

# From architecture to algorithms: Lessons from the **FAWN** project

David Andersen, Vijay Vasudevan, Michael Kaminsky\*, Michael A. Kozuch\*, Amar Phanishayee, Lawrence Tan, Jason Franklin, Iulian Moraru, Sang Kil Cha, Hyeontaek Lim, Bin Fan, Reinhard Munz, Nathan Wan, Jack Ferris, Hrishikesh Amur\*\*, Wolfgang Richter, Michael Freedman\*\*\*, Wyatt Lloyd\*\*\*, Padmanabhan Pillali\*, Dong Zhou

Carnegie Mellon University

\*\* Princeton University

\*Intel Labs Pittsburgh

\*\*\* Georgia Tech

# Power limits computing



# Infrastructure: PUE

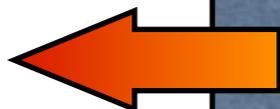
2005: 2-3

2012: ~1.1

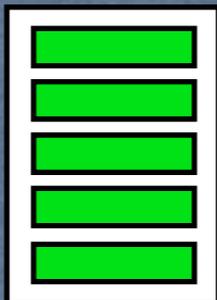
*Leave it to industry*



1000W

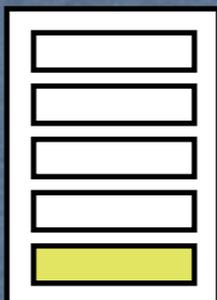


100%



Servers

20%



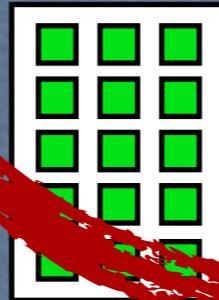
750W



200W

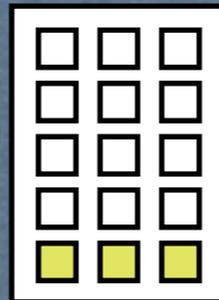
**Efficiency**

100%



FAWNs

20%

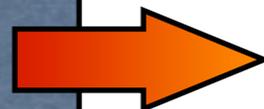


300W



**Combined...**

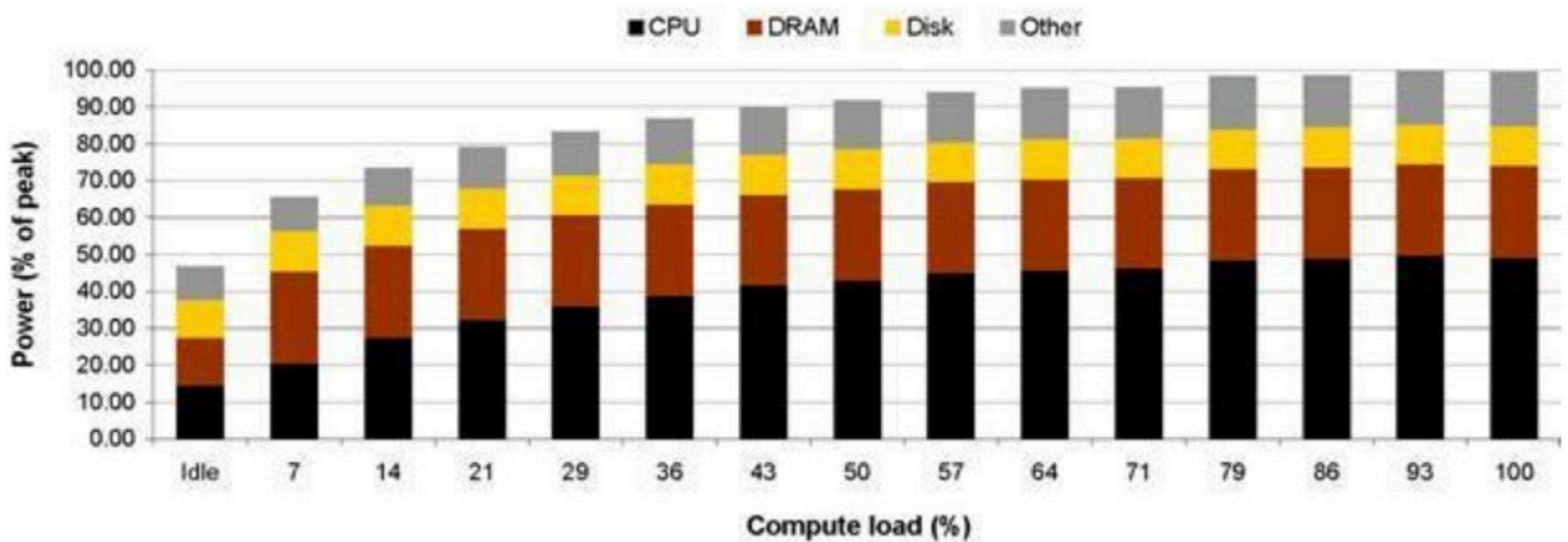
< 100W



# Builds up to...

- Computation / total energy =  
$$\frac{1}{\text{PUE}} * \frac{1}{\text{SPUE}} * \frac{\text{Computational Work}}{\text{Total Energy to Components}}$$

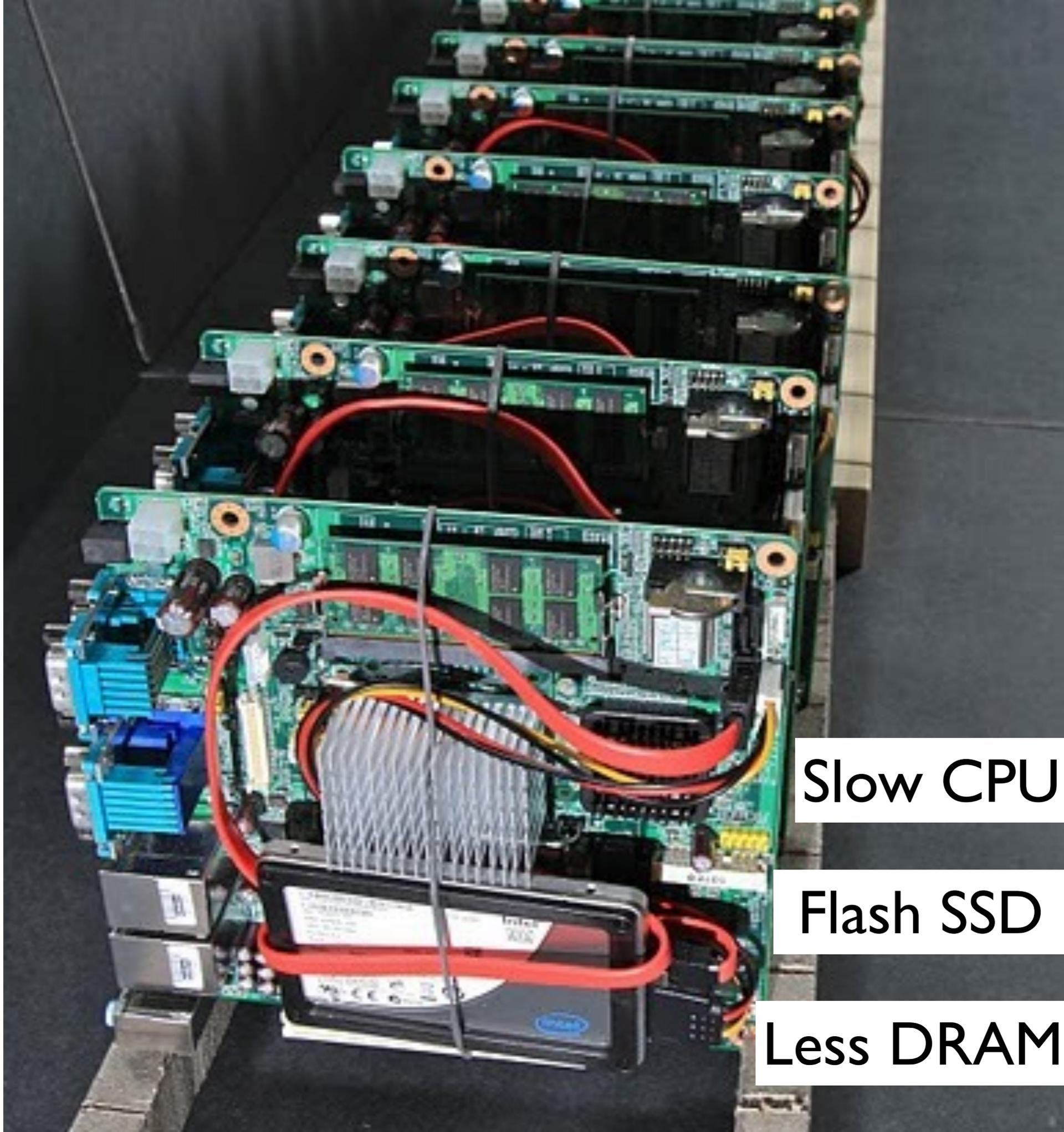
# Efficiency vs load



Source: WSC book

# Points to make

- DVFS used to be effective; not so good anymore
- Core is decently proportional; rest of components aren't.



