not asynchronous. This means that when FAWN-DS reads from the storage device, the reading thread blocks until the I/O (or I/Os, in the case of multiread) complete. Many applications program to the synchronous I/O model, though some systems support native asynchronous I/O though `libaio`-like interfaces (asynchronous functions with callbacks).

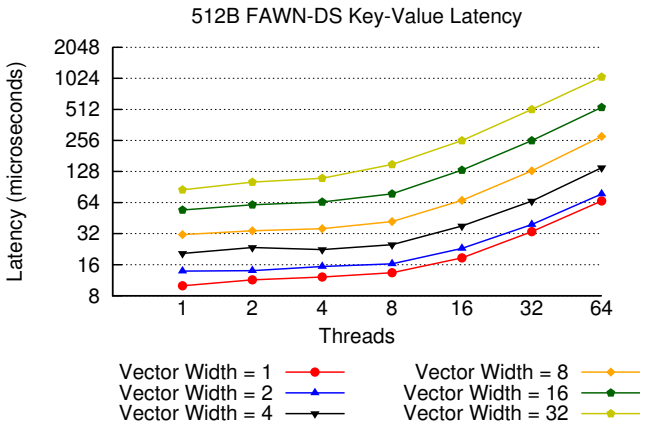

Figure 8: High vector widths don't have proportionally higher latency: Growing vector width by a factor of 32 increases latency by only a factor of 8.

Figure 7 shows the performance of 512B key-value lookups[2] with an increasing number of threads. Single I/O submission reaches a maximum of about one million key-value lookups per second even when using 32 threads, whereas multireads of size 16 or 32 can provide approximately 1.8 million key-value lookups per second at 32 threads. Maintaining a high effective queue depth is critical to saturating these devices: asynchronous I/O through `libaio` (single I/O) or vector interfaces will be necessary for local applications to obtain full performance. Networked applications will benefit further from choosing vector interfaces over asynchronous single-I/O interfaces (Section 5).

Vector interfaces and latency: Vector batching has a complex effect on latency. Batching waits for multiple requests before beginning execution. It similarly does not complete until the last request in the batch completes. Both of these effects add latency. On the other hand, it reduces the amount of work to be done by eliminating redundant work, such as avoiding unnecessary interrupts and repeatedly acquiring potentially contended locks. It also achieves higher throughput, and so on a busy server reduces the time that requests spend waiting for others to complete. These effects reduce latency, particularly at high load.

Figure 8 shows the how batching modestly increases latency at low load. The bottom left point in the graph shows the minimum latency for retrieving a key-value pair using the NVM platform, which is approximately 10μs. Doubling the vector width for a given number of threads does not double the latency of the entire operation until the device nears throughput saturation. This means that modest amounts of batching can improve throughput more than they increase latency. We also observe this behavior in the next section when evaluating a combined RPC and storage environment, finding that increases in vector width significantly increase throughput with almost no additional increase in latency: the optimal points in the latency vs. throughput curve are obtained using varying vector widths.

[2]32B I/O was no faster than 512B I/O on our platform, and larger I/Os quickly saturate device bandwidth.

5. VECTOR INTERFACES TO NETWORKED KEY-VALUE STORAGE

Exporting vector interfaces to non-volatile memory devices and to local key-value storage systems provides several throughput, latency, and efficiency benefits. Next, we examine vector interfaces to RPCs and their interaction with the vector interfaces provided by the storage device and key-value storage APIs.

5.1 Experimental setup

In contrast to the previous section whose experiments involved only local I/O, the experiments in this section all involve queries made over the network using the FAWN-KV distributed key-value storage system. A cluster of Intel Atom-based machines generates the query workload, using as many nodes as needed to ensure that the backend node, not the load generators, are the bottleneck. The experiments described here use between 20 and 30 query generation nodes. Queries are sent in an *open loop*: The query generators issue key-value queries at a specific rate regardless of the rate of replies from the backend. The benchmark client uses three threads: one fills a token bucket at the specified rate, another removes tokens to issue *asynchronous* requests to the backend device; and the final thread receives and discards the responses.

For each experimental configuration, we use as many threads on the backend node as necessary to maximize throughput without unnecessarily increasing latency. In practice, running the FAWN-KV software with between 8–16 independent threads to send I/Os to the NVM platform was sufficient.

Asynchrony: The benchmark utility issues asynchronous key-value requests because synchronous requests would be bottlenecked by the end-to-end network latency of our environment. Although some applications may desire a synchronous interface, an asynchronous interface can benchmark the limits of the backend system using a relatively small number of clients and threads.

