FAWN: A Fast Array of Wimpy Nodes

Paper ID: 188 (14 pages)

Abstract

This paper presents a new cluster architecture for low-power data-intensive computing. FAWN couples low-power embedded CPUs to small amounts of local flash storage, and balances computation and I/O capabilities to enable efficient, massively parallel access to data.

The key contributions of this paper are the principles of the FAWN architecture 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 400 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 [9]), LinkedIn ("Voldemort" [29]), and Facebook (`memcached` [23]).

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 are typically small, e.g. 1 KB values for thumbnail images, 100s of bytes for wall posts, twitter messages, etc.

The clusters that serve these workloads must provide both high performance and low cost operation. Unfortunately, small 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 both expensive and consume a surprising amount of power—two 2 GB DIMMs consume as much energy as a 1 TB disk.

The power draw of these clusters is becoming an increasing fraction of their cost—-up to 50% of the three-year total cost of owning a computer. The density of the data centers 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 [19]. Future data centers may require as much as 200 MW [19], and data centers 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 size, availability, throughput, and latency requirements?

In this paper, we present the FAWN architecture—a Fast Array of Wimpy Nodes—that is designed to address this question. FAWN couples low-power, efficient embedded CPUs with flash storage to provide efficient, fast, and cost-effective access to large, random-access data. Flash is significantly faster than disk, much cheaper than the equivalent amount of DRAM, and consumes less power than both. Thus, it is a particularly suitable choice for FAWN and its workloads. FAWN creates a well-matched system architecture around flash: each node can use the full capacity of the flash without memory or bus bottlenecks, but does not waste excess power.

To show that it is *practical* to use these constrained nodes as the core of a large system, we have designed and built the FAWN-KV cluster-based key-value store, which provides storage functionality similar to that used in several large enterprises [9, 29, 23]. FAWN-KV is designed specifically with the FAWN hardware in mind, and is able 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 data store* 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* [39] 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 on node insertion, and they minimize the

time the database is locked during such operations—for a single node failure and recovery, the database is blocked for at most 100 milliseconds.

We have built a prototype 21-node FAWN cluster using 500 MHz embedded CPUs. Each node can serve up to 1700 256 byte queries per second, exploiting nearly all of the raw I/O capability of their attached flash devices, and consumes under 5 W when network and support hardware is taken into account. The FAWN cluster achieves 364 queries per second per Watt—two orders of magnitude better than traditional disk-based clusters.

In Section 5, we compare a FAWN-based approach to other architectures, finding that the FAWN approach provides significantly lower total cost and power for a significant set of large, high-query-rate applications.

2 Why FAWN?

The FAWN approach to building *well-matched* cluster systems has the potential to achieve high performance and be fundamentally more power-efficient than conventional architectures for serving massive-scale I/O and data-intensive workloads. We measure system performance in queries per second and measure power-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 last 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. Techniques to mask the CPU-memory bottleneck come at the cost of energy efficiency. Branch prediction, speculative execution, and increasing the amount of on-chip caching all require additional processor die area; modern processors dedicate as much as half their die to L2/3 caches [15]. 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 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 100 million instructions per joule, assuming all cores are active and avoid stalls or mispredictions. Lower-frequency single-issue CPUs, in contrast, can provide over 1 billion instructions per joule—an order of magnitude more efficient while still running at $1/3$rd 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 [41], but modern CPUs already operate near minimum voltage at the highest frequencies. In addition, transistor leakage currents quickly become a dominant power cost as the frequency is reduced [7].

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 [2], requiring that all components be scaled back with demand. As a result, running a system at 20% of its capacity may still consume over 50% of its peak power [38]. Despite improved power scaling technology, systems remain most power-efficient when operating at peak power.

A promising path to energy proportionality is turning machines off entirely [6]. Unfortunately, these techniques do not apply well to our workloads: key-value systems must often meet service-level agreements for query response throughput and latency of hundreds of milliseconds; the inter-arrival time and latency bounds of the requests prevents shutting machines down (and taking many seconds to wake them up again) during low load [2].

Finally, energy proportionality alone is not a panacea: systems ideally should be both proportional *and* efficient at 100% load. In this paper, we show that there is significant room to improve energy efficiency, and the FAWN approach provides a simple way to do so.

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 data store (Section 3.3), and the FAWN-KV cluster key-value lookup system (Section 3.4.1), including caching, 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 data store 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 are organized into a ring using consistent hash-

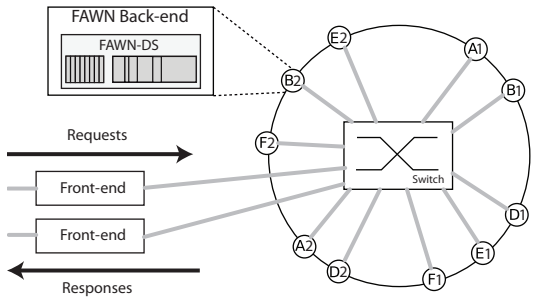

Figure 1: FAWN-KV Architecture.

ing. As in systems such as Chord [36], 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* ("vnodes"). Each physical node is thus responsible for V different (non-contiguous) key ranges. The data for each virtual node is stored using FAWN-DS.

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. These characteristics of flash underlie the design of the FAWN-KV system described throughout this section:

1. **Fast random reads**: ($\ll 1$ ms), up to 175 times faster than random reads on magnetic disk [24, 28].
2. **Low power consumption**: Flash devices consume less than one watt even under heavy load, whereas mechanical disks can consume over 10 W at load.
3. **Slow random writes**: Small writes on flash are very expensive. Updating a single page requires first erasing an entire erase block (128 KB–256 KB) of pages, and then writing the modified page in its entirety. As a result, updating a single byte of data is as expensive as writing an entire block of pages [26].

Modern devices improve random write performance using write buffering and preemptive block erasure. These techniques improve performance for short bursts of writes, but recent studies show that sustained random writes still perform poorly on these devices [28].

These performance problems motivate log-structured techniques for flash filesystems and data structures [25, 26, 17]. These same considerations inform the design of FAWN's node storage management system, described next.

3.3 The FAWN Data Store

FAWN-DS is a log-structured key-value store that runs on each virtual node. It acts to clients like a disk-based hash table that supports `store`, `lookup`, and `delete`.[1]

FAWN-DS is designed specifically to perform well on flash storage: all writes to the data store are sequential, and reads require a single random access. To provide this property, FAWN-DS maintains an in-memory hash table (Hash Index) that maps keys to an offset in the append-only Data Log on the flash (Figure 2a). This log-structured design is similar to several append-only filesystems [30, 13], which avoid random seeks on magnetic disks for writes.

