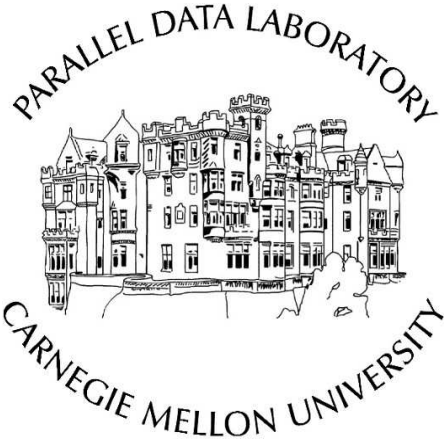

FAWN: A Fast Array of Wimpy Nodes

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

CMU-PDL-08-108
May 2008

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

This paper introduces the FAWN—Fast Array of Wimpy Nodes—cluster architecture for providing fast, scalable, and power-efficient key-value storage. A FAWN links together a large number of tiny nodes built using embedded processors and small amounts (2–16GB) of flash memory into an ensemble capable of handling 700 queries per second per node, while consuming fewer than 6 watts of power per node. We have designed and implemented a clustered key-value storage system, FAWN-DHT, that runs atop these node. Nodes in FAWN-DHT use a specialized log-like back-end hash-based database to ensure that the system can absorb the large write workload imposed by frequent node arrivals and departures. FAWN uses a two-level cache hierarchy to ensure that imbalanced workloads cannot create hot-spots on one or a few wimpy nodes that impair the system's ability to service queries at its guaranteed rate.

Our evaluation of a small-scale FAWN cluster and several candidate FAWN node systems suggest that FAWN can be a practical approach to building large-scale storage for seek-intensive workloads. Our further analysis indicates that a FAWN cluster is cost-competitive with other approaches (e.g., DRAM, multitudes of magnetic disks, solid-state disk) to providing high query rates, while consuming 3-10x less power.

Acknowledgements: We thank the members and companies of the CyLab Corporate Partners and the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, Network Appliance, Oracle, Seagate, Symantec, and VMware) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grant CNS-0619525, and by the Army Research Office, under agreement number DAAD19–02–1–0389.

Keywords: cluster, flash, databases, performance, power, efficiency

1 Introduction

This paper presents the design, implementation, and evaluation of a new scalable cluster key/value storage architecture targeting two important trends: the need for power-efficiency and the need to provide tight performance guarantees across widely-ranging workload characteristics. In this work, we focus on meeting these needs for the specific, but important, class of *primary-key, seek-bound* applications that perform a large number of retrievals of small data objects. A few examples of this class of applications include serving large root DNS zones or managing the catalog for large e-commerce sites.

Power is becoming an increasingly large financial and scaling burden for computing and society. The costs of running large data-centers are becoming dominated by power and cooling to the degree that Google, Microsoft, and others have started building new datacenters close to major hydroelectric power sources [12] or in areas such as Iceland with free environmental cooling and ready access to geothermal energy. On a smaller scale, power and cooling are serious impediments to the achievable density in datacenters [23] and large systems, and heat is one of the major scaling challenges for individual processors. Increasing demand for and techniques to support consolidation make these barriers even more severe [13, 4, 2].

Primary-key, read-mostly workloads with small objects are growing in importance with existing and emerging Internet applications. Social networking and blogging sites such as Livejournal, for instance, have already been forced to create cluster-based memory caches such as `memcached` [10] to handle a huge number of requests per second for objects on the order of tens to thousands of bytes. The same challenges are faced by e-commerce sites such as Amazon.com, which have catalogs of hundreds of millions or more objects [8], mostly accessed by a unique object ID. The `.com` DNS zone faces similar challenges of primary-key lookups in a database of hundreds of millions of DNS entries. In this work, we turn a particular eye towards the *queries per joule* (equivalently, the queries/second per watt) that different system designs can achieve in serving requests for these small data objects.

Unfortunately, these workloads are poorly suited to conventional disk-based architectures. Disk seek speeds have long been an impediment to such applications, and the gap between seek times and other computer speeds grows worse each year. While intelligent use of caches and clustering can speed access, many of these applications exist in an environment where accesses to "rare" items are likely, either because of user access patterns, adversarial access, or a stringent need to meet response time guarantees regardless of workload [8]. As we show in Section 5, storing these workloads entirely in DRAM is both costly and power-intensive, and often unnecessary to meet the performance requirements of the applications.

This paper introduces a new architecture targeted to serving these workloads quickly and efficiently. FAWN—a Fast Array of Wimpy Nodes—offers a superior alternative that is cost-competitive and vastly more power-efficient than using either magnetic disk or storing terabytes of data in costly DRAM. The goal of this work is to create a cluster architecture where the available computational power, memory, and IO are *well-matched*. To illustrate this point, consider an example of a poorly-matched system: using a modern, dual-processor quad-core system (2x80W for the CPUs alone) to serve a seek-bound workload from a fast hard drive. Such a system might satisfy 200 queries per second and would consume 300 watts, achieving a terrible $\frac{2}{3}$ query per joule. As we show in Section 5, even less obviously-poor architectures still provide poor power efficiency.

FAWN takes a very different approach to building a cluster, using a large number of tiny nodes (e.g., based on 800Mhz XScale chips with 256MB RAM) coupled to a small amount of Flash memory (2-16GB). The key contributions of this work center around the architecture and the techniques needed to make use of this architecture:

A well-matched system architecture for seek-bound workloads. The FAWN nodes consume little power while providing a consistent and high query rate.

The design of the FAWN-DHT primary-key distributed storage system atop the FAWN nodes. The FAWN nodes have two features that introduce challenges beyond those in more conventional distributed hash tables such as Amazon's Dynamo [8] or Chord [29]: The nodes are relatively "crippled" by today's standards, and they use flash memory, which has performance peculiarities somewhere between disk and RAM—Flash is particularly slow with small random write workloads. We present the design features of this system that enable it to run efficiently on a large network of wimpy nodes. These techniques center around ensuring fast bulk-loading of data when nodes join or leave the system and on ensuring that writes to flash are sequential and large, both of which prove necessary for obtaining high performance.

A two-level cache hierarchy to prevent "hot spots" on the wimpy nodes. While FAWN nodes can individually maintain a query rate higher than that of a single magnetic disk, no single node is capable of handling nearly the aggregate rate required of the cluster. In Section 3.5, we examine the two-level cache hierarchy that we use in FAWN-DHT to ensure that hot-spots do not slow overall system performance.

Our evaluation of individual FAWN nodes and of a small-scale FAWN cluster (Section 5) show that FAWN-DHT nodes can serve 700 small requests per second while consuming under 6 watts. In aggregate, a complete FAWN cluster is capable of handling one half million lookups per second while consuming 5kW of power, achieving power efficiency nearly 50x that of magnetic disk and 4x that of low-power conventional processors coupled with solid-state disks. We believe that with custom hardware instead of the off-the-shelf nodes used in our evaluation, the power savings could be twice as high.