Slow CPU

Flash SSD

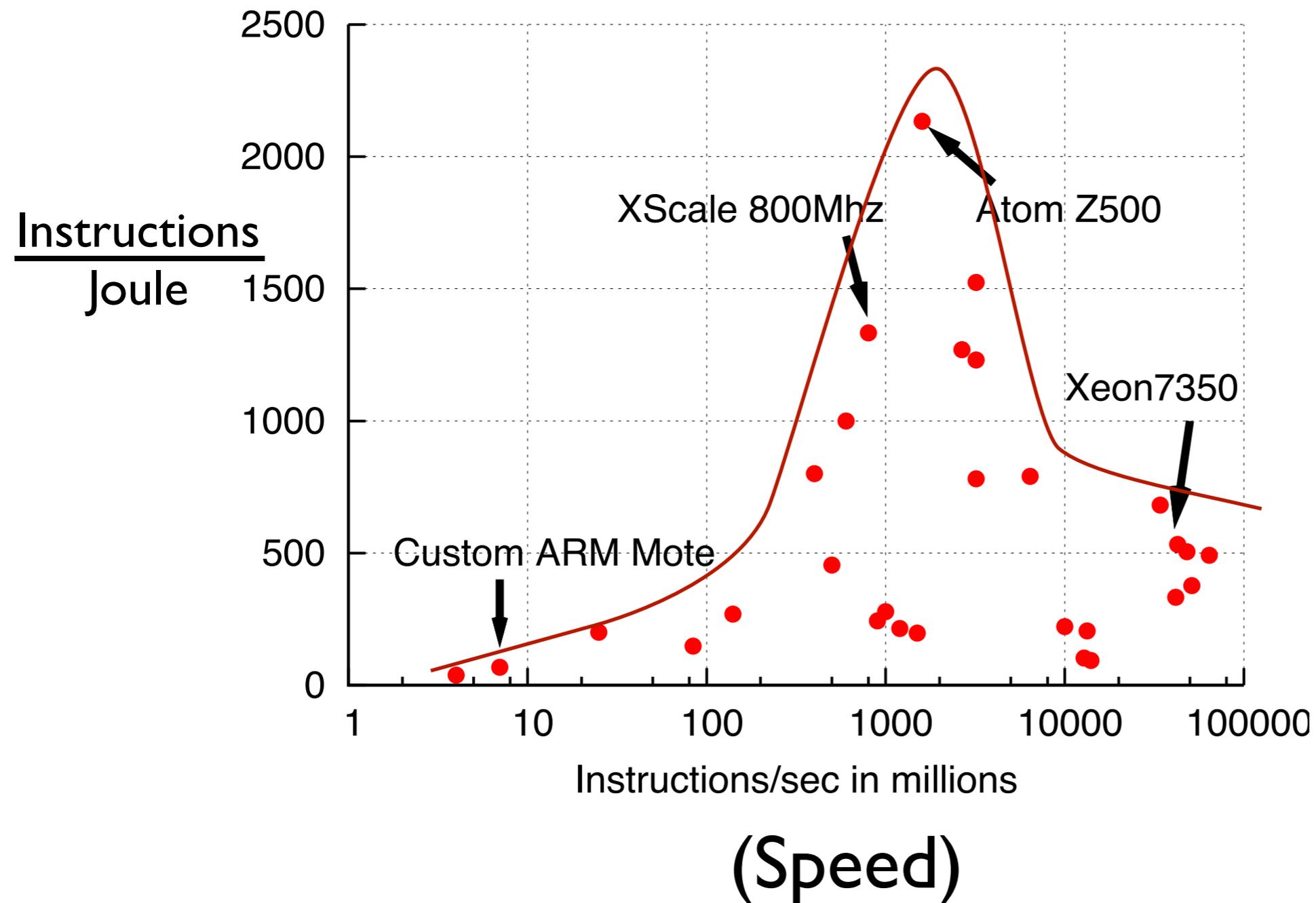
Less DRAM

Efficiency is not free

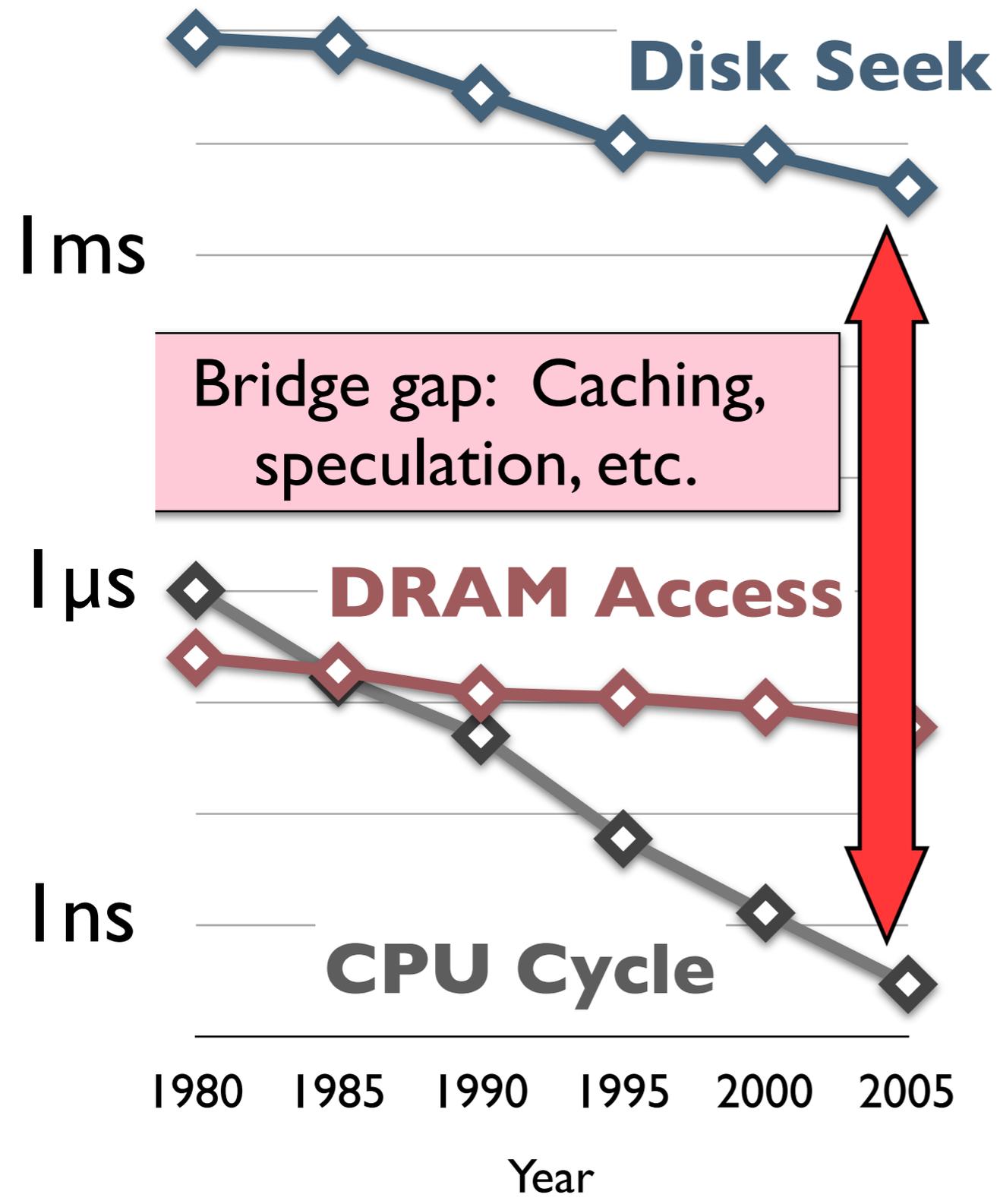
**Because speed is  
not cheap**

# Gigahertz is not free

Speed and power calculated from specification sheets  
Power includes "system overhead" (e.g., Ethernet)

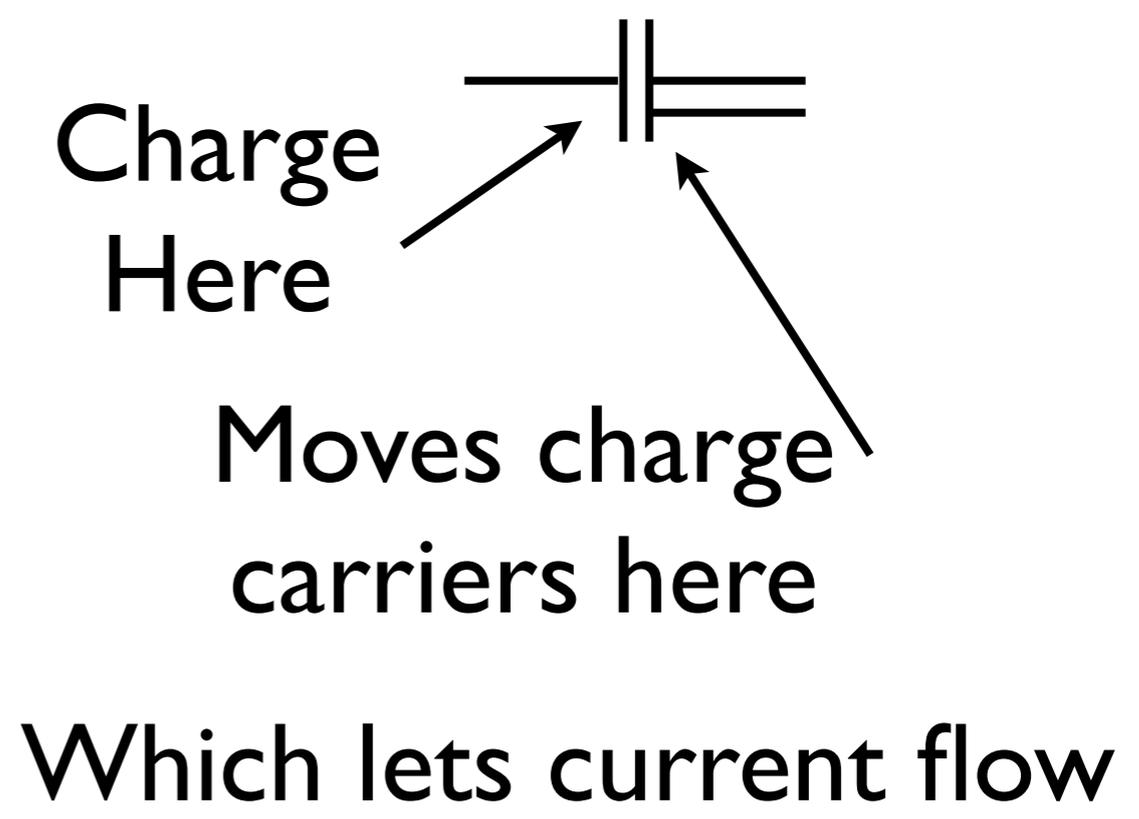


# The Memory Wall



# Transistors

Have the soul of a capacitor



# Two pillars

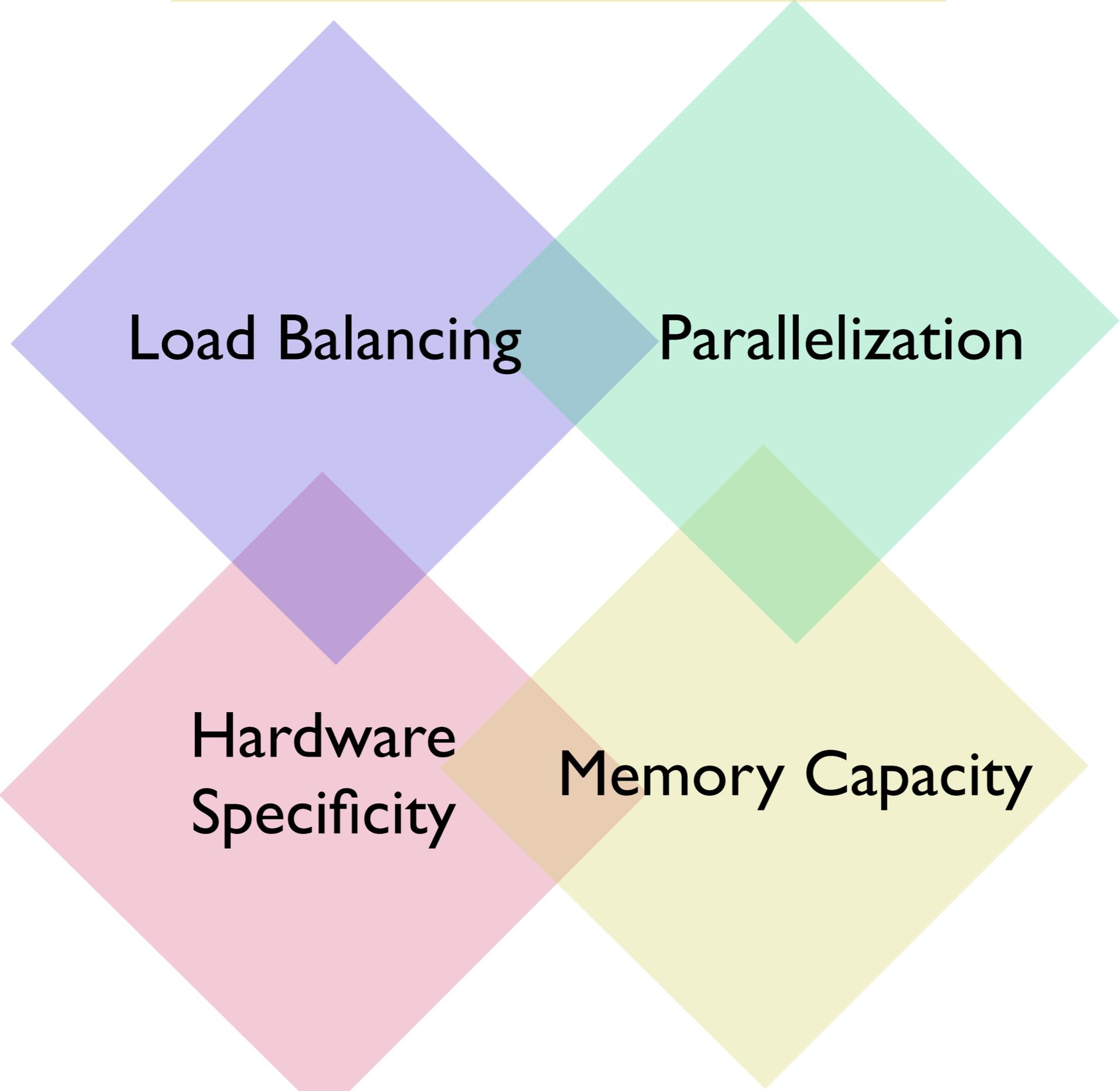
- Gigahertz costs twice:
  - Once for the switching speed
  - Once for the memory wall
- Memory capacity costs (at least) once:
  - Longer buses < efficient

# “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*

# The FAWN Quad of Pain



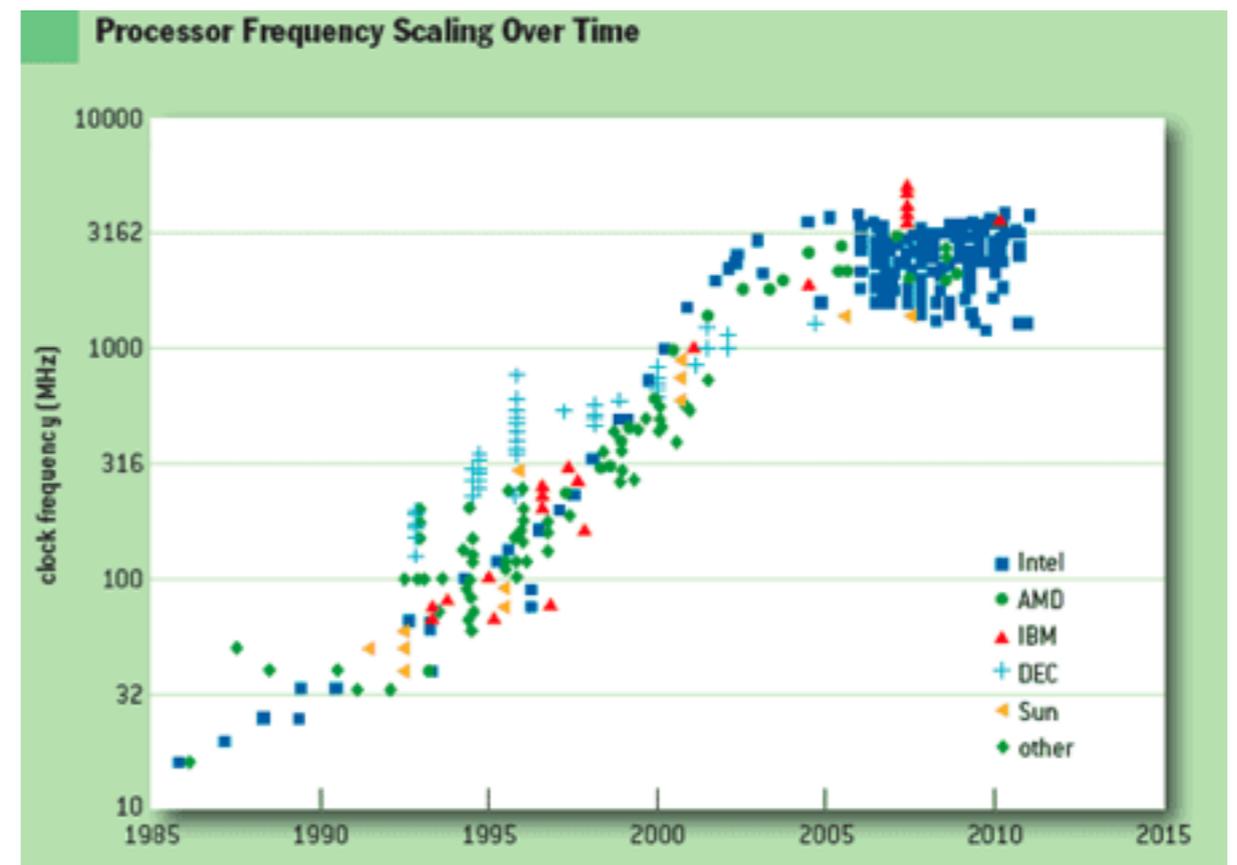
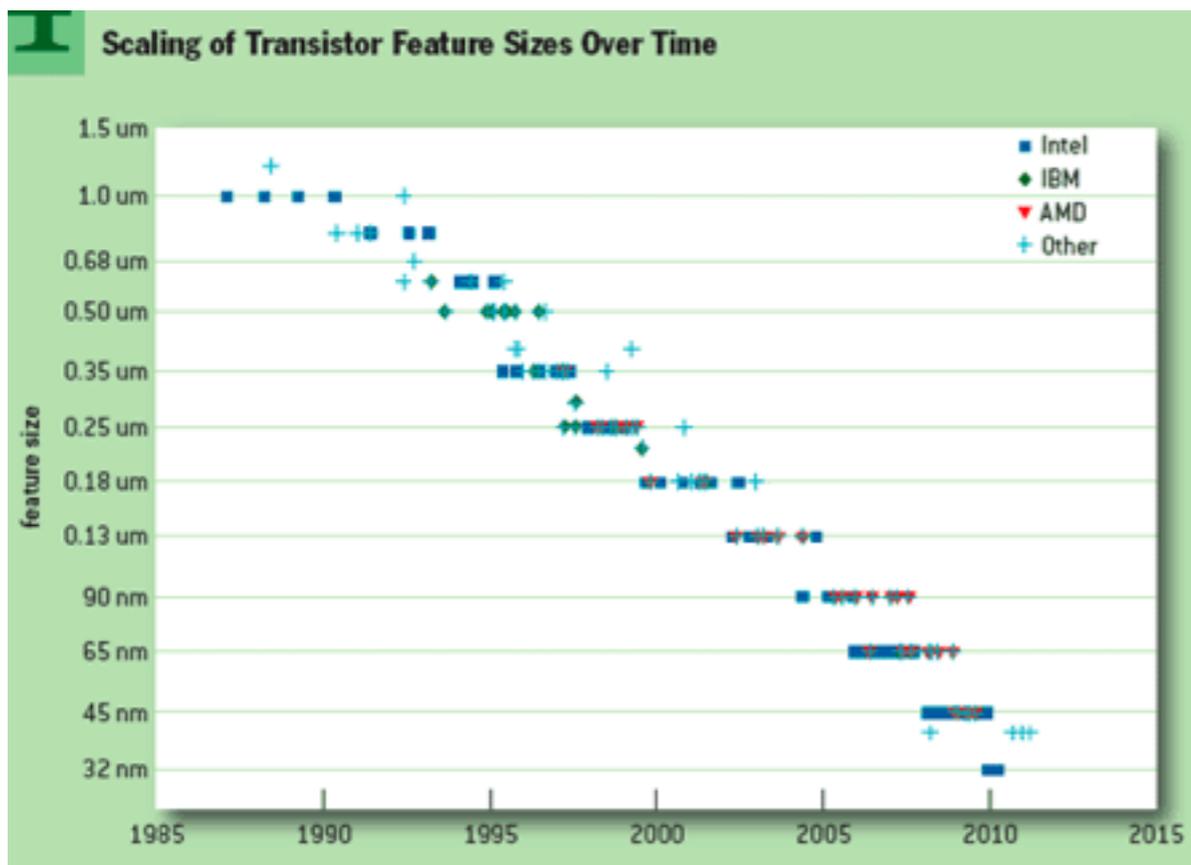
Bigger Clusters