Mapping a Key to a Value. Each FAWN-DS vnode uses its in-memory Hash Index to map 160-bit keys to a value in the Data Log. The maximum number of entries in the hash index limits the number of key-value pairs the node can store. Limited memory on the FAWN nodes is therefore precious. FAWN-DS conserves memory by allowing the Hash Index to infrequently be wrong, requiring another (relatively fast) random read from flash to find the correct entry.

To map a key using the Hash Index, FAWN-DS 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 index into the Hash Index, which contains 2^i hash buckets. Each bucket in the Hash Index is six bytes: a 15-bit key fragment, a valid bit, and a 4-byte pointer to the corresponding value stored in the Data Log.

After locating a bucket using the index bits, FAWN-DS compares the key fragment from the key to that stored in the hash bucket. If the key fragments do not match, FAWN-DS uses standard hash chaining to pick another bucket. If the fragments do match, FAWN-DS reads the data entry from the Data Log. This in-memory comparison helps avoid incurring multiple accesses to flash for a single key lookup without storing the full key in memory: There is a 1 in 2^{15} chance of a collision in which the bits match for different entries.

Each entry in the Data Log contains the full key, data length, and a variable-length data blob. Upon retrieving the entry, FAWN-DS compares the full 160-bit key to the one in the entry. If they do not match (a collision), FAWN-DS continues chaining through the Hash Index until it locates the correct value.

Reconstruction. Using this design, the Data Log contains all the information necessary to reconstruct the hash index from scratch. As an optimization, FAWN-DS checkpoints the hash index by periodically writing it to flash. The checkpoint includes the Hash Index plus a pointer to the last log entry. After a failure, FAWN-DS uses the checkpoint as a starting point to reconstruct the in-memory Hash Index quickly.

[1] We differentiate data store from database to emphasize that we do not provide a transactional or relational interface.

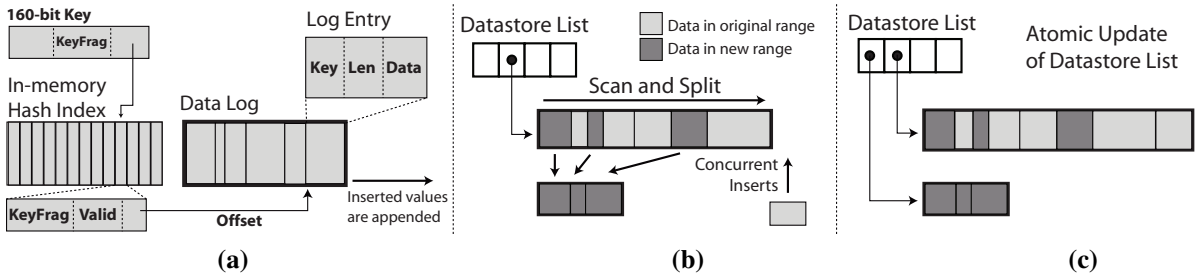

Figure 2: (a) FAWN-DS appends writes to the end of the data region. (b) Split requires a sequential scan of the data region, transferring out-of-range entries to the new store. (c) After scan is complete, the data store list is atomically updated to add the new store. Compaction of the original store will clean up out-of-range entries.

Vnodes and Semi-random Writes. Each virtual node in the FAWN-KV ring has a single FAWN-DS file that contains the data for that vnode's hash range. A physical node therefore has a separate data store file for each of its virtual nodes, and FAWN-DS appends new or updated data items to the appropriate data store. Sequentially appending to a small number of files is termed *semi-random writes*. Prior work by Nath and Gibbons observed that with many flash devices, these semi-random writes are nearly as fast as a single sequential append [25]. 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 below. We show in Section 4 that these semi-random writes perform sufficiently well.

3.3.1 Basic functions: Store, Lookup, Delete

`Store` appends an entry to the log, updates the corresponding hash table entry to point to this offset 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 by clearing the valid flag and writing 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.

3.3.2 Maintenance: Split, Merge, Compact

Inserting a new vnode into the ring causes one key range to split into two, with the new vnode taking responsibility for the first half of it. A vnode must therefore `Split` its data store into two files, one for each range. When a vnode departs, two adjacent key ranges must similarly be `Merged` into a single file. In addition, a vnode must periodically `Compact` a data store to clean up stale or orphaned entries created by `Split`, `Store`, and `Delete`.

The design of FAWN-DS ensures that these maintenance functions work well on flash, requiring only scans of one data store and sequential writes into another. We briefly discuss each operation in turn.

`Split` parses the data log entries sequentially, writing the entry in a new data store if its key falls in the new data store's range. `Merge` writes every log entry from one data store into the other data store; because the key ranges are independent, it does so as an append. Split and Merge propagate delete entries into the new data store.

`Compact` cleans up entries in a data store, similar to garbage collection in a log-structured filesystem. It skips entries that fall outside of the data store's key range, which may be left-over 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 data store.

3.3.3 Concurrent Maintenance and Operation

All FAWN-DS maintenance functions allow concurrent read and write access to the data store. `Stores` and `deletes` only modify hash table entries and write to the end of the log.

The 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 data store logfile. The maintenance operations therefore run until they reach the end of the log, and then briefly lock the data store, ensure that all values flushed to the old log have been processed, update the FAWN-DS data store list to point to the newly created log, and release the lock (Figure 2c). The lock must be held while writing in-flight appends to the log and updating data store list pointers, which typically takes 20–30 ms at the end of a `Split` or `Merge` (Section 4.3).

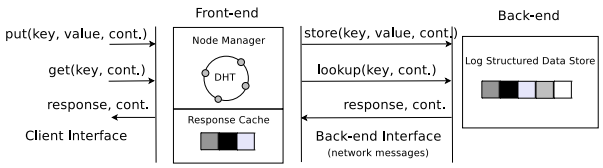

Figure 3: FAWN-KV Interfaces – Front-ends manage back-ends, route requests, and cache responses. Back-ends use FAWN-DS to store key-value pairs.

3.4 The FAWN Key-Value System

Figure 3 depicts FAWN-KV request processing. Client applications send requests to front-ends using a standard put/get interface. Front-ends send the request to the back-end vnode that owns the key space for the request. The back-end vnode satisfies the request using its FAWN-DS and replies to the front-ends.

In a basic FAWN implementation, clients link against a front-end library and send requests using a local API. Extending the front-end protocol over the network is straightforward—for example, we have developed a drop-in replacement for the `memcached` distributed memory cache that enables a collection of FAWN nodes to appear as a single, robust memcached server.