2 Background and Motivation

We begin by briefly reiterating well-known scaling trends that challenge systems designers: the growing (and varying) gap between disk, memory, and CPU speeds. We discuss the relevant metrics that we study in this paper (particularly cost-per-gigabyte and operations-per-joule). We end this section with an examination of the characteristics of Flash memory that are relevant to its use in computers, databases, and our FAWN clusters.

The Increasing CPU-Disk Gap One of the most constant and consternating trends of modern systems is the growing gap between the performance of CPUs and hard drives. HD transfer rates scale linearly with density increases, whereas capacity scales quadratically. HD seek times scale only with platter speed increases, size decreases, and mechanical improvements to read head technology. As a result, the fastest single drives available today have seek times in the range of 2 to 4 ms. For seek-bound workloads, a 4ms seek time provides only 250 requests/second to a CPU capable of billions of instructions per second. The throughput story is still daunting, though somewhat less severe: magnetic disks offer sustained transfer rates of 30-80MB/s; in contrast, a modern CPU can stream gigabytes/sec of data from memory.

This gap represents a prime opportunity to reduce power consumption without reducing performance by balancing the CPU and system capability with that of the storage devices. While simply reducing the speed of the CPU would provide great benefits, this reduction takes advantage of a second scaling challenge to modern CPUs:

CPU power consumption increases faster than speed. High-end CPUs dedicate large amounts of area and power to keeping the processor "fed" using caches, branch prediction, and speculation. These components do not *increase* the speed at which the processor can perform basic operations, but exist to ensure that the CPU-memory speed gap does not cripple system performance. Slower CPUs, in contrast, devote significantly more of their transistors to actual computational duties. As a result, these smaller CPUs are much more efficient in *instructions per joule* (equivalently, instructions per second per watt).

For example, a quad-core Xeon 7350 operates 4 cores at 2.6Ghz, consuming about 80W. Assuming optimal pipeline performance (4 operations per cycle), this processor optimistically operates at 520M instructions per joule—if it can issue enough instructions per clock cycle and does not stall or mispredict. On the other hand, a single-core Xscale processor running at 800Mhz consumes 0.5W, or 1600M instructions per joule. The *performance-to-power* ratio of the Xscale is at least three times that of the Xeon, and is likely higher given real-world pipeline performance.

Tellingly, the numbers are even more severe when considered as IO operations per joule: The XScale PXA320 has a 260MHz memory bus; the Xeon has a 1GHz memory bus, five times faster but at 160 times the power.

2.1 Finding the Sweet Spot

Given these trends, we ask: What is the right balance of storage technology and processor technology to serve a seek-intensive, small read workload? Our resulting architecture, which we present in Section 3, relies on "wimpy" processors with flash memory. To understand this decision, we highlight the key differences between magnetic disk, flash storage, and DRAM using four metrics:

- cost per gigabyte [1]
- throughput
- access time
- power consumption

Cost per Gigabyte: For storage under 4GB per device, Compact Flash(CF)/Secure Digital(SD) cards can be found for about $5 to $10.00/GB. Magnetic disks holding less than 4GB are nowhere to be found: the smallest magnetic disk drive commonly available contains 40GB. For small amounts of storage, CF/SD cards are most efficient in terms of form factor and cost per gigabyte. Market forces in the consumer photography sector are likely to help drive down the cost of CF/SD cards even further. Server-quality registered DRAM costs roughly $50.00/GB, depending on chip configuration and bus speed support.

For larger storage sizes, magnetic disk provides the best cost/GB, from $0.20/GB for a 320GB+ hard drive to $0.50/GB for the smallest (40GB) hard drive we found. In comparison, there are few CF/SD cards larger than 16GB, though they cost the same amount per gigabyte as smaller cards. Larger solid-state flash disks are still new enough that their prices vary widely; current 64GB models list for around $1050, or $16/GB.

Finally, it is worth noting that the achievable density of DRAM tends to be limited by the available motherboards. Most "standard" motherboards support 8–16GB of DRAM, but systems that support 32 and 64GB tend to be large server-class systems that draw particularly large amounts of power.

[1] Price statistics were gathered by surveying popular online computer electronic stores such as Newegg.com and Buy.com.

Throughput: Magnetic disk provides sustained read throughput from 20-80MB/s, and Compact Flash/SD from 3-20MB/s. DDR2 RAM provides peak transfer rates between 3.2GBps (PC2-3200) to 8.5GBps (PC2-8500). When throughput is more important than cost or capacity, RAM is the clear choice. On the other hand, providing enough RAM for a wide-range of query-based workloads requires large machines with support for many memory banks; such systems are likely to consume a lot of power. Furthermore, since RAM is volatile storage, additional secondary storage is required to hold the data more permanently, thus increasing the cost and power consumption of these systems.

Seek/Access Time: The fastest magnetic disks available today have seek times from 2-4ms. Compact Flash access times range from 0.1ms to 4ms depending on the product and interface; access times for Flash are likely to decrease as they are not limited by mechanical movement. DDR2 RAM access times are on the order of 10-100 nanoseconds. Intriguingly, recent work in databases has shown that even these low latencies cannot saturate a modern CPU, and CPU-cache-aware techniques are needed to get the most out of a database system [3].

Power Consumption: Magnetic disks spin platters at high RPM. Consumer magnetic disk technologies use about 10W of power when active [25] and 5W when idle. DRAM requires a significant amount of power to perform active refresh, and often requires between 4-10W per DIMM [25]. Flash technology requires between 1-3000 microJoules per read operation [20], using less than 1W under load.

For the moderate-sized (several 10s to 1000s of GB), seek-bound workloads we consider in this paper, DRAM systems can provide extremely high performance, but only at extremely high costs. In practice, the cost of hundreds or thousands of gigabytes of DRAM is too high for many uses. In this paper, we therefore consider moving towards a Flash-based architecture as a power and cost-effective middle ground between DRAM and disk.

Metric	**Server-class Node**	**Wimpy Flash-based Node**
CPU	Quad-core 2.3Ghz x86	800Mhz ARM
Memory (DRAM)	4-32GB	128MB
Memory cost/GB ($)	50	50
Storage capacity (GB)	40GB-1TB (per disk)	512MB-16GB (Flash)
Storage cost per GB ($)	0.30	10
Disk speed (read latency)	4ms	1ms
Queries per second (data in memory)	80000	3000
Queries per second (data on disk)	170	150-300
Power consumption	300W	0.75W
Total Cost (power + HW) over 3 years	9000	453

Table 1. Comparison of a high-end machine to a wimpy flash-storage node.

