# POWER EFFICIENT DATA CENTER FOR A KEY-VALUE STORE

Aaron Gorenstein

CSC 458

2/14/2011

# Problem Area, Motivation

- Data Centers
- Currently use “typical” computers
- Core issue: power!
  - 10-20kW per rack, 10-20MW per center, projected up to 200MW
- 50% of 3 year cost of ownership goes to the power bill
- This paper: a data center designed to address a particular problem area
  - Small key-value pairs—Facebook wall posts, tweets

# Why in this class?

- Massively parallel IO
- This topic is how to provide power-efficient, parallel access to a key-value data-store
  - Read, Writes, and Deletes

# Aspects of Improvement

- Power:
  - Processor: energy requirement grows super-linearly with speed
    - Processor scaling isn't sufficient—sacrifice 80% of computation for just 50% energy savings
    - Shutting down machines works, but bad with traffic spikes
    - We just don't need the processing power—weak processors!
  - Storage: keeping hard drives spinning or DRAMS active
    - Hard drives are inefficient for queries serviceable per watt
    - DIMMS are quite expensive
- Speed:
  - Hard drives are always slow, KV particularly bad area

# The Solution

- FAWN:
  - Fast Array of Wimpy Nodes
- Completely different hardware
  - Power efficient
  - Flash storage
- Custom software exploits the hardware properties

# Rest of the Talk

- Node-level:
  - The hardware in a FAWN node
  - The software (file system) in a FAWN node
- Array Level:
  - Design and implementation of a data store with FAWN Nodes
- Results: was it worth it?
- What's Next?

# Node-Level Hardware

- Processor: 500 megahertz
- Storage: camera flash drive
  - Read speed: up to 175x faster random reads
  - Write speed: Must do whole block rewrites, sustained random writes bad
  - Power requirement: 1 watt under heavy load (4GB), contrast with 10 watts for 2GB DIMMs

# Node-level Software

- File system: FAWN-DS (FAWN Data Store)
- Designed for Flash:
- Log-structured
  - Append-only filesystem
  - All writes are sequential
  - All reads require *single* random access
    - Hashmap maintained in memory, can be reconstructed from log
- Store, Lookup, Delete
- Compacting

# Node-level Software Cont'd

- Parallel:
  - Stores, Deletes modify hash table, write to the end of the log
  - Deletes delayed to avoid random writes
- Node Maintenance (20-30ms locks)
  - Merge
  - Split
  - Compact

# FAWN-KV (Key Value)

- Parallel IO
- Parallel Considerations:
  - Problem Decomposition
  - Tasks and their dependencies
  - Load Balancing
- Distributed Considerations
  - Replication
  - Consistency

# FAWN-KV

- Supporting a request:
  - Front end owns a slice of the pie
  - Each front end has 80 back ends
- How does that work for load balancing?
  - Consistent hashing, from Chord
  - One master node determines the pie slicing
  - Queries go directly to the front-end nodes, who fix any address errors themselves

# FAWN-KV

- Replication
  - Each VID is in a length-R replication chain
  - Consistency follows from sequential log
- Joins
  - Copying all necessary information requires single log-scan
  - Split the key range owned by a node
- Leaves
  - Merge over key range, then provide new duplicates
  - Merge the key range owned by neighboring nodes

# Results

- Storing new data: two million 1 KB entries into a single log was 23.2 MBS/s, 96% of raw speed
- Compared to general-purpose BerkeleyDB, tried seven million 200 byte entries, 0.07 MB/s
- Power: on idle, 3 watts per node, less than 4 when under load
  - 10 watts per switch.
  - On heavy load, switches are 20% of power draw

# What's Next?

- Scaling
  - No data about how well it scales
  - Exploiting chain replication for more robust load-balancing
- Complaints:
  - Data store, not data base
  - Limited application
  - Larger Files

# Citations


- David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: a Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (SOSP'09). ACM, New York, NY, USA, 1-14.