

# FAWN: A Fast Array of Wimpy Nodes

By David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan

## Abstract

**This paper presents a fast array of wimpy nodes—FAWN—an approach for achieving low-power data-intensive data-center computing. FAWN couples low-power processors to small amounts of local flash storage, balancing computation and I/O capabilities. FAWN optimizes for per node energy efficiency to enable efficient, massively parallel access to data.**

**The key contributions of this paper are the principles of the FAWN approach and the design and implementation of FAWN-KV—a consistent, replicated, highly available, and high-performance key-value storage system built on a FAWN prototype. Our design centers around purely log-structured datastores that provide the basis for high performance on flash storage, as well as for replication and consistency obtained using chain replication on a consistent hashing ring. Our evaluation demonstrates that FAWN clusters can handle roughly 350 key-value queries per Joule of energy—two orders of magnitude more than a disk-based system.**

## 1. INTRODUCTION

Large-scale data-intensive applications, such as high-performance key-value storage systems, are growing in both size and importance; they now are critical parts of major Internet services such as Amazon (Dynamo<sup>7</sup>), LinkedIn (Voldemort), and Facebook (memcached).

The workloads these systems support share several characteristics: They are I/O, not computation, intensive, requiring random access over large datasets; they are massively parallel, with thousands of concurrent, mostly independent operations; their high load requires large clusters to support them; and the size of objects stored is typically small, for example, 1KB values for thumbnail images, hundreds of bytes for wall posts, and twitter messages.

The clusters that serve these workloads must provide both high performance and low-cost operation. Unfortunately, small-object random-access workloads are particularly ill served by conventional disk-based or memory-based clusters. The poor seek performance of disks makes disk-based systems inefficient in terms of both system performance and performance per Watt. High-performance DRAM-based clusters, storing terabytes or petabytes of data, are expensive and power-hungry: Two high-speed DRAM DIMMs can consume as much energy as a 1TB disk.

The power draw of these clusters is becoming an increasing fraction of their cost—up to 50% of the 3 year total cost of owning a computer. The density of the datacenters that house them is in turn limited by their ability to supply and

cool 10–20 kW of power per rack and up to 10–20 MW per datacenter.<sup>12</sup> Future datacenters may require as much as 200 MW,<sup>12</sup> and datacenters are being constructed today with dedicated electrical substations to feed them.

These challenges necessitate the question: Can we build a cost-effective cluster for data-intensive workloads that uses less than a tenth of the power required by a conventional architecture, but that still meets the same capacity, availability, throughput, and latency requirements?

The FAWN approach is designed to address this question. FAWN couples low-power, efficient CPUs with flash storage to provide efficient, fast, and cost-effective access to large, random-access data. Flash is faster than disk, cheaper than DRAM, and consumes less power than either. Thus, it is a particularly suitable choice for FAWN and its workloads. FAWN represents a class of systems that targets both system balance and per node energy efficiency: The 2008-era FAWN prototypes used in this work used embedded CPUs and CompactFlash, while today a FAWN node might be composed of laptop processors and higher-speed SSDs. Relative to today's highest-end computers, a contemporary FAWN system might use dual or quad-core 1.6 GHz CPUs with 1–4 GB of DRAM.

To show that it is *practical* to use these constrained nodes as the core of a large system, we designed and built the FAWN-KV cluster-based key-value store, which provides storage functionality similar to that used in several large enterprises.<sup>7</sup> FAWN-KV is designed to exploit the advantages and avoid the limitations of wimpy nodes with flash memory for storage.

The key design choice in FAWN-KV is the use of a *log-structured per node datastore* called FAWN-DS that provides high-performance reads and writes using flash memory. This append-only data log provides the basis for replication and strong consistency using *chain replication*<sup>21</sup> between nodes. Data is distributed across nodes using consistent hashing, with data split into contiguous ranges on disk such that all replication and node insertion operations involve only a fully in-order traversal of the subset of data that must be copied to a new node. Together with the log structure, these properties combine to provide fast failover and fast node insertion, and they minimize the time the affected datastore's key range is locked during such operations.

The original version of this paper was published in *Proceedings of the 22nd ACM Symposium of Operating Systems Principles*, October 2009.

We have built a prototype 21-node FAWN cluster using 500 MHz embedded CPUs. Each node can serve up to 1300 256 byte queries/s, exploiting nearly all of the raw I/O capability of their attached flash devices, and consumes under 5W when network and support hardware is taken into account. The FAWN cluster achieves 330 queries/J—two orders of magnitude better than traditional disk-based clusters.

## 2. WHY FAWN?

The FAWN approach to building *well-matched* cluster systems has the potential to achieve high performance and be fundamentally more energy-efficient than conventional architectures for serving massive-scale I/O and data-intensive workloads. We measure system performance in queries per second and measure energy efficiency in queries per Joule (equivalently, queries per second per Watt). FAWN is inspired by several fundamental trends:

**Increasing CPU-I/O gap:** Over the past several decades, the gap between CPU performance and I/O bandwidth has continually grown. For data-intensive computing workloads, storage, network, and memory bandwidth bottlenecks often cause low CPU utilization.

**FAWN approach:** To efficiently run I/O-bound data-intensive, computationally simple applications, FAWN uses wimpy processors selected to reduce I/O-induced idle cycles while maintaining high performance. The reduced processor speed then benefits from a second trend.

**CPU power consumption grows super-linearly with speed:** Higher frequencies require more energy, and techniques to mask the CPU-memory bottleneck come at the cost of energy efficiency. Branch prediction, speculative execution, out-of-order execution and large on-chip caches all require additional die area; modern processors dedicate as much as half their die to L2/3 caches.<sup>9</sup> These techniques do not increase the speed of basic computations, but do increase power consumption, making faster CPUs less energy efficient.