2.2 Flash memory characteristics

NAND Flash provides a non-volatile memory store, offering several significant benefits over typical magnetic hard disks:

1. Improved read access times, ranging from 1us to 1ms in modern chips, providing over an order of magnitude improvement [20].
2. Increased random read performance [19]: up to 18.5 times better random read throughput than on magnetic disk. As opposed to disk, accessing two sequential pages is no faster than

accessing two random pages.

3. Low power consumption: Flash devices utilize under a Watt of power even under heavy load, whereas disks are mechanical devices and require up to 10W at load.

On the other hand, Flash memory also operates similarly to disk in certain ways, including poor random write performance.

In Flash storage, reads and writes typically operate on pages on the order of 512-2048 bytes [19, 20]. Pages are organized into groups of 32 or 64, called erase blocks. Data in erase blocks have bits initialized to 1, and can be written at the bit granularity to change a 1 to a 0, but not from 0 to 1. Thus, a write operation requires that an entire block be erased before writing new data, because the 0 bits must be reinitialized to 1. As a result, Flash writes tend to be expensive in comparison to reads.

Small writes on flash are very expensive, because an update of a single page requires the equivalent of writing an entire block of pages. Previous work has shown that the energy and time to write to Flash requires a fixed initial access cost and a linear per-page access cost for each incremental page written [20], so it is in the interest of storage systems to write to Flash in as large chunks as possible to amortize the initial access cost of a write.

The Flash Transition Layer (FTL) on most Flash devices performs a number of additional duties, such as wear-leveling to prevent erase block wear-out. Some of these functions also add to the already-significant overhead of small writes on Flash devices. As we discuss in more detail in Section 7, these performance problems have motivated the wide use of log-structured techniques for Flash filesystems and data structures. These same considerations underlie the design of FAWN's node storage management system.

3 Design

The FAWN architecture, depicted in Figure 1, uses a large number of "wimpy" back-end nodes that act as data storage/retrieval nodes, and a much smaller number of more powerful front-end nodes that act as protocol gateways between an application protocol and the back-end nodes. Our design consists of three major components:

- The overall system architecture of wimpy nodes and front-end nodes and the division of responsibility between these nodes.
- The DHT-like system that assigns data to nodes, handles PUTs and GETs, and manages node joins and leaves.
- The techniques for managing data on individual nodes in a way that operates well on the highly resource-constrained back-end nodes.

We describe each of these components below. In Section 5, we show that each is necessary to creating a FAWN system that is fast, efficient, and that scales well.

3.1 Design Goals

Our design of FAWN and the FAWN-DHT strives to meet 3 goals:

1. *Power efficiency*: Maximize the queries performed per joule of energy.
2. *Meet service-level agreements*: Regardless of workload, the FAWN cluster must support a minimum number of queries per second. Exceeding this rate is nice, but need not be a primary design goal.

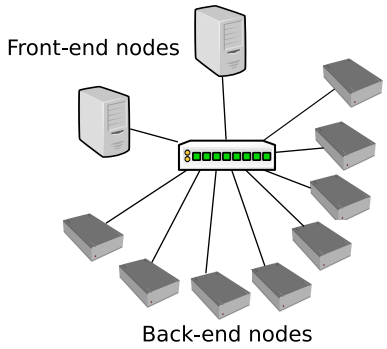

Figure 1. FAWN system architecture.

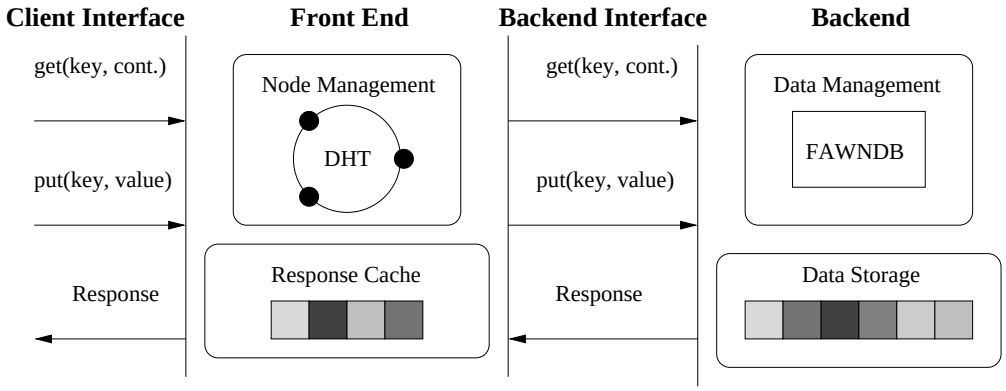

Figure 2. The FAWN API and division of responsibilities.

3. *Operate under continuous node arrivals/failures.* In a FAWN system of any size, failures will be the norm. The system must meet its goals despite frequent membership changes. This goal directly supports the notion that a FAWN architecture should be easily and incrementally expandable by simply adding additional nodes.

3.2 FAWN Architecture

The FAWN architecture, depicted in Figure 2, consists of four parts: the client interface, a front-end node manager with response cache, the back-end interface, and a back-end storage manager.

Client applications interact with FAWN via a key-based get/put interface, as they might with a more conventional (distributed) hash table such as Chord or Dynamo. For relatively simple applications such as DNS, the gateway from the client application to the put/get interface will run on the front-end node itself, or applications might be written to use the get/put interface itself. FAWN's interface is similar to that of many prior systems, though FAWN allows clients to specify a state continuation so that the front-end nodes can operate statelessly:

$$\texttt{get}(\texttt{key}, \texttt{continuation}) \rightarrow \text{value, continuation}$$
$$\texttt{put}(\texttt{key}, \texttt{value}) \rightarrow \text{status}$$

The front-end nodes maintain a one-hop DHT-like system (they know the identities of every back-end node) to route messages to nodes, and they maintain a small cache (Section 3.5) of recent

queries.[2] As described in the next section, the front-end managers handle join and leave events for new back-end nodes.

3.3 The FAWN Distributed Hash Table

FAWN's back-end nodes are organized into a storage ring-structure based upon the Chord DHT [29]. We omit the Chord routing algorithms in favor of a one-hop hierarchical design in which the front-end nodes maintain the entire node membership list (and can therefore route directly to the node containing a particular data item). This design requires that front-ends track on the order of hundreds or a few thousands of back-end nodes, which does not need the massive scalability of DHT $O(\log n)$ algorithms. While we suspect that readers are by now familiar with the consistent hashing techniques used in these systems, we outline briefly the operation of FAWN's storage manager, highlighting important differences with prior work where appropriate.