Wimpy Nodes

# 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

The screenshot shows the Facebook homepage from 2007. On the left, there is a login form with fields for "Email:" and "Password:", a "Login" button, and a "Forgot Password?" link. The main content area features the Facebook logo, the text "Facebook is a social utility that connects you with the people around you.", and a "Sign Up" button. Below this, there is a section with links for "upload photos", "publish notes", "get the latest news", "post videos", "tag your friends", "privacy settings", and "join a network". At the bottom, there is a "Find your friends" link.

The screenshot shows the Twitter homepage from 2010. At the top, there is a search bar with the text "Search for a keyword or phrase" and a "Search" button. Below the search bar, there is a navigation bar with links for "Home", "Direct Messages", "Profile", "Settings", and "Sign Out". The main content area is divided into several sections: "See who's here" with a grid of user avatars, "Top tweets" with a list of tweets, and "New to Twitter?" with a "Let me in" button. The footer contains copyright information and links for "About Us", "Contact", "Blog", "Status", "Goodies", "API", "Business", "Help", "Jobs", "Terms", "Privacy", and "Language: English".

# Small record, random access

Select name,photo from users where uid=513542;

Select wallpost from posts where pid=89888333522;

Select name,photo from users where uid=818503;

Select wallpost from posts where pid=13821828188;

Select name,photo from users where uid=474488;

Select name,photo from users where uid=124111;

Select name,photo from users where uid=42223;

Select name,photo from users where uid=468883;

Select wallpost from posts where pid=12314144887;

Select name,photo from users where uid=007788;

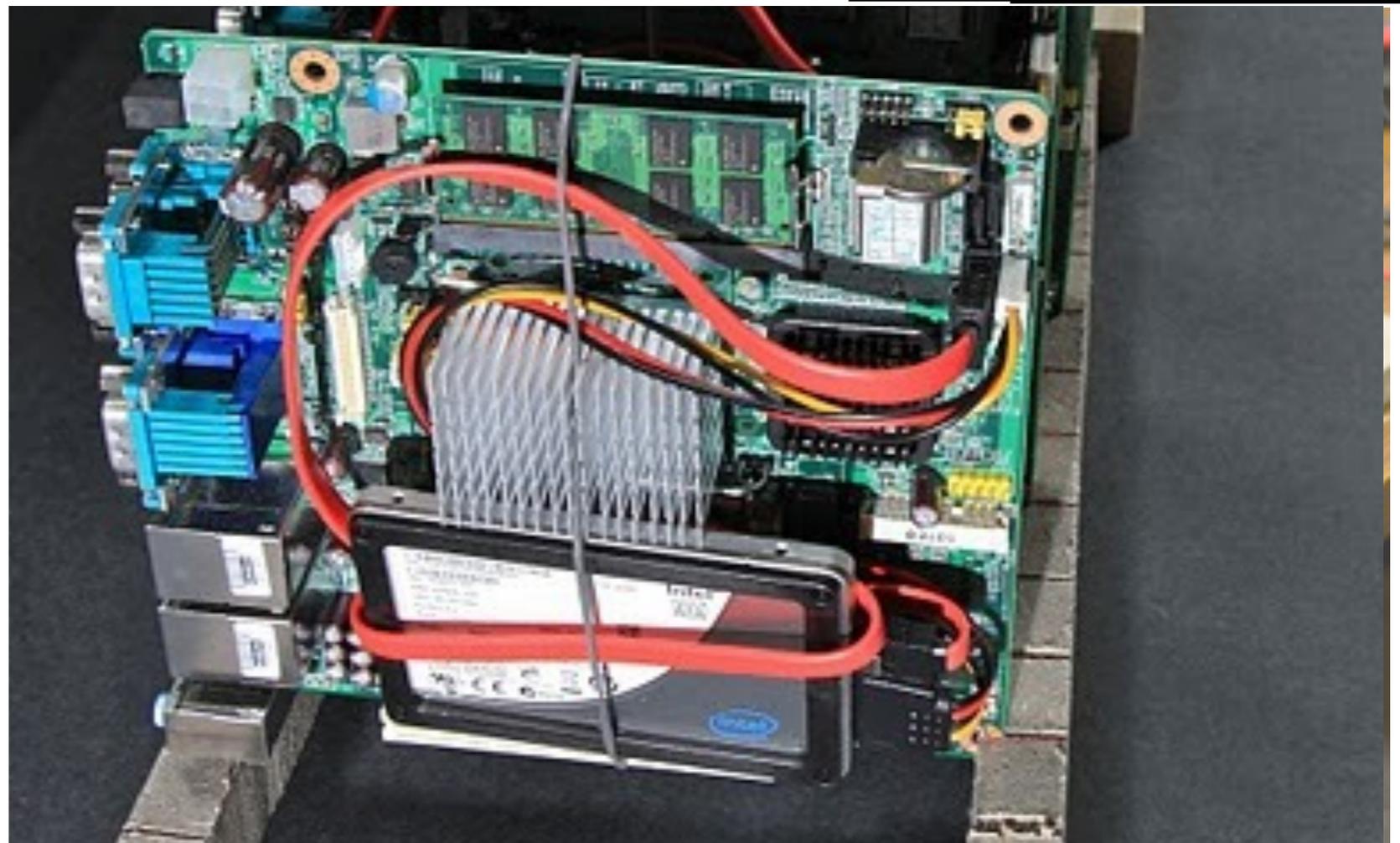
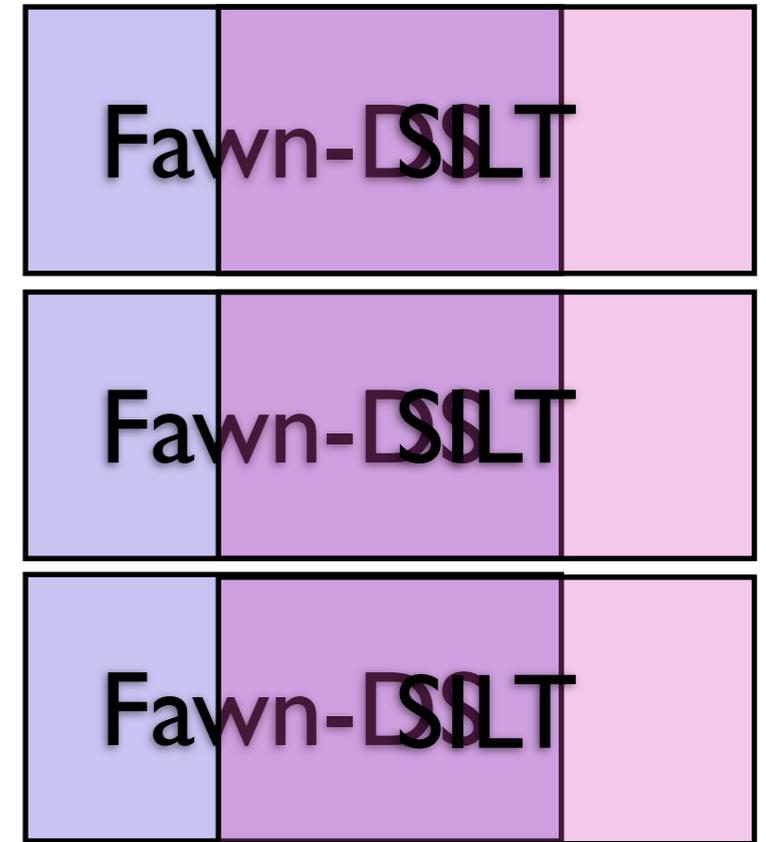
Select wallpost from posts where pid=738838402;

Select name,photo from users where uid=357845;

FAWN-DS    FAWN-KV    SILT    Small Cache    Cuckoo

key-value, parallel, fast, distributed store  
 backed by distributed  
 memory-optimized  
 hyper-optimized store  
 optimized for wimpy  
 multi-processor churning  
 using NAND flash  
 and large flash  
*cuckoo hashing*

Cuckoo  
 FAWN-KV  
 Cache



# FAWN-DS and -KV: Key-value Storage System

Goal: improve **Queries/Joule**

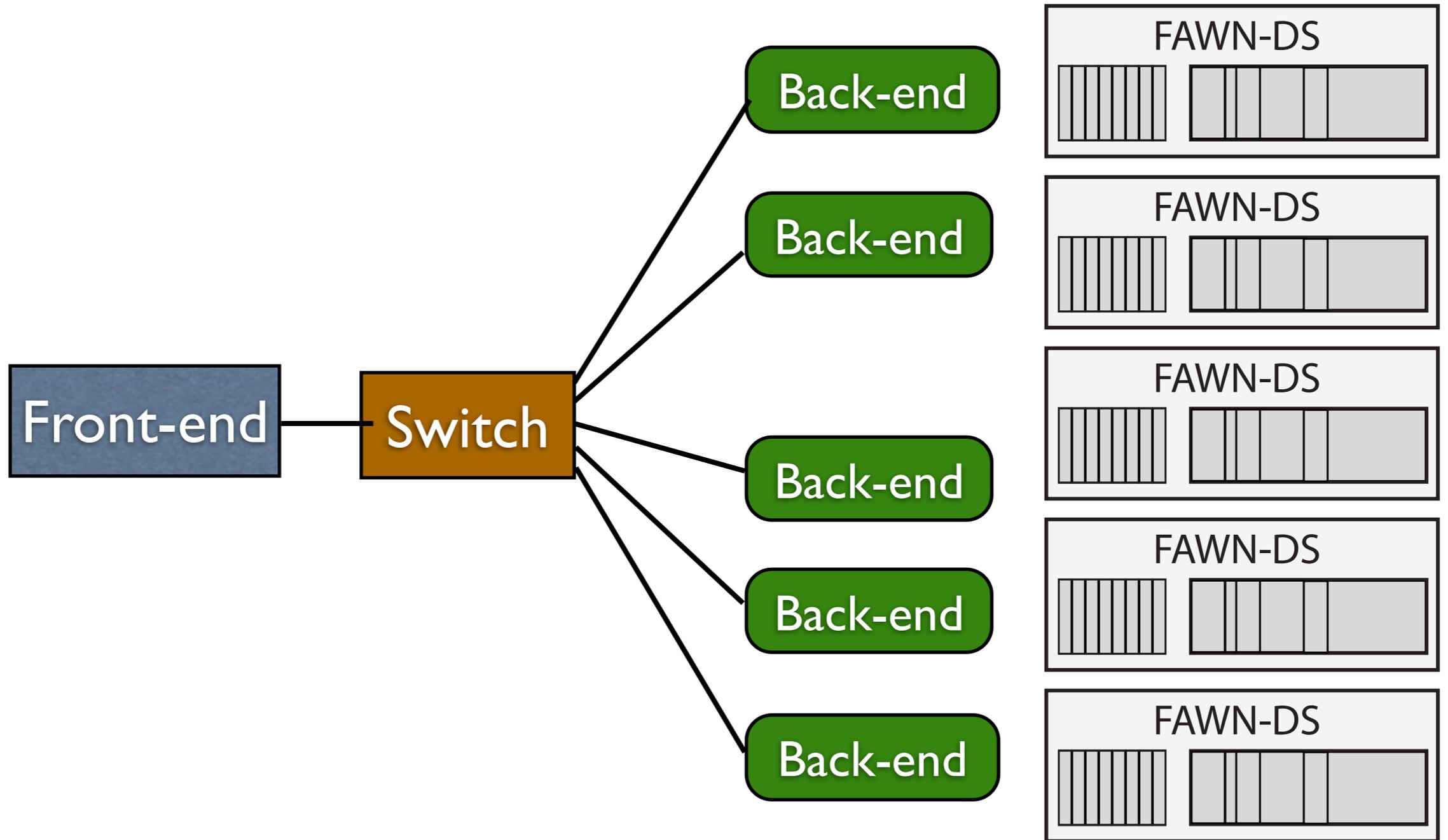
## Unique Challenges:

- Wimpy CPUs, limited DRAM
- Flash poor at small random writes
- Sustain performance during membership changes

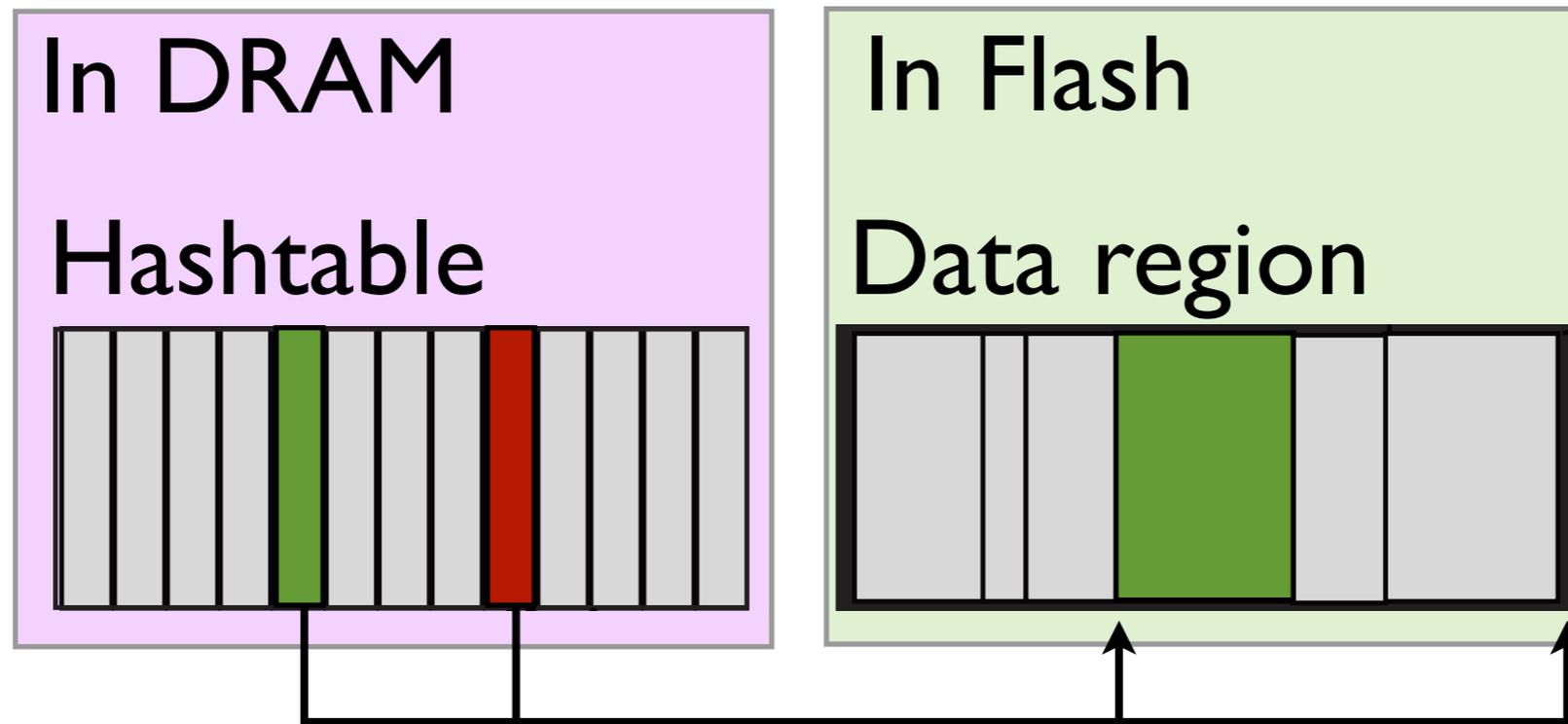


256MB DRAM  
4GB CompactFlash

# FAWN-KV Architecture



# Avoiding random writes



Put  $K_1, V$

Get  $K_2$

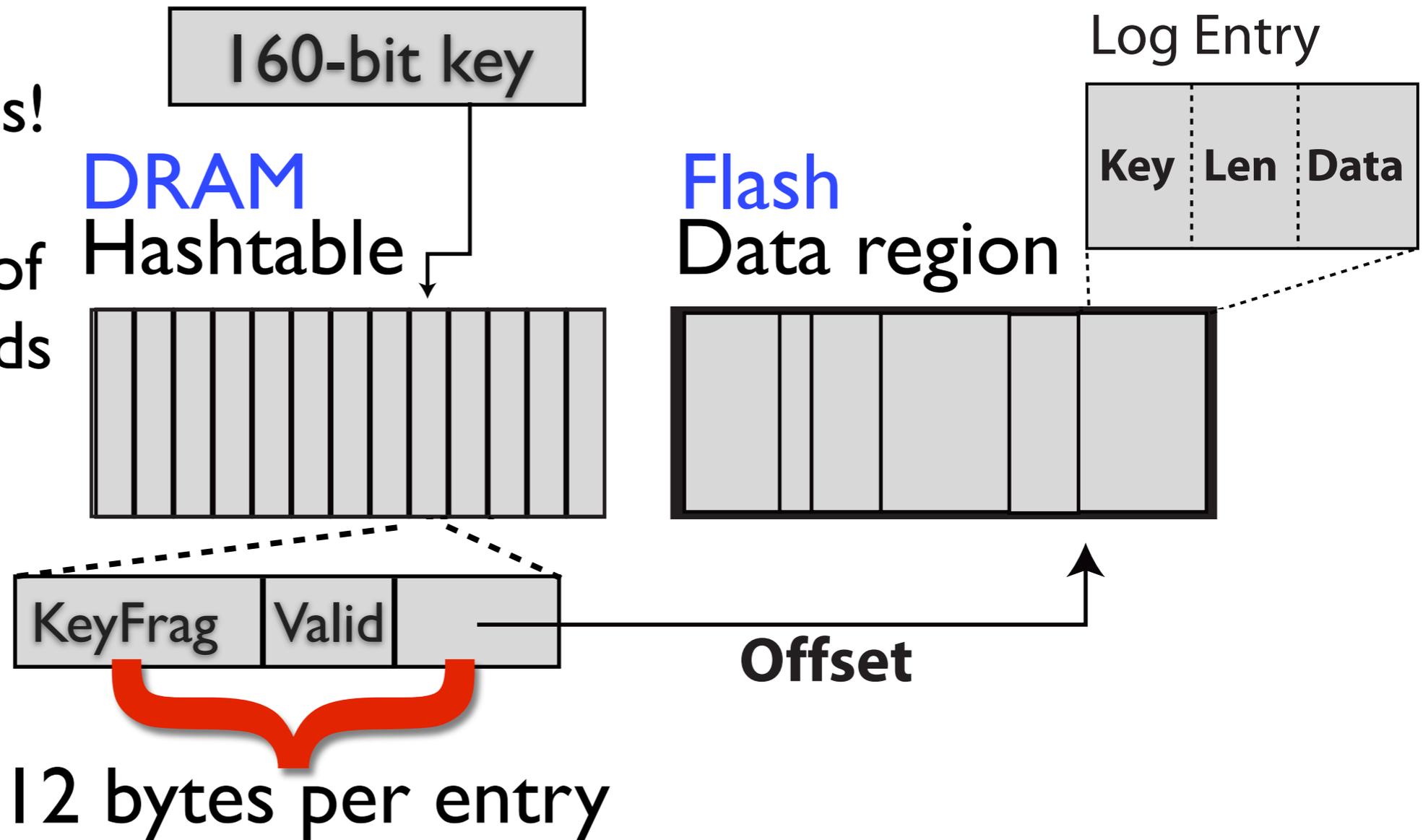
**All writes to Flash are sequential**

# With limited DRAM

KeyFrag != Key

Potential collisions!

