To appear in *22nd ACM Symposium on Operating Systems Principles (SOSP'09)*
This version is reformatted from the official version that appears in the conference proceedings.

FAWN: A Fast Array of Wimpy Nodes

David G. Andersen[1], Jason Franklin[1], Michael Kaminsky[2],
Amar Phanishayee[1], Lawrence Tan[1], Vijay Vasudevan[1]

[1]*Carnegie Mellon University,* [2]*Intel Labs*

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 350 key-value *queries per Joule* of energy—two orders of magnitude more than a disk-based system.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Systems*; D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

Keywords

Design, Energy Efficiency, Performance, Measurement, Cluster Computing, Flash

SOSP'09, October 11-14, 2009, Big Sky, MT, USA.

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 [10]), LinkedIn (Voldemort [41]), and Facebook (`memcached` [33]).

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, 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-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 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 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 [25]. Future datacenters may require as much as 200 MW [25], 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?

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 either. 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 [10, 41, 33]. 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 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* [54] 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—for a single node failure and recovery, the affected key range 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 1300 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 Joule—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 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 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. Operating processors at higher frequency requires more energy, and techniques to mask the CPU-memory bottleneck come at the cost of energy efficiency. Branch prediction, speculative execution, out-of-order/superscalar 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 [21]. 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 approximately 100 million instructions per Joule, assuming all cores are active and avoid stalls or mispredictions. Lower-frequency in-order 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 [56], 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 [3], requiring that all components be scaled back with demand. As a result, running a modern, DVFS-enabled system at 20% of its capacity may still consume over 50% of its peak power [52]. 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 [7]. Unfortunately, these techniques do not apply well to FAWN-KV's target 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 [3].

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 datastore (Section 3.3), and the FAWN-KV cluster key-value lookup system (Section 3.4), 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 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 are organized into a ring using consistent hashing. As in systems such as Chord [48], 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 (non-contiguous) key ranges. The data associated with each virtual ID is stored on flash 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. Three 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 [35, 40].
2. **Efficient I/O**: Flash devices consume less than one Watt even under heavy load, whereas mechanical disks can consume over 10 W at load. Flash is over two orders of magnitude more efficient than mechanical disks in terms of queries/Joule.
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 block in its entirety. As a result, updating a single byte of data is as expensive as writing an entire block of pages [37].

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 [40].

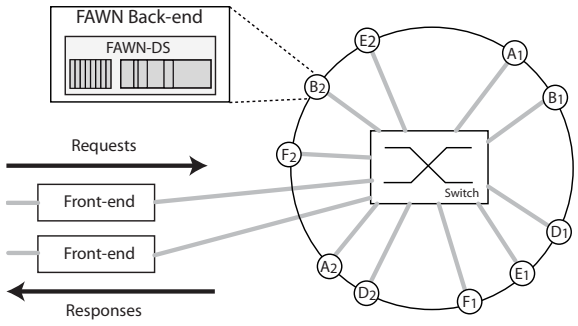

Figure 1: FAWN-KV Architecture.

These performance problems motivate log-structured techniques for flash filesystems and data structures [36, 37, 23]. 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. 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`.[1]

FAWN-DS is designed specifically 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 [42, 15], 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 six bytes of DRAM per key-value pair).

Figure 3 shows the pseudocode that implements this design for `Lookup`. 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 select a bucket from the Hash Index, which contains 2^i hash buckets. Each bucket is only six 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.

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

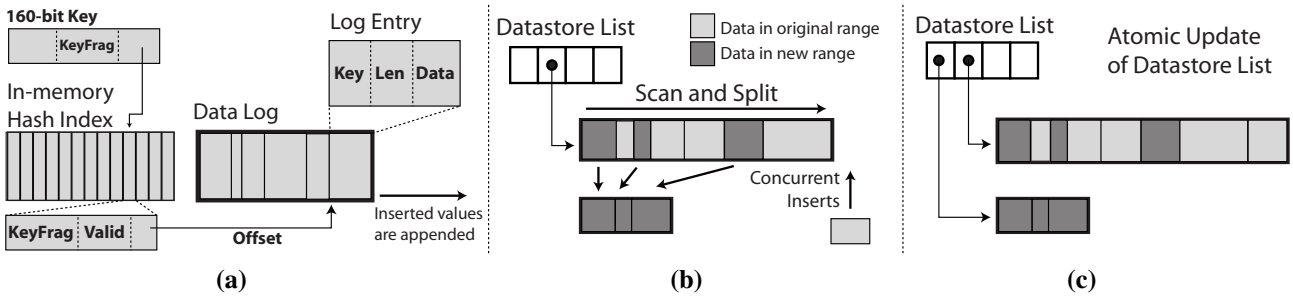

Figure 2: (a) FAWN-DS appends writes to the end of the Data Log. (b) Split requires a sequential scan of the data region, transferring out-of-range entries to the new store. (c) After scan is complete, the datastore list is atomically updated to add the new store. Compaction of the original store will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