Storing data: FAWN uses a 160-bit circular ID space for its hash table. Nodes identify themselves as v "virtual" nodes in the ring, with IDs derived from the hash of their Ethernet MAC address. Each node is responsible for items for which it is the item's "successor" in the ring space (it is the node immediately clockwise in the ring). In addition, items are replicated at the r following nodes in the ring.[3] As an example, consider the cluster depicted in Figure 3 with three physical nodes. Each node is represented as two virtual nodes, and items are replicated onto two replicas.

The dashed line shows the area of the ring for which virtual node n1-1 is responsible as a primary and as a replica (secondary). In total, node N1 will store 5 different items from four different ranges. The distinction between these ranges plays a key part in efficiently handling node arrival and departures, as we discuss in the next subsection.

The FAWN-DHT front-end handling a PUT request sends the request to *every* node in the replica group. FAWN-DHT does not currently ensure consistency between these nodes if failures occur; in the future, we plan to add vector clocks and a gossip protocol for eventual consistency, but such support is outside the scope of our current work.

Fetching data: To service requests, the front-end looks up the replica group for a given key k. In our weakly-consistent model, only one node is needed to service a GET request, so the front-end directs all requests for key k to the node's owner (the first node counter-clockwise in the ring).

3.4 Per-Node Storage Management

FAWN back-end nodes use a storage architecture that is designed to make efficient use of their Flash memory and limited CPU and RAM. As described in Section 2, Flash memory behaves more like disk than is often perceived: random writes are more expensive than sequential writes, and small random writes are even worse. Prior work has shown that a log-like approach works well on Flash devices, and we use this observation as the basis for our design.

The workloads we envision for FAWN are read-mostly (representative examples again being serving huge DNS zones or e-commerce catalogs). With this workload, however, FAWN nodes must

[2] To focus primarily on the implementation issues surrounding wimpy nodes and flash memory, in this work we deliberately ignore some issues of consistency—our FAWN prototype provides only weak consistency and may return inconsistent results. We do not make use of this relaxed consistency to improve performance, but adding consistency would add additional communication load to the front-end nodes.

[3] As an optimization, replica groups skip over virtual nodes mapped to a physical node already represented in the group. In practice, this matters mostly for small clusters.

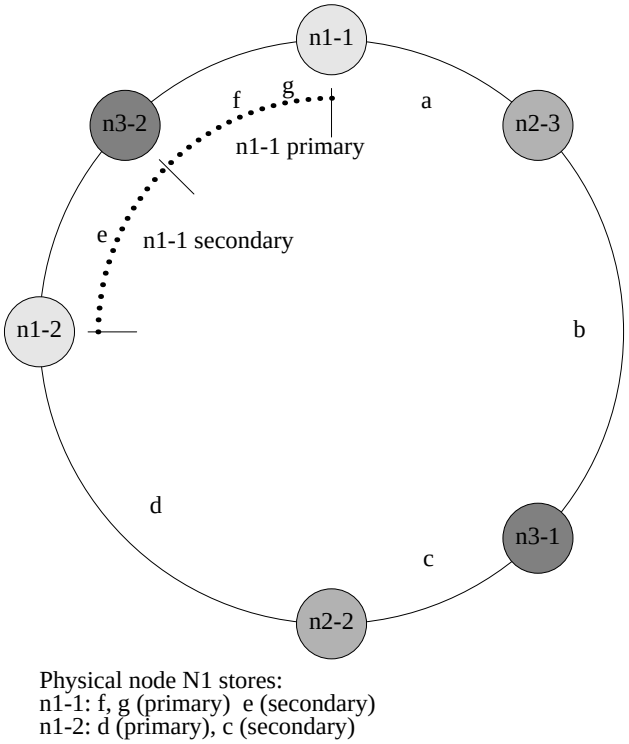

Figure 3. An example ring with three physical nodes. Items a, b, ..., g, are data items mapped to their locations in the ring.

still be able to write data rapidly when a node joins or leaves the system, and at a moderate rate during normal operation. Our storage architecture specifically optimizes for the traffic patterns that occur during membership changes.

Node join and leave traffic: Examining the ring in Figure 3, observe what would happen if a new node X was inserted between `n1-2` and `n3-2` on the left-hand side of the ring:

1. Node `n1-1` would no longer need to store objects falling between `n1-1` and the new node X and could free that memory.
2. Node X would need to obtain objects from nodes `n1-2` and `n2-2`.

In practice, of course, the node would insert at *multiple* locations because of its v virtual nodes, and so would cause changes at $r * v$ places in the ring.

As the experiments we present in Section 5 show, simpler architectural choices (e.g., simply using BDB to manage the back-end database) can be an order of magnitude slower than FAWN for handling these bulk data moves.

Node Storage Architecture: Our resulting architecture has two features: First, each FAWN node stores one database file per hash-range (e.g., the objects falling between two nodes on the ring) that it is responsible for. The node's *storage manager* handles each of these independent database files. With the parameters we choose for replication (3) and virtual nodes (≤ 10), each storage manager will have about 30 of these files (Figure 4).

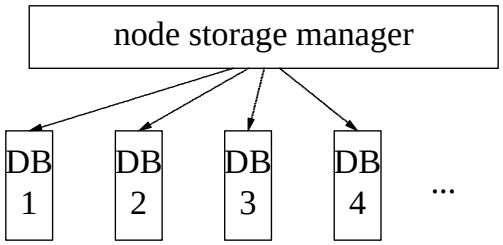

Figure 4. The FAWN per-node storage manager divides data into one file per hash-range that the node is responsible for to facilitate management and bulk data copy operations.

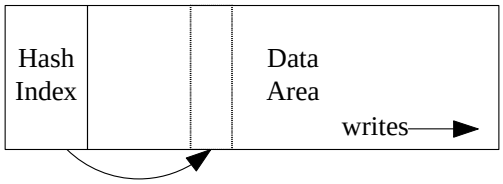

Figure 5. The FAWNDB back-end storage format. The index and data areas are separate, and all data inserts are sequentially written to the end of the data area, ensuring log-like performance.

Each FAWN database file is a hash-based database with a separate index and data region (Figure 5). Data is inserted by appending it to the end of the data region and updating the index pointer to it. The nodes are sized so that the index fits in memory, though most of the data will not.

This layout confers several advantages to FAWN nodes. Consider the above example of a node joining. To support this requires three operations:

1. Node `n1-1` splits one file into two using a single pass *partition* operation, and then deletes one of the resulting files. The partition operation takes a 160-bit hash value and splits a database file into two, one with objects above the partition value and one that contains all objects below the value.

2. Nodes `n3-2` and `n1-2` each send one entire hash-range file to the new node X. X can immediately begin using these files without re-indexing or touching their content.

Key to this approach is noting that upon joining, *all* operations on the flash devices are single-pass bulk read and write operations. Other than the single partition operation, these operations touch only the data that must be moved to the new node.