**FAWN approach:** A FAWN cluster's slower CPUs dedicate proportionally more transistors to basic operations. These CPUs execute significantly more *instructions per Joule* than their faster counterparts: Multi-GHz superscalar quad-core processors can execute approximately 100 million instructions/J, assuming all cores are active and avoid stalls or mispredictions. Lower-frequency in-order CPUs, in contrast, can provide over 1 billion instructions/J—an order of magnitude more efficient while running at 1/3 the frequency.

Worse yet, running fast processors below their full capacity draws a disproportionate amount of power.

**Dynamic power scaling on traditional systems is surprisingly inefficient:** A primary energy-saving benefit of dynamic voltage and frequency scaling (DVFS) was its ability to reduce voltage as it reduced frequency, but modern CPUs already operate near minimum voltage at the highest frequencies.

Even if processor energy was completely proportional to load, non-CPU components such as memory, motherboards, and power supplies have begun to dominate energy

consumption,<sup>2</sup> requiring that all components be scaled back with demand. As a result, a computer may consume over 50% of its peak power when running at only 20% of its capacity.<sup>20</sup> Despite improved power scaling technology, systems remain most energy efficient when operating at peak utilization.

A promising path to energy proportionality is turning machines off entirely.<sup>6</sup> Unfortunately, these techniques do not apply well to FAWN-KV's target workloads: Key-value systems must often meet service-level agreements for query throughput and latency of hundreds of milliseconds; the inter-arrival time and latency bounds of the requests prevent shutting machines down (and taking many seconds to wake them up again) during low load.<sup>2</sup>

Finally, energy proportionality alone is not a panacea: Systems should be both proportional *and* efficient at 100% load. FAWN specifically addresses efficiency, and cluster techniques that improve proportionality should apply universally.

## 3. DESIGN AND IMPLEMENTATION

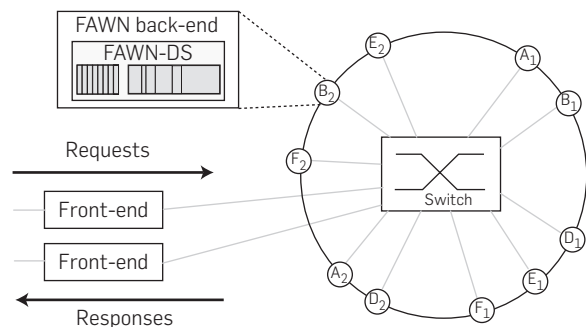
We describe the design and implementation of the system components from the bottom up: a brief overview of flash storage (Section 3.2), the per node FAWN-DS datastore (Section 3.3), and the FAWN-KV cluster key-value lookup system (Section 3.4), including replication and consistency.

### 3.1. Design overview

Figure 1 gives an overview of the entire FAWN system. Client requests enter the system at one of several *front ends*. The front-end nodes forward the request to the *back-end* FAWN-KV node responsible for serving that particular key. The back-end node serves the request from its FAWN-DS datastore and returns the result to the front end (which in turn replies to the client). Writes proceed similarly.

The large number of back-end FAWN-KV storage nodes is organized into a ring using consistent hashing. As in systems such as Chord,<sup>18</sup> keys are mapped to the node that follows the key in the ring (its *successor*). To balance load and reduce failover times, each *physical node* joins the ring as a small number ( $V$ ) of *virtual nodes*, each virtual node representing a *virtual ID* (“VID”) in the ring space. Each physical node is thus responsible for  $V$  different (noncontiguous) key ranges. The data associated with each virtual ID is stored on flash using FAWN-DS.

Figure 1. FAWN-KV architecture.



### 3.2. Understanding flash storage

Flash provides a non-volatile memory store with several significant benefits over typical magnetic hard disks for random-access, read-intensive workloads—but it also introduces several challenges. Three characteristics of flash underlie the design of the FAWN-KV system described in this section:

1. **Fast random reads:** ( $\ll 1$  ms) up to 175 times faster than random reads on magnetic disk.<sup>17</sup>
2. **Efficient I/O:** Many flash devices consume less than 1 W even under heavy load, whereas mechanical disks can consume over 10 W at load.
3. **Slow random writes:** Small writes on flash are expensive. Updating a single page requires first erasing an entire erase block (128–256KB) of pages and then writing the modified block in its entirety. Updating a single byte of data is therefore as expensive as writing an entire block of pages.<sup>16</sup>

Modern devices improve random write performance using write buffering and preemptive block erasure. These techniques improve performance for short bursts of writes, but sustained random writes still underperform.<sup>17</sup>

These performance problems motivate log-structured techniques for flash filesystems and data structures.<sup>10, 15, 16</sup> These same considerations inform the design of FAWN’s node storage management system, described next.

### 3.3. The FAWN datastore

FAWN-DS is a log-structured key-value store. Each store contains values for the key range associated with one virtual ID. It acts to clients like a disk-based hash table that supports Store, Lookup, and Delete.

FAWN-DS is designed to perform well on flash storage and to operate within the constrained DRAM available on wimpy nodes: All writes to the datastore are sequential, and reads require a single random access. To provide this property, FAWN-DS maintains an in-DRAM hash table (Hash Index) that maps keys to an offset in the append-only Data Log on flash (Figure 2a). This log-structured design is similar to several append-only filesystems such as the Google File System (GFS) and Venti, which avoid random seeks on magnetic disks for writes.