Low probability of multiple Flash reads

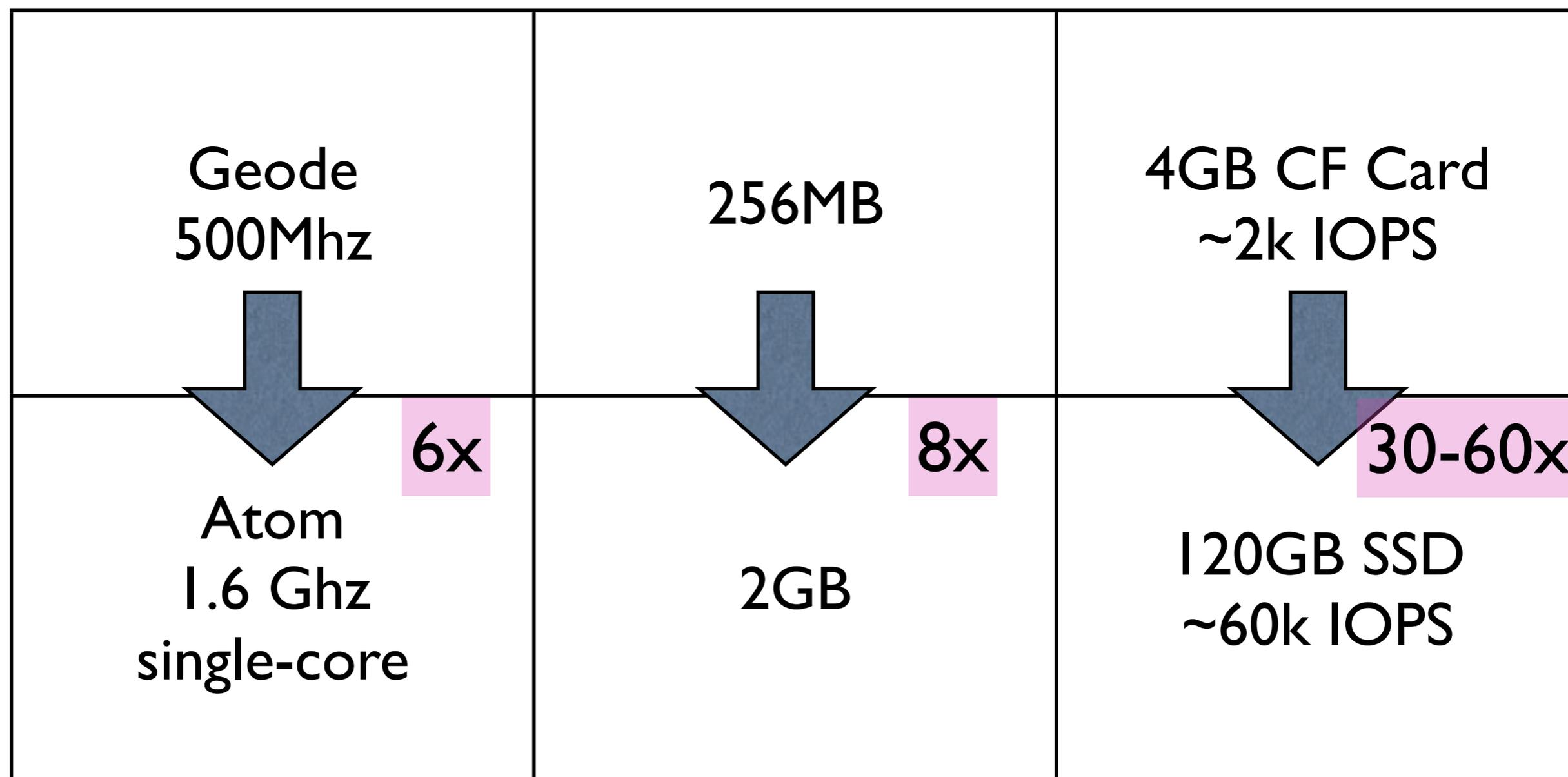


## Partial-key hashing

# 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



FAWN-DS

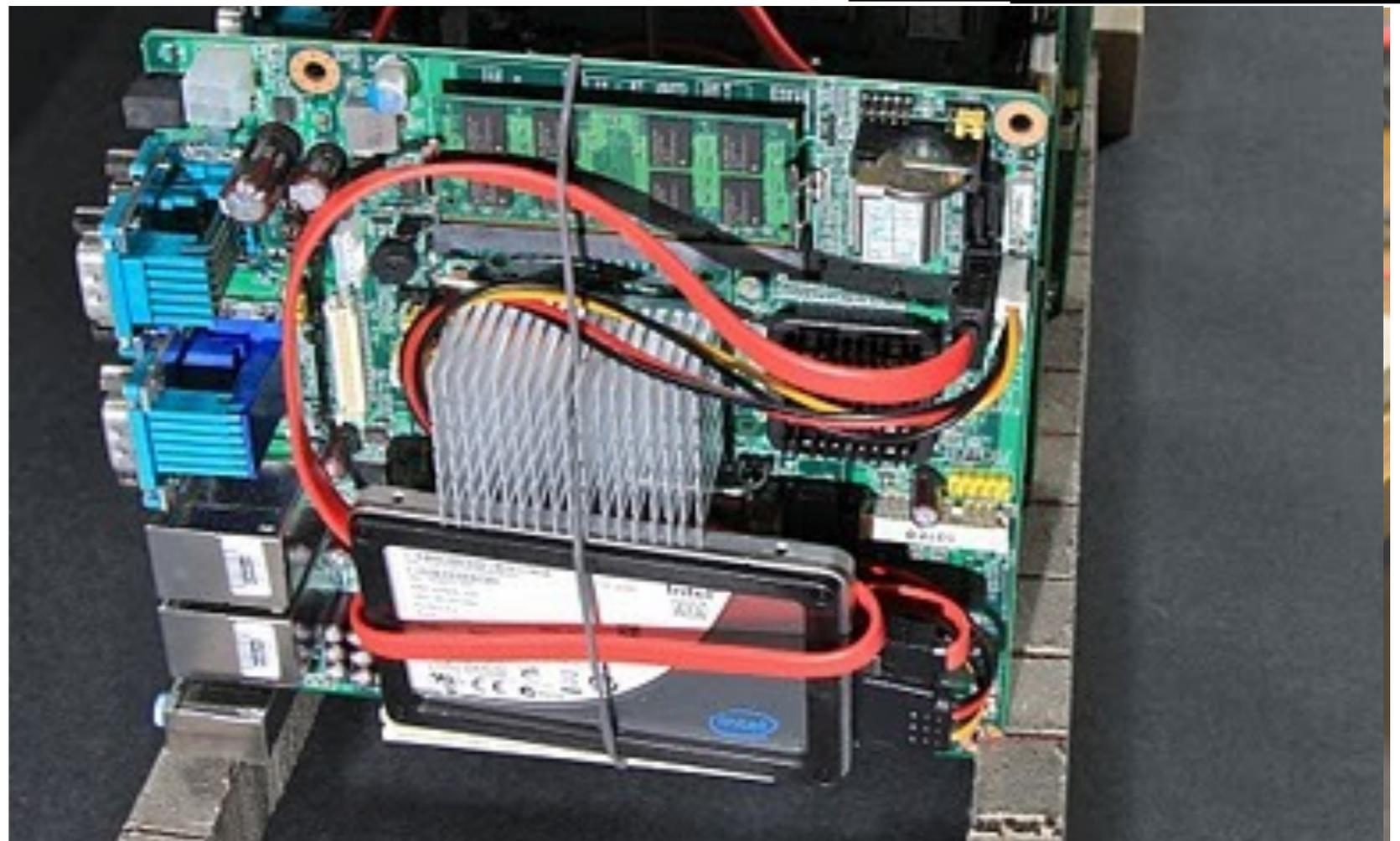
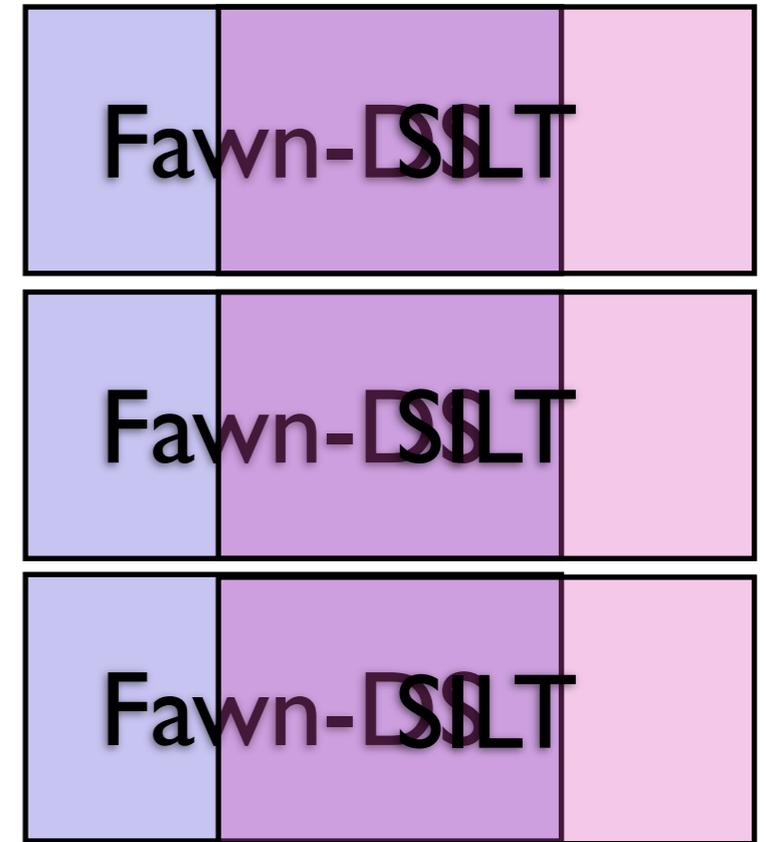
FAWN-KV

SILT

Small Cache

Cuckoo

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)

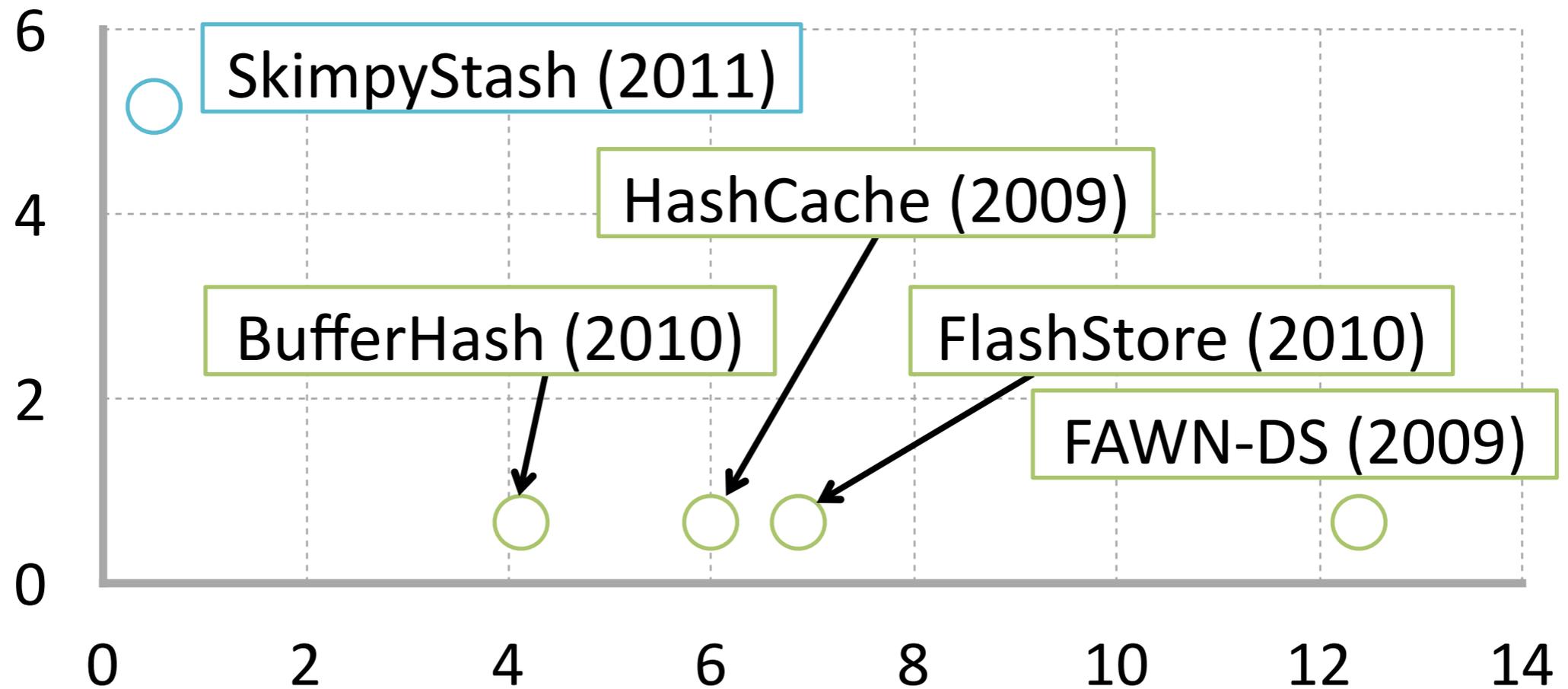
**Read amplification** = Flash reads per query

- Limits **query throughput**
- Ideally 1 (no wasted flash reads)

**Write amplification** = Flash writes per entry

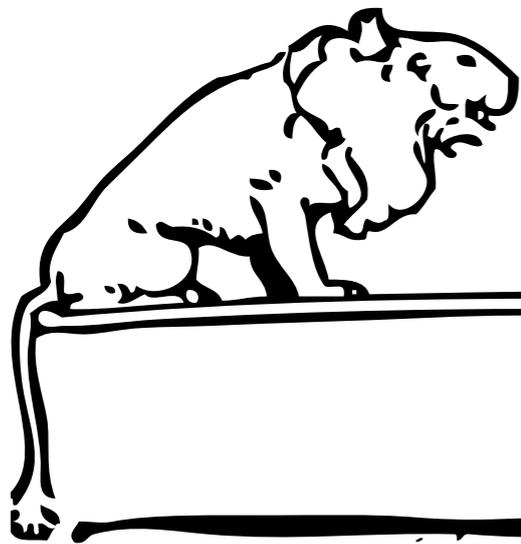
- Limits **insert throughput**
- Also reduces **flash life expectancy**
  - Must be small enough for flash to last a few years

## Read amplification



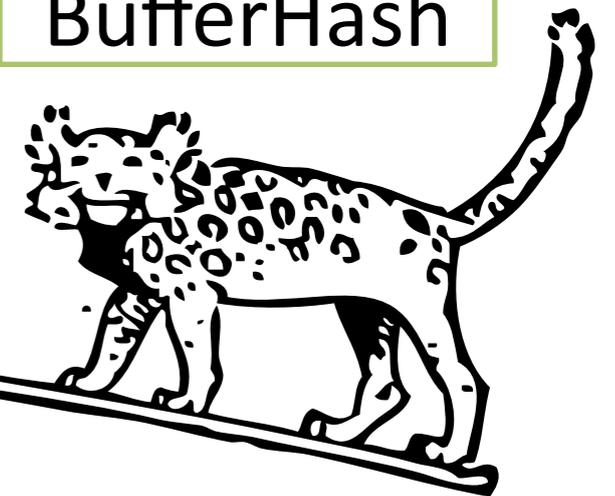
**Memory overhead** (bytes/entry)