3.5 Caching in FAWN

FAWN uses caching to ensure that it always meets or exceeds its minimum query rate, even if an abnormal fraction of the total requests hit a single (wimpy) back-end node. This use, avoiding worst-case behavior, is quite different from the use of caching in traditional systems, where caching is used to boost average-case behavior. FAWN takes advantage of this difference to ensure high performance with comparatively tiny amounts of cache at the front-end nodes.

Load balance assumptions. FAWN uses consistent hashing to map items to nodes. We assume that while the distribution of requests for individual *objects* may be extremely unbiased, the

distribution of popular objects to FAWN nodes is random. We assume that an adversary cannot explicitly retrieve, e.g., a wide range of objects that all map to a single node. To guarantee this, FAWN might need to internally use a keyed hash function (which it does not currently do, but doing so is a simple change). Under this assumption, FAWN ensures that accesses are load-balanced by using the Chord technique of having $O(\log n)$ virtual nodes for each physical node.

Two-Level cache hierarchy. FAWN caches data using a two-level cache hierarchy. Each node implicitly maintains a small cache of recently accessed data in its filesystem buffer cache. Under a worst-case (completely random) workload, the node serves all data from flash, but as the workload reaching the node becomes biased, it serves data more rapidly from cache. As noted previously in Table 1, the wimpy (Soekris) nodes serve 150-300 queries/sec from flash, but can serve up to 3000 queries/sec from memory.

The FAWN front-end maintains a high-speed, small query cache to ensure that even if *all* queries would have gone to a single back-end node, the system can meet its guarantees. Fortunately, this cache easily fits in memory: it must hold only $N \log N$ entries, where N is the number of back-end nodes. With 512 byte entries and 1000 back-end nodes, the front-end would need only 512KB of cache.

Cache size analysis: The FAWN cluster can meet its guaranteed query rate when queries are being handled by all N back-end nodes. If the most popular $N \log N$ items are cached, and items are assigned randomly to back-end nodes (buckets), then by the coupon collectors problem, *all* of the nodes will be in use for at least one of these objects. As a result, for the workload to *not* result in cache hits, it must be requesting a sufficiently diverse set of items such that those items come from all nodes in the cluster. In this case, the cluster is able to meet its performance goals by serving the requests directly from the back-end nodes.

This analysis elides a small constant factor from the $O(N \log N)$ analysis for load balancing. Without that factor, the load on back-end nodes may be unbalanced by a factor of $\log N$. Fortunately, the first-level memory cache on the back-end nodes absorbs this imbalance—cached accesses are about 20 times faster on the back-end nodes. We confirm our analytical results with empirical performance results in Section 5.4.

4 Implementation

FAWN is implemented in C (4000 lines of code) and uses GLib2.0. The front-end node runs two threads – one thread services requests from applications and forwards them to the second thread which interacts with the back-end nodes and also maintains an internal cache. Put requests are directly forwarded to the appropriate back-end nodes. Get requests are serviced from the front-end cache if possible, or else are routed to the appropriate back-end node. The response is then cached in a cache-manager using LRU replacement. The front-end and back-end components interact over UDP. The maximum per port buffer size for UDP sockets is set to 1MB. Requests and responses are decoupled, i.e. the front-end makes a request and is not blocked for the response. Requests and responses are correlated using state continuations.

We have deployed the FAWN code on a small cluster that we use for the evaluation in Section 5. We measured the architecture on a number of node types, but found that most embedded systems failed in one way or another: the Soekris hardware upon which we measure our system is several years old and therefore consumes 3.75 watts per node. Lower-power embedded systems that provide similar performance (e.g., the Gumstix platform [1]) were typically deficient in other ways, such

as providing only 16 bit PIO access to their compact flash slots, that crippled their use as FAWN nodes (despite having an overall faster CPU). In the following section, we discuss the hardware upon which we have deployed FAWN. In Section 6, we discuss the requirements for the FAWN hardware we plan to build to support this system.

4.1 Prototype Cluster

We have deployed the FAWN prototype on a cluster of Soekris Net4801-60 nodes.

CPU	266Mhz SC1100 (Pentium-like)
Memory	256MB
Network	100Mbit ethernet
Flash	SanDisk ExtremeIII 2GB CF
Flash access	DMA

The cluster is configured to netboot from the front-end node. The nodes are linked to one 8-port and one 5-port 100Mbit/1Gbit switches to which the front end is also connected.

5 Evaluation

Our evaluation of FAWN follows closely from its design goals. We first compare the energy/performance and (estimated) cost/performance of a FAWN system to several conventional architectures; our results show that FAWN is competitive on cost while offering large energy savings. We next examine the behavior of the FAWN system in more detail to understand the effects of its major design features, examining the time required to handle node arrivals and failures and comparing our design decisions with designs used in more conventional architectures.

Measuring power consumption: Unless otherwise noted, we measure power consumption at the wall, including all conversion losses, using a "Kill-a-Watt" P4400.[4]

5.1 Performance and Power

We begin by comparing the queries/joule achieved by seven systems that occupy varying points on the power/performance curve. We compare a server, two desktop, and one laptop (because of its focus on power efficiency) with several hard drives and solid-state disks, along with three low-power embedded systems.

- **Server**: Quad Processor, Dual-Core Intel Xeon 5130, 12GB DRAM, 1Gbit/s Ethernet, 15x HD RAID-5.
- **Desktop**: Asus P2-P5945G barebones w/ 2.4Ghz Intel Core 2 Duo CPU, 2GB DRAM, 1Gbit/s Ethernet, 7200RPM Hard Drive. A power-efficient, small-form-factor machine.
- **Desktop+SSD**: The Desktop machine above equipped with an IDE 32GB solid-state Flash drive.
- **Macbook Pro**: A MacBook Pro laptop. 2.33Ghz Intel Core 2 Duo CPU, 2GB DRAM, 1Gbit/s Ethernet, 5400RPM Hard Drive.
- **Alix**: An Alix.1c embedded computer. 500MHz AMD Geode LX CPU, 256MB DRAM, 100Mbit/s Ethernet.

[4] http://www.p3international.com/products/special/P4400/P4400-CE.html, 0.2% accuracy.

System	QPS	Watts	Queries/Joule
Alix	704	6	117
Soekris (1)	334	3.75	89
Soekris (8)	2431	30	81
Desktop+SSD	2728	80	34
Gumstix	50	2	25
Macbook Pro	53	29	1.8
Desktop	160	87	1.8
Server	600	400*	1.2

Table 2. Query rates and power costs using different machine configurations. (* Server power is estimated–our power measuring device does not accept the 220V feed that the server uses.)