Request Size: This evaluation inserts and retrieves 4-byte values (prior sections used 512 byte values), because transferring larger values over the network quickly saturates the server's 1Gbps network. Internally, the server still reads 512 bytes per request from storage. This change will slightly over-state the relative benefits of vector RPC (below): *Requests* are unchanged, internal storage I/Os are unchanged, but the amount of time spent copying data into responses—something not helped greatly by network vectorization—will decrease. We believe, however, that these effects are modest.

Multiget and Multiread: Our evaluation in this section measures the impact of multiread, multiget, and their combination when applied to FAWN-KV running on top of the NVM platform with vector interface support. As shown in the previous section, reads and writes perform roughly the same on this device, but we note that specific NVM technologies would likely have different read/write characteristics. For the purposes of identifying and evaluating operation-independent overheads, we focus solely on the read/data retrieval path. We use the term *multiread* to describe vector I/Os to the storage device, and we use the term *multiget* to describe vector RPCs over the network. In figures, we refer to a multiget vector width of N as "GN" and a multiread vector width of M as "RM". We use multiread as a proxy for multiwrite, and multiget as a proxy for other types of RPCs such as multiput.

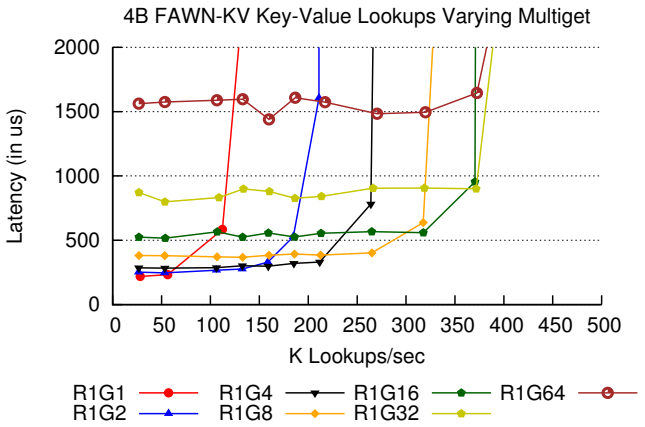

Figure 9: Networked 4-byte throughput vs. latency as a function of the multiread width.

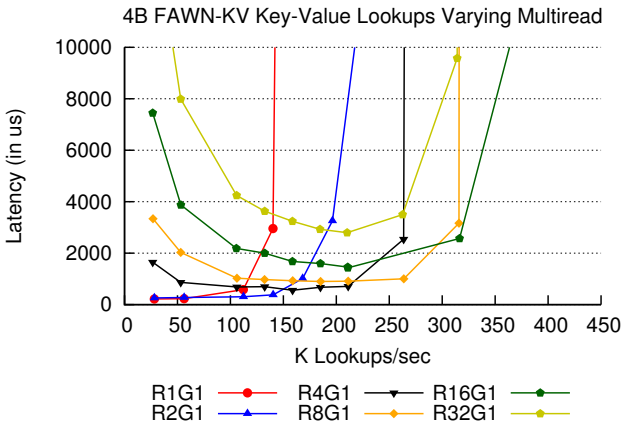

Figure 10: Networked 4-byte throughput vs. latency as a function of the multiread vector width. Multiread creates batches of network activity that reduce interrupt and packet rate.

5.2 Results

We begin by measuring baseline throughput and latency without vectors. The R1G1 line in Figure 9 shows the throughput versus latency as the query load increases. Because the measurement is open-loop, latency increases as the system approaches full capacity.

At low load, the median latency is 220μs. As shown previously, the NVM device access time at low load is only 10μs, so most of this latency is due to network latency (approximately 100μs baseline), kernel, RPC, and FAWN-KV application processing. Without vector interfaces, the system is capable of about 112K key-value lookup/sec at a median latency of 500μs.

5.2.1 *Multiget (vector RPC) alone*

Network key-value throughput using standard storage and RPC interfaces ("R1G1") is over an order of magnitude lower than device capability and eight times lower than local I/O performance. At this point, the overhead from I/O, local datastore lookups, RPC processing, and network I/O has greatly exceeded the CPU available on the backend node. We begin by examining how much the Vector RPC interface can help reduce this overhead.

The remaining lines in Figure 9 show load versus latency for larger multiget widths. Multiget improves the throughput (at reasonable latency) from 112K key-value lookups/sec to 370K for a multiget width of 16. Increasing the vector width further increases latency without improving throughput.

Vector RPCs increase peak throughput more than they increase latency. For a multiget width of 4 (G4), the latency at 50,000 key-value lookups/sec is 300μs compared to 220μs for single get. But the throughput achieved for G4 is roughly 210,000 lookups/second, about twice the throughput of G1 at a latency of 330μs. The additional latency comes from 1) the time to assemble a batch of key-value requests on the client, 2) the time to enqueue and dequeue all requests and responses in a batch, and 3) the response time for all I/Os in a batch to return from the NVM platform.