SkimpyStash



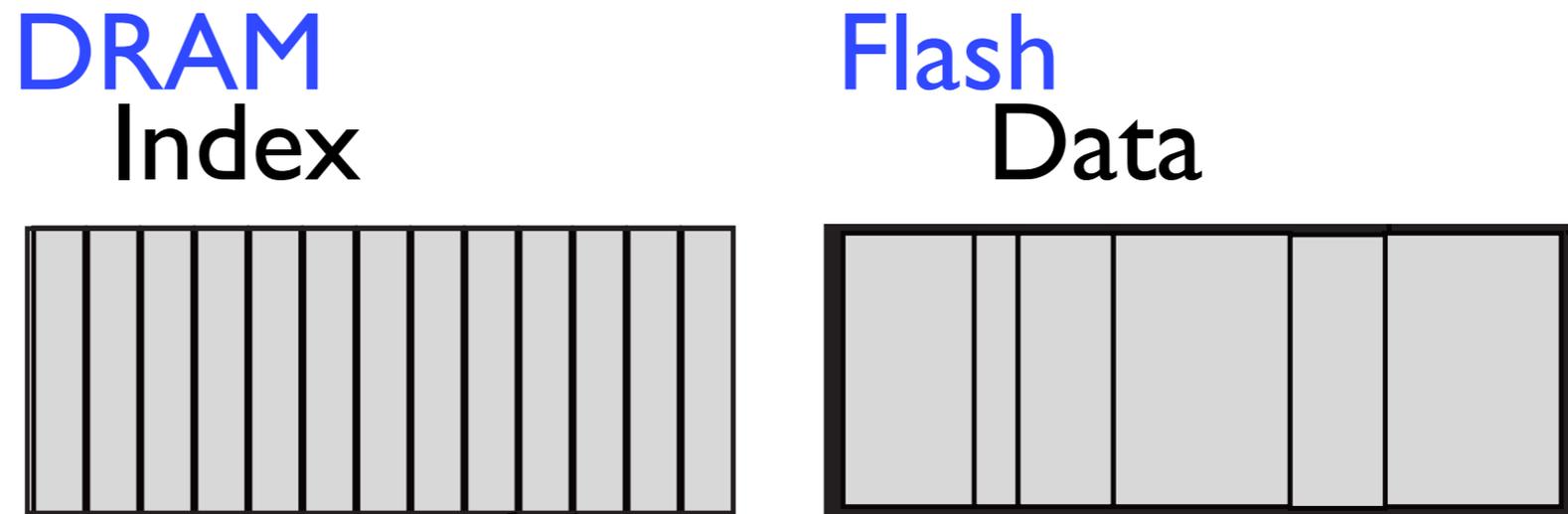
Memory efficiency

FAWN-DS  
FlashStore  
HashCache  
BufferHash



High performance

# (static) “External Dictionary”

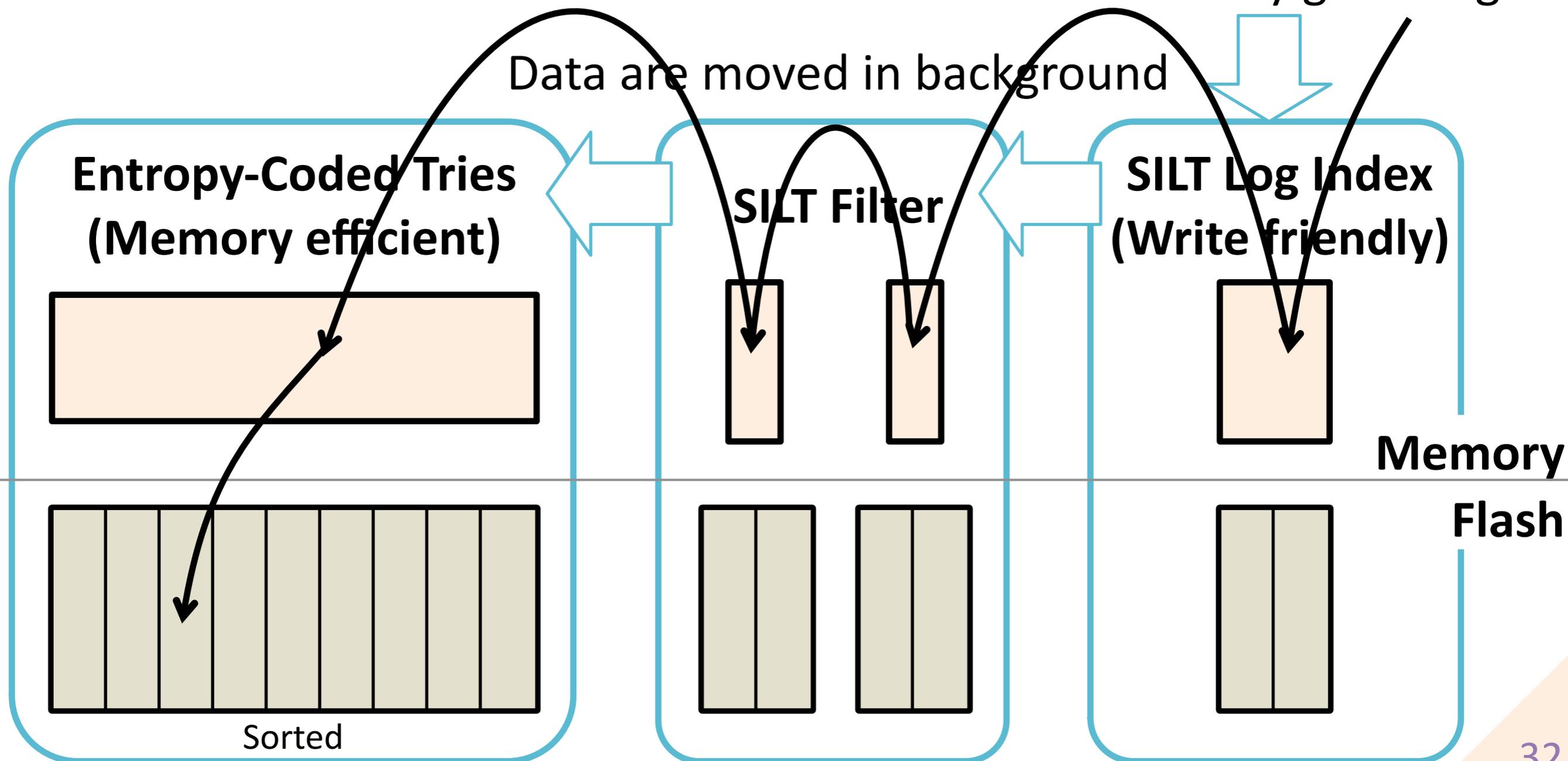


- Prior state of the art: “EPH”:  $\sim 3.8$  bits/entry
- Ours: Entropy-coded tries,  $\sim 2.5$  bits/entry
- Important considerations:
  - Construction speed; query speed
  - Aw, it’s read-only...

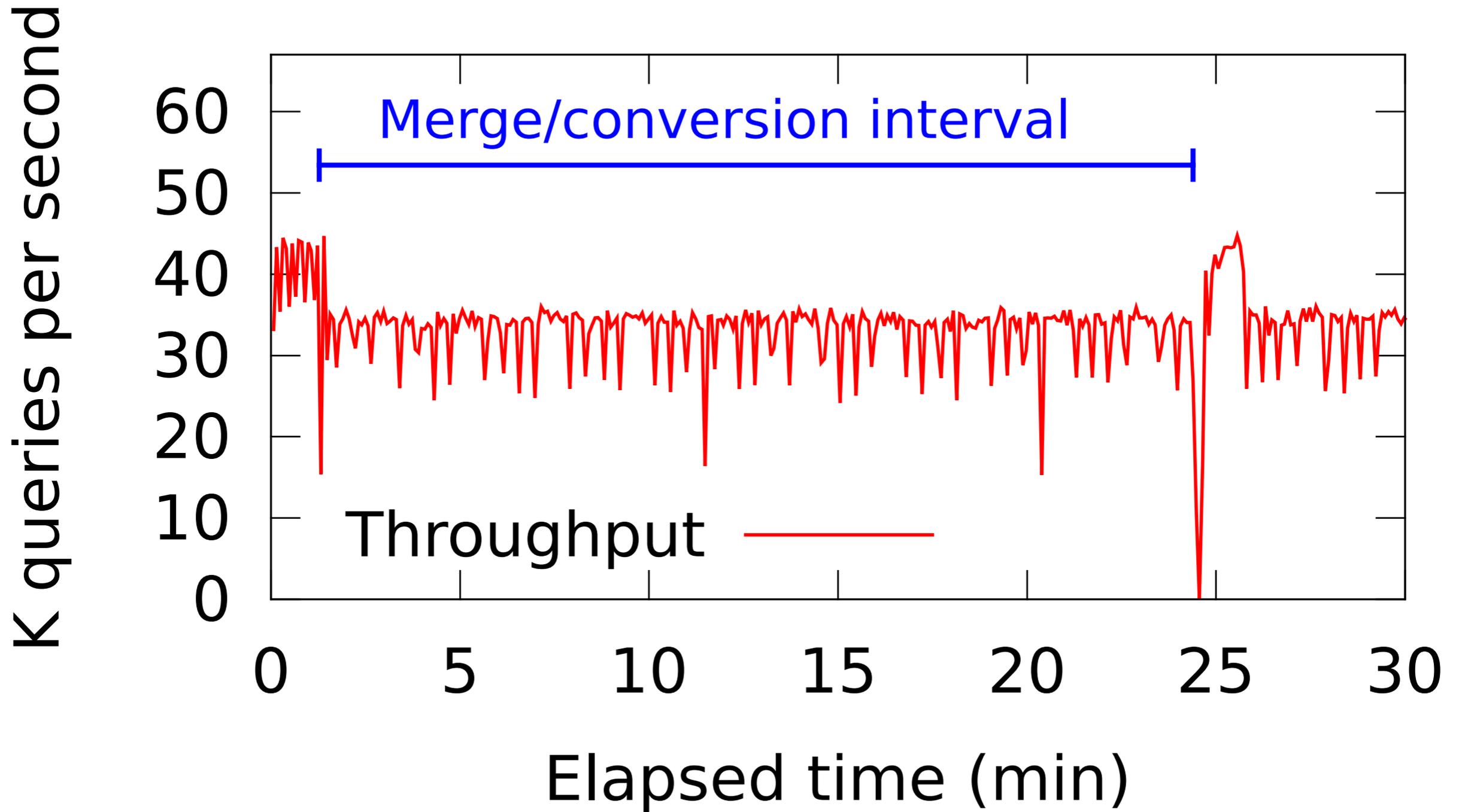
# Solution: (1) Three Stores with (2) New Index Data Structures

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

Inserts only go to Log



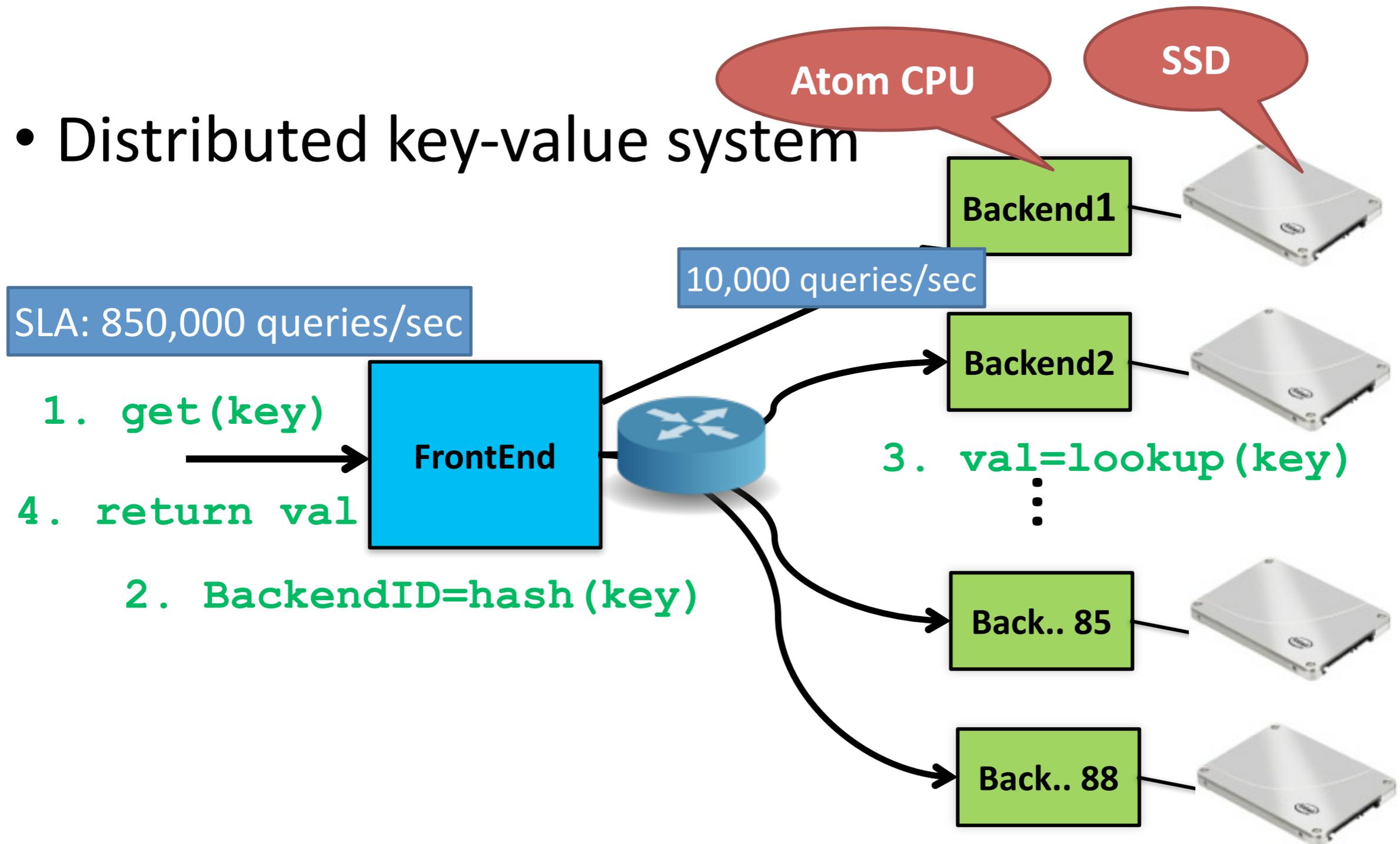
Workload: 90% GET (100~ M keys) + 10% PUT



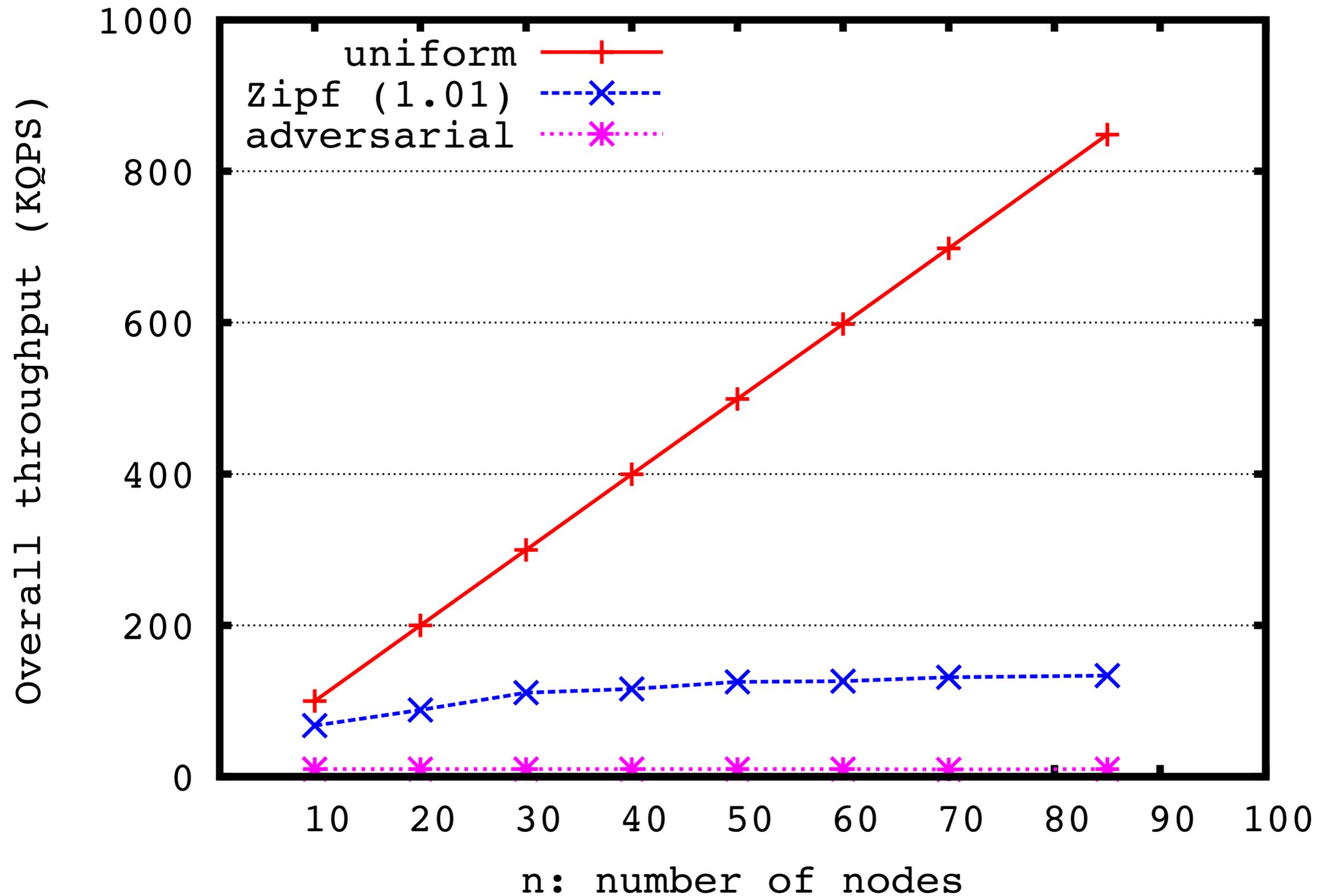
Caveat: Not on wimpies. Still working on reducing CPU cost! :-)