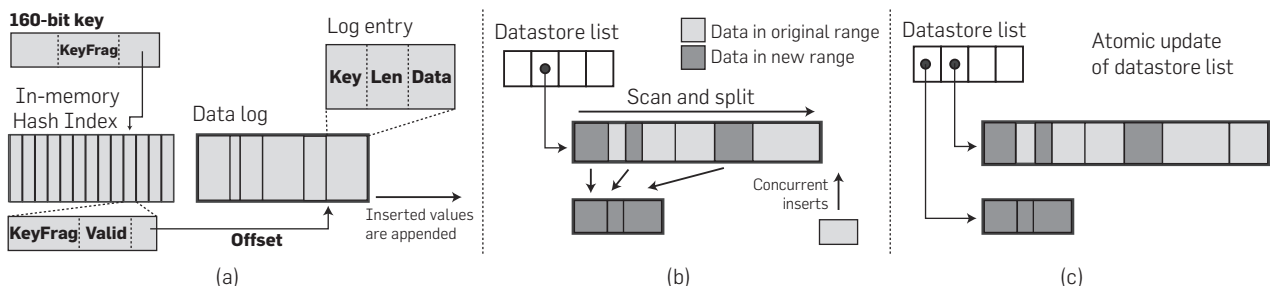
**Mapping a key to a value:** FAWN-DS uses an in-memory (DRAM) Hash Index to map 160 bit keys to a value stored in the Data Log. It stores only a fragment of the actual key in memory to find a location in the log; it then reads the full key (and the value) from the log and verifies that the key it read was, in fact, the correct key. This design trades a small and configurable chance of requiring two reads from flash (we set it to roughly 1 in 32,768 accesses) for drastically reduced memory requirements (only 6 bytes of DRAM per key-value pair).

FAWN-DS’s Lookup procedure extracts two fields from the 160 bit key: The  $i$  low order bits of the key (the *index bits*) and the next 15 low order bits (the *key fragment*). FAWN-DS uses the index bits to select a bucket from the Hash Index, which contains  $2^i$  hash buckets. Each bucket is 6 bytes: a 15 bit key fragment, a valid bit, and a 4 byte pointer to the location in the Data Log where the full entry is stored.

Lookup proceeds, then, by locating a bucket using the index bits and comparing the key against the key fragment. If the fragments do not match, FAWN-DS uses hash chaining to continue searching the hash table. Once it finds a matching key fragment, FAWN-DS reads the record off of the flash. If the stored full key in the on-flash record matches the desired lookup key, the operation is complete. Otherwise, FAWN-DS resumes its hash chaining search of the in-memory hash table and searches additional records. With the 15-bit key fragment, only 1 in 32,768 retrievals from the flash will be incorrect and require fetching an additional record.

The constants involved (15 bits of key fragment, 4 bytes of log pointer) target the prototype FAWN nodes described in Section 4. A typical object is between 256 bytes and 1KB, and the nodes have 256MB of DRAM and approximately 4GB of flash storage. Because each physical node is responsible for  $V$  key ranges (each with its own datastore file), it can address  $4GB * V$  bytes of data. Expanding the in-memory storage to 7 bytes per entry would permit FAWN-DS to address 1TB of data per key range. While some additional optimizations are possible, such as rounding the size of objects stored in flash or reducing the number of bits used for the key fragment (and thus incurring, e.g., a 1-in-1000 chance of having to do two reads from flash), the current design works well for the key-value workloads we study.

**Figure 2. (a) FAWN-DS appends writes to the end of the Data Log. (b) Split requires a sequential scan of the data region, transferring out-of-range entries to the new store. (c) After scan completes, the datastore list is atomically updated to add the new store. Compaction of the original store cleans up out-of-range entries.**



**Reconstruction:** The Data Log contains all the information necessary to reconstruct the Hash Index from scratch. As an optimization, FAWN-DS periodically checkpoints the index by writing the Hash Index and a pointer to the last log entry to flash. After a failure, FAWN-DS uses the checkpoint as a starting point to reconstruct the in-memory Hash Index.

**Virtual IDs and semi-random writes:** A physical node has a separate FAWN-DS datastore file for each of its virtual IDs, and FAWN-DS appends new or updated data items to the appropriate datastore. Sequentially appending to a small number of files is termed *semi-random writes*. With many flash devices, these semi-random writes are nearly as fast as a single sequential append.<sup>15</sup> We take advantage of this property to retain fast write performance while allowing key ranges to be stored in independent files to speed the maintenance operations described in the following.

### 3.3.1. Basic functions: Store, lookup, delete

*Store* appends an entry to the log, updates the corresponding hash table entry to point to the offset of the newly appended entry within the Data Log, and sets the valid bit to true. If the key written already existed, the old value is now *orphaned* (no hash entry points to it) for later garbage collection.

*Lookup* retrieves the hash entry containing the offset, indexes into the Data Log, and returns the data blob.

*Delete* invalidates the hash entry corresponding to the key and writes a *Delete entry* to the end of the data file. The delete entry is necessary for fault tolerance—the invalidated hash table entry is not immediately committed to non-volatile storage to avoid random writes, so a failure following a delete requires a log to ensure that recovery will delete the entry upon reconstruction. Because of its log structure, FAWN-DS deletes are similar to store operations with 0 byte values. Deletes do not immediately reclaim space and require compaction to perform garbage collection. This design defers the cost of a random write to a later sequential write operation.

### 3.3.2. Maintenance: Split, merge, compact

Inserting a new virtual node into the ring causes one key range to split into two, with the new virtual node gaining responsibility for the first part of it. Nodes handling these *VIDs* must therefore *Split* their datastore into two datastores, one for each key range. When a virtual node departs the system, two adjacent key ranges must similarly *Merge* into a single datastore. In addition, a virtual node must periodically *Compact* its datastores to clean up stale or orphaned entries created by *Split*, *Store*, and *Delete*.

These maintenance functions are designed to work well on flash, requiring only scans of one datastore and sequential writes into another.

*Split* parses the Data Log sequentially, writing each entry in a new datastore if its key falls in the new datastore's range.

*Merge* writes every log entry from one datastore into the other datastore; because the key ranges are independent, it does so as an append. *Split* and *Merge* propagate delete

entries into the new datastore.

*Compact* cleans up entries in a datastore, similar to garbage collection in a log-structured filesystem. It skips entries that fall outside of the datastore's key range, which may be leftover after a split. It also skips orphaned entries that no in-memory hash table entry points to, and then skips any delete entries corresponding to those entries. It writes all other valid entries into the output datastore.