Despite the 3× throughput improvement compared to single get throughput, the system is still far below the 1.8M key-value lookups that the device can provide. Although multiget improves performance, per-RPC processing is not the only bottleneck in the system.

5.2.2 *Multiread (vector storage) alone*

Next, we hold the multiget width at 1 and vary the multiread width to understand to what degree multiread can improve performance without multiget. Varying multiread alone is useful for environments where only local changes to the storage server are possible.

Figure 10 plots throughput versus latency for multiread widths of 1, 2, 4, ..., 32. Grouping 8 reads together (from different RPCs) improves throughput from 112K to 320K key-value lookups/sec while maintaining relatively low latency. Multiread widths beyond 8 do not increase throughput without a corresponding large increase in median latency.

The low-load latency behavior is different for multiread than for multiget. At very low load, latency scales linearly with the multiread width, drops rapidly as offered load increases, and then increases as the system approaches peak capacity. This behavior is caused by the queue structure that collects similar RPC requests together: Using a multiread width N, the first key-value RPC that arrives into an idle system must wait for the arrival of $N-1$ other key-value requests from the network before the batch of I/Os are issued to the device. At low load, these $N-1$ other requests enter the system slowly and as load increases, request inter-arrival time shortens and reduces the time the first request in the queue must wait.

Multiread's implicit benefits: To our surprise, multiread both improves the efficiency of client network performance—even without multiget—and increases cache efficiency on the backend. We term these two improvements *implicit benefits*, in contrast with the explicit design goals of vector interfaces: multiread explicitly reduces the total number of commands sent to the storage device, and correspondingly the number of interrupts received; multi*get* explicitly reduces the number of `send()` and `recv()` system calls, serialization and deserialization overhead for individual RPC messages, and number of packets (when Nagle's algorithm is disabled).

Improved client efficiency: A multiread storage I/O contains multiple key-value responses. Because we use a one-thread-per-connection model, these responses are destined to the same client, and so the RPC handling thread sends these responses closely in time. The client receives a burst of several responses in one larger packet due to packet coalescing, reducing significantly the number of

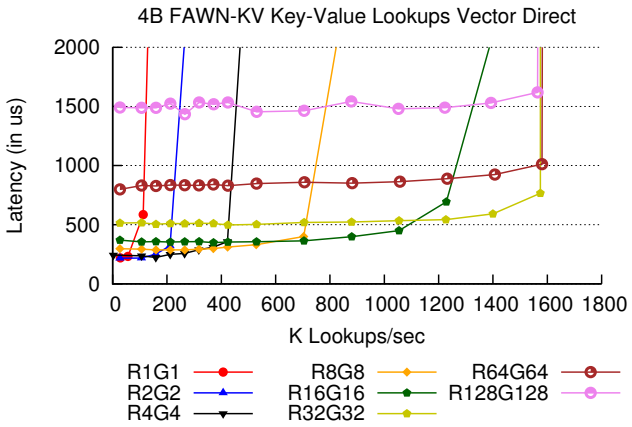

Figure 11: Throughput vs. latency for matched vector widths. Vector interfaces enable a single server to provide 1.6M key-value lookups per second at a latency below 1ms.

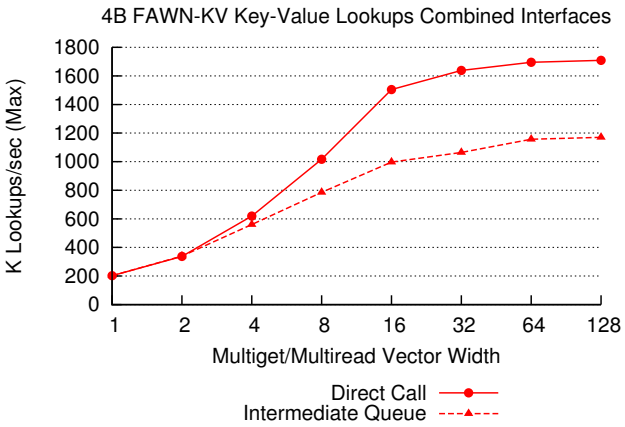

Figure 12: Throughput increases as storage and multiget widths increase. Using an intermediate queue between the RPC and storage layer significantly degrades performance at large vector widths.

ACKs sent back to the server: sequence numbers refer to bytes, not packets, so one ACK is generated for the entire burst. This improves the efficiency and performance of the server because it incurs fewer network interrupts, but this behavior also can improve efficiency on clients: In Figure 10, the backend server is the bottleneck, but when we reduced the number of clients to the point where clients were the bottleneck, enabling only multiread on the server improved the unmodified clients' performance by 10%.