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 size is between 256 B to 1 KB, and the nodes have 256 MB of DRAM and approximately 4 GB of flash storage. Because each node is responsible for V key ranges (each of which has its own datastore file), a single physical node can address 4 GB * V bytes of data. Expanding the in-memory storage to 7 bytes per entry would permit FAWN-DS to address 512 GB 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 target key-value workloads we study.

Reconstruction. Using this design, 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 quickly.

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*. 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 [36]. 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.

```
/* KEY = 0x93df7317294b99e3e049, 16 index bits */
INDEX = KEY & 0xffff; /* = 0xe049; */
KEYFRAG = (KEY >> 16) & 0x7fff; /* = 0x19e3; */
for i = 0 to NUM_HASHES do
  bucket = hash[i](INDEX);
  if bucket.valid && bucket.keyfrag==KEYFRAG &&
    readKey(bucket.offset)==KEY then
    return bucket;
  end if
  {Check next chain element...}
end for
return NOT_FOUND;
```

Figure 3: Pseudocode for hash bucket lookup in FAWN-DS.

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. 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`.

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

`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 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 datastore.

3.3.3 Concurrent Maintenance and Operation

All FAWN-DS maintenance functions allow concurrent read and write access to the datastore. `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 datastore logfile. The maintenance operations therefore 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). The lock must be held while writing in-flight appends to the log and updating datastore list pointers, which typically takes 20–30 ms at the end of a `Split` or `Merge` (Section 4.3).

3.4 The FAWN Key-Value System

Figure 4 depicts FAWN-KV request processing. Client applications send requests to front-ends using a standard put/get

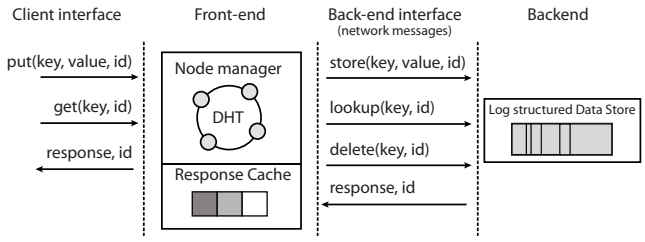

Figure 4: FAWN-KV Interfaces—Front-ends manage back-ends, route requests, and cache responses. Back-ends use FAWN-DS to store key-value pairs.

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.

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, enabling 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 *VIDs* into a storage ring-structure using consistent hashing, similar to the Chord DHT [48]. 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 *VID* 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 [27], 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.[2]

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 *VID* at a time. We

[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.

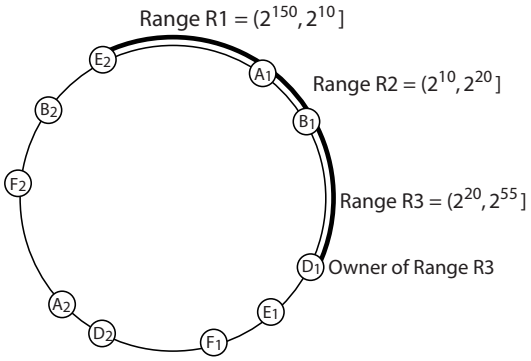

Figure 5: Consistent Hashing with 5 physical nodes and 2 virtual IDs each.

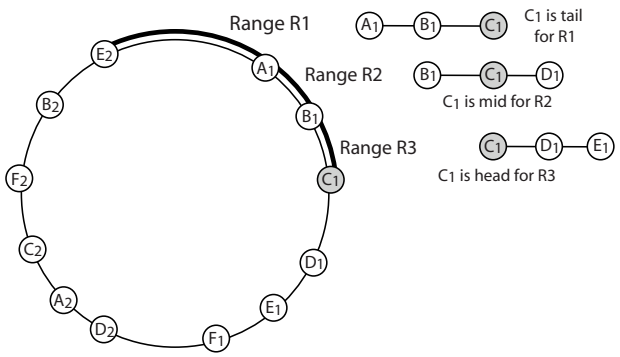

Figure 6: Overlapping Chains in the Ring – Each node in the consistent hashing ring is part of $R = 3$ chains.