### 3.3.3. Concurrent maintenance and operation

All FAWN-DS maintenance functions allow concurrent reads and writes to the datastore. *Stores* and *Deletes* only modify hash table entries and write to the end of the log.

Maintenance operations (*Split*, *Merge*, and *Compact*) sequentially parse the Data Log, which may be growing due to deletes and stores. Because the log is append only, a log entry once parsed will never be changed. These operations each create one new output datastore logfile. The maintenance operations run until they reach the end of the log, and then briefly lock the datastore, ensure that all values flushed to the old log have been processed, update the FAWN-DS datastore list to point to the newly created log, and release the lock (Figure 2c).

## 3.4. The FAWN key-value system

In FAWN-KV, client applications send requests to front ends using a standard put/get interface. Front ends send the request to the back-end node that owns the key space for the request. The back-end node satisfies the request using its FAWN-DS and replies to the front ends.

### 3.4.1. Consistent hashing: Key ranges to nodes

A typical FAWN cluster will have several front ends and many back ends. FAWN-KV organizes the back-end *VIDs* into a storage ring-structure using consistent hashing.<sup>18</sup> Front ends maintain the entire node membership list and directly forward queries to the back-end node that contains a particular data item.

Each front-end node manages the *VID* membership list and queries for a large contiguous chunk of the key space. A front end receiving queries for keys outside of its range forwards the queries to the appropriate front-end node. This design either requires clients to be roughly aware of the front-end mapping or doubles the traffic that front ends must handle, but it permits front ends to cache values without a cache consistency protocol.

The key space is allocated to front ends by a single management node; we envision this node being replicated using a small Paxos cluster,<sup>13</sup> but we have not (yet) implemented this. There would be 80 or more back-end nodes per front-end node with our current hardware prototypes, so the amount of information this management node maintains is small and changes infrequently—a list of 125 front ends would suffice for a 10,000 node FAWN cluster. When a back-end node joins, it obtains the list of front-end IDs. It uses this list to determine which front ends to contact to join the ring, one *VID* at a time. We chose this design so that the system would be robust to front-end node failures: The back-end node identifier (and thus, what keys

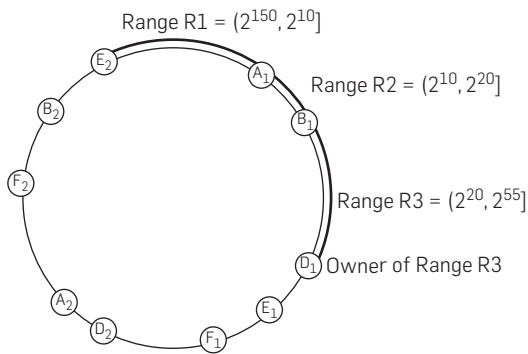
it is responsible for) is a deterministic function of the back-end node ID. If a front-end node fails, data does not move between back-end nodes, though virtual nodes may have to attach to a new front end.

FAWN-KV uses a 160 bit circular ID space for *VIDs* and keys. Virtual IDs are hashed identifiers derived from the node's address. Each *VID* owns the items for which it is the item's successor in the ring space (the node immediately clockwise in the ring). As an example, consider the cluster depicted in Figure 3 with five physical nodes, each of which has two *VIDs*. The physical node *A* appears as *VIDs* *A1* and *A2*, each with its own 160 bit identifiers. *VID* *A1* owns key range *R1*, *VID* *B1* owns range *R2*, and so on.

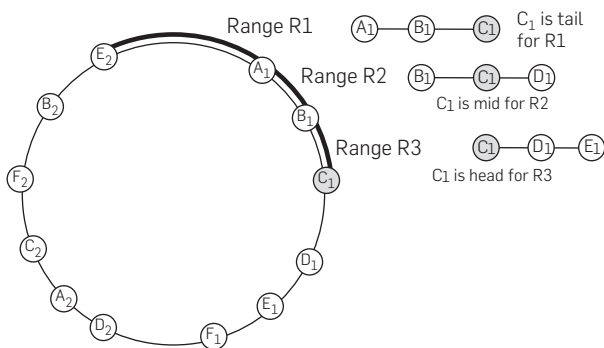
### 3.4.2. Replication and consistency

FAWN-KV offers a configurable replication factor for fault tolerance. Items are stored at their successor in the ring space and at the  $R - 1$  following virtual IDs. FAWN-KV uses chain replication<sup>21</sup> to provide strong consistency on a per key basis. Updates are sent to the head of the chain, passed along to each member of the chain via a TCP connection between the nodes, and queries are sent to the tail of the chain. By mapping chain replication to the consistent hashing ring, each virtual ID in FAWN-KV is part of  $R$  different chains: it is the "tail" for one chain, a "mid" node in  $R - 2$  chains, and the "head" for one. Figure 4 depicts a ring with

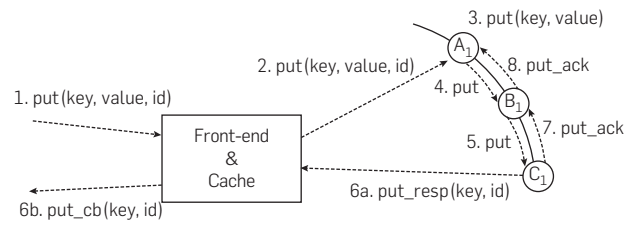
**Figure 3. Consistent hashing with five physical nodes and two virtual IDs each.**



**Figure 4. Overlapping chains in the ring—each node in the ring is part of  $R = 3$  chains.**



**Figure 5. Life cycle of a put with chain replication—puts go to the head and are propagated through the chain. Gets go directly to the tail.**



six physical nodes, where each has two virtual IDs ( $V = 2$ ), using a replication factor of 3. In this figure, node *C1* is the tail for range *R1*, mid for range *R2*, and tail for range *R3*.