Improved cache efficiency: A by-product of organizing requests into queues for multiread is that it improves the backend's cache efficiency, and hence sequential performance. On each get request, the servicing thread performs a network `read()`, processes the get RPC, and inserts the request into a queue, repeating *N* times in a row while keeping requisite data structures and code in cache. When the thread issues the multiread request, the hardware interrupt generated will cause the processor's cache to flush, but many of the structures needed for performing I/O are no longer needed. This better cache locality increases the instructions per cycle (measured using CPU performance counters) and contributes partially to the improvement that multiread provides over single reads in the networked server evaluation. These benefits are similar to those found in Cohort Scheduling [17], where executing a group of similar operations improves data and code locality and hence sequential performance.

5.3 Combining vector interfaces

Despite their benefits, multiread and multiget in isolation cannot achieve full system performance of 1.8M key-value lookups per second. Multiread improves networked key-value retrieval by roughly 2x by reducing the storage interrupt load, freeing the CPU for the costly network and RPC processing. Multiget provides a similar 2x improvement by reducing the number of packets sent across the network and the RPC processing overhead. In this section, we show that the two combine synergistically to increase total throughput by 14×.

When the multiget width is less than the multiread width, a queue structure is required to collect key-value requests from multiple RPCs. We call this implementation the "intermediate queue" pattern. If the multiget width and multiread width are the same, no intermediate queues are necessary, and the backend can execute a multiread directly—we call this implementation the "direct" pattern.

5.3.1 *Matching multiread and multiget widths*

The combinations of parameters R and G are numerous; thus, we begin by investigating the performance when the multiget width equals the multiread width. In this case, we use the "direct" pattern that omits intermediate queues.

Figure 11 shows the latency versus throughput for equal vector widths using the direct pattern. Combining multiread and multiget provides up to 1.6M key-value lookups per second from the backend, nearly saturating the NVM device, and does so without increasing median latency beyond 1ms. Achieving this only requires vector widths of 32. Because no intermediate queues are required, the latency behavior at low load is identical to that of just using multigets.

The seemingly small overheads of enqueuing and dequeuing requests in the intermediate queue significantly reduce performance at high load. Figure 12 shows the peak query throughput with increasing vector width. The dashed line depicts the performance using an intermediate queue, while the direct call is shown as the solid line. With a queue, the overhead of inserting and removing each get RPC's data limits the performance to 1.2M key-value IOPS, whereas the direct call implementation performs 1.7M IOPS at peak (unbounded latency). This highlights one of the important benefits that vector interfaces offers over simple batching: Vector interfaces allow the system to propagate vectors of work efficiently, whereas even intelligent batching would require intermediate queues to stage requests.

5.3.2 *Unequal vector widths*

Is it ever advisable to set the size of multiget and multiread widths differently? We consider scenarios where the multiget width is fixed at different values as the independent variable and show how varying the multiread width affects throughput and latency. When the multiget width is less than the multiread width, we use an intermedi-

ate queue to collect requests together. When the multiget width is greater than the multiread width, we use the direct implementation but issue reads to the device in sizes of the specified multiread width.

For low multiget widths, it is advisable to keep the multiread parameter low. Waiting for the multiread queue to fill at low load creates long queuing delays. As load increases, however, it is beneficial to increase the multiread width higher than the multiget width because it allows the system to achieve higher rates than otherwise possible. Thus, the best strategy is to scale the multiread width higher as load increases to get the best tradeoff of throughput and latency.

For high multiget widths, regardless of load, it is always best to match RPC and multiread widths. When multiread is low, issuing each request serially and taking an interrupt for each get request in the multiget batch increases latency significantly. At low load, using a low multiread width approximately doubles median latency, while at high load it is necessary to have a high multiread width to achieve higher throughput. The queue structure required when multiread is greater than multiget, though, reduces the performance benefits having a high multiread width can provide. Existing algorithms that vary vector RPC width with load might be adapted to also vary storage vector width to provide high throughput at high load and low latency at low load [20].

In summary, vector interfaces used in isolation improve throughput by amortizing the cost of RPCs and reducing interrupts and packet processing, but still provide only a small fraction of the underlying storage platform's throughput capabilities. By carefully combining both types of vector interfaces, however, we have shown both that such a system is capable of 90% of optimal throughput and also how unequal vector widths trade between throughput and latency.

6. DISCUSSION AND FUTURE WORK

Using vector RPC and storage interfaces can significantly improve performance for a networked key-value storage system. In this section, we ask: when are vector interfaces generally useful and when should they be avoided?