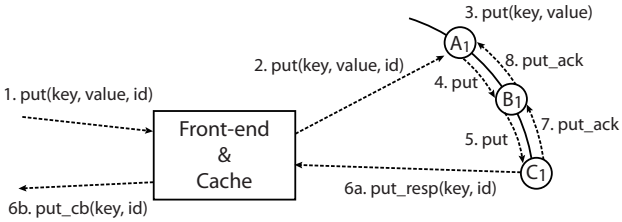

Figure 7: Lifecycle of a put with chain replication—puts go to the head and are propagated through the chain. Gets go directly to the tail.

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.

The FAWN-KV ring 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 5 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.

Consistent hashing provides incremental scalability without global data movement: adding a new *VID* moves keys only at the successor of the *VID* 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 read at about 1300 queries per second from flash, they can locally retrieve 85,000 queries 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 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 [54] 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 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 6 depicts a ring with six physical nodes, where each has two virtual IDs ($V = 2$), 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 7 shows a put request for an item in range $R1$. The front-end routes 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 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 datastore, and increment it when they receive an acknowledgment. By using a purely log structured datastore, chain replication with FAWN-KV becomes simply a process of streaming the growing datastore 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 virtual node causes one key range to split into two.
2. The new virtual node 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 virtual node as a head or tail for requests in the appropriate key ranges.
4. Virtual nodes 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 can occur concurrently with the rest (the split and data transmission can 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 datastore 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 8 where node $C1$ joins as a new middle replica for range $R2$.

Phase 1: Datastore pre-copy. Before any ring membership changes occur, the current tail for the range (VID $E1$) begins sending the new node $C1$ a copy of the datastore 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, log flush and play-forward. After $C1$'s pre-copy phase has completed, the front-end sends a *chain membership message* that flushes through the chain. This message plays two roles: first, it updates each node's neighbor state to add $C1$ to the chain; second, it ensures that any in-flight updates sent after the pre-copy phase completed are flushed to $C1$.

More specifically, this message propagates in-order through $B1$, $D1$, and $E1$, and is also sent to $C1$. Nodes $B1$, $C1$, and $D1$ update their neighbor list, and nodes in the current chain forward the message to their successor in the chain. Updates arriving at $B1$ after the reception of the chain membership message now begin streaming 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 datastore, 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.

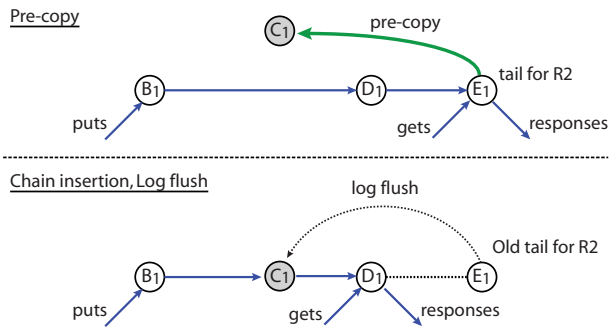

Figure 8: Phases of join protocol on node arrival.

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

All put requests sent to $B1$ after it received the chain membership message are replicated at $B1$, $C1$, and $D1$, and $D1$ forwards the chain membership message directly to $E1$. Thus, the receipt of the chain membership message at $E1$ signals that no further updates to this range will arrive at $E1$. The old tail $E1$ then pushes all entries that might have arrived in the time after $C1$ received the log copy and before $C1$ was inserted in the chain, and $C1$ adds these entries to the $R2$ datastore. At the end of this process, $E1$ sends the chain membership message back to $C1$, confirming that all in-flight entries have been flushed. $C1$ then merges (appends) the temporary log to the end of the $R2$ datastore, updating its in-memory hash table as it does so. The node briefly locks the temporary log at the end of the merge to flush these in-flight writes.

After phase 2, C1 is a functioning member of the chain with a fully consistent copy of the datastore. This process occurs R times for the new virtual ID—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 correct node. 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 2)—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 [47]. Each front-end exchanges heartbeat messages with its back-end nodes 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 slow-down, 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 nodes 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 energy 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 cluster 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, 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 27 W Intel 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.

Seq. Read	*Rand Read*	*Seq. Write*	*Rand. Write*
28.5 MB/s	1424 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	72352	85012
125 MB	51968	65412
250 MB	6824	5902
500 MB	2016	2449
1 GB	1595	1964
2 GB	1446	1613
3.5 GB	1150	1298

Table 2: Local random read performance of FAWN-DS.

