From architecture to algorithms: Lessons from the FAWN project


Carnegie Mellon University        *Intel Labs Pittsburgh
** Princeton University        *** Georgia Tech

Tuesday, April 9,
Power limits computing
Infrastructure: PUE
2005: 2–3
2012: ~1.1

Proportionality
1000W
750W
200W

Efficiency
300W
<100W

Combined...

Leaving it to industry
Builds up to...

- Computation / total energy = \( \frac{1}{PUE} \times \frac{1}{SPUE} \times \frac{\text{Computational Work}}{\text{Total Energy to Components}} \)
Efficiency vs load

Source: WSC book
Points to make

• DVFS used to be effective; not so good anymore

• Core is decently proportional; rest of components aren’t.
Efficiency is not free

Because speed is not cheap
Gigahertz is not free

Speed and power calculated from specification sheets
Power includes “system overhead” (e.g., Ethernet)
The Memory Wall

Disk Seek

Bridge gap: Caching, speculation, etc.

DRAM Access

CPU Cycle

Year


Nanoseconds

Year


Transistors

Have the soul of a capacitor

Charge

Moves charge carriers here

Which lets current flow

Principles

Key-Value Systems

FAWN-KV Design

Evaluation

Key-Value Systems Evaluation

FAWN-KV Design

The Memory Wall

Disk Seek

Bridge gap: Caching, speculation, etc.

DRAM Access

CPU Cycle

Year


Nanoseconds

Year

Two pillars

• Gigahertz costs twice:
  • Once for the switching speed
  • Once for the memory wall

• Memory capacity costs (at least) once:
  • Longer buses < efficient
“Wimpy” Nodes

1.6 GHz Dual-core Atom
32-160 GB Flash SSD
Only 1 GB DRAM!
“Each decimal order of magnitude increase in parallelism requires a major redesign and rewrite of parallel code” - Kathy Yelick
It’s not just masochism

Moore

Dennard

(Figures from Danowitz, Kelley, Mao, Stevenson, and Horowitz: CPU DB)

All systems will face this challenge over time
FAWN:
It started
with a key-value store
Key-value storage systems

- Critical infrastructure service
- Performance-conscious
- Random-access, read-mostly, hard to cache
Small record, random access

- Select name, photo from users where uid=513542;
- Select name, photo from users where uid=818503;
- Select name, photo from users where uid=474488;
- Select name, photo from users where uid=124566;
- Select name, photo from users where uid=097788;
- Select name, photo from users where uid=357845;
- Select name, photo from users where uid=42223;
- Select name, photo from users where uid=124111;
- Select wallpost from posts where pid=89888333522;
- Select wallpost from posts where pid=13821828188;
- Select wallpost from posts where pid=738838402;
- Select wallpost from posts where pid=12314144887;
- Select wallpost from posts where pid=738838402;
- Select wallpost from posts where pid=357845;
A cluster-distributed key-value store optimized for wimpy nodes and flash.

Fawn-KV: Hyper-optimized for low DRAM and large flash.

Fawn-DS: Small cache, provable load balancing using a tiny cache.

Parallel, fast, memory-efficient memcached using optimistic cuckoo hashing.
FAWN-DS and -KV: Key-value Storage System

Goal: improve Queries/Joule

Unique Challenges:
- Wimpy CPUs, limited DRAM
- Flash poor at small random writes
- Sustain performance during membership changes
FAWN-KV Architecture

Front-end - Switch - Back-end

FAWN-DS

Principles
Key-Value Systems
FAWN-KV Design
Evaluation
Avoiding random writes

In DRAM
Hashtable

In Flash
Data region

Put $K_1, V$

Get $K_2$

All writes to Flash are sequential
Three characteristics of flash underlie the design of the FAWN-KV system described next. Considerations inform the design of FAWN's node storage management system. Sustained random writes still perform poorly on these devices [62]. FAWN-DS maintains the benefits over typical magnetic hard disks for random access using an in-DRAM hash table [Hash Index] that maps keys to an offset in the Flash Data region. To provide this property, FAWN-DS maintains an in-DRAM hash table [Hash Index] that maps keys to an offset in the Flash Data region. To provide this property, FAWN-DS maintains an in-DRAM hash table [Hash Index] that maps keys to an offset in the Flash Data region.

