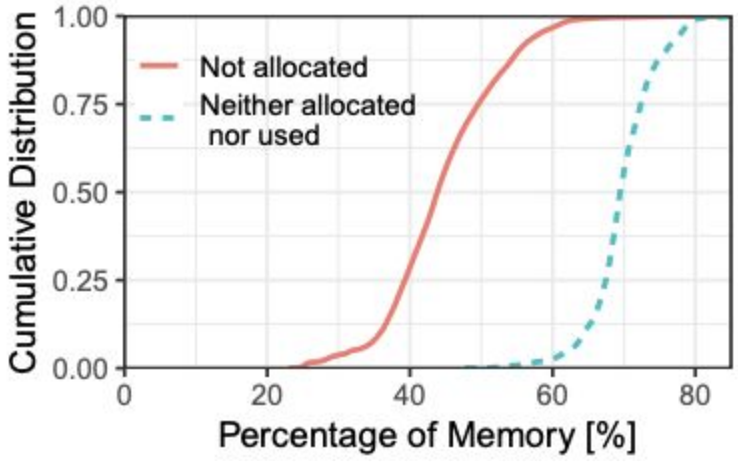
CompuCache : Remote Computatble Caching using Spot VMs

Remote Caching With Spot VMs

- Data intensive applications benefit from large in-memory caches
  - Memory is very expensive
  - Limited local memory and workload changes