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 vnodes into a storage ring-structure using consistent hashing, similar to the Chord DHT [36]. FAWN-KV does not use DHT routing—instead, 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 vnode membership list and queries for a large contiguous chunk of the key space (in other words, the circular key space is divided into pie-wedges, each owned by a front-end). 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 [21], but we have not (yet) implemented this. Because there are typically 80 or more back-end nodes per front-end node, 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.[2]

[2]We do not use consistent hashing to determine this mapping because the number of front-end nodes may be too small to achieve good load balance.

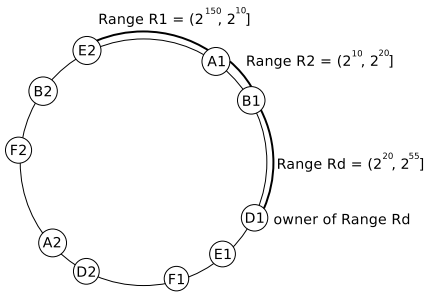

Figure 4: Consistent Hashing with 5 physical nodes and 2 virtual nodes each.

When a back-end node joins, it obtains the list of front-end IDs. Each of its virtual nodes uses this list to determine which front-end to contact to join the ring, one at a time. We chose this design so that the system was robust to front-end node failures: The back-end node identifier (and thus, what keys it are 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 vnodes, though vnodes may have to attach to a new front-end.

The FAWN-KV ring uses a 160-bit circular ID space for vnodes and keys. Vnode IDs are the hash of the concatenation of the (node IP address, vnode number). Each vnode *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 4 with five physical nodes, each of which has two vnodes. The physical node A appears as vnodes $A1$ and $A2$, each with its own 160-bit identifiers. Vnode $A1$ owns key range $R1$, vnode $B1$ owns range $R2$, and so on.

Consistent hashing provides incremental scalability without global data movement: adding a new vnode moves keys only at the successor of the vnode being added. We discuss below (Section 3.4.4) how FAWN-KV uses the single-pass, sequential `Split` and `Merge` operations in FAWN-DS to handle such changes efficiently.

3.4.2 Caching Prevents Wimpy Hot-Spots

FAWN-KV caches data using a two-level cache hierarchy. Back-end nodes implicitly cache recently accessed data in their filesystem buffer cache. While our current nodes (Section 4) can serve about 1700 queries/sec from flash, they serve 55,000 per second if the working set fits completely in buffer cache. The FAWN front-end maintains a small, high-speed query cache that helps reduce latency and ensures that if the load becomes skewed to only one or a few keys, those keys are served by a fast cache instead of all hitting a single back-end node.

3.4.3 Replication

FAWN-KV offers a configurable replication factor for fault tolerance. Items are stored at their successor in the

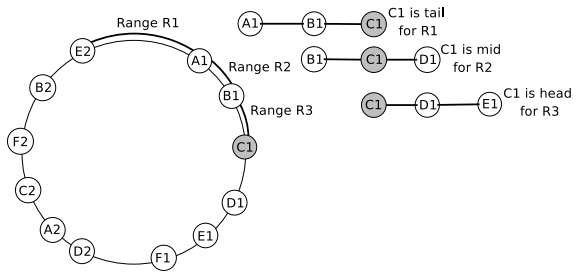

Figure 5: Overlapping Chains in the Ring – Each node in the consistent hashing ring is part of $R = 3$ chains.

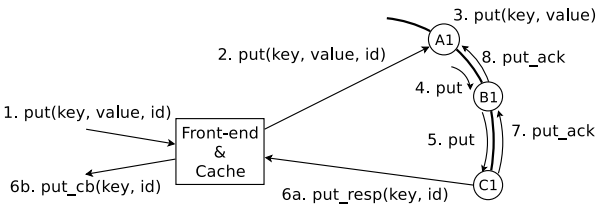

Figure 6: Lifecycle of a put with chain replication—puts go to the head and are propagated through the chain. Gets go directly to the tail.

ring space and at the $R-1$ following virtual nodes. FAWN uses chain replication [39] 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 the chain replication to the consistent hashing ring, each virtual node 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 5 depicts a ring with six physical nodes, where each has two virtual nodes ($V = 2$), using a replication factor of three. In this figure, node $C1$ is thus the tail for range $R1$, mid for range $R2$, and tail for range $R3$.

Figure 6 shows a put request for an item in range $R1$. The front-end routes the put to the key's successor, vnode $A1$, which is the head of the replica chain for this range. After storing the value in its data store, $A1$ forwards this request to $B1$, which similarly 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 data store, and increment it when they receive an acknowledgment. By using a purely log structured data store, chain replication with FAWN-KV becomes simply a process of streaming the growing datafile from node to node.

Gets proceed as in chain replication—the front-end directly routes the get to the tail of the chain for range $R1$, node $C1$, which responds to the request. Chain replication ensures that any update seen by the tail has also been applied by other replicas in the chain.

3.4.4 Joins and Leaves

When a node joins a FAWN-KV ring:

1. The new vnode causes one key range to split into two.
2. The new vnode must receive a copy of the R ranges of data it should now hold, one as a primary and $R-1$ as a replica.
3. The front-end must begin treating the new vnode as a head or tail for requests in the appropriate key ranges.
4. Vnodes down the chain may free space used by key ranges they are no longer responsible for.

The first step, key range splitting, occurs as described for FAWN-DS. While this operation actually occurs concurrently with the rest (the split and data transmission overlap), for clarity, we describe the rest of this process as if the split had already taken place.

After the key ranges have been split appropriately, the node must become a working member of R chains. For each of these chains, the node must receive a consistent copy of the data store file corresponding to the key range. The process below does so with minimal locking and ensures that if the node fails during the data copy operation, the existing replicas are unaffected. We illustrate this process in detail in Figure 7 where node $C1$ joins as a new middle replica for range $R2$.

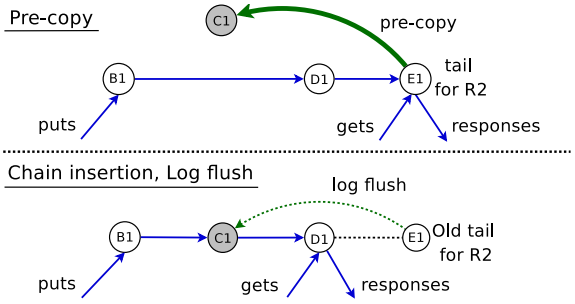

Figure 7: Phases of join protocol on node arrival.

Phase 1: data store pre-copy. Before any ring membership changes occur, the current tail for the range (vnode $E1$) begins sending the new node $C1$ a copy of the data store log file. This operation is the most time-consuming part of the join, potentially requiring hundreds of seconds. At the end of this phase, $C1$ has a copy of the log that contains all records committed to the tail.