- **Soekris**: A Soekris net4801-66. 266MHz SC1100 CPU, 128MB DRAM, 100 Mbit/sec Ethernet.
- **Gumstix**: A Gumstix Verdex XL6P. 600MHz Marvell PXA270 XScale CPU, 128MB RAM. 16-bit PIO Compact Flash interface. 100Mbit/s Ethernet.

We first evaluate the power and query rate that these systems can achieve using the FAWN database on a single node. The tests requested the data over the network to measure the rate at which these nodes could serve data if they were acting in a FAWN cluster. Table 2 shows the results. The embedded systems are decisively more power-efficient than even the low-power desktop node with a modern SSD. The best-performing node, the Alix.1c, achieved 117 queries per joule. Laptops and desktops serving queries from a single hard drive achieved only 1.8 queries per joule (and were considerably slower). A high-end server with a 15-disk RAID5 array achieved 1.2 queries per joule.

Scaling with more FAWN nodes To determine whether the FAWN architecture would scale, we performed a *small*-scale scaling test using the 8 available Soekris nodes in our cluster. The results, in Figure 6, show that at this scale, the FAWN database scales linearly with an increasing number of back-end nodes. This cluster is quite small, but there is little in the architecture that impairs scaling to the small thousands of nodes: the front-end nodes track only a small amount of data per back-end node, and there is no inter-node communication. The FAWN design is very similar to other massively-scalable designs such as Amazon's Dynamo [8]. While these scaling results are as expected as they are encouraging, we plan to conduct larger-scale tests in the future.

5.2 Power and Cost of a FAWN Cluster

A FAWN cluster is not simply an independent set of nodes; it requires front-end nodes and a network infrastructure to link them together. In this section, we first extrapolate the queries/joule that could be provided by a complete FAWN cluster. For this analysis, we use as a baseline the Alix.1c system (Table 2) instead of the relatively old (2003) Soekris nodes from our test cluster. The analysis includes the power overhead of the FAWN nodes, the front-end nodes, and the network infrastructure. We conclude by briefly speculating about the cost and performance of a custom FAWN cluster.

Front-end nodes: The **Desktop** node (Table 2) with a small amount of flash consumes 80W and can forward 81,000 queries/second to back-end FAWN nodes. As a result, using Alix-based

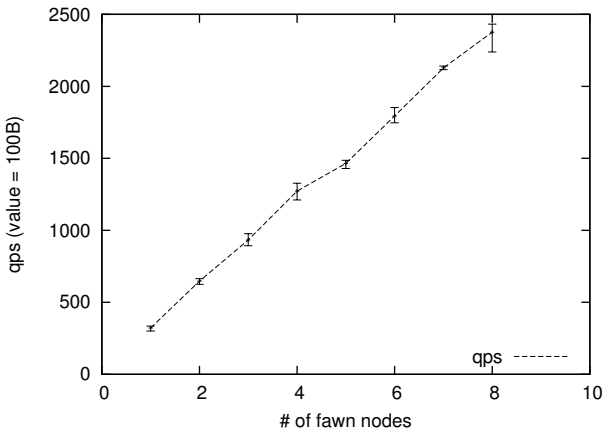

Figure 6. The FAWN cluster scales linearly with an increasing number of nodes to the limit of the cluster size.

back-end nodes, a FAWN cluster requires one front-end node per 115 back-ends, for a total of 7 front-end nodes for 720 back-end nodes.

FE power draw	560W
FE cost	$4,200

Network infrastructure: With 100 byte requests, each FAWN node sends approximately 720Kbit/s of traffic under full load ($128\text{bytes} * 704\frac{\text{queries}}{\text{sec}} * 8\frac{\text{bits}}{\text{byte}}$). This low query rate means that a single gigabit uplink can support 1000 back-end nodes, and so FAWN can use a relatively flat network hierarchy of 16-port switches ($65, 1.5 watts)[5] connected to a single 48-port 100/1000 switch ($750, 66 watts).[6] We note that the network infrastructure adds nearly 0.2 watts per node; while this is not a large increase with the Alix nodes, it would represent a 20% increase with a 0.8W node.

Network power draw	134W
Network cost	$3,675

Back-end nodes: 720 Alix.1c-based FAWN back-end nodes with 16GB of compact flash per node.

node power draw	4320W
node cost	$191,520

Alix node FAWN cluster: A cluster of 720 Alix.1c-based FAWN nodes, with 7 front-end nodes and associated network infrastructure.

FAWN total power draw	5014W
FAWN total cost	$199,395

[5]Listed DC power input for Linksys EZXS16W is 3.3V, 300mA. 1.5W value multipled by 1.5 for power supply inefficiency.

[6]Maximum listed power draw for HP ProCurve 2610.

With a 3x replication factor, this FAWN could serve a 3.8-terabyte dataset at 506,000 queries/second, or 101 queries per joule.

The power drawn by the FAWN is similar to that of a 16-node cluster of 1U rack-mount, dual processor machines. Given that the Alix nodes are over-provisioned for our needs in a FAWN cluster (they supply video, USB, PCI slots, multiple IO ports, etc), we believe that a more custom solution could achieve over 120 queries per joule.

Alternate architectures: The comparison with alternate architectures depends heavily on a design choice of whether to store the primary dataset in flash or on disk (in other words, whether and how much to replicate data on the FAWN nodes).

To achieve the same half-million queries per second would require $\sim$ 2000 disks, which alone would consume 20kW of power and cost $80,000 without machines. The disks would have the advantage of not needing a replication factor of three—even 40GB hard drives would provide more than enough storage space for the dataset. Achieving the same query rate with DRAM would require roughly 4TB of memory to hold the working set (assuming primary copies were stored on disk), at a cost of about $150,000 of DRAM, plus the cost of $\geq$ 128 server-class machines to hold the memory, which would themselves consume over 20kW. It is worth noting that, using power consumption figures from Rivoire et al. [25], the memory alone would consume approximately 4kW. A final alternative is using 192 nodes of the "Desktop-SSD" class equipped with 64GB solid-state drives. Such a cluster would consume 15kW and cost about 50% more than the equivalent FAWN cluster.

5.3 Leave and Join Performance

Several of FAWN's design goals center around ensuring that the system performs well when nodes join and leave. (Similar challenges arise when initially populating a FAWN cluster with data.)

Splitting and Merging: As described in Section 3.4, a newly-joined node must obtain the database files for the key ranges it is now responsible for. Multiple nodes must split a database file into two smaller files and transfer them to the new node. On a leave, multiple nodes must merge two database files into one larger file.

A FAWNDB split sequentially scans the data portion of the database and creates two new database files, appending data to each file in a log-like fashion. As it appends data to each of the smaller files, the index of each database is updated correspondingly. Because the index is small enough to fit in memory, this does not result in random writes to Flash. A FAWNDB merge operates in a similar log-like fashion. Figure 9 shows the time required for split and merge operations using FAWNDB: the merge operations shown combine two equally sized database files, whereas split splits one file into two files. The FAWNDB split and merge operations increase linearly with the size of the database being split or merged: Split requires a scan over an entire file, whereas merge requires a scan of only half of the file in this experiment, explaining the factor of two performance difference.