Figure 5 shows a put request for an item in range *R1*. The front end sends the put to the key's successor, *VID* *A1*, which is the head of the replica chain for this range. After storing the value in its datastore, *A1* forwards this request to *B1*, which stores the value and forwards the request to the tail, *C1*. After storing the value, *C1* sends the put response back to the front end and sends an acknowledgment back up the chain indicating that the response was handled properly.

For reliability, nodes buffer put requests until they receive the acknowledgment. Because puts are written to an append-only log in FAWN-DS and are sent in-order along the chain, this operation is simple: nodes maintain a pointer to the last unacknowledged put in their datastore and increment it when they receive an acknowledgment. By using a log-structured datastore, chain replication in FAWN-KV reduces to simply streaming the datastore from node to node.

Get requests proceed as in chain replication—the front end directly routes gets to the tail of the chain for range *R1*, node *C1*, which responds to requests. Any update seen by the tail has therefore also been applied by other replicas in the chain.

## 4. EVALUATION

We begin by characterizing the baseline I/O performance of a node. We then show that FAWN-DS's performance is similar to the node's baseline I/O capability. To illustrate the advantages of FAWN-DS's design, we compare its performance to an implementation using the general-purpose BerkeleyDB, which is not optimized for flash writes. We then study a prototype FAWN-KV system running on a 21-node cluster, evaluating its energy efficiency in queries per second per Watt.

**Evaluation hardware:** Our FAWN cluster has 21 back-end nodes built from commodity PEngine Alix 3c2 devices, commonly used for thin clients, kiosks, network firewalls, wireless routers, and other embedded applications. These devices have a single-core 500MHz AMD Geode LX processor, 256MB DDR SDRAM operating at 400 MHz, and 100Mbit/s Ethernet. Each node contains one 4GB Sandisk Extreme IV CompactFlash device. A node consumes 3W when idle and a maximum of 6W when using 100% CPU, network, and flash. The nodes are connected to each other

and to a 27 W Intel Atom-based front-end node using two 16-port Netgear GS116 GigE Ethernet switches.

**Evaluation workload:** We show query performance for 256 byte and 1KB values. We select these sizes as proxies for small text posts, user reviews or status messages, image thumbnails, and so on. They represent a quite challenging regime for conventional disk-bound systems and stress the limited memory and CPU of our wimpy nodes.

#### 4.1. Individual node performance

We benchmark the I/O capability of the FAWN nodes using iotop and Flexible I/O tester. The flash is formatted with the ext2 filesystem. These tests read and write 1KB entries, the lowest record size available in iotop. The filesystem I/O performance using a 3.5GB file is shown in Table 1.

##### 4.1.1. FAWN-DS single node local benchmarks

**Lookup speed:** This test shows the query throughput achieved by a local client issuing queries for randomly distributed, existing keys on a single node. We report the average of three runs (the standard deviations were below 5%). Table 2 shows FAWN-DS 1KB and 256 byte random read queries/s as a function of the DS size. If the datastore fits in the buffer cache, the node locally retrieves 50,000–85,000 queries/s. As the datastore exceeds the 256MB of RAM available on the nodes, a larger fraction of requests go to flash.

FAWN-DS imposes modest overhead from hash lookups, data copies, and key comparisons; and it must read slightly more data than the iotop tests (each stored entry has a header). The query throughput, however, remains high: Tests reading a 3.5 GB datastore using 1 KB values achieved 1,150 queries/s compared to 1,424 queries/s from the filesystem. Using 256 byte entries achieved 1,298 queries/s from a 3.5 GB datastore. By comparison, the raw filesystem achieved 1,454 random 256 byte reads/s using Flexible I/O.

**Bulk store speed:** The log structure of FAWN-DS ensures that data insertion is entirely sequential. Inserting 2 million

**Table 1. Baseline CompactFlash statistics for 1KB entries.**  
QPS = Queries/second.

Seq. Read	Rand Read	Seq. Write	Rand. Write
28.5MB/s	1424 QPS	24MB/s	110 QPS

**Table 2. Local random read speed of FAWN-DS.**

DS Size	1KB Rand Read (in queries/s)	256 bytes Rand Read (in queries/s)
10KB	72352	85012
125MB	51968	65412
250MB	6824	5902
500MB	2016	2449
1GB	1595	1964
2GB	1446	1613
3.5GB	1150	1298

entries of 1KB each (2GB total) into a *single* FAWN-DS log proceeds at 23.2MB/s (nearly 24,000 entries/s), which is 96% of the raw speed that the flash can be written through the filesystem.

**Put speed:** Each FAWN-KV node has  $R * V$  FAWN-DS files: Each virtual ID adds one primary data range, plus an additional  $R - 1$  replicated ranges. A node receiving puts for different ranges will concurrently append to a small number of files (“semi-random writes”). Good semi-random write performance is central to FAWN-DS’s per range data layout that enables single-pass maintenance operations. Our recent work confirms that modern flash devices can provide good semi-random write performance.<sup>1</sup>

##### 4.1.2. Comparison with BerkeleyDB

To understand the benefit of FAWN-DS’s log structure, we compare with a general purpose disk-based database that is *not* optimized for flash. BerkeleyDB provides a simple put/get interface, can be used without heavy-weight transactions or rollback, and performs well vs. other memory or disk-based databases. We configured BerkeleyDB using both its default settings and using the reference guide suggestions for flash-based operation.<sup>3</sup> The best performance we achieved required 6 hours to insert 7 million, 200 byte entries to create a 1.5GB B-Tree database. This corresponds to an insert rate of 0.07MB/s.