Phase 2: Chain insertion, inconsistent. Next, as shown in Figure 7, the front-end tells the head node $B1$ to point to the new node $C1$ as its successor. $B1$ immediately begins streaming updates to $C1$, and $C1$ relays them properly to $D1$. $D1$ becomes the new tail of the chain.

At this point, $B1$ and $D1$ have correct, consistent views of the data store, but $C1$ may not: A small amount of time passed between the time that the pre-copy finished and when $C1$ was inserted into the chain.

To cope with this, $C1$ logs updates from $B1$ in a temporary data store, *not* the actual data store file for range $R2$, and does not update its in-memory hash table. During this phase, $C1$ is not yet a valid replica.

Phase 3: Log flush and play-forward. After it was inserted in the chain, $C1$ requests any entries that might have arrived in the time after it received the log copy and before it was inserted in the chain. The old tail $E1$ pushes these entries to $C1$, who adds them to the $R2$ data store. At the end of this process, $C1$ then merges (appends) the temporary log to the end of the $R2$ data store, updating its in-memory hash table as it does so. The node locks the temporary log at the end of the merge for 20–30ms to flush in-flight writes.

After phase 3, C1 is a functioning member of the chain with a fully consistent copy of the data store. This process occurs R times for the new virtual node—e.g., if $R = 3$, it must join as a new head, a new mid, and a new tail for one chain.

Joining as a head or tail: In contrast to joining as a middle node, joining as a head or tail must be coordinated with the front-end to properly direct requests to the vnode. The process for a new head is identical to that of a new mid. To join as a tail, a node joins before the current tail and replies to put requests. It does not serve get requests until it is consistent (end of phase 3)—instead, its predecessor serves as an interim tail for gets.

Leave: The effects of a voluntary or involuntary (failure-triggered) leave are similar to those of a join, except that the replicas must *merge* the key range that the node owned. As above, the nodes must add a new replica into each of the R chains that the departing node was a member of. This replica addition is simply a join by a new node, and is handled as above.

Failure Detection: Nodes are assumed to be fail-stop [35]. Each front-end exchanges heartbeat messages with its back-end vnodes every t_{hb} seconds. If a node misses $fd_{threshold}$ heartbeats, the front-end considers it to have failed and initiates the leave protocol. Because the Join protocol does not insert a node into the chain until the majority of log data has been transferred to it, a failure during join results only in an additional period of slowdown, not a loss of redundancy.

We leave certain aspects of failure detection for future work. In addition to assuming fail-stop, we assume that the dominant failure mode is a *node* failure or the failure of a link or switch, but our current design does not cope with a communication failure that prevents one node in a chain from communicating with the next while leaving each able to communicate with the front-ends. We plan to augment the heartbeat exchange to allow vnodes to report their neighbor connectivity.

4 Evaluation

We begin by characterizing the I/O performance of a wimpy node. From this baseline, we then evaluate how well FAWN-DS performs on this same node, finding that its performance is similar to the node's baseline I/O capability. To further illustrate the advantages of FAWN-DS's design, we compare its performance to an implementation using the general-purpose Berkeley DB, which is not optimized for flash writes.

After characterizing individual node performance, we then study a prototype FAWN-KV system running on a 21 node cluster. We evaluate its power efficiency, in queries per second per watt, and then measure the performance effects of node failures and arrivals. In the following section, we then compare FAWN to a more traditional datacenter architecture designed to store the same amount of data and meet the same query rates.

Evaluation Hardware: Our FAWN cluster has 21 back-end nodes built from commodity PCEngine Alix 3c2 devices, commonly used for thin-clients, kiosks, network firewalls, wireless routers, and other embedded applications. These devices have a single-core 500 MHz AMD Geode LX processor with 256 MB DDR SDRAM operating at 400 MHz, and 100 Mbit/s Ethernet. Each node contains one 4 GB Sandisk Extreme IV CompactFlash device. A node consumes 3 W when idle and a maximum of 6 W when deliberately using 100% CPU, network and flash. The nodes are connected to each other and to a 40 W Atom-based front-end node using two 16-port Netgear GS116 GigE Ethernet switches.

Evaluation Workload: FAWN-KV targets read-intensive, small object workloads for which key-value systems are often used. The exact object sizes are, of course, application dependent. In our evaluation, we show query performance for 256 byte and 1 KB 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 iozone [16] and Flexible I/O tester [1]. The flash

Seq. Read	*Rand Read*	*Seq. Write*	*Rand. Write*
28.5 MB/s	1872 QPS	24 MB/s	110 QPS

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

DS Size	*1 KB Rand Read* (in queries/sec)	*256 B Rand Read* (in queries/sec)
10 KB	50308	55947
125 MB	39261	46079
250 MB	6239	4427
500 MB	1975	2781
1 GB	1563	1916
2 GB	1406	1720
3.5 GB	1125	1697

Table 2: Random read performance of FAWN-DS.

is formatted with the ext2 filesystem and mounted with the `noatime` option to prevent random writes for file access [24]. These tests read and write 1 KB entries, the lowest record size available in iozone. The filesystem I/O performance is shown in Table 1.

4.1.1 FAWN-DS Single Node 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 1 KB and 256 byte random read queries/sec as a function of the DS size. If the data store fits in the buffer cache, the node serves 35–55 thousand queries per second. As the data store exceeds the 256 MB 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 iozone tests (each stored entry has a header). The resulting query throughput, however, remains high: tests using 1 KB values achieved 1,125 queries/sec compared to 1,872 queries/sec from the filesystem. Using the 256 byte entries that we focus on below achieved 1,697 queries/sec from a 3.5 GB datastore. By comparison, the raw filesystem achieved 1,972 random 256 byte reads per second using Flexible I/O.

Alix Node Performance			
Queries/sec	*Idle Power*	*Active Power*	*Queries/J*
1697	3 W	4 W	424

Bulk `store` speed: The log structure of FAWN-DS ensures that data insertion is entirely sequential. As a consequence, inserting two million entries of 1 KB each (2 GB total) into a *single* file sustains an insert rate of 23.2 MB/s (or nearly 24,000 entries per second), which is 96% of the raw speed that the flash can be written.

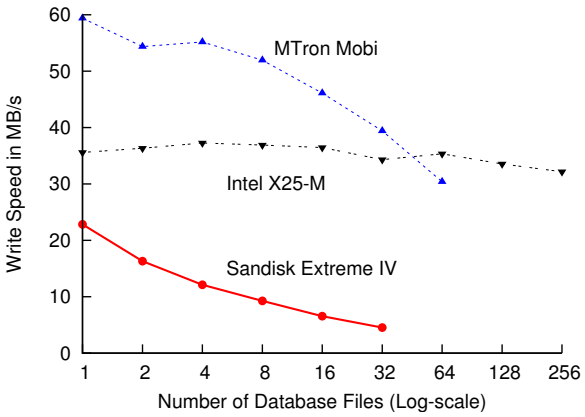

Figure 8: Sequentially writing to multiple FAWN-DS files results in semi-random writes.

