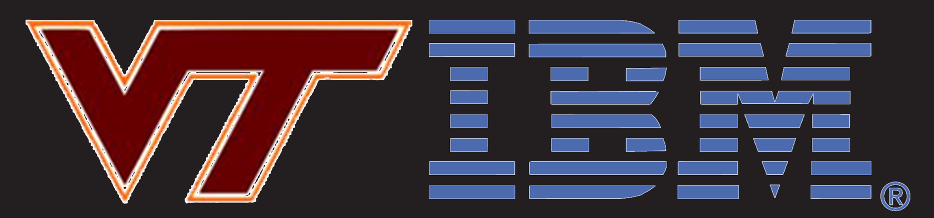
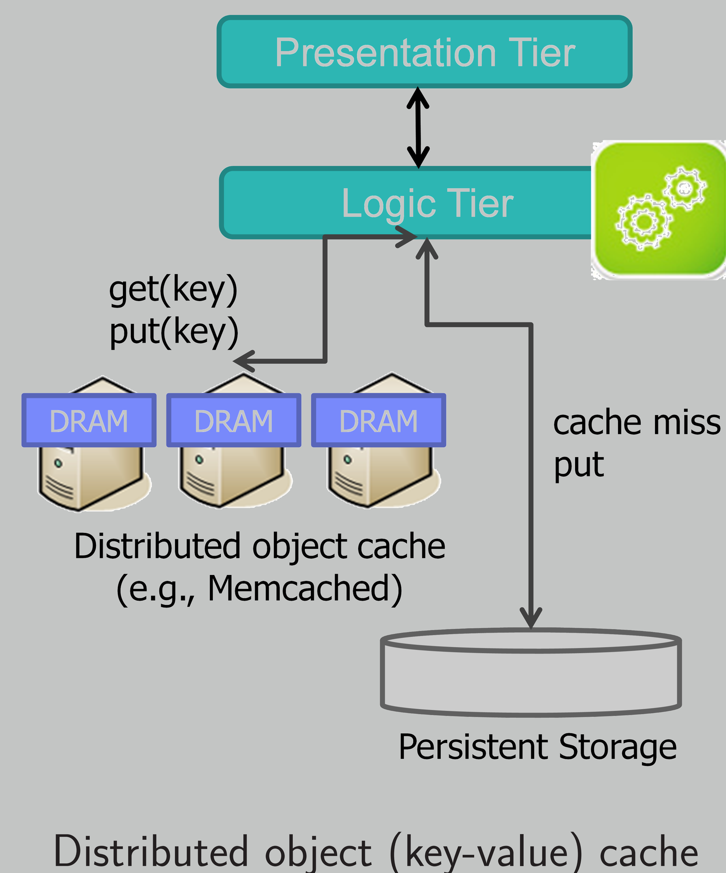
High Performance In-memory Caching through Flexible Fine-grained Services

Yue Cheng†, Aayush Gupta and Anna Povzner‡, and Ali R. Butt†
†Virginia Tech, ‡IBM Almaden Research Center
{yuec, butta}@cs.vt.edu, {guptaaa, apovzne}@us.ibm.com