4.1 Individual Node Performance

We benchmark the I/O capability of the FAWN nodes using iozone [22] and Flexible I/O tester [1]. The flash is formatted with the ext2 filesystem and mounted with the `noatime` option to prevent random writes for file access [35]. These tests read and write 1 KB entries, the lowest record size available in iozone. The filesystem I/O performance using a 3.5 GB 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 1 KB and 256 byte random read queries/sec as a function of the DS size. If the datastore fits in the buffer cache, the node locally retrieves 50–85 thousand queries per second. As the datastore 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 reading a 3.5 GB datastore using 1 KB values achieved 1,150 queries/sec compared to 1,424 queries/sec from the filesystem. Using the 256 byte entries that we focus on below achieved 1,298 queries/sec from a 3.5 GB datastore. By comparison, the raw filesystem achieved 1,454 random 256 byte reads per second using Flexible I/O.

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* FAWN-DS log sustains an insert rate of 23.2 MB/s

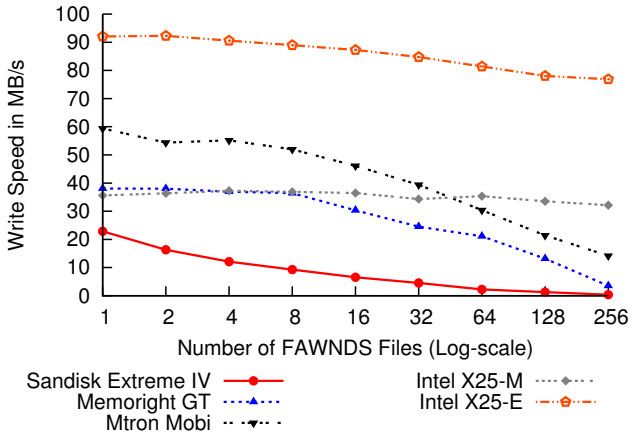

Figure 9: Sequentially writing to multiple FAWN-DS files results in semi-random writes.

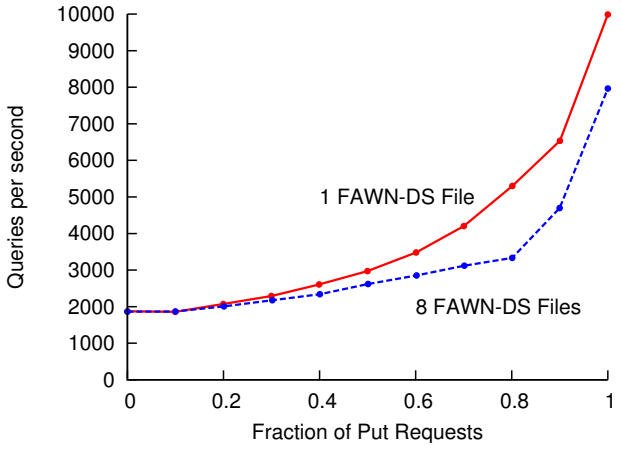

Figure 10: FAWN supports both read- and write-intensive workloads. Small writes are cheaper than random reads due to the FAWN-DS log structure.

(or nearly 24,000 entries per second), which is 96% of the raw speed that the flash can be written through the filesystem.

`Put` Speed: In FAWN-KV, each FAWN 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. We therefore evaluate its performance using five flash-based storage devices.

Semi-random performance varies widely by device. Figure 9 shows the aggregate write performance obtained when inserting 2GB of data into FAWN-DS using five different flash drives as the data is inserted into an increasing number of datastore files. All SATA-based flash drives measured below use an Intel Atom-based chipset because the Alix3c2 lacks a SATA port. The relatively low-performance CompactFlash write speed slows with an increasing number of files. The 2008 Intel X25-M and X25-E, which use log-structured writing and preemptive block erasure, retain high performance with up to 256 concurrent semi-random writes for the 2 GB of data we inserted; both the Mtron Mobi and Memoright GT drop in performance 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 heavy-weight 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 [4]. The best performance we achieved required 6 hours (B-Tree) and 27 hours (Hash) to insert seven million, 200 byte entries to create a 1.5 GB 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. We evaluated BDB on top of NILFS2 [39], 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 200 MB worth of 256 B key-value pairs generated 3.5 GB of metadata. Other existing Linux log-structured flash filesystems, such as JFFS2 [23], 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 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

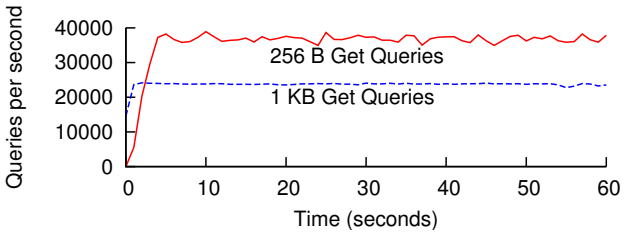

Figure 11: Query throughput on 21-node FAWN-KV system for 1 KB and 256 B entry sizes.