`Put` Speed: In FAWN-KV, each FAWN node has $R * V$ FAWN-DS files: each virtual node is responsible for its own data range, plus the number of ranges it is a replica for. A physical 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. We therefore evaluate its performance using three flash-based storage devices.

Semi-random performance varies widely by device. Figure 8 shows the aggregate write performance obtained when inserting 2GB of data using three different flash drives, as the data is inserted into an increasing number of files. The relatively low-performance CompactFlash write speed slows with an increasing number of files. The 2008 Intel X25-M, which uses log-structured writing and preemptive block erasure, retains high performance with up to 256 concurrent semi-random writes for the 2 GB of data we inserted; the 2007 Mtron Mobi shows higher performance than the X25-M that drops somewhat as the number of files increases. The key take-away from this evaluation is that Flash devices are *capable* of handling the FAWN-DS write workload extremely well—but a system designer must exercise care in selecting devices that actually *do* so.

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 heavyweight transactions or rollback, and performs well versus other memory or disk-based databases. We configured BerkeleyDB using both its default settings and using the reference guide suggestions for Flash-based operation [3]. The best performance we achieved required 6 hours (B-

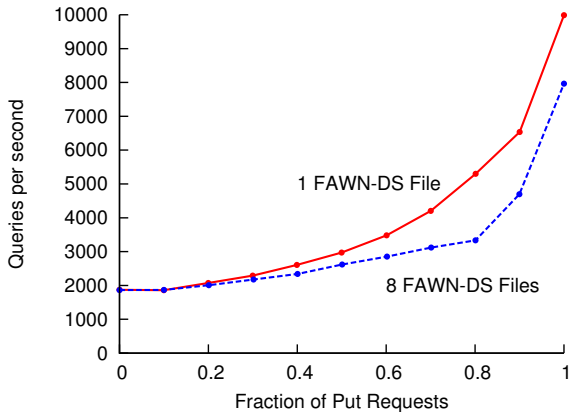

Figure 9: FAWN supports both read- and write-intensive workloads. Small writes are cheaper than random reads due to the FAWN-DS log structure.

Tree) and 27 hours (Hash) to insert seven million, 200 byte entries to create a 1.5GB database. This corresponds to an insert rate of 0.07 MB/s.

The problem was, of course, small writes: When the BDB store was larger than the available RAM on the nodes (< 256 MB), both the B-Tree and Hash implementations 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's characteristics. Unfortunately, we were not able to explore using BDB on a log-structured filesystem, which might be able to convert the random writes into sequential log writes. Existing Linux log-structured flash filesystems, such as JFFS2 [17], 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. It remains to be seen whether these approaches can speed up naive DB performance on flash, but the pure log structure of FAWN-DS remains necessary even if we could use a more conventional backend: 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 at least some writes. For example, Facebook's memcached workloads have a 1:6 ratio of application-level puts to gets [18]. We therefore measured the aggregate query rate as the fraction of puts ranged from 0 (all gets) to 1 (all puts) on a single node (Figure 9).

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

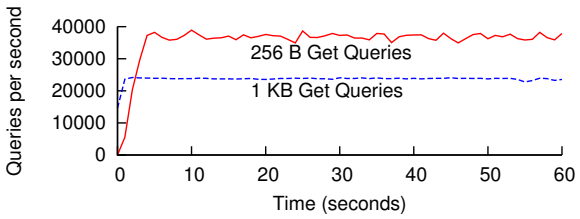

Figure 10: Query throughput on 21-node FAWN-KV system for 1 KB and 256 B entry sizes.

devices is worse than purely sequential writes, it still achieves higher throughput than pure random read performance.

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. A pure write workload would require frequent cleaning, reducing the throughput by half—still faster than the get rate. In the next section, we mostly evaluate read-intensive workloads because it represents the *worst-case* scenario for FAWN-KV.

4.2 FAWN-KV System Benchmarks

In this section, we evaluate the query rate and power draw of our 21-node FAWN-KV system.

System Throughput: To measure query throughput, we populated the KV cluster with 20 GB of values, and then measured the maximum rate at which the front-end received query responses for random keys. We disabled front-end caching for this experiment. Figure 10 shows that the cluster sustained roughly 36,000 256 byte gets per second (1,700 per second per node) and 24,000 1 KB gets per second (1,100 per second per node)—roughly 80% of the sustained rate that a single FAWN-DS could handle with local queries. The first reason for the difference was load balance: with random keys, some back-end nodes received more queries than others, slightly reducing system performance.[3] Second, this test involves network overhead and request marshaling and unmarshaling.

System Power Consumption: Using a WattsUp [40] 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 11 shows that, when idle, the cluster uses about 83 W, or 3 watts per node and 10 W per switch. During gets, power consumption increases to 99 W, and during insertions, power consump-

[3]This problem is fundamental to random load-balanced systems. A recent in-submission manuscript by Terrace and Freedman devised a mechanism for allowing queries to go to any node using chain replication; in future work, we plan to incorporate this to allow us to direct queries to the least-loaded replica, which has been shown to drastically improve load balance.

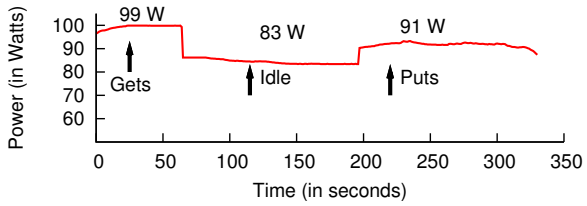

Figure 11: Power consumption of 21-node FAWN-KV system for 256 B values during Puts/Gets.

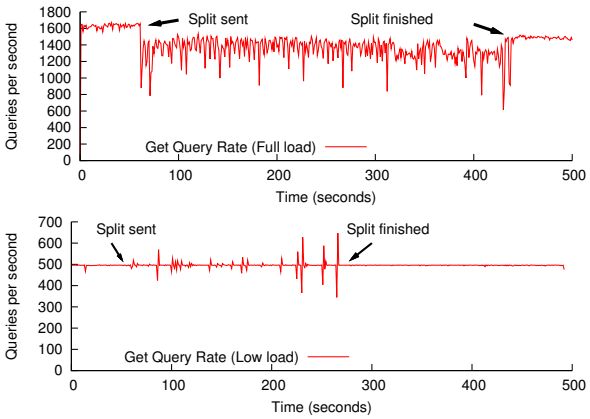

Figure 12: Get query rates during background split for high load (top) and low load (bottom).

tion is 91 W.[4] Peak get performance reaches about 36,000 256 B queries/sec for the cluster, so this system, excluding the frontend, provides 364 queries/joule.