Background

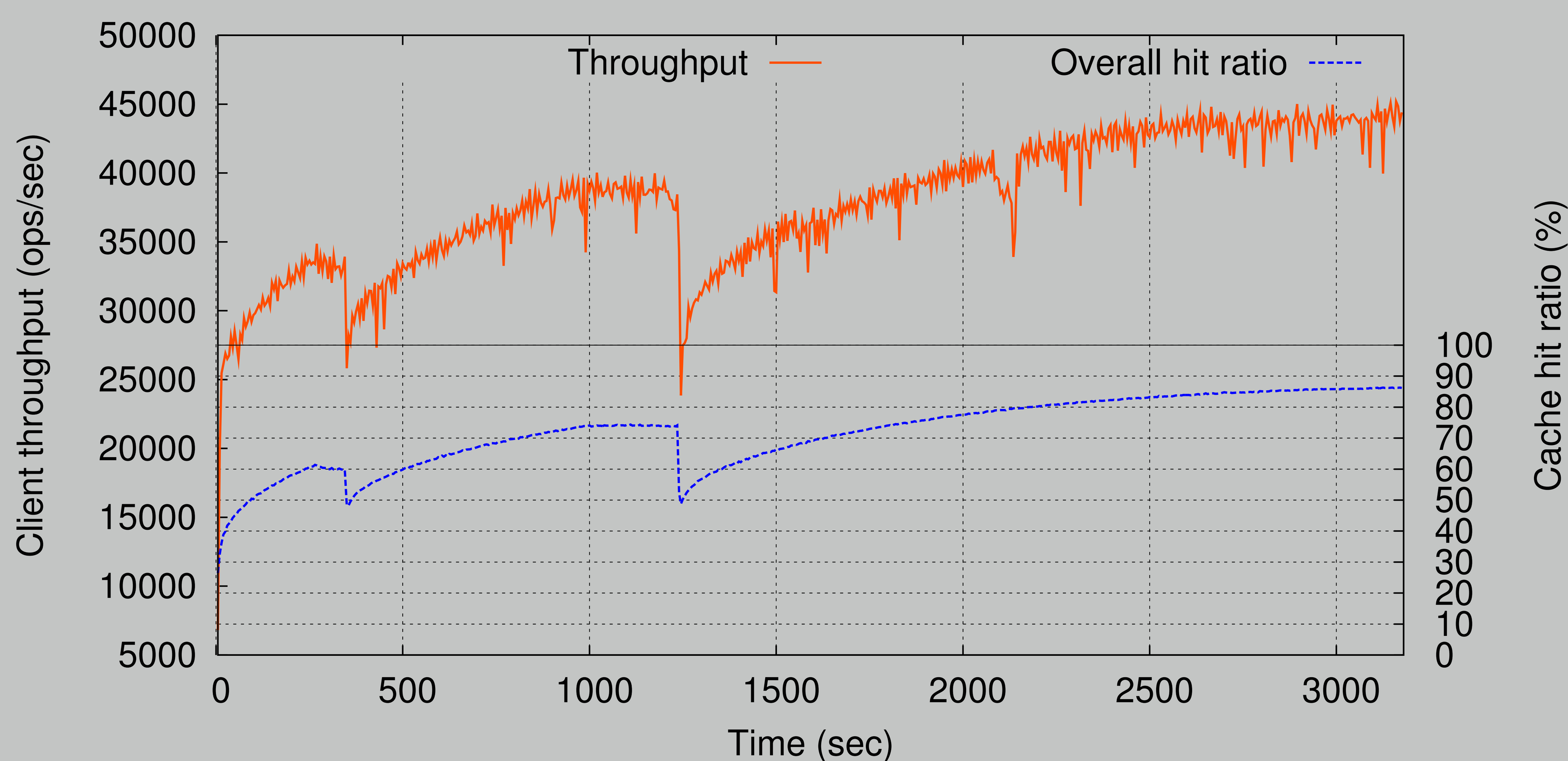
- In-memory object caches extensively used in public/private clouds and web installations
  - Low-latency access to data
  - Scalability
- The state-of-the-art
  - Amazon Web Service Elasticache
  - Facebook Memcache, TAO
  - Masstree [EuroSys'12]
  - MemC3 [NSDI'13]



Distributed object (key-value) cache

Motivation

- Most systems adopt monolithic storage models and engineer optimizations on specific workload characteristics or operations such as GET
  - The main focus of most optimizations is on performance improvement on one single dimension
  - Large-scale cloud workloads exhibit temporal and spatial shifts
- They either do not or support dynamic membership but with significantly high overhead
  - Cold cache warm-up causes intermittent performance degradation