The principal scenarios where vector interfaces are useful share three properties: The services expose a narrow set of interfaces, exhibiting a high degree of "operator redundancy"; the work being batched together shares common work; and the requests in an initial vector follow a similar path to ensure that vectors are propagated together throughout the distributed system.

Narrow Interfaces: Key-value and other storage systems often export a small number of external interfaces to clients. Under high load, any such system will be frequently invoking the same operators, providing the opportunity to eliminate redundant work found across these similar operations. If the interface is not narrow but the operator mix is skewed towards a common set, then operator redundancy will be high enough that vector interfaces can provide a benefit.

Similarity: The computational similarity found among get requests in a multiget allows us to eliminate redundant work common across otherwise independent requests. In contrast, consider a vector "SELECT" interface for SQL used to batch independent select queries. If the queries in a vector do not operate on the same table or process the resulting data similarly, then there may be few opportunities for optimization because redundancy may not exist. The cost of serializing independent requests that are computationally-unique would outweigh the benefits the small amount of work sharing provides [15].

Vector Propagation: For maximum advantage, vectors of similar work should propagate together through the system. In FAWN-KV, the key-value lookups in a multiget from a client to the backend remain together throughout the lifetime of the operation. A mixed put/get workload, in contrast, would diverge once the request arrives at the backend; the backend would need to inspect the elements in the vector to separate the puts and gets into two separate vector operations. The system may then need to re-combine these two vectors before sending a response, adding further coordination and serialization overhead. Other systems have successfully used the "stages with queues" model to implement re-convergence for graphics pipelines [30]; PacketShader, for example, demonstrated that packet processing does not significantly diverge in execution to erase the benefits that GPU-accelerated vector processing can provide [14].

A related issue is whether to return partial results, and how to expose that interface to the clients. Partial results can help reduce latency for requests where parts of the vector take longer than other parts. However, our results suggest that there remain substantial efficiency benefits from avoiding queueing and re-queueing (the "direct" read pattern in the previous section), so we approach this issue cautiously.

6.1 Using Vector Interfaces at Large Scale

Vector RPCs can be easily used when communicating with a single backend storage server. When the storage system consists of tens to hundreds of machines, each storing a portion of the entire dataset, a client's key requests might not map to the same backend storage server, requiring that the client issue several RPCs each consisting of requests for fewer keys. The ability to issue large multiget RPCs depends on three factors: key naming, access pattern, and the replication strategy.

Key naming: Backend nodes are typically responsible for a subset of the key-value pairs in a distributed storage system; a single storage node serves keys for multiple continuous partitions of the key space, and an index (often a distributed B-tree or a simple map) maps keys to nodes. Most systems structure accesses to key-value storage based on how keys are named. If the key is a hash of an application-specific name, then a multiget from a single client will contain keys that are randomly distributed throughout the key space, reducing the likelihood that a multiget request can be served by a single backend. If the keys are not hashed, then the application access pattern and key naming policy determines how a set of key requests in a multiget map to backends.

Access pattern: Some applications, like webmail services, will exhibit a high degree of request locality: users access and search only over their own mail. If keys corresponding to a particular user are prefixed with a unique user ID or some other identifier that specifies locality in the key space, then requests for multiple keys might be served by a small number of backend nodes, allowing the multiget RPC to maintain the original vector width. For other applications, the access pattern might not exhibit locality: Social network graphs can be difficult to partition well [13], so requests from a single client may need to access a large fraction nodes in the backend storage system.

Replication: Data in large-scale key-value storage systems are often replicated for fault-tolerance, as well as higher performance and load balancing. Because replication provides multiple choices

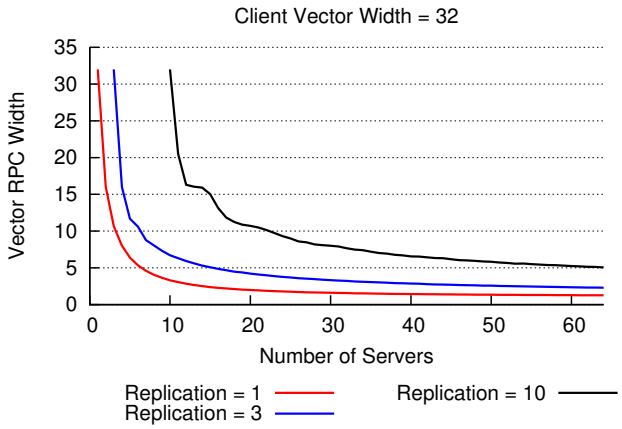

Figure 13: Simulation results showing the expected vector width of multiget RPCs, with the client making requests for 32 random keys, as a function of cluster size and replication factor. Data is assumed to be distributed evenly across all nodes, and replica nodes are chosen randomly (without replacement).

from which to obtain the same data, it is possible to use replication to reduce the total number of nodes one needs to contact when using vector RPCs: a multiget for two keys that map to different physical nodes can be satisfied by a single node if it contains a replica for both keys.