The front-end has a 1 Gbit/s connection to the backend nodes, 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 backends (80 nodes * 1500 queries/sec/node * 1 KB / query = 937 Mbit/s). Our prototype front-end uses 40 W, which adds 0.5 W per node amortized over 80 nodes, providing 330 queries/joule for the entire system.

4.3 Impact of Ring Membership Changes

Node joins, leaves, or failures require existing nodes to merge, split, and transfer data while still handling puts and gets. We discuss the impact of a split on query throughput and query latency on a per-node basis; the impact of merge and compact is similar.

Query throughput during Split: In this test, we split a 512 MB FAWN-DS data store file while issuing random get requests to the node from the front-end. Figure 12(top)

[4]Flash writes and erase require higher currents and voltages than reads do, but the overall put power was lower because FAWN's log-structured writes enable efficient bulk writes to flash, so the system spends more time idle.

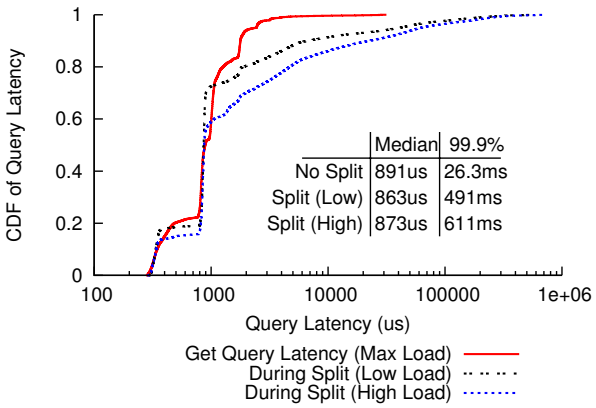

Figure 13: CDF of query latency under normal and split workloads.

shows query throughput during a split started at time t=50 when the request rate is high—about 1600 gets per second. During the split, throughput drops to 1000-1400 queries/sec. Because our implementation does not yet resize the hash index after a split, the post-split query rate is slightly reduced, at 1425 queries/sec, until the old data store is cleaned.

The split duration depends both on the size of the log and on the incoming query load. Without query traffic, the split runs at the read or write speed of the flash device (24 MB/s), depending on how much of the key range is written to a new file. Under heavy query load the split runs at only 1.45 MB/s—the random read queries greatly reduce the rate of the split reads and writes. In Figure 12(bottom) we repeat the same experiment but with a lower external query rate of 500 queries/sec. At low load, the split has very little impact on query throughput, and the split itself takes only half the time to complete.

Impact of Split on query latency: Figure 13 shows the distribution of query latency for three workloads: a pure get workload issuing gets at the maximum rate (Max Load), a 500 requests per second workload with a concurrent Split (Split-Low Load), and a high-rate request workload with a concurrent Split (Split-High Load).

In general, accesses that hit buffer cache are returned in 300μs including processing and network latency. When the accesses go to flash, the median response time is 800μs. Even during a split, the median response time remains under 1 ms.

Most key-value systems care about 99.9th percentile latency guarantees as well as fast average-case performance. During normal operation, request latency is very low: 99.9% of requests take under 26.3ms, and 90% take under 2ms. During a split with low external query load, the additional processing and locking extend 10% of requests above 10ms. Query latency increases briefly at the

end of a split when the datastore is locked to atomically add the new datastore. The lock duration is 20–30 ms on average, but can rise to 100 ms if the query load is high, increasing queuing delay for incoming requests during this period. The 99.9%-ile response time is 491 ms.

For a high-rate request workload, the incoming request rate is occasionally higher than can be serviced during the split. Incoming requests are buffered and experience additional queuing delay: the 99.9%-ile response time is 611 ms. Fortunately, these worst-case response times are still on the same order as those worst-case times seen in production key-value systems [9].

With larger values (1KB), query latency during Split increases further due to a lack of flash device parallelism—a large write to the device blocks concurrent independent reads, resulting in poor worst-case performance. Modern SSDs, in contrast, support and *require* request parallelism to achieve high flash drive performance [28]; a future switch to these devices could greatly reduce the effect of background operations on query latency.

5 Alternative Architectures

When is the FAWN approach likely to beat traditional architectures? We examine this question in two ways. First, we examine how much power can be saved on a conventional system using standard scaling techniques. Next, we compare the three-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.

5.1 Characterizing Conventional Nodes

We first examine a low-power, conventional desktop node configured to conserve power. The system uses an Intel quad-core Q6700 CPU with 2 GB DRAM, an Mtron Mobi SSD, and onboard gigabit Ethernet and graphics.

Power saving techniques: We configured the system to use DVFS with three p-states (2.67 GHz, 2.14 GHz, 1.60 GHz). To maximize idle time, we ran a tickless Linux kernel (version 2.6.27) and disabled non-system critical background processes. We enabled power-relevant BIOS settings including ultra-low fan speed and processor C1E support. Power consumption was 64 W when idle with only system critical background processes and 83-90 W with significant load.

Query throughput: Both raw (iozone) and FAWN-DS random reads achieved 5,800 queries/second, which is the limit of the flash device.

The resulting full-load query efficiency was 70 queries/Joule, compared to the 424 queries/Joule of a fully populated FAWN cluster. Even a three-node FAWN cluster that achieves roughly the same query throughput as the desktop, including the full power draw of an unpopulated 16-port gigabit Ethernet switch (10 W), achieved 240 queries per joule. As expected from the small idle-active power gap of the desktop (64 W idle, 83 W active), the system had little room for "scaling down"—the queries/Joule became drastically worse as the load decreased. The idle power of the desktop is dominated by fixed power costs, while half of the idle power consumption of the 3-node FAWN cluster comes from the idle (and under-populated) Ethernet switch.

System / Storage	QPS	Watts	Queries/sec / Watt
Embedded Systems			
Alix3c2 / Sandisk(CF)	1697	4	424
Soekris / Sandisk(CF)	334	3.75	89
Traditional Systems			
Desktop / Mobi(SSD)	5800	83	69.9
MacbookPro / HD	66	29	2.3
Desktop / HD	171	87	1.96

Table 3: Query performance and efficiency for different machine configurations.