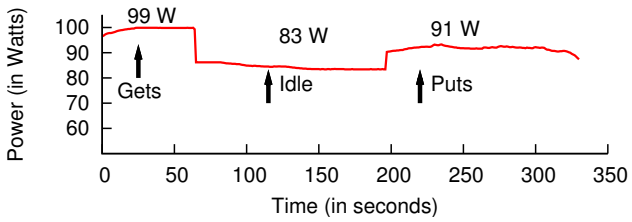

Figure 12: Power consumption of 21-node FAWN-KV system for 256 B values during Puts/Gets.

of application-level puts to gets [24]. 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 10).

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/sec 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

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 11 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). A single node serving a 512 MB 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 local queries. The primary reason for the difference is the addition of network overhead and request marshaling and unmarshaling. Another reason for difference is load balance: with random key distribution, some back-end nodes receive more queries than others, slightly reducing system performance.[3]

System Power Consumption: Using a WattsUp [55] 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 12 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 consumption is 91 W.[4] Peak get performance reaches about 36,000 256 B queries/sec for the cluster serving the 20 GB dataset, so this system, excluding the front-end, 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 27 W, which adds nearly 0.5 W per node amortized over 80 nodes, providing 330 queries/Joule for the entire system.

Network switches currently account for 20% of the power used by the entire system. Our current cluster size affords the use of a flat network hierarchy, but providing full bisection bandwidth for a large cluster would require many more network switches, increasing the ratio of network power to FAWN node power. Scaling networks to support large deployments is a problem that affects today's clusters and remains an active area of research [2, 18, 16, 19]. While improving the network energy consumption of large FAWN clusters is a topic of ongoing work, we note that recent fat-tree network topology designs using many small commodity, low-power switches [2] would impose only a fixed per-node network power overhead. Should the application design permit, sacrificing full bisection bandwidth can trade reduced communication flexibility for improved network energy efficiency.

[3] This problem is fundamental to random load-balanced systems. Terrace and Freedman [51] recently 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.

[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.

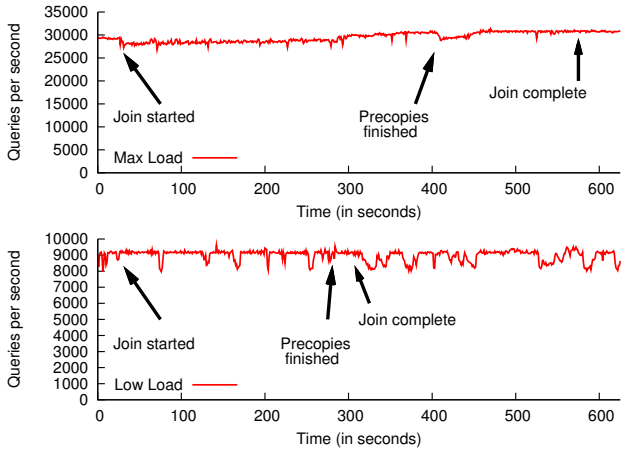

Figure 13: Get query rates during node join for max load (top) and low load (bottom).

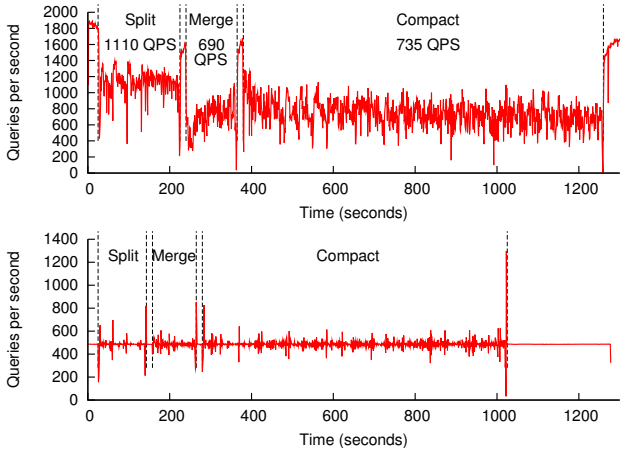

Figure 14: Get query rates during background operations for high (top) and low (bottom) external query loads.

4.3 Impact of Ring Membership Changes

Node joins, leaves, or failures require existing nodes to split merge, and transfer data while still handling puts and gets. In this section we evaluate the impact of node joins on system query throughput and the impact of maintenance operations such as local splits and compaction on single node query throughput and latency.