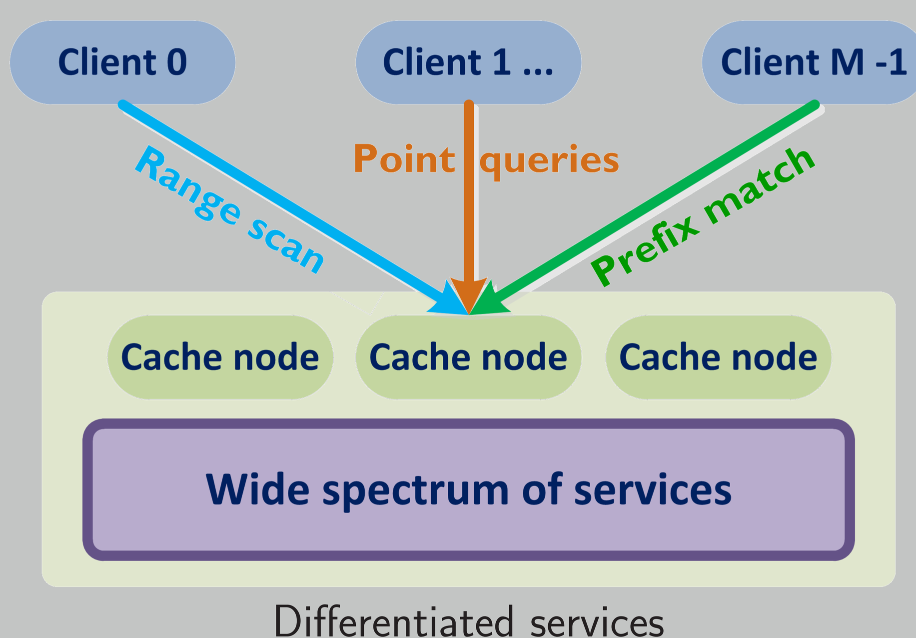
YCSB benchmarking with 10 GB data and caching tier enabled. Systems start up with 4 cache nodes. At sec 340 and 1240, 4 new cache nodes are added in respectively. While warming up, overall throughput reduces up to 41% and performance recovery takes up to 10 min.

The Idea

- The fine-grained modular design within one cache instance
  - Partition both data and metadata the independent entities called *Cachelet*
  - Hash table module, B+ tree module, trie module, etc.
  - APIs (services):
    - `GET()`
    - `SET()`
    - `RANGE_SCAN()`
    - `PREFIX_MATCH()`
    - etc... Flexible, customizable, extensible

What other benefits can the system get from the fine-grained modular design?

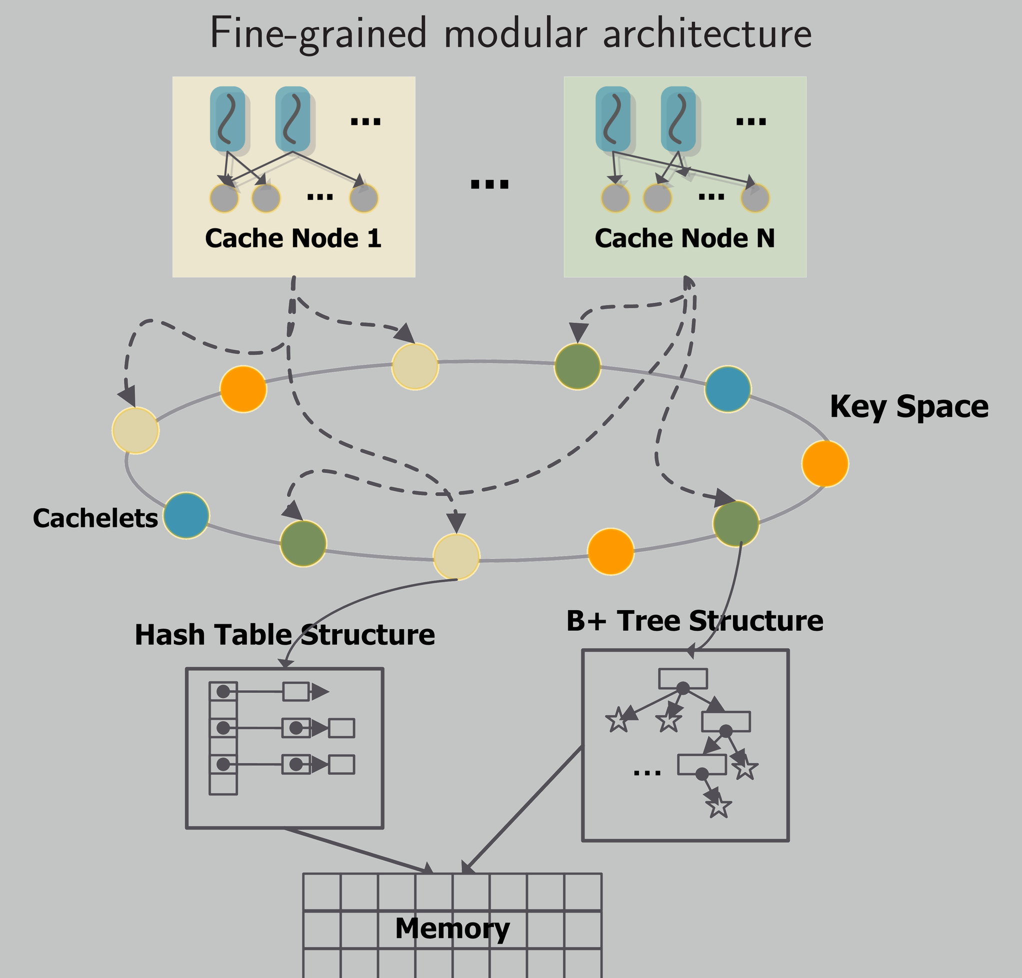
- Enables seamless per-instance resource re-provisioning and low-overhead dynamic membership management