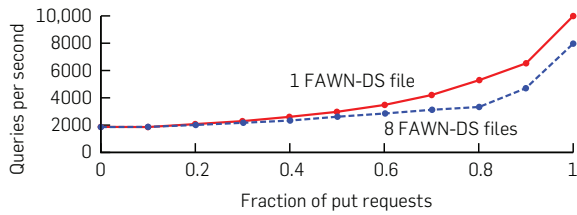
The problem was, of course, small writes: When the BDB store was larger than the available RAM on the nodes (<256MB), BDB had to flush pages to disk, causing many writes that were much smaller than the size of an erase block.

That comparing FAWN-DS and BDB seems unfair is exactly the point: Even a well-understood, high-performance database will perform poorly when its write pattern has not been specifically optimized to flash characteristics. We evaluated BDB on top of NILFS2, a log-structured Linux filesystem for block devices, to understand whether log-structured writing could turn the random writes into sequential writes. Unfortunately, this combination was not suitable because of the amount of metadata created for small writes for use in filesystem checkpointing and rollback, features not needed for FAWN-KV—writing 200MB worth of 256 bytes key-value pairs generated 3.5GB of metadata. Other existing Linux log-structured flash filesystems, such as JFFS2, are designed to work on raw flash, but modern SSDs, compact flash, and SD cards all include a Flash Translation Layer that hides the raw flash chips. While future improvements to filesystems can speed up naive DB performance on flash, the pure log structure of FAWN-DS remains necessary even if we could use a more conventional back end: It provides the basis for replication and consistency across an array of nodes.

##### 4.1.3. Read-intensive vs. write-intensive workloads

Most read-intensive workloads have some writes. For example, Facebook’s memcached workloads have a 1:6 ratio of application-level puts to gets.<sup>11</sup> We therefore measured the aggregate query rate as the fraction of puts ranging from 0

**Figure 6. FAWN supports both read- and write-intensive workloads. Small writes are cheaper than random reads due to the FAWN-DS log structure.**



(all gets) to 1 (all puts) on a single node (Figure 6).

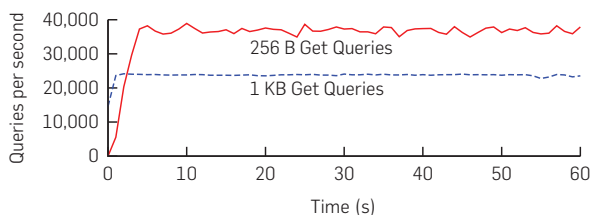
FAWN-DS can handle more puts per second than gets because of its log structure. Even though semi-random write performance across eight files on our CompactFlash devices is worse than purely sequential writes, it still achieves higher throughput than pure random reads.

When the put-ratio is low, the query rate is limited by the get requests. As the ratio of puts to gets increases, the faster puts significantly increase the aggregate query rate. On the other hand, a pure write workload that updates a small subset of keys would require frequent cleaning. In our current environment and implementation, both read and write rates slow to about 700–1000 queries/s during compaction, bottlenecked by increased thread switching and system call overheads of the cleaning thread. Last, because deletes are effectively 0 byte value puts, delete-heavy workloads are similar to insert workloads that update a small set of keys frequently. In the next section, we mostly evaluate read-intensive workloads because it represents the target workloads for which FAWN-KV is designed.

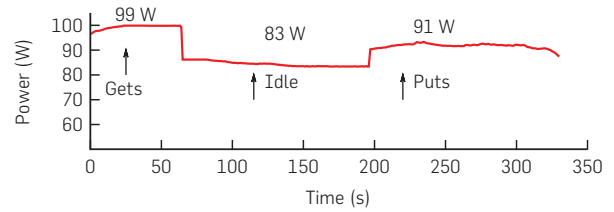
#### 4.2. FAWN-KV system benchmarks

**System throughput:** To measure query throughput, we populated the KV cluster with 20GB of values and then measured the maximum rate at which the front end received query responses for random keys. Figure 7 shows that the cluster sustained roughly 36,000 256 byte gets per second (1,700 per second per node) and 24,000 1KB gets per second (1,100 per second per node). A single node serving a 512MB datastore over the network could sustain roughly 1,850 256 byte gets per second per node, while Table 2 shows that it could serve the queries locally at 2,450 256 byte queries per second per node. Thus, a single node serves roughly 70% of the sustained rate that a single FAWN-DS could handle with

**Figure 7. Query throughput on 21-node FAWN-KV system for 1KB and 256 bytes entry sizes.**



**Figure 8. Power consumption of 21-node FAWN-KV system for 256 bytes values during Puts/Gets.**



local queries. The primary reasons for the difference are the addition of network overhead, request marshaling and unmarshaling, and load imbalance—with random key distribution, some back-end nodes receive more queries than others, slightly reducing system performance.

**System power consumption:** Using a WattsUp power meter that logs power draw each second, we measured the power consumption of our 21-node FAWN-KV cluster and two network switches. Figure 8 shows that, when idle, the cluster uses about 83 W, or 3 W/node and 10 W/switch. During gets, power consumption increases to 99 W, and during insertions, power consumption is 91 W. Peak get performance reaches about 36,000 256 bytes queries/s for the cluster serving the 20GB dataset, so this system, excluding the front end, provides 364 queries/J.

The front end connects to the back-end nodes through a 1 Gbit/s uplink on the switch, so the cluster requires about one low-power front end for every 80 nodes—enough front ends to handle the aggregate query traffic from all the back ends (80 nodes \* 1500 queries/s/node \* 1KB/query = 937 Mbit/s). Our prototype front end uses 27 W, which adds nearly 0.5 W/node amortized over 80 nodes, providing 330 queries/J for the entire system. A high-speed (4 ms seek time, 10 W) magnetic disk by itself provides less than 25 queries/J—two orders of magnitude fewer than our existing FAWN prototype.