However, the type of replication used determines whether replication can maintain the benefit of using vector RPCs for higher performance. For example, if a system uses a hot-spare replication scheme (a replica node contains the exact same data as its master), then two keys that map to different masters will not map to the same physical node. On the other hand, a mapping system that randomly distributes data replicas slightly increases the probability that two keys can be served by the same node and allows a client to issue a single multiget RPC to retrieve both key-value pairs.

Application-specific replication schemes can do far better in ensuring that multiple client requests can hit only one server. For example, SPAR is a middleware layer that partitions and replicates data for online social networks to ensure that a user's data (including one-hop neighbors) exists on a single server, while simultaneously minimizing the overhead of replication [25].

Simulation of worst-case pattern: To understand how a random access workload interacts with vector RPCs and replication, we use a Monte Carlo simulation to find the expected average width of vector RPCs as a key-value storage cluster scales. We assume that keys map uniformly at random to nodes (including replicas of keys). We then fix the *desired* vector width at 32, vary the number of servers N that data is distributed evenly across, and calculate the minimum number of nodes a client must contact to retrieve 32 random keys (out of a large set of keys).[3] Each experiment is run 1000 times and we record the average minimum node count. We then divide the desired vector width (32) by this number to calculate the average expected vector RPC width.

Figure 13 shows how increasing the number of servers affects the average vector RPC width. For a replication factor of 1, the vector width starts at 32 for 1 server, reduces to 16 for 2 servers, etc. This line follows the formula $f(x) = \frac{32}{x-(x\times(1-\frac{1}{x})^{32})}$, which can be derived from a balls and bins analysis assuming 32 balls and x bins.

Increasing replication allows a client to maintain a higher vector width as the number of backend servers increases, because the additional choice provided by replication allows the client to pick a smaller set of nodes to cover all keys. For example, with a replication factor of 10 and 128 servers, a client can expect to contact minimum of ∼9 servers to retrieve the 32 random keys in the vector request, resulting in an average vector width of ∼4 key requests sent to each server.

Although random replication can improve the ability to maintain higher vector RPC widths for random access patterns, these simulation results highlight the need to cluster keys intelligently (e.g., application-specific locality hints or clustered replication techniques [25]) to ensure that vectors can be propagated with maximum efficiency.

6.2 How to expose vector interfaces

Vector storage interfaces can be implemented without requiring global modifications, whereas vector RPC interfaces require coordinated changes. Therefore, where and how should these vector interfaces be exposed to other components in a distributed system?

One option, as we have chosen, is that vector RPC interfaces are exposed directly to clients interacting with key-value storage systems. For example, a social networking application may issue a multiget corresponding to key-value requests for each friend, instead of requesting data serially one friend after another. Multiget in systems like `memcached` and Redis suggest that applications today already have opportunities to use these vector interfaces.

Alternatively, a client library can implicitly batch synchronous RPC requests originating from different threads into a single queue, issuing the vector request once a timeout or a threshold has been reached. Unfortunately, this creates the same opaque latency versus throughput tradeoffs found in TCP's Nagle option. A program using synchronous vectors cannot advance until the entire vector is complete, giving full control to the application developer to decide how to trade latency for throughput. Some cluster services separate clients from backend infrastructure using load balancers or caching devices. Load balancers can coalesce requests arriving from different clients destined to the same backend, but should use adaptive algorithms to control coalescing based on load [20].

6.3 Vector Network Interfaces

We have discussed vectorizing storage and RPC interfaces, but we did not need to vectorize the network socket layer interfaces because our TCP socket access patterns already work well with existing Ethernet interrupt coalescing. Our structuring of threads and queues ensures that we write in bursts to a single stream at a time. As we showed in Section 5.2.2, when several packets arrive over the network close in time, Ethernet interrupt coalescing will deliver multiple packets to the kernel with a single interrupt (which, in turn, often triggers only one outgoing ACK packet from the kernel for the combined sequence range). The application `recv()` will process this larger stream of bytes in one system call rather than one for each packet. On the sending side, Nagle's algorithm can coalesce outbound packets, though the application must still incur a mode switch for each system call.[4]

[3] This requires calculating a minimum set cover where the universe is the 32 keys requested and the sets are the N mappings of server number to keys.

[4] The RPC package we use (Apache Thrift) explicitly disables Nagle's algorithm by default to reduce the RPC latency added by batching outgoing packets in the kernel.