Table 3 extends this comparison to clusters of several other systems.[5] As expected, systems with disks are limited by seek times: the desktop above serves only 171 queries per second, and so provides only 1.96 queries/joule—two orders of magnitude lower than a fully-populated FAWN. This performance is not too far off from what the disks themselves can do: they draw 10 W at load, providing only 17 queries/joule. Low-power laptops with magnetic disks fare little better. The desktop (above) with an SSD performs best of the alternative systems, but is still far from the performance of a FAWN cluster.

5.2 General Architectural Comparison

A general comparison requires looking not just at the queries per joule, but the total system cost. In this section, we examine the 3-year total cost of ownership (TCO), which we define as the sum of the capital cost and the 3-year power cost at 10 cents per kWh.

Because the FAWN systems we have built use several-year-old technology, we study a theoretical 2008 FAWN node using a low-power Intel Atom CPU; private communication with a large manufacturer indicates that such a node is feasible, consuming 6-8W and costing ∼$150 in volume. We in turn give the benefit of the doubt to the server systems we compare against—we assume a 1 TB disk exists that serves 300 queries/sec 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, SSDs, or DRAM.

[5]The Soekris is a five-year-old embedded communications board.

System	Cost	W	QPS	Queries/Joule	GB/Watt	TCO/GB	TCO/QPS
Traditionals:							
5-1 TB HD	$5K	250	1500	6	20	1.1	3.85
160 GB SSD	$8K	220	200K	909	0.7	53	0.04
64 GB DRAM	$3K	280	1M	3.5K	0.2	58	0.003
FAWNs:							
1 TB Disk	$350	15	250	16	66	0.4	1.56
32 GB SSD	$550	7	35K	5K	4.6	17.8	0.02
2 GB DRAM	$250	7	100K	14K	0.3	134	0.002

Table 4: Traditional and FAWN node statistics

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 tradeoff between per-node storage capacity, query rate, and power cost.

Table 4 shows these cost and performance statistics for several candidate systems. The "traditional" nodes use 200W servers that cost $3,000 each. *Traditional+Disk* pairs a single server with five 1 TB high-speed disks capable of 300 queries/sec, each disk consuming 10 W. *Traditional+SSD* uses two PCI-E Fusion-IO 80 GB Flash SSDs, each also consuming about 10 W (Cost: $3k). *Traditional+DRAM* uses eight 8 GB server-quality DRAM modules, each consuming 10 W. *FAWN+Disk* nodes use one 1 TB 7200 RPM disk: FAWN nodes have fewer connectors available on the board. *FAWN+SSD* uses one 32 GB Intel SATA Flash SSD, consuming 2 W ($400). *FAWN+DRAM* uses a single 2 GB, slower DRAM module, also consuming 2 W.

Figure 14 shows which base system has the lowest cost for a particular dataset size and query rate, with dataset sizes between 100 GB and 10 PB and query rates between 100 K and 1 billion per second. The dividing lines represent a boundary across which one system becomes more favorable than another.

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.

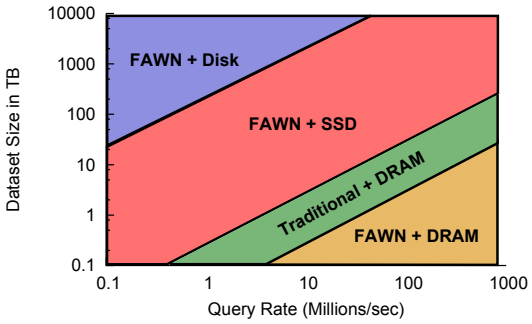

Figure 14: Solution space for lowest 3-year TCO as a function of dataset size and query rate.

Small datasets, high query rates: *FAWN+DRAM* costs the fewest dollars per queries/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 [4] by avoiding the memory wall. The FAWN nodes can only accept 2 GB 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 (64 GB vs 2 GB per node).

Middle range: *FAWN+SSD*s provide the best balance of storage capacity, query rate, and total cost. As SSD capacity improves, this combination is likely to continue expanding into the range served by *FAWN+Disk*; as SSD performance improves, so will it reach into DRAM territory. It is therefore conceivable that *FAWN+SSD* could become the dominant architecture for a wide range of random-access workloads.

Are traditional systems obsolete? We emphasize that this analysis applies only to small, random access workloads. 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 [12]—but they augment their wimpy processors with custom floating point units to do so. Finally, of course, our analysis assumes that cluster software developers can engineer away the human costs of management—an optimistic assumption for all architectures. We similarly discard issues such as ease of programming, though we ourselves selected an x86-based wimpy platform precisely for ease of development.

6 Related Work

FAWN follows in a long tradition of ensuring that systems are balanced in the presence of scaling challenges

and of designing systems to cope with the performance challenges imposed by hardware architectures.

System Architectures: JouleSort [32] is a recent energy-efficiency benchmark; its authors developed a SATA disk-based "balanced" system coupled with a low-power (34 W) CPU that significantly out-performed prior systems in terms of records sorted per joule. A major difference with our work is that the sort workload can be handled with large, bulk I/O reads using radix or merge sort. FAWN targets even more seek-intensive workloads for which even the efficient CPUs used for JouleSort are excessive, and for which disk is inadvisable.

The Gordon [5] hardware architecture pairs an array of flash chips and DRAM with low-power CPUs for low-power data intensive computing. A primary focus of their work is on developing a Flash Translation Layer suitable for pairing a single CPU with several raw flash chips. Simulations on general system traces indicate that this pairing can provide improved energy-efficiency. Our work leverages commodity embedded low-power CPUs and flash storage, enabling good performance on flash regardless of FTL implementation.

Considerable prior work has examined ways to tackle the "memory wall." The Intelligent RAM (IRAM) project combined CPUs and memory into a single unit, with a particular focus on energy efficiency [4]. An IRAM-based CPU could use a quarter of the power of a conventional system to serve the same workload, reducing total system energy consumption to 40%. FAWN takes a thematically similar view—placing smaller processors very near flash—but with a significantly different realization. Similar efforts, such as the Active Disk project [31], focused on harnessing computation close to disks. Schlosser et al. [34] proposed obtaining similar benefits from coupling MEMS with CPUs.

Databases and Flash: Much ongoing work is examining the use of flash in databases. Recent work concluded that NAND flash might be appropriate in "read-mostly, transaction-like workloads", but that flash was a poor fit for high-update databases [24]. This work, along with FlashDB [26], also noted the benefits of a log structure on flash; however, in their environments, using a log-structured approach slowed query performance by an unacceptable degree. In contrast, FAWN-KV's sacrifices range queries by providing only primary-key queries, which eliminates complex indexes: FAWN's separate data and index can therefore support log-structured access without reduced query performance. Indeed, with the log structure, FAWN's performance actually *increases* with a higher percentage of writes.