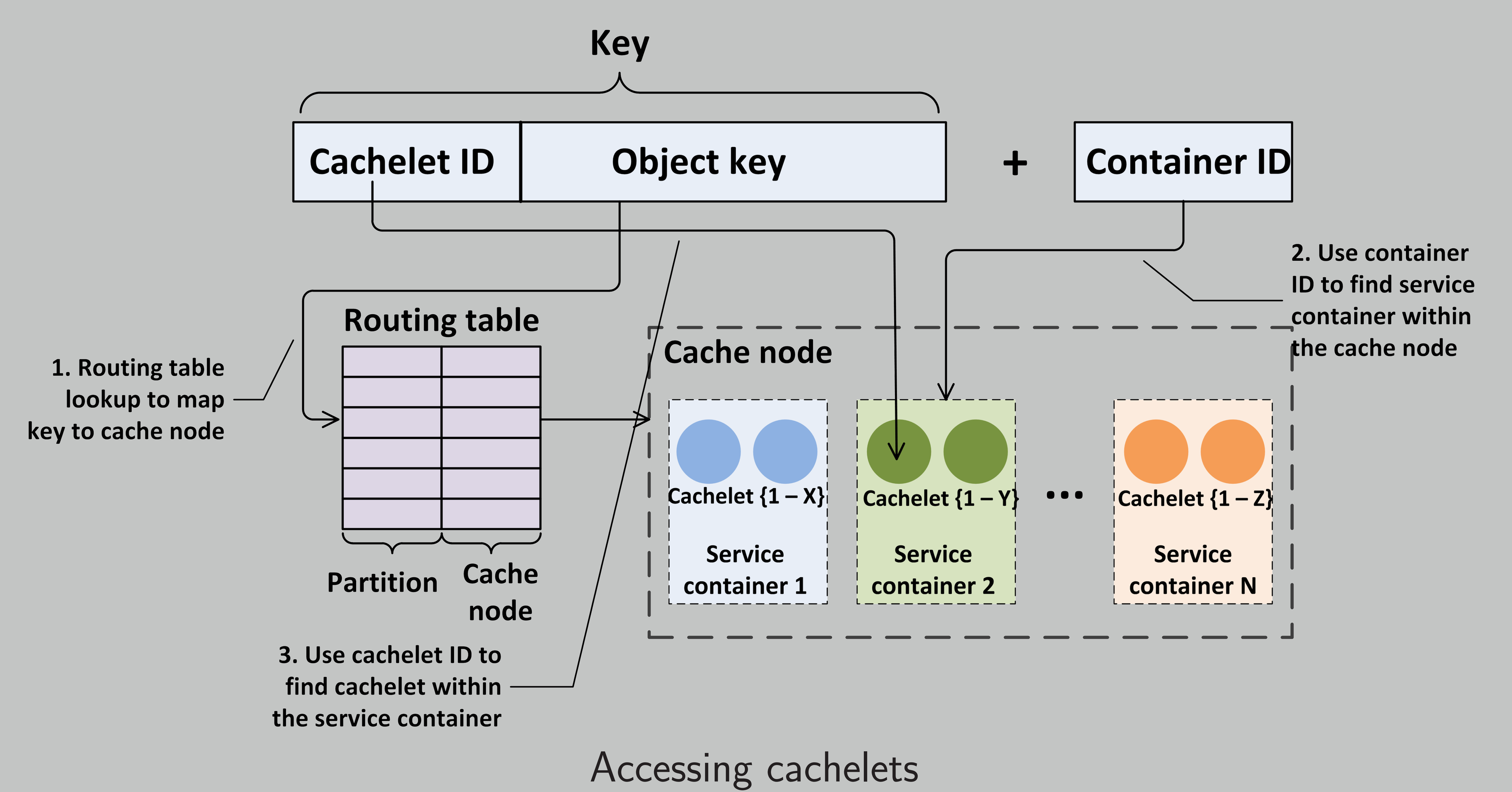
Differentiated services

System Design

- Each cache instance is a "fat server" comprising multiple service abstractions
- Cachelet type abstracts the service provided to the clients
- Services (query types, resources allocated: CPU, DRAM etc.) are configurable
- Data is stored in relatively small partitions spanning multiple cachelets



⇓



Accessing cachelets

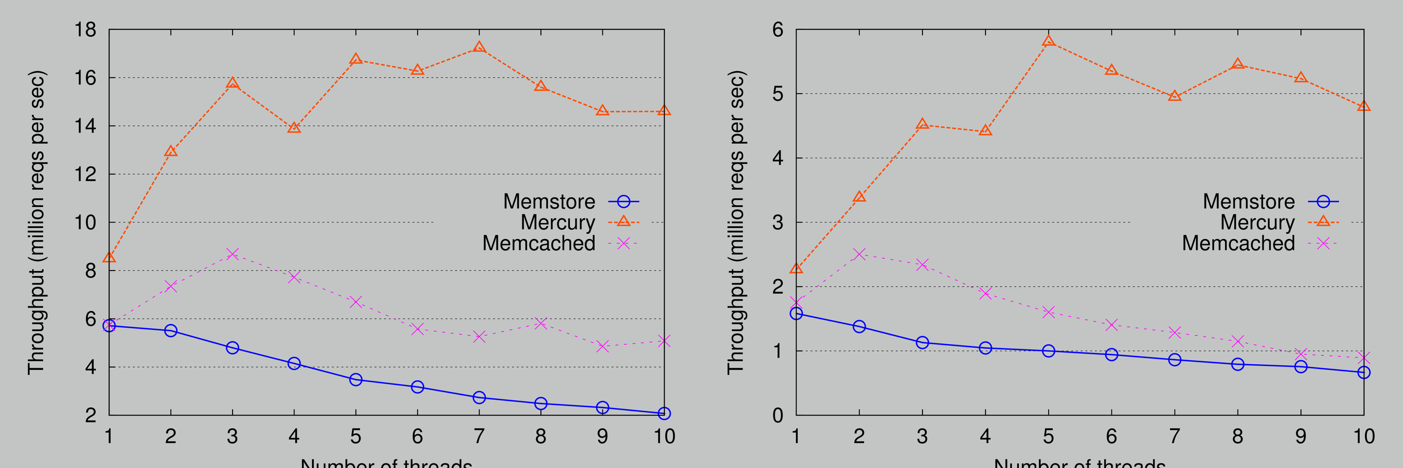
⇐ Naturally enables smooth warm-up transition!

How?

- Client-side routing to reduce logic complexity of the cache side
- Optionally support ordered partitioning in key space (tweaking the consistent hashing) for range queries
- Seamless per-instance resource re-provisioning
  - Priority-driven resource multiplexing - high resource utilization
- Low-overhead dynamic membership management
  - Data migration in granularity of cachelets - efficiently elastic
  - Lazy client view update - upon the completion of cachelet migration, old cache nodes respond with the updated view

Major module performance evaluation: Hash table

- Experiment setup (point query, client aggregators operating mode for avoiding network overhead)
  - 6 core, 2.67 GHz, 12 GB DRAM
  - Memstore: our lock-free hash table; Mercury: Memcached hash table with fine-grained bucket locks; Memcached: original 1.4.13 version



(a) GET performance (Zipfian) (b) SET performance (Zipfian)

Current Status

- Integrate B+ tree and trie module into the system
- Implement client side simplistic consistent hashing + data migration scheme
- Build different case studies to demonstrate the benefits of our cache framework