Query Throughput During Node Join: In this test, we start a 20-node FAWN-KV cluster populated with 10 GB of key-value pairs and begin issuing get requests uniformly at random to the entire key space. At t=25, we add a node to the ring and continue to issue get requests to the entire cluster. For this experiment, we set $R = 3$ and $V = 1$. Figure 13 shows the resulting cluster query throughput during a node join.

The joining node requests pre-copies for $R = 3$ ranges, one range for which it is the tail and two ranges as the head and mid. The three nodes that pre-copy their datastores to the joining node experience a one-third reduction in external query throughput, serving about 1,000 queries/sec. Pre-copying data does not cause significant I/O interference with external requests for data—the pre-copy operation requires only a sequential read of the datastore and bulk sends over the network. The lack of seek penalties for concurrent access on flash together with the availability of spare network capacity results in only a small drop in performance during pre-copying. The other 17 nodes in our cluster are not affected by this join operation and serve queries at their normal rate. The join operation completes long after pre-copies finished in this experiment due to the high external query load, and query throughput returns back to the maximum rate.

The experiment above stresses the cluster by issuing requests at the maximum rate the cluster can handle. But most systems offer performance guarantees only for loads below maximum capacity. We run the same experiment above but with an external query load at about 30% of the maximum supported query rate. The three nodes sending pre-copies have enough spare resources available to perform their pre-copy without affecting their ability to serve external queries, so the system's throughput does not drop when the new node is introduced. The join completes shortly after the pre-copies finishes.

Query Throughput During Maintenance Operations: Maintenance operations perform sequential reads of one file and sequential writes into another. In the node join experiment above, we deferred performing the local split/merge operations until after the node join completed to minimize the performance impact during the node join.

Figure 14(top) shows the impact of split, merge, and compaction on external get queries sent at high load to the 512 MB datastore. In this experiment, the key range is initially split unevenly: 25% of the original key space is split into a second FAWN-DS datastore. As a result, the split operation only writes 25% of its records into the second datastore. Merging the two datastores back into one is more "intense" than a split because the merge requires a read and write of nearly every record in the datastore being merged rather than just a fraction of the records. Consequently, the FAWN-DS file with fewer records should always be merged into the larger store to minimize the completion time of the merge operation.

Compaction has a query impact between both split and merge—compaction must write most of the entries in the log, except for out-of-range, deleted, or orphaned entries. However, because it must read and write every valid record in the datastore, the length of the operation is typically longer than either split and merge.

Figure 14(bottom) shows the same experiment with a query rate set at 30% of the maximum supported, showing that the impact of maintenance operations on query rate is minimal when the incoming rate is below half of the node's maximum query capacity.

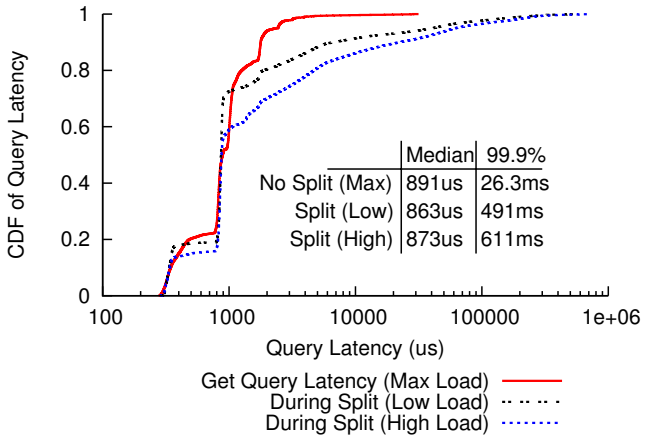

Figure 15: Query latency CDF for normal and split workloads.

Impact of Split on Query Latency: Figure 15 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 1500 requests per second workload with a Split (Split-High Load).

In general, accesses that hit buffer cache are returned in 300 μs including processing and network latency. When the accesses go to flash, the median response time is 800 μs. Even during a split, the median response time remains under 1 ms. The median latency increases with load, so the max load, get-only workload has a slightly higher median latency than the lower load splits.

Many 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 26.3 ms, and 90% take under 2 ms. During a split with low external query load, the additional processing and locking extend 10% of requests above 10 ms. 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 resulting 99.9%-ile response time during the low-activity split 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 [10].

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 [40]; a future switch to these devices could greatly reduce the effect of background operations on query latency.