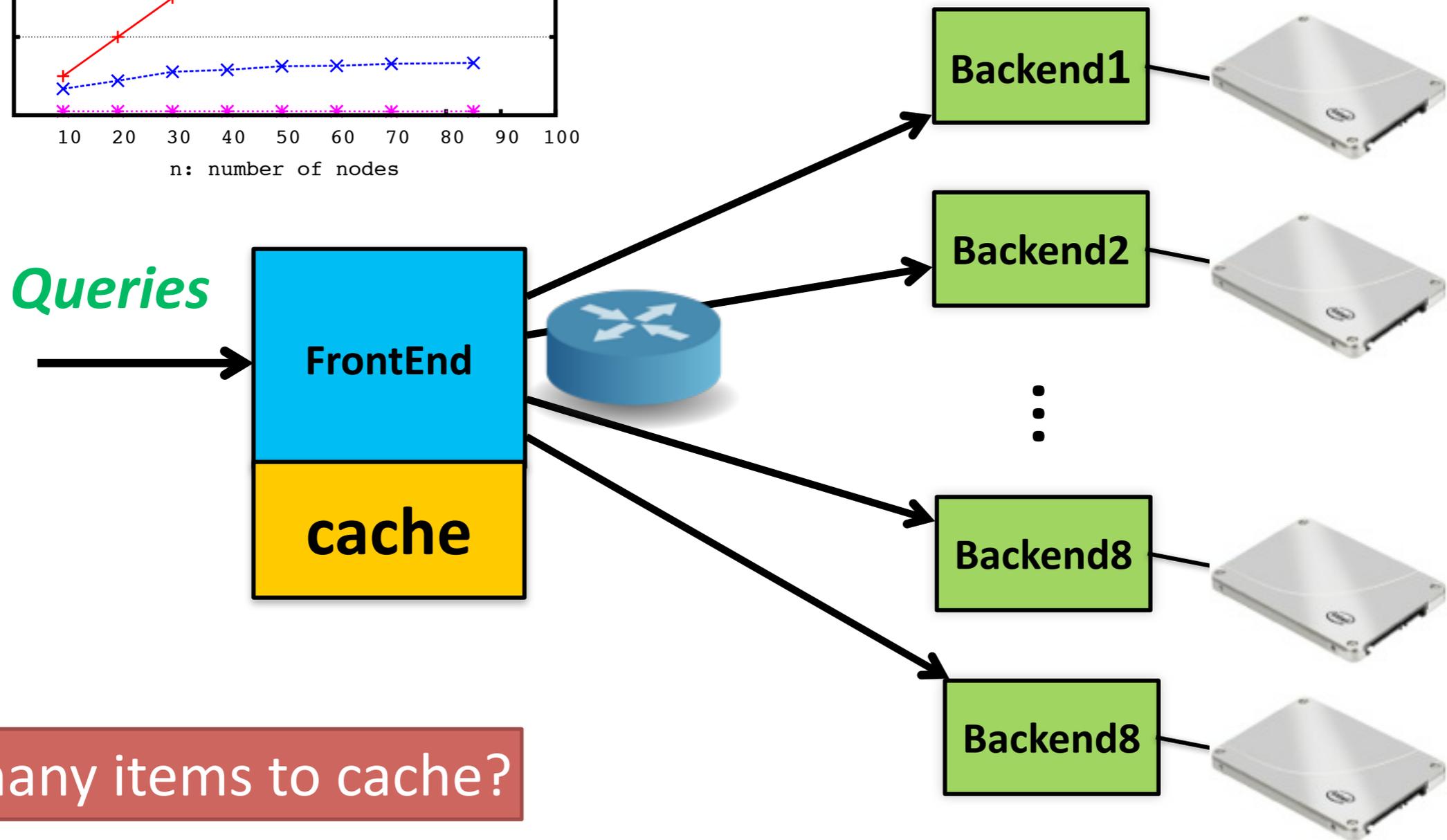
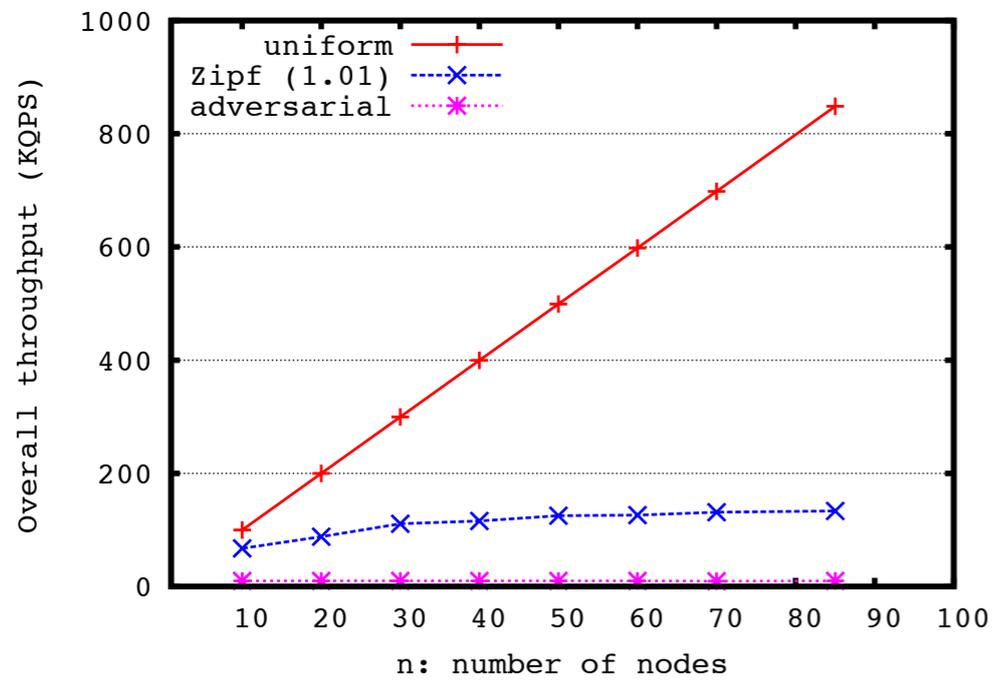
# And now... Load imbalance

- Distributed key-value system



# Measured tput on FAWN testbed



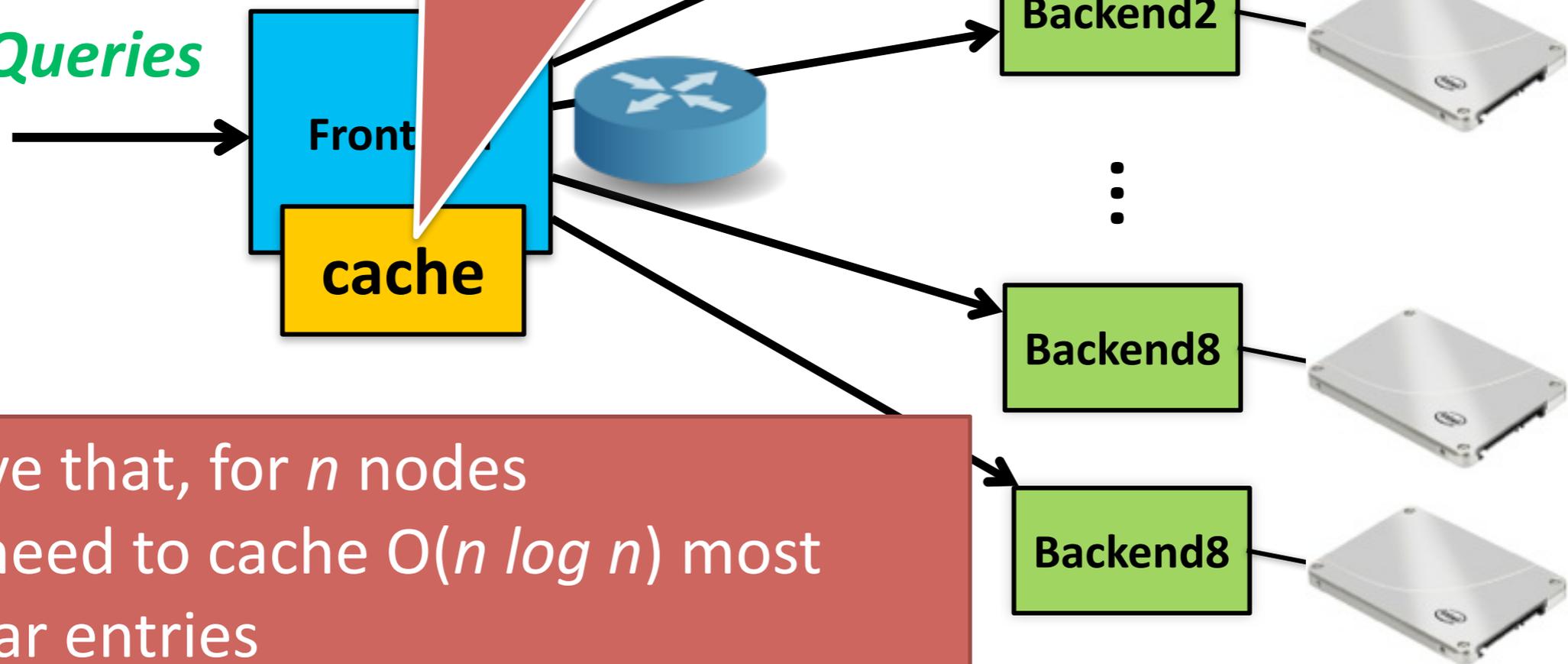


How many items to cache?