However, vector networking interfaces (such as `multi_send()`, `multi_recv()`, or `multi_accept()`) can be useful in other environments when code repeatedly invokes the same type of network function call with different arguments close in time: for example, an event loop handler sending data might frequently call `send()` for multiple sockets, and amortizing the cost of this system call across multiple connections may be useful. Many event-based systems fall under this pattern, including systems build on `libevent` [24].

Vector network interfaces therefore can be useful depending on the structure of network event processing in a key-value storage system. For a highly-concurrent server for which event-based systems use significantly lower memory, we believe that implementing vector interfaces to the network might be necessary.

6.4 Vector interfaces for vector hardware

Programming to vector interfaces creates the potential for compiler-assisted code specialization for vector hardware. This creates a unique opportunity to better use vector hardware available on emerging server platforms.

For example, preparing the multiread I/O command to the underlying storage device requires creating and populating a structure describing the unique offset and size of each individual I/O. The multiget/multiread code path in the backend server contains 10 for loops that iterate over vectors whose width matches the multiget factor. A significant component of the increased latency at high vector widths comes from sequentially iterating through these vectors: while we have eliminated redundant work, we have not yet vectorized the unique work. SSE hardware today is capable of operating on 256-bit registers, and GPU hardware is capable of much wider widths. Exporting the computations within these simple loops to specialized vector hardware instead of general-purpose sequential cores could dramatically reduce latency at larger batch sizes. This is not merely wishful thinking: developers today have tools to harness vector hardware, such as CUDA and Intel's SPMD compiler (`http://ispc.github.com/`).

7. RELATED WORK

Vector interfaces: Prior systems have demonstrated the benefits of using an organization similar to vector interfaces. "Event batches" in SEDA [34] amortize the cost of invoking an event handler, improving code and data cache locality. Vector interfaces also share many common benefits with Cohort Scheduling [17] such as lower response time under certain conditions and improved CPU efficiency. However, Cohort Scheduling benefits only from the implicit batching that scheduling similar work in time provides, whereas vector interfaces can completely avoid re-executing the same work for every request.

Batched execution is a well-known systems optimization that has been applied in a variety of contexts, including recent software router designs [9, 14, 19] and batched system call execution [29, 27, 26]. Each system differs in the degree to which the work in a batch is similar. The multi-call abstraction simply batches together system calls regardless of type [27], whereas FlexSC argues for (but does not evaluate) specializing cores to handle a batch of specific system calls for better cache locality. Vector interfaces target the far end of the spectrum where the work in a vector is nearly identical, providing opportunities to amortize and eliminate the redundant computation that would be performed if each request in the vector were handled independently [32].

In High Performance Computing, interfaces similar to our vector interfaces have been developed. Examples include Multicollective I/O [21], POSIX listio and `readx()`/`writex()` extensions. These extensions provide batches of I/O to an intermediate I/O director, which can near-optimally schedule requests to a distributed storage system by reordering or coalescing requests. The existence of these interfaces suggests that application designers are willing and able to use explicit vector interfaces to achieve higher performance.

Key-value stores: In recent years, several key-value storage systems optimized for flash storage have emerged to take advantage of flash storage and non-volatile memory improvements. BufferHash [4], SkimpyStash [8], FlashStore [7], and SILT [18] are examples of recent key-value systems optimized for low-memory footprint deduplication or Content Addressable Memory systems. These systems evaluate performance on a prior generation of SSDs capable of a maximum of 100,000 IOPS, whereas our work looks ahead to future SSD generations capable of much higher performance. Their evaluations typically use synthetic benchmarks or traces run on a single machine. In contrast, our work demonstrates that achieving high performance for a *networked* key-value storage system is considerably more difficult, and that achieving the performance of local microbenchmarks may require redesigning parts of local key-value systems.

Non-volatile memory uses: Several studies have explored both the construction and use of high-throughput, low-latency storage devices and clusters [6, 28, 3, 33, 22]. Most closely related is Moneta [6], which both identified and eliminated many of the existing overheads in the software I/O stack, including those from I/O schedulers, shared locks in the interrupt handler, and the context switch overhead of interrupts themselves. Our work is orthogonal in several ways, as vector interfaces to storage can be used on top of their software optimizations to yield similar benefits. In fact, our userspace interface to the NVM device begins where they left off, allowing us to explore opportunities for further improvement by using vector interfaces. Finally, we demonstrate that having the capability to saturate a NVM device using local I/O does not imply that achieving that performance over the network is straightforward.

Programming model: Our implementation of vector interfaces requires programmers to explicitly use vector abstractions (queues, barriers). In some cases, converting an operation to using a multi-interface can be difficult. Libraries like Tame [16] provide novice programmers with basic non-vector interfaces and event-based abstractions, but can execute operations using vectorized versions of those interfaces when possible.