By way of comparison, our initial implementation of FAWN used Berkeley DB 4.6 (BDB) as the back-end storage manager, and we found that the split and merge operations on BDB were expensive. As shown in Figure 9, splitting and merging 128MB databases took 840 seconds, an order of magnitude longer than FAWNDB. Only the first 4 points are shown because of the time required to split/merge larger files.

Data Insertion: Inserts into BDB caused many more small writes to the Flash device, and its performance proved unacceptable. For example, we inserted 7 million records (comprising 1.8GB

of data), finding that the time required to insert depended on the size of the files generated:

Number of Files	Insertion Time (seconds)
1	46240
8	11867
32	8757

Figure 7. BDB Insert Times

Inserting into 32 different files sequentially improved performance because random writes touched a working set of only 58MB, which could be stored in memory. However, the time required to insert this data was still longer than for FAWNDB.

In contrast, inserting data into FAWNDB produced insertion times that do not depend on the number or size of the database files because data is always inserted sequentially regardless of the file size:

Number of Files	Insertion Time (seconds)
1	578
8	590
32	596

Figure 8. FAWNDB Insert Times

Splitting or merging a 1GB file requires less than 10 minutes, transferring the file over a 10Mbps takes about 15 minutes. FAWN therefore requires about 25 minutes to copy all data to a new node, and 10 minutes to initially populate a node.

5.4 Workload variance and caching

A significant design question in FAWN was whether to use higher-powered front-end nodes or to keep the architecture homogeneous. Our major reason for having a heterogeneous architecture was to provide caching at the (faster) front-ends to keep up with highly skewed workloads. Figure 10 shows the FAWN cluster's query rate as the workload goes from uniform (far left) to completely biased to a single object (far right), with and without the front-end cache. To model a power-law type distribution, the probability of selecting the popular key set is proportional to the number of popular keys, e.g. at 99% bias, 1% of the keys are selected 99% of the time.

As the workload bias increases, the query rate goes up regardless of the use of a front-end cache, as the back-end nodes become able to satisfy more requests from memory. Once the bias begins to reduce the number of *nodes* that the queries hit, however, the cluster without a front-end cache experiences a drastic reduction in throughput, finally being reduced to the in-memory performance of a single wimpy node. In contrast, the small cache at the FAWN front-end allows the cluster to maintain high performance regardless of workload bias.

6 Discussion and Future Work

Our investigation of the FAWN architecture is still early, but the results are promising: Even using many-year-old (and now discontinued) embedded systems, FAWN achieved many times the

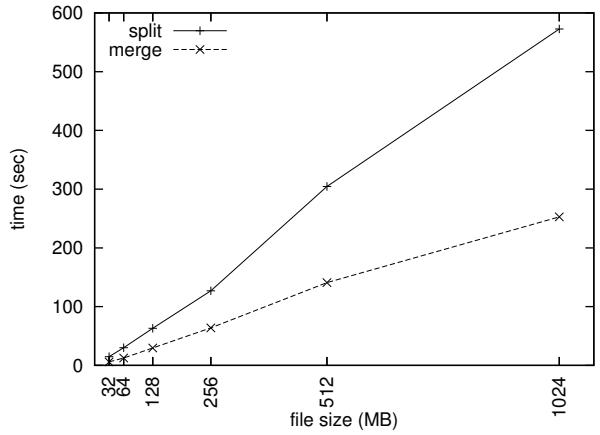

Figure 9. Split and merge times for FAWNDB and BDB for several database sizes.

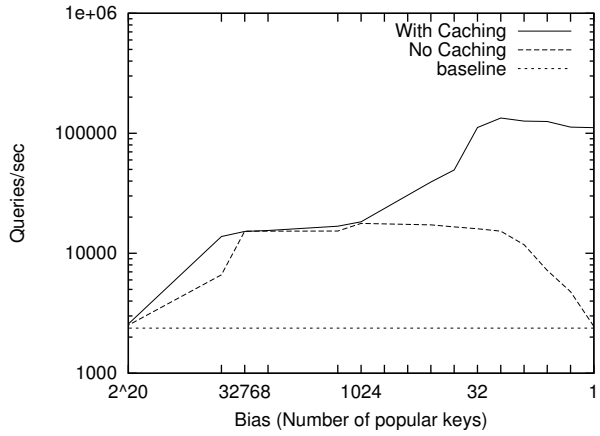

Figure 10. Queries per second (Y) vs. bias, where the probability of selecting the popular keys is inversely proportional to the number of popular keys. FAWN's two-level cache hierarchy prevents the throughput collapse that occurs when a single node becomes overloaded (nocache).

queries/joules of a modern system, even using solid-state disk. Important future work includes conducting our evaluation at significantly larger scale, with modern hardware tailored for use in a FAWN, and pushing the boundaries to understand for what other workloads FAWN provides superior cost and power-performance.

6.1 FAWN Hardware

As we noted in Section 4, our experience with the available hardware platforms has been disappointing. The right mix of hardware capabilities appears critical to getting the most out of the nodes while consuming very low power. We are therefore planning to create a custom embedded system to act as the next FAWN prototype; we believe that with modern hardware with proper Flash support, the power savings from FAWN can greatly exceed those we have shown in this paper. Briefly, the planned FAWN nodes comprise:

CPU	800Mhz XScale CPU Marvell PXA 320
RAM	128 or 256MB
Network	10 Mbit/s Ethernet (or 100)
Storage	Compact Flash slot
Interface	32-bit DMA capable

The cost estimates for such a system are slightly under $100 without Flash; external CompactFlash costs today about $8 per gigabyte. The power draw from the nodes is about 0.8W.

6.2 Future Workloads

Our efforts in FAWN thus far have examined read-mostly workloads. We plan to extend our work to understanding whether FAWN can perform well for write-intensive workloads that require fast random access. Our initial target is network packet logging systems [16] that log all or most network traffic and allow, for a time, post-hoc examination of previously received traffic that matches particular criteria. While flash's write performance is slower than its read performance, the FAWN back-end storage model of log-like writes with a small in-memory index might allow these workloads to achieve high write rates (5-10MB/sec per FAWN node) with very rapid random access.

7 Related Work

Any design for creating an array of smaller devices to replace one large component owes an intellectual debt to RAID [22]. FAWN leverages similar "dis-efficiencies" of scale for *computation* to those that RAID did for disks. In addition to this obvious descent, 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.