Figure 2: FAWN-DS appends writes to the end of the Data Log. Split requires a sequential scan of the data region: transfer, ring out, of, range entries to the new store/ fib2 After scan is complete: the datastore list is atomically updated to add the new store/. Figure 3: Pseudocode for hash bucket lookup in FAWN-DS. Inserted values for the key range associated with one virtual ID. It acts to make a hash table of the KeyFrag/Valid/KeyFrag values.

KeyFrag != Key Potential collisions!

Low probability of multiple Flash reads

160-bit Key

DRAM Hashtable

Flash Data region

Offset

Log Entry

KeyLenData

Partial-key hashing

12 bytes per entry

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry

KeyLenData

KeyFragValid

Log Entry
Evaluation Takeaways

- 2008: FAWN-based system 6x more efficient than traditional systems
- Partial-key hashing enabled memory-efficient DRAM index for flash-resident data
- Can create high-performance, predictable storage service for small key-value pairs
And then we moved to Atom + SSD

<table>
<thead>
<tr>
<th></th>
<th>Geode 500Mhz</th>
<th>256MB</th>
<th>4GB CF Card</th>
<th>~2k IOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Atom 1.6 Ghz single-core</td>
<td>2GB</td>
<td>120GB SSD</td>
<td>~60k IOPS</td>
</tr>
<tr>
<td>Speed Increase</td>
<td>6x</td>
<td>8x</td>
<td>30-60x</td>
<td></td>
</tr>
</tbody>
</table>
backend store
hyper-optimized
for low DRAM
and large flash
Flash Must be Used Carefully

Random reads / sec | 48,000

→ Fast, but not THAT fast

$ / GB | 1.83

→ Space is precious

Another long-standing problem: 
random writes are slow and bad for flash life (wearout)
Three Metrics to Minimize

**Memory overhead** = Index size per entry
- Ideally 0 bytes/entry (no memory overhead)

**Read amplification** = Flash reads per query
- Limits **query throughput**
- Ideally 1 (no wasted flash reads)

**Write amplification** = Flash writes per entry
- Limits **insert throughput**
- Also reduces **flash life expectancy**
  - Must be small enough for flash to last a few years
Memory efficiency

High performance

SkimpyStash

FAWN-DS
FlashStore
HashCache
BufferHash
(static) “External Dictionary”

- Prior state of the art: “EPH”: ~3.8 bits/entry
- Ours: Entropy-coded tries, ~2.5 bits/entry

- Important considerations:
  - Construction speed; query speed
  - Aw, it’s read-only...
Solution: (1) Three Stores with (2) New Index Data Structures

Queries look up stores in sequence (from new to old)

Inserts only go to Log

Data are moved in background

Entropy-Coded Tries (Memory efficient)

SILT Filter

SILT Log Index (Write friendly)

Memory

Flash

Sorted
Workload: 90% GET (100~ M keys) + 10% PUT

Caveat: Not on wimpies. Still working on reducing CPU cost! :-)

Tuesday, April 9,
And now... Load imbalance

- Distributed key-value system

1. get(key)
2. BackendID=hash(key)
3. val=lookup(key)
4. return val

SLA: 850,000 queries/sec
10,000 queries/sec
Measured tput on FAWN testbed

Overall throughput (KQPS)

- uniform
- Zipf (1.01)
- adversarial

n: number of nodes
How many items to cache?
small/fast cache is enough!

E.g., for 1KB (k,v) pair, 85 nodes, 3MB needed, fitting in CPU L3 cache

We prove that, for $n$ nodes
- Only need to cache $O(n \log n)$ most popular entries
- worst case perf. = $(1 - \varepsilon) \times n \times$ single node capacity
Cache forces near-uniform dist.

Popularity

Cached Keys

Uncached keys

KeyId
Worst case? Now best case

Overall throughput (KQPS)

n: number of nodes

uniform
Zipf (1.01)
adversarial

Tuesday, April 9,
Thus...
“Brawny” server

Insanely Fast Cache

O(N log N)

Multi-reader parallel cuckoo hashing

[“small cache” socc 2011]

“Wimpy” servers

[SILT, SOSP 2011]

SILT

SILT

SILT

[SILT, SOSP 2011]

Entropy-coded tries

Partial-key cuckoo hashing

[SILT + under submission]

Cuckoo filter

[FAWN, SOSP 2009]

[under submission]
highly parallel, lower-GHz, (memory-constrained?):

Architectures, algorithms, and programming