We also measured the latency of put requests during normal operation. With R=1, median put latency was about 500μs, with 99.9%ile latency extending to 24.5 ms. With R=3, put requests in chain replication are expected to incur additional latency as the requests get routed down the chain. Median latency increased by roughly three times to 1.58 ms, with 99.9%ile latency increasing only to 30 ms.[5]

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: Raw (iozone) random reads achieved 4,771 (256 B) queries/sec and FAWN-DS achieved 4,289 queries/second. The resulting full-load query efficiency was 52 queries/Joule, compared to the 346 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/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 consump-

[5] When the workload consisted of a mixture of puts and gets, 99.9%ile latency increased significantly—our naive implementation used a single queue for all requests, so puts propagating between neighbors would often get queued behind a large set of external get requests, further increasing latency. Using separate queues for external messages and neighbor messages would reduce this worst-case latency.

System / Storage	QPS	Watts	Queries / Joule
Embedded Systems			
Alix3c2 / Sandisk(CF)	1298	3.75	346
Soekris / Sandisk(CF)	334	3.75	89
Traditional Systems			
Desktop / Mobi(SSD)	4289	83	51.7
MacbookPro / HD	66	29	2.3
Desktop / HD	171	87	1.96

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

tion of the 3-node FAWN cluster comes from the idle (and under-populated) Ethernet switch.

Table 3 extends this comparison to clusters of several other systems.[6] 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 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 efficiency 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 2009 FAWN node using a low-power CPU that consumes 10–20 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/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.

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:

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

System	Cost	W	QPS	Queries / Joule	GB / Watt	TCO / GB	TCO / QPS
Traditionals:							
5-2TB HD	$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
FAWNs:							
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

Table 4: Traditional and FAWN node statistics

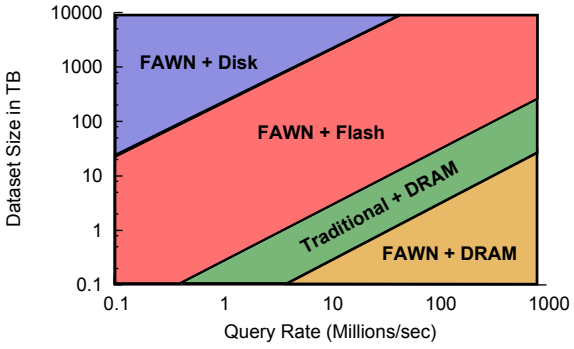

Figure 16: Solution space for lowest 3-year TCO as a function of dataset size and query rate.

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 4 shows these cost and performance statistics for several candidate systems. The "traditional" nodes use 200W servers that cost $1,000 each. *Traditional+Disk* pairs a single server with five 5 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 2 TB 7200 RPM disk: FAWN nodes have fewer connectors available on the board. *FAWN+SSD* uses one 32 GB Intel SATA Flash SSD capable of 35,000 random reads/sec [40] and consuming 2 W ($400). *FAWN+DRAM* uses a single 2 GB, slower DRAM module, also consuming 2 W.

Figure 16 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.

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 [5] by avoiding the memory wall. We assume 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 [14]—but they augment their wimpy processors with custom floating point units to do so.

Our definition of "total cost of ownership" also 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, 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 [44] 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.

More recently, several projects have begun using low-power processors for datacenter workloads to reduce energy consumption [6, 34, 11, 50, 20, 30]. The Gordon [6] hardware architecture argues for pairing 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 for cluster key-value applications, enabling good performance on flash regardless of FTL implementation. CEMS [20], AmdahlBlades [50], and Microblades [30] also leverage low-cost, low-power commodity components as a building block for datacenter systems, similarly arguing that this architecture can provide the highest work done per dollar and work done per joule. Microsoft has recently begun exploring the use of a large cluster of low-power systems called Marlowe [34]. This work focuses on taking advantage of the very low-power sleep states provided by this chipset (between 2–4 W) to turn off machines and migrate workloads during idle periods and low utilization, initially targeting the Hotmail service. We believe these advantages would also translate well to FAWN, where a lull in the use of a FAWN cluster would provide the opportunity to significantly reduce average energy consumption in addition to the already-reduced peak energy consumption that FAWN provides. Dell recently designed and has begun shipping VIA Nano-based servers consuming 20–30 W each for large webhosting services [11].

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 [5]. 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 [43], focused on harnessing computation close to disks. Schlosser et al. proposed obtaining similar benefits from coupling MEMS with CPUs [46].