# small/fast cache is enough!

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

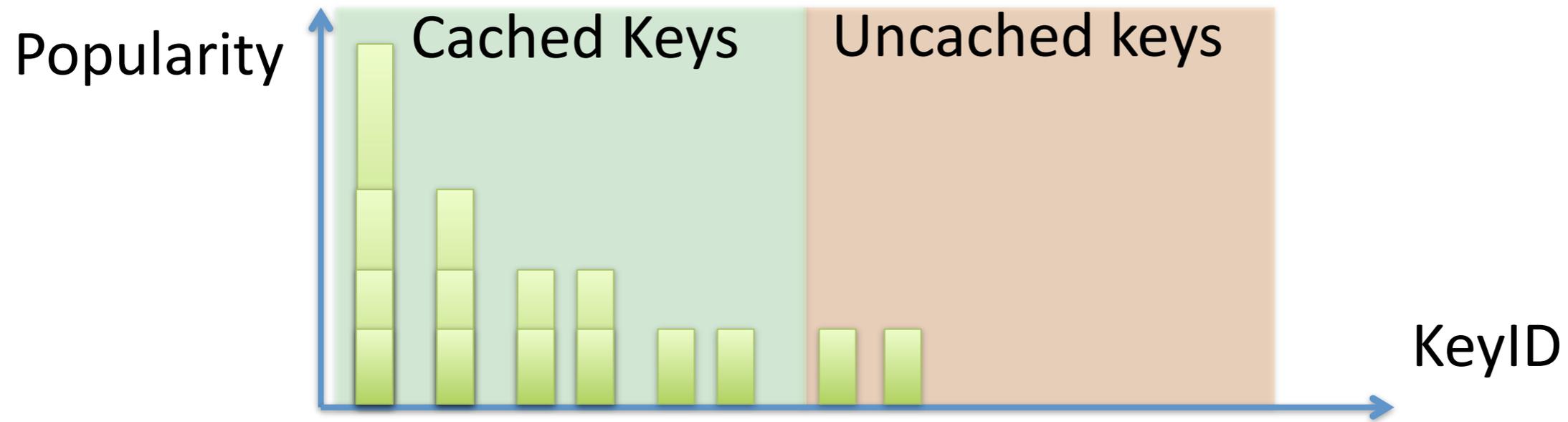
*Queries*



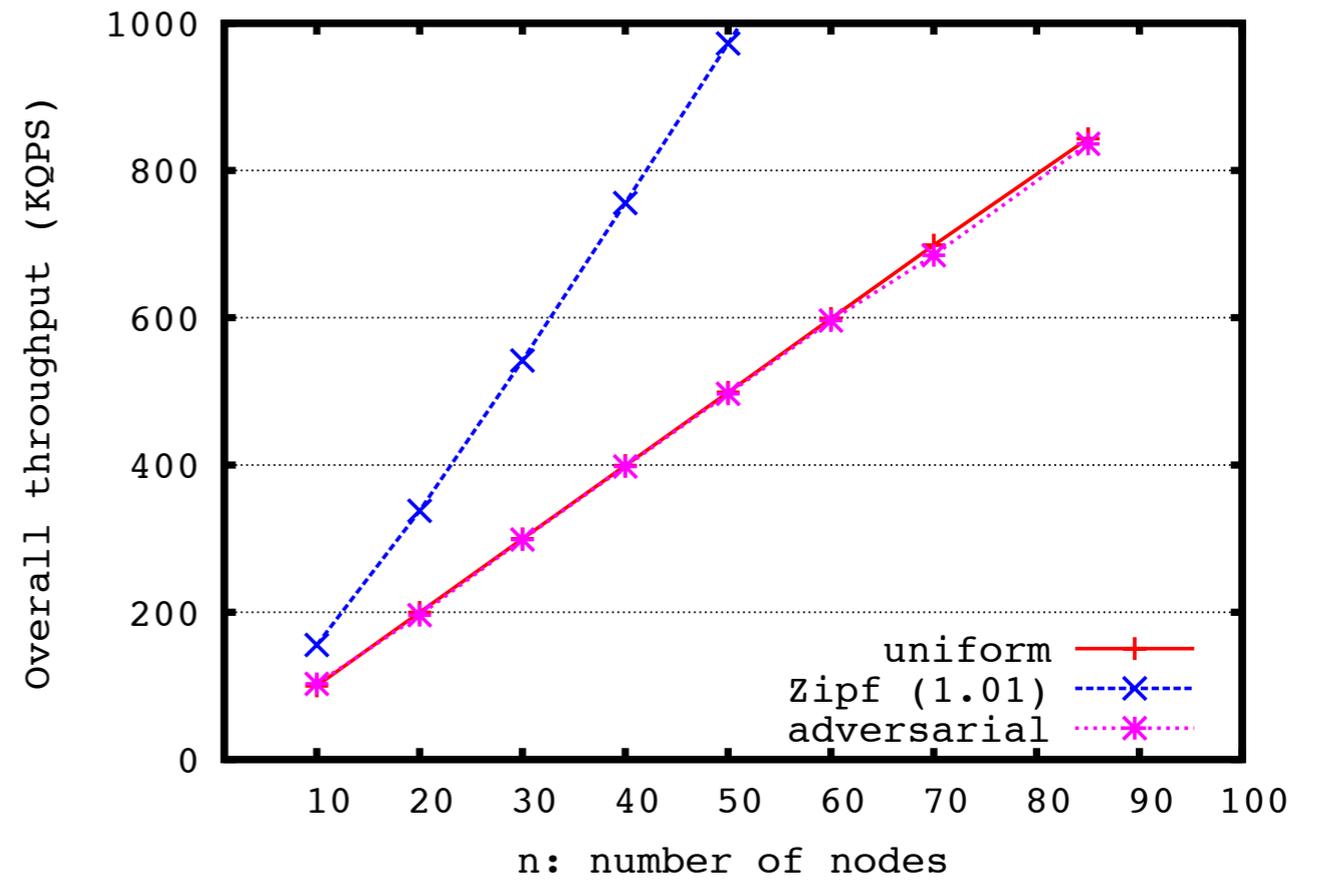
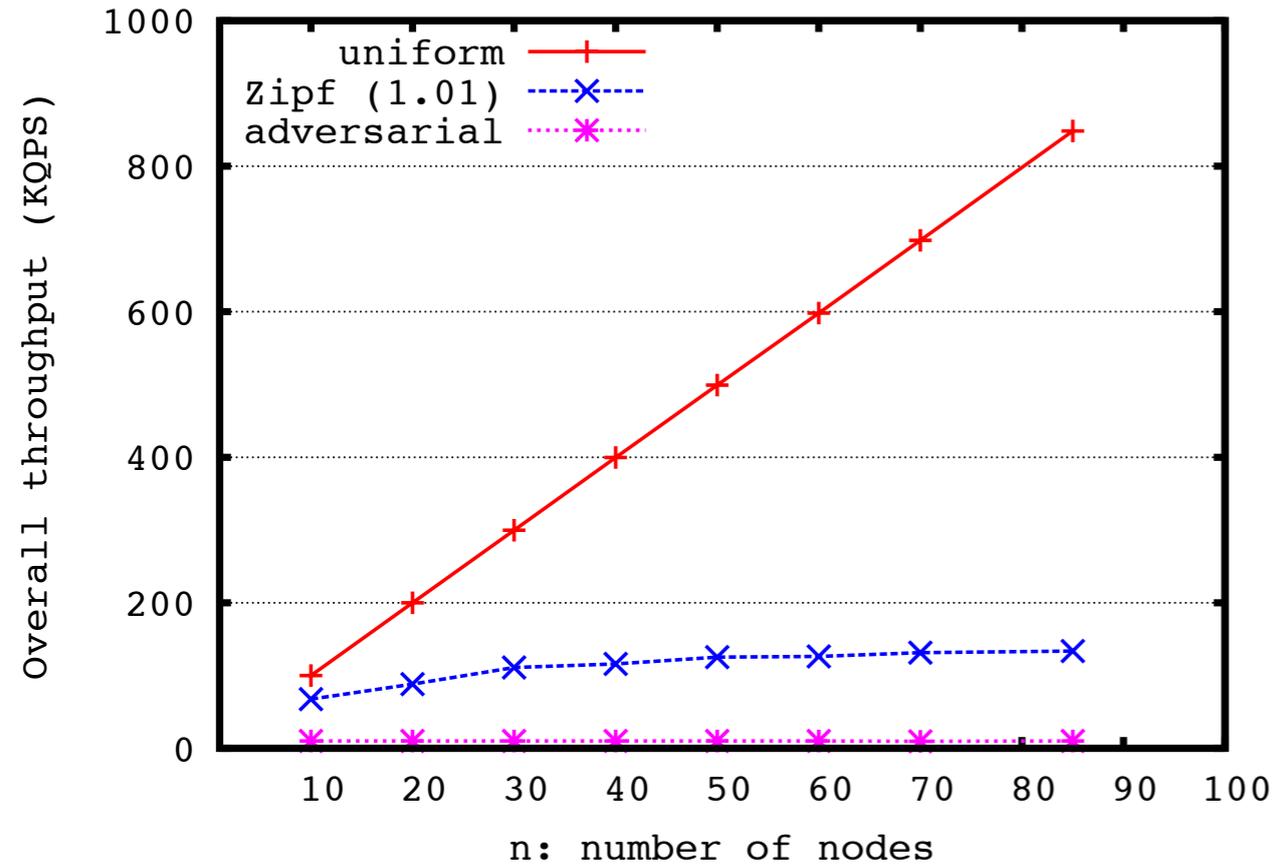
We prove that, for  $n$  nodes

- Only need to cache  $O(n \log n)$  most popular entries
- worst case perf. =  $(1 - \epsilon) * n * \text{single node capacity}$

# Cache forces near-uniform dist.



# Worst case? Now best case



**Thus...**

FAWN-DS

FAWN-KV

SILT

Small Cache

Cuckoo

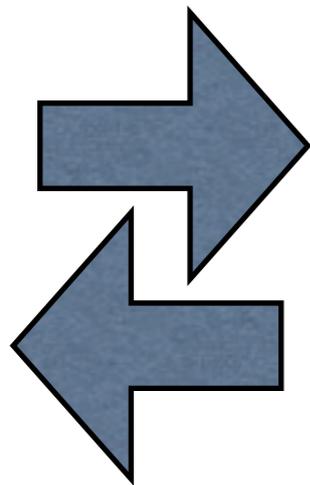
“Wimpy” servers

[FAWN, SOSP 2009]



[SILT, SOSP 2011]

“Brawny” server



$O(N \log N)$

[“small cache” socc 2011]

Multi-reader  
parallel cuckoo  
hashing

[under submission]

Entropy-coded tries

[SILT + under submission]

Partial-key cuckoo hashing

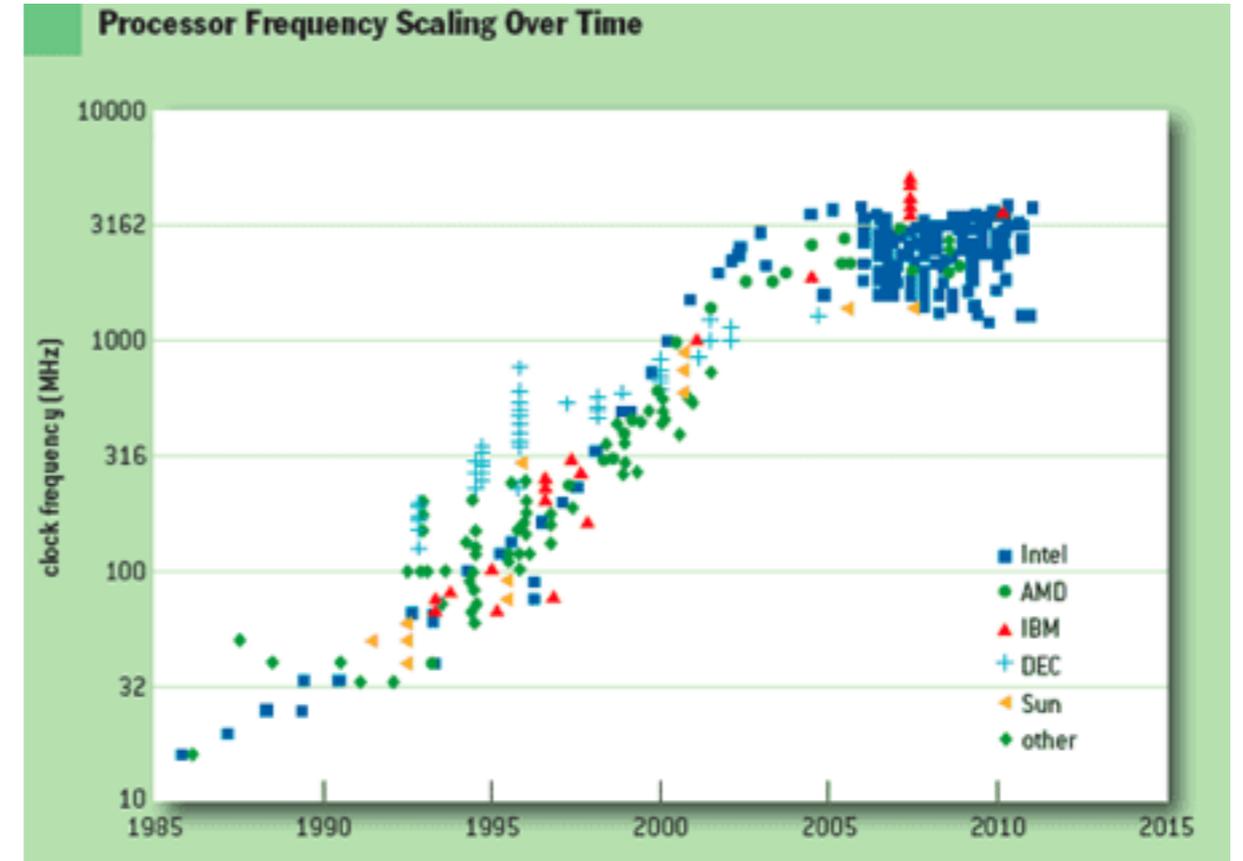
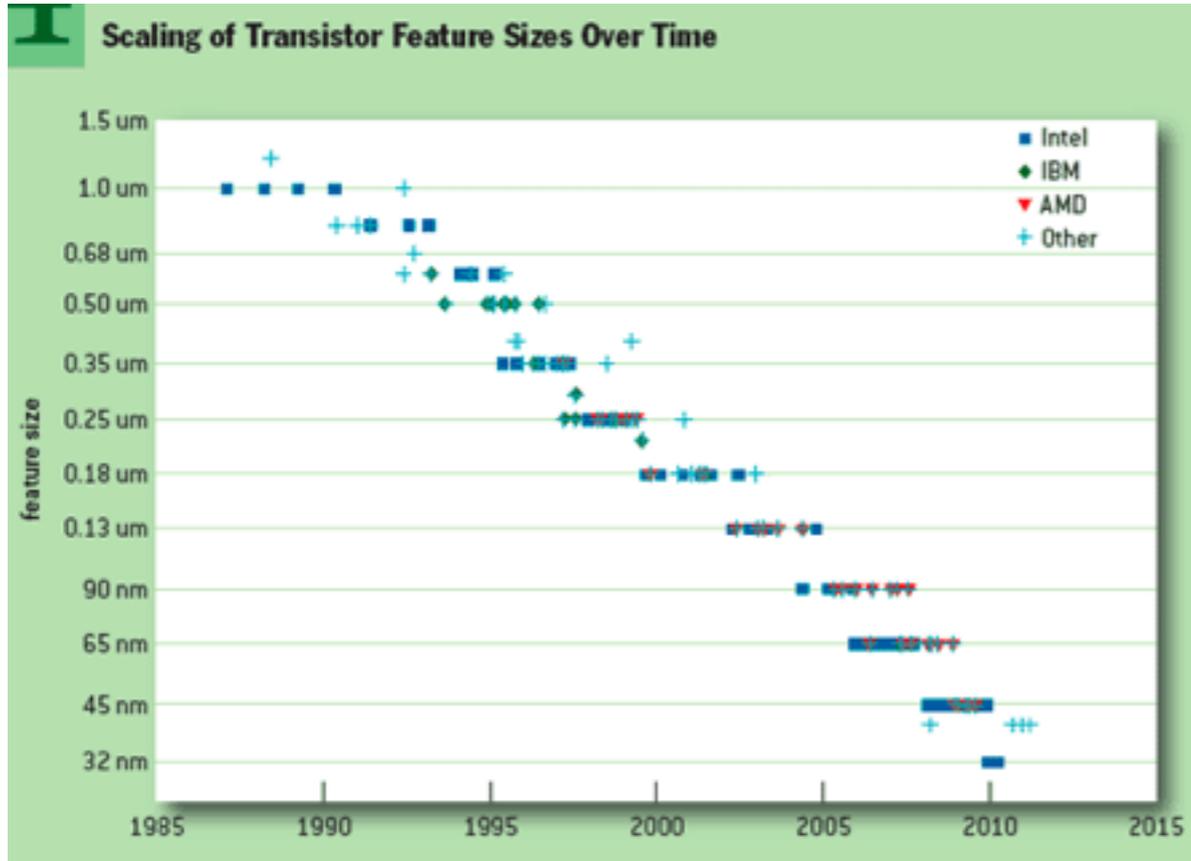
Cuckoo filter



Moore



Dennard



highly parallel, lower-GHz, (memory-constrained?):

*Architectures, algorithms, and programming*