Network switches currently account for 20% of the power used by the entire system. Moving to FAWN requires roughly one 8-to-1 aggregation switch to make a group of FAWN nodes look like an equivalent-bandwidth server; we account for this in our evaluation by including the power of the switch when evaluating FAWN-KV. As designs such as FAWN reduce the power drawn by servers, the importance of creating scalable, energy-efficient datacenter networks will grow.

#### 5. ALTERNATIVE ARCHITECTURES

When is the FAWN approach likely to beat traditional architectures? We examine this question by comparing the 3 year total cost of ownership (TCO) for six systems: Three “traditional” servers using magnetic disks, flash SSDs, and DRAM; and three hypothetical FAWN-like systems using the same storage technologies. We define the 3 year TCO as the sum of the capital cost and the 3 year power cost at 10 cents/kWh.

Because the FAWN systems we have built use several-year-old technology, we study a theoretical 2009 FAWN node

using a low-power CPU that consumes 10W–20 W and costs ~\$150 in volume. We in turn give the benefit of the doubt to the server systems we compare against—we assume a 2 TB disk exists that serves 300 queries/s at 10 W.

Our results indicate that both FAWN and traditional systems have their place—but for the small random-access workloads we study, traditional systems are surprisingly absent from much of the solution space, in favor of FAWN nodes using either disks, flash, or DRAM.

Key to the analysis is a question: *Why does a cluster need nodes?* The answer is, of course, for both storage space and query rate. Storing a  $DS$  gigabyte dataset with query rate  $QR$  requires  $N$  nodes:

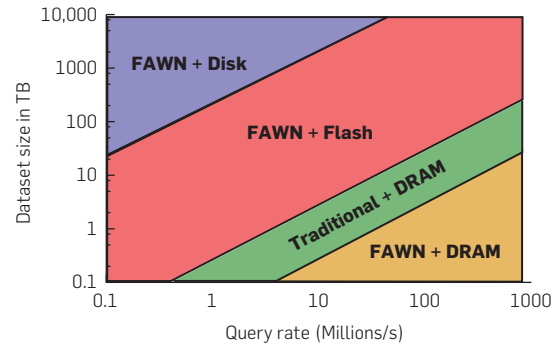
$$N = \max \left( \frac{DS}{\frac{gb}{node}}, \frac{QR}{\frac{qr}{node}} \right)$$

With large datasets with low query rates, the number of nodes required is dominated by the storage capacity per node: Thus, the important metric is the total cost per GB for an individual node. Conversely, for small datasets with high query rates, the per node query capacity dictates the number of nodes: the dominant metric is queries per second per dollar. Between these extremes, systems must provide the best trade-off between per node storage capacity, query rate, and power cost.

Table 3 shows these cost and speculative performance statistics for several candidate systems circa 2009; while the numbers are outdated, the trends likely still apply. The “traditional” nodes use 200 W servers that cost \$1,000 each. *Traditional+Disk* pairs a single server with five 2 TB high-speed (10,000 RPM) disks capable of 300 queries/s, each disk consuming 10 W. *Traditional+SSD* uses two PCI-E Fusion-IO 80GB flash SSDs, each also consuming about 10 W (Cost: \$3K). *Traditional+DRAM* uses 8GB server-quality DRAM modules, each consuming 10 W. *FAWN+Disk* nodes use one 2 TB 7200 RPM disk: FAWN nodes have fewer connectors available on the board. *FAWN+SSD* uses one 32GB Intel SATA flash SSD capable of 35,000 random reads/s,<sup>17</sup> consuming 2 W (\$400). *FAWN+DRAM* uses a single 2GB, slower DRAM module, also consuming 2 W.

Figure 9 shows which base system has the lowest cost for a particular dataset size and query rate, with dataset sizes between 100GB and 10PB and query rates between 100 K

**Figure 9. Solution space for lowest 3 year TCO as a function of dataset size and query rate.**



and 1 billion/s.

**Large datasets, low query rates:** *FAWN+Disk* has the lowest total cost per GB. While not shown on our graph, a traditional system wins for exabyte-sized workloads if it can be configured with sufficient disks per node (over 50), though packing 50 disks per machine poses reliability challenges.

**Small datasets, high query rates:** *FAWN+DRAM* costs the fewest dollars per queries per second, keeping in mind that we do *not* examine workloads that fit entirely in L2 cache on a traditional node. This somewhat counterintuitive result is similar to that made by the intelligent RAM project, which coupled processors and DRAM to achieve similar benefits<sup>4</sup> by avoiding the memory wall. We assume the FAWN nodes can only accept 2GB of DRAM per node, so for larger datasets, a traditional DRAM system provides a high query rate and requires fewer nodes to store the same amount of data (64GB vs. 2GB/node).

**Middle range:** *FAWN+SSDs* provide the best balance of storage capacity, query rate, and total cost. If SSD cost per GB improves relative to magnetic disks, this combination is likely to continue expanding into the range served by *FAWN+Disk*; if the SSD cost per performance ratio improves relative to DRAM, so will it reach into DRAM territory. It is therefore conceivable that *FAWN+SSD* could become the dominant architecture for many random-access workloads.

*Are traditional systems obsolete?* We emphasize that this analysis applies only to small, random-access workloads.

**Table 3. Traditional and FAWN node statistics.**

System	Cost	W	QPS	Queries/Joule	GB/Watt	TCO/GB	TCO/QPS
<i>Traditionals</i>							
5–2TB Disks	\$2K	250	1500	6	40	0.26	1.77
160GB PCIe SSD	\$8K	220	200K	909	0.72	53	0.04
64GB DRAM	\$3K	280	1M	3.5K	0.23	59	0.004
<i>FAWNs</i>							
2TB Disk	\$350	20	250	12.5	100	0.20	1.61
32GB SSD	\$500	15	35K	2.3K	2.1	16.9	0.015
2GB DRAM	\$250	15	100K	6.6K	0.13	134	0.003



Sequential-read workloads are similar, but the constants depend strongly on the per byte processing required. Traditional cluster architectures retain a place for CPU-bound workloads, but we do note that architectures such as IBM's BlueGene successfully apply large numbers of low-power, efficient processors to many supercomputing applications—but they augment their wimpy processors with custom floating point units to do so.

Our definition of “total cost of ownership” ignores several notable costs: In comparison to traditional architectures, FAWN should reduce power and cooling infrastructure but may increase network-related hardware and power costs due to the need for more switches. Our current hardware prototype improves work done per volume, thus reducing costs associated with datacenter rack or floor space. Finally, our analysis assumes that cluster software developers can engineer away the human costs of management—an optimistic assumption for all architectures. We similarly ignore issues such as ease of programming, though we selected an x86-based wimpy platform for ease of development.

## 6. RELATED WORK

Several projects are using low-power processors for datacenter workloads to reduce energy consumption.<sup>5, 8, 14, 19</sup> These systems leverage low-cost, low-power commodity components for datacenter systems, similarly arguing that this approach can achieve the highest work per dollar and per Joule. More recently, ultra-low power server systems have become commercially available, with companies such as SeaMicro, Marvell, Calxeda, and ZT Systems producing low-power datacenter computing systems based on Intel Atom and ARM platforms.


FAWN builds upon these observations by demonstrating the importance of re-architecting the software layers in obtaining the potential energy efficiency such hardware can provide.

## 7. CONCLUSION

The FAWN approach uses nodes that target the “sweet spot” of per node energy efficiency, typically operating at about half the frequency of the fastest available CPUs. Our experience in designing systems using this approach, often coupled with fast flash memory, has shown that it has substantial potential to improve energy efficiency, but that these improvements may come at the cost of re-architecting software or algorithms to operate with less memory, slower CPUs, or the quirks of flash memory: The FAWN-KV key-value system presented here is one such example. By successfully adapting the software to this efficient hardware, our then four-year-old FAWN nodes delivered over an order of magnitude more queries per Joule than conventional disk-based systems.

Our ongoing experience with newer FAWN-style systems shows that its energy efficiency benefits remain achievable, but that further systems challenges—such as high kernel I/O overhead—begin to come into play. In this light, we view our experience with FAWN as a potential harbinger of the systems challenges that are likely to arise for future many-core energy-efficient systems.

## Acknowledgments

This work was supported in part by gifts from Network Appliance, Google, and Intel Corporation, and by grant CCF-0964474 from the National Science Foundation, as well as graduate fellowships from NSF, IBM, and APC. We extend our thanks to our OSDI and SOSP reviewers, Vyas Sekar, Mehul Shah, and to Lorenzo Alvisi for shepherding the work for SOSP. Iulian Moraru provided feedback and performance-tuning assistance. 

## References

1. Andersen, D.G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, October 2009).
2. Barroso, L.A., Hölzle, U. The case for energy-proportional computing. *Computer* 40, 12 (2007), 33–37.
3. Memory-only or Flash configurations. <http://www.oracle.com/technology/documentation/berkeley-db/db/ref/program/ram.html>
4. Bowman, W., Cardwell, N., Kozyrakis, C., Romer, C., Wang, H. Evaluation of existing architectures in IRAM systems. In *Workshop on Mixing Logic and DRAM, 24th International Symposium on Computer Architecture* (Denver, CO, June 1997).
5. Caulfield, A.M., Grupp, L.M., Swanson, S. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)* (San Diego, CA, March 2009).
6. Chase, J.S., Anderson, D., Thakar, P., Vahdat, A., Doyle, R. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Banff, AB, Canada, October 2001).
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (Stevenson, WA, Oct. 2007).
8. Hamilton, J. Cooperative expendable micro-slice servers (CEMS): Low cost, low power servers for Internet scale services. [http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton\\_CEHS.pdf](http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_CEHS.pdf) (2009).
9. Penryn Press Release. <http://www.intel.com/pressroom/archive/releases/20070328fact.htm>
10. The Journaling Flash File System. <http://sources.redhat.com/jffs2/>
11. Johnson, B. Facebook, personal communication (November 2008).
12. Katz, R.H. Tech titans building boom. *IEEE Spectrum* (February 2009). <http://spectrum.ieee.org/green-tech/buildings/tech-titans-building-boom>
13. Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst.*, 16, 2, (1998), 133–169.
14. Lim, K., Ranganathan, P., Chang, J., Patel, C., Mudge, T., Reinhardt, S. Understanding and designing new server architectures for emerging warehouse-computing environments. In *International Symposium on Computer Architecture (ISCA)* (Beijing, China, June 2008).
15. Nath, S., Gibbons, P.B. Online maintenance of very large random samples on flash storage. In *Proceedings of VLDB* (Auckland, New Zealand, August 2008).
16. Nath, S., Kansal, A. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of ACM/IEEE International Conference on Information Processing in Sensor Networks* (Cambridge, MA, April 2007).
17. Polte, M., Simsa, J., Gibson, G. Enabling enterprise solid state disks performance. In *Proceedings of the Workshop on Integrating Solid-State Memory into the Storage Hierarchy* (Washington, DC, March 2009).
18. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. August 2001. <http://portal.acm.org/citation.cfm?id=383071>
19. Szalay, A., Bell, G., Terzis, A., White, A., Vandenberg, J. Low power Amdahl blades for data intensive computing, 2009. <http://portal.acm.org/citation.cfm?id=1740407&dl=ACM>
20. Tolia, N., Wang, Z., Marwah, M., Bash, C., Ranganathan, P., Zhu, X. Delivering energy proportionality with non energy-proportional systems—optimizing the ensemble. In *Proceedings of HotPower* (Palo Alto, CA, December 2008).
21. van Renesse, R. Schneider, F.B. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th USENIX OSDI* (San Francisco, CA, December 2004).

David G. Andersen, Jason Franklin, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan, Carnegie Mellon University

Michael Kaminsky, Intel Labs