Filesystems for Flash: Several filesystems are specialized for use on flash. Most are partially log-structured [33], such as the popular JFFS2 (Journaling Flash File System) for Linux. Our observations about flash's performance characteristics follow a long line of research [10, 24, 43, 26, 28]. Past solutions to these problems include the eNVy filesystem's use of battery-backed SRAM to buffer copy-on-write log updates for high performance [42], followed closely by purely flash-based log-structured filesystems [20].

High-throughput storage and analysis: Recent work such as Hadoop or MapReduce [8] running on GFS [13] has examined techniques for scalable, high-throughput computing on massive datasets. More specialized examples include SQL-centric options such as the massively parallel data-mining appliances from Netezza [27]. As opposed to the random-access workloads we examine for FAWN-KV, these systems provide bulk throughput for massive datasets with low selectivity or where indexing in advance is difficult. We view these workloads as a promising next target for the FAWN approach.

Distributed Hash Tables: Related cluster and wide-area hash table-like services include Distributed data structure (DDS) [14], a persistent data management layer designed to simplify cluster-based Internet services. FAWN's major points of differences with DDS are a result of FAWN's hardware architecture, use of flash, and focus on power efficiency—in fact, the authors of DDS noted that a problem for future work was that "disk seeks become the overall bottleneck of the system" with large workloads, precisely the problem that FAWN-DS solves. These same differences apply to systems such as Dynamo [9] and Voldemort [29]. Systems such as Boxwood [22] focus on the higher level primitives necessary for managing storage clusters. Our focus was on the lower-layer architectural and data-storage functionality.

Sleeping Disks: A final set of research examines how and when to put disks to sleep; we believe that the FAWN approach compliments them well. Hibernator [44], for instance, focuses on large but low-rate OLTP database workloads (a few hundred queries/sec). Ganesh et al. proposed using a log-structured filesystem so that a striping system could perfectly predict which disks must be awake for writing [11]. Finally, Pergamum [37] used nodes much like our wimpy nodes to attach to spun-down disks for archival storage purposes, noting that the wimpy nodes consume much less power when asleep. The system achieved low power, though its throughput was limited by the wimpy nodes' Ethernet.

7 Conclusion

FAWN pairs low-power embedded nodes with flash storage to provide fast and energy efficient processing of random read-intensive workloads. Effectively harnessing these more efficient but memory and compute-limited nodes into a usable cluster requires a re-design of many of the lower-layer storage and replication mechanisms. In this paper, we have shown that doing so is both possible

and desirable. FAWN-KV begins with a log-structured per-node datastore to serialize writes and make them fast on flash. It then uses this log structure as the basis for chain replication between cluster nodes, providing reliability and strong consistency, while ensuring that all maintenance operations—including failure handling and node insertion—require only efficient bulk sequential reads and writes. By delivering over an order of magnitude more queries per Joule than conventional disk-based systems, the FAWN architecture demonstrates significant potential for many I/O intensive workloads.

References

[1] Flexible I/O Tester. `http://freshmeat.net/projects/fio/`.

[2] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[3] BerkeleyDB Reference Guide. Memory-only or Flash configurations. `http://www.oracle.com/technology/documentation/berkeley-db/db/ref/program/ram.html`.

[4] W. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of existing architectures in IRAM systems. In *Workshop on Mixing Logic and DRAM, 24th International Symposium on Computer Architecture*, June 1997.

[5] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proc. ASPLOS*, Mar. 2009.

[6] J. S. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[7] P. de Langen and B. Juurlink. Trade-offs between voltage scaling and processor shutdown for low-energy embedded multiprocessors. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2007.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX OSDI*, Dec. 2004.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, Oct. 2007.

[10] F. Douglis, F. Kaashoek, B. Marsh, R. Caceres, K. Li, and J. Tauber. Storage alternatives for mobile computers. In *Proc İst USENIX OSDI*, pages 25–37, Nov. 1994.

[11] L. Ganesh, H. Weatherspoon, M. Balakrishnan, and K. Birman. Optimizing power consumption in large scale storage systems. In *Proc. HotOS XI*, May 2007.

[12] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, et al. Overview of the Blue Gene/L system architecture. *IBM J. Res and Dev.*, 49(2/3), May 2005.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. SOSP*, Oct. 2003.

[14] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. 4th USENIX OSDI*, Nov. 2000.

[15] Intel. Penryn Press Release. `http://www.intel.com/pressroom/archive/releases/20070328fact.htm`.

[16] Iozone. Filesystem Benchmark. `http://www.iozone.org`.

[17] JFFS2. The Journaling Flash File System. `http://sources.redhat.com/jffs2/`.

[18] B. Johnson. Facebook, personal communication, Nov. 2008.

[19] R. H. Katz. Tech titans building boom. IEEE Spectrum, Feb. 2009.

[20] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proc. USENIX Annual Technical Conference*, Jan. 1995.

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16 (2):133–169, 1998. ISSN 0734-2071.

[22] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proc. 6th USENIX OSDI*, Dec. 2004.

[23] Memcached. A distributed memory object caching system. `http://www.danga.com/memcached/`.

[24] D. Myers. On the use of NAND flash memory in high-performance relational databases. M.S. Thesis, MIT, Feb. 2008.

[25] S. Nath and P. B. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage. In *Proc. VLDB*, Aug. 2008.

[26] S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *Proc. ACM/IEEE Intl. Conference on Information Processing in Sensor Networks*, Apr. 2007.

[27] Netezza. Business intelligence data warehouse appliance. `http://www.netezza.com/`, 2006.

[28] 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*, Mar. 2009.

[29] Project Voldemort. A distributed key-value storage system. `http://project-voldemort.com`.

[30] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, Jan. 2002.

[31] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001.

[32] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: A balanced energy-efficient benchmark. In *Proc. ACM SIGMOD*, June 2007.

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

[34] S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger. Filling the memory access gap: A case for on-chip magnetic storage. Technical Report CMU-CS-99-174, Carnegie Mellon University, Nov. 1999.

[35] F. B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984. ISSN 0734-2071.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, Aug. 2001.

[37] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proc. USENIX Conference on File and Storage Technologies*, Feb. 2008.

[38] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering energy proportionality with non energy-proportional systems – optimizing the ensemble. In *Proc. HotPower*, Dec. 2008.

[39] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. 6th USENIX OSDI*, Dec. 2004.

[40] WattsUp. .NET Power Meter. `http://wattsupmeters.com`.

[41] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc İst USENIX OSDI*, pages 13–23, Nov. 1994.

[42] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. ASPLOS*, Oct. 1994.

[43] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *Proc. FAST*, Dec. 2005.

[44] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proc. SOSP*, Oct. 2005.