Vector interfaces bear similarity to the "SIMT" programming model used in the graphics community, where computations are highly-parallelizable, independent, but similar in operation. GPU fixed function and programmable shader hardware matches well to these workloads where each functional core performs work in lockstep with potentially hundreds of other threads in a warp [12, 30]. The popularity of CUDA programming suggests that exposing non-vector interfaces to programmers and using vector-style execution for performance-critical sections can provide the best of both worlds, provided that vector interfaces are exposed where needed.

8. CONCLUSION

As non-volatile memories continue to improve in speed, the software interface must also evolve to take full advantage of these speed increases. We demonstrated that using vector interfaces pervasively

throughout a distributed key-value storage system can improve performance by over an order of magnitude when accessing a storage device capable of delivering millions of I/Os per second. We also showed that using vectors at both the RPC and storage layers is required to achieve this level of performance.

Acknowledgments

We would like to acknowledge our collaborators at Intel for providing us with the hardware that enabled this work. We would also like to thank Luiz Barroso, Garth Gibson, and Greg Ganger for their helpful feedback.

References

[1] FAWN-KV: A distributed key-value store for FAWN. http://github.com/vrv/FAWN-KV.

[2] Apache Thrift. https://thrift.apache.org/, 2011.

[3] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A prototype phase change memory storage array. In *Proc. HotStorage*, Portland, OR, June 2011.

[4] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, Apr. 2010.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[6] A. M. Caulfield, A. De, J. Coburn, T. Mollov, R. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *IEEE Micro*, Dec. 2010.

[7] B. Debnath, S. Sengupta, and J. Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, Sept. 2010.

[8] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash. In *Proc. ACM SIGMOD*, Athens, Greece, June 2011.

[9] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[10] R. Freitas, J. Slember, W. Sawdon, and L. Chiu. GPFS scans 10 billion files in 43 minutes. IBM Whitepaper, http://www.almaden.ibm.com/storagesystems/resources/GPFS-Violin-white-paper.pdf, 2011.

[11] fusion-io. Fusion-IO. http://www.fusionio.com.

[12] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, Nov. 2010.

[13] J. Hamilton. Scaling at MySpace. http://perspectives.mvdirona.com/2010/02/15/ScalingAtMySpace.aspx, 2010.

[14] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, New Delhi, India, Aug. 2010.

[15] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To share or not to share? In *Proc. VLDB*, Vienna, Austria, Sept. 2007.

[16] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proc. USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[17] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *Proc. USENIX Annual Technical Conference*, Berkeley, CA, June 2002.

[18] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[19] T. Marian. *Operating Systems Abstractions for Software Packet Processing in Datacenters*. PhD thesis, Cornell University, Jan. 2011.

[20] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[21] G. Memik, M. T. Kandemir, W. Liao, and A. Choudhary. Multicollective i/o: A technique for exploiting inter-file access patterns. volume 2, Aug. 2006.

[22] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. In *Operating Systems Review*, volume 43, pages 92–105, Jan. 2010.

[23] M. Polte, J. Simsa, and G. Gibson. Enabling enterprise solid state disks performance. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, Washington, DC, Mar. 2009.

[24] N. Provos. libevent. http://monkey.org/~provos/libevent/.

[25] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proc. ACM SIGCOMM*, New Delhi, India, Aug. 2010.

[26] A. Purohit, C. P. Wright, J. Spadavecchia, and E. Zadok. Cosy: Develop in user-land, run in kernel-mode. In *Proc. HotOS IX*, Lihue, Hawaii, May 2003.

[27] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlicting. Cassyopia: Compiler assisted system optimization. In *Proc. HotOS IX*, Lihue, Hawaii, May 2003.

[28] E. Seppanen, M. T. O'Keefe, and D. J. Lilja. High performance solid state storage under linux. In *26th IEEE Symposium on Massive Storage Systems and Technologies*, May 2010.

[29] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, Oct. 2010.

[30] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. GRAMPS: A programming model for graphics pipelines. In *ACM Transactions on Graphics*, Jan. 2009.

[31] V. Vasudevan, D. G. Andersen, M. Kaminsky, L. Tan, J. Franklin, and I. Moraru. Energy-efficient cluster computing with FAWN: Workloads and implications. In *Proc. e-Energy 2010*, Passau, Germany, Apr. 2010. (invited paper).

[32] V. Vasudevan, D. G. Andersen, and M. Kaminsky. The case for VOS: The vector operating system. In *Proc. HotOS XIII*, Napa, CA, May 2011.

[33] S. Venkataraman, N. Tolia, P. Ranganathan, and R. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, San Jose, CA, Feb. 2011.

[34] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.