Databases and Flash: Much ongoing work is examining the use of flash in databases, examining how database data structures and algorithms can be modified to account for flash storage strengths and weaknesses [53, 28, 35, 37, 29]. 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 [35]. This work, along with

FlashDB [37] and FD-Trees [29], 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. Prior work in sensor networks [8, 32] has employed flash in resource-constrained sensor applications to provide energy-efficient filesystems and single node object stores. In contrast to the above work, FAWN-KV 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 moderate percentage of writes. FAWN-KV also applies log-structured data organization to speed maintenance and failover operations in a clustered, datacenter environment.

Filesystems for Flash: Several filesystems are specialized for use on flash. Most are partially log-structured [45], such as the popular JFFS2 (Journaling Flash File System) for Linux. Our observations about flash's performance characteristics follow a long line of research [12, 35, 58, 37, 40]. 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 [57], followed closely by purely flash-based log-structured filesystems [26].

High-throughput Storage and Analysis: Recent work such as Hadoop or MapReduce [9] running on GFS [15] 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 [38]. 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) [17], 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 energy 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 [10] and Voldemort [41]. Systems such as Boxwood [31] 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 [59], 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 [13]. Finally, Pergamum [49] 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. Our 4-year-old FAWN nodes delivered over an order of magnitude more queries per Joule than conventional disk-based systems, and our preliminary experience using Intel Atom-based systems paired with SATA-based Flash drives shows that they can provide over 1000 queries/Joule, demonstrating that the FAWN architecture has significant potential for many I/O-intensive workloads.

Acknowledgments

This work was supported in part by gifts from Network Appliance, Google, and Intel Corporation, and by grant CNS-0619525 from the National Science Foundation. Jason Franklin is supported in part by an NSF Graduate Research Fellowship. Amar Phanishayee was supported by an IBM Fellowship. Vijay Vasudevan is supported by a fellowship from APC by Schneider Electric. We extend our thanks to the excellent feedback from our OSDI and SOSP reviewers, Vyas Sekar, Mehul Shah, and to Lorenzo Alvisi for shepherding the work for SOSP. Iulian Moraru provided both feedback and extensive performance tuning assistance on the wimpy nodes.

References

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

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity, data center network architecture. In *Proc. ACM SIGCOMM*, Aug. 2008.

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

[4] BerkeleyDB Reference Guide. Memory-only or Flash configurations. `http://www.oracle.com/technology/`

documentation/berkeley-db/db/ref/program/ram.html.

[5] 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.

[6] A. M. Caulfield, L. M. Grupp, and S. Swanson. 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 (ASPLOS09)*, Mar. 2009.

[7] 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.

[8] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004.

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

[10] 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. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[11] Dell XS11-VX8. Dell fortuna. "http://www1.euro.dell.com/content/topics/topic.aspx/emea/corporate/pressoffice/2009/uk/en/2009_05_20_brk_000", 2009.

[12] F. Douglis, F. Kaashoek, B. Marsh, R. Caceres, K. Li, and J. Tauber. Storage alternatives for mobile computers. In *Proc. 1st USENIX OSDI*, pages 25–37, Nov. 1994.

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

[14] 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.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[16] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proc. ACM SIGCOMM*, Aug. 2009.

[17] 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.

[18] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A scalable and fault-tolerant network structure for data centers. In *Proc. ACM SIGCOMM*, Aug. 2008.

[19] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *Proc. ACM SIGCOMM*, Aug. 2009.

[20] J. Hamilton. Cooperative expendable micro-slice servers (CEMS): Low cost, low power servers for Internet scale services. http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_CEMS.pdf, 2009.

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

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

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

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

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

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

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

[28] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In *Proc. ACM SIGMOD*, June 2008.

[29] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proceedings of 25th International Conference on Data Engineering*, Mar. 2009.

[30] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *International Symposium on Computer Architecture (ISCA)*, June 2008.

[31] 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.

[32] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Oct. 2006.

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

[34] Microsoft Marlowe. Peering into future of cloud computing. http://research.microsoft.com/en-us/news/features/ccf-022409.aspx, 2009.

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

[36] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. In *Proc. VLDB*, Aug. 2008.

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

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

[39] nilfs. Continuous snapshotting filesystem for Linux. http://www.nilfs.org.

[40] 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.

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

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] A. Szalay, G. Bell, A. Terzis, A. White, and J. Vandenberg. Low power Amdahl blades for data intensive computing, 2009.

[51] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proc. USENIX Annual Technical Conference*, June 2009.

[52] 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.

[53] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *Proc. ACM SIGMOD*, June 2009.

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

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

[56] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. 1st USENIX OSDI*, pages 13–23, Nov. 1994.

[57] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. 6th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1994.

[58] 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. 4th USENIX Conference on File and Storage Technologies*, Dec. 2005.

[59] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.