7.1 System Architectures

JouleSort [25] is a recent energy-efficiency benchmark; its authors developed a SATA disk-based "balanced" system coupled with a low-power (34W) CPU that significantly out-performed prior systems in terms of sorted records 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 with the smaller sort stages being performed in-memory. In this regard, JouleSort systems are more similar to the

high-throughput storage and analysis systems discussed below (in fact, a major step in MapReduce is performing a sort on all intermediate keys). FAWN, in contrast, targets even more seek-intensive workloads for which even the efficient CPUs used for JouleSort are excessive, and for which disk is inadvisable.

Considerable prior work has examined ways to tackle the "memory wall" (memory access times and bandwidth not keeping up with CPU speeds) and the equivalent problems with disk.

One of the most ambitious efforts in this area was the Intelligent RAM (IRAM) project, which examined an architecture that combined the CPU and memory into a single unit[5]. A particular focus of this project was on energy efficiency for portable systems; the authors found that 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 very different realization.

Similar efforts, such as the Active Disk project [24], examined a similar approach focused on harnessing computation close to disks. Schlosser et al. proposed obtaining similar benefits from coupling MEMS [28] with CPUs.

7.2 Disks with Flash

Recent trends have started placing moderate amounts of Flash either on-disk or on-controller to act as a cache of disk or filesystem data. These efforts build on significant past work examining the use of Flash memory as a cache to delay disk writes to save power and improve speed for power-constrained mobile devices, from work by Marsh et al. in 1994 [17], to related efforts in the last few years [6].

Today, several vendors offer "hybrid hard drives" that place a large Flash memory cache on disk to speed boot times and permit longer sleep intervals before writes. While these approaches have the potential to significantly speed average case performance, from FAWN's minimum-query-rate perspective, the hard drive component is unnecessary baggage: to meet its performance guarantees, FAWN must store all of its data in Flash anyway.

7.3 Databases and Flash

One of the most related efforts is recent attention to the use of large amounts of Flash memory for databases, up to and including placing the entire database in Flash. Recent work by Myers and Madden concluded that NAND Flash might be appropriate in "read-mostly, transaction-like workloads", but that Flash was a poor fit for high-update databases[19]. Their work, along with Nath & Kansal's FlashDB [20] also noted the log-like benefits of Flash, but in their environments, using a log-structured approach slowed query performance by an unacceptable degree. In contrast, FAWN's primary-key focus eliminates the need for more complicated B-tree-like indexing, and its separate data and index can therefore support log-structured access with no reduction of query performance.

7.4 Filesystems for Flash

Several filesystems are specialized for use on Flash memory. Most use some form of log-structured filesystem [26], such as the popular JFFS (Journaling Flash File System) for Linux, or the journaled ext3 filesystem. Our observations about Flash's performance characteristics follow a long line of research [9, 19, 31, 20]. 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 [30], followed closely by purely Flash-based log-structured Flash filesystems [14].

7.5 High-throughput storage and analysis.

A flurry of recent work has examined techniques for scalable, high-throughput computing on massive datasets. Major examples in this field include Hadoop or MapReduce [7] running on GFS [11]. More specialized examples include SQL-centric options such as the massively parallel data-mining appliances from Netezza [21]. In general, all of these approaches aim to allow users to perform data processing at "spindle speeds", streaming bulk data as rapidly as possible from tens to thousands of hard drives concurrently. In general, these systems solve an orthogonal problem of providing bulk throughput for massive datasets with low selectivity (most of the data is of interest); they are not optimized for the high-selectivity, seek-bound workloads that FAWN targets.

7.6 Distributed Hash Tables

FAWN-DHT takes advantage of much prior work in the design of distributed hash tables (DHTs). Its consistent hashing algorithm is based on Chord [29]. It borrows many features from existing systems, and *omits* many of the features that distinguish prior systems, such as distributed one-hop routing [15], $O(N \log N)$ routing scalability [29, 27, 18], or flexible consistency management [8]. Instead, FAWN-DHT focuses on the design aspects necessary to create a scalable hash table out of a huge array of resource-constrained nodes and to do so in a way that works well with Flash-based storage. Each of these systems has features that complement those in FAWN, and FAWN's low-power and Flash-friendly design should apply equally to these prior systems.

8 Conclusion

This paper presented the design and evaluation of a Fast Array of Wimpy Nodes specialized for supplying high response rates to seek-bound, small-read workloads. FAWN uses a combination of balanced hardware (wimpy nodes with power-efficient CPU and IO capabilities) and software specialized to the peculiarities of this environment. Through this dual-pronged approach, FAWN achieves power efficiency 3-50 times higher than existing systems, while remaining a cost-effective way to meet the demand for high query rates.

Our experience with FAWN showed that building a system that really takes advantage of the scaling trends that FAWN exploits requires considerable attention to detail at both the hardware level and in the design of the database layered atop the nodes. Our evaluation of seven commodity platforms identified several promising candidates, but also identified a gap between FAWN's needs and the hardware capabilities that could be filled to further reduce FAWN's power consumption.

While our work with FAWN is still progressing, we believe that its power savings and high achievable query throughput show that its approach is worthwhile in an era where computing density and costs are becoming increasingly dominated by factors such as power and cooling.

References

[1] Gumstix – dream, design, deliver.

[2] VMware: Virtualization, Virtual Machine and Virtual Server Consolidation.

[3] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 3:192–215.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization.

[5] W. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of Existing Architectures in IRAM Systems.

[6] F. Chen, S. Jiang, and X. Zhang. SmartSaver: turning flash drive into a disk energy saver for mobile computers.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters.

[8] 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.

[9] F. Douglis, F. Kaashoek, B. Marsh, R. Caceres, K. Li, and J. Tauber. Storage Alternatives for Mobile Computers. Pages 25–37.

[10] B. Fitzpatrick. LiveJournal's Backend: A History of Scaling.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System.

[12] K. Gray. Port deal with Google to create jobs.

[13] P. R. N. Jouppi. Enterprise IT Trends and Implications for Architecture Research.

[14] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system.

[15] B. Leong and J. Li. Achieving One-Hop DHT Lookup and Strong Stabilization by Passing Tokens.

[16] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching Network Security Analysis with Time Travel. (To appear).

[17] B. Marsh, F. Douglis, and P. Krishnan. Flash Memory File Caching for Mobile Computers.

[18] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric.

[19] D. Myers. On the use of NAND Flash Memory in High-Performance Relational Databases.

[20] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash.

[21] Netezza. Business intelligence data warehouse appliance.

[22] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID).

[23] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers.

[24] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active Disks for Large-Scale Data Processing. *IEEE Computer*, **34**(6):68–74.

[25] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: A Balanced Energy-Efficient Benchmark.

[26] 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.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.

[28] 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.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.

[30] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system.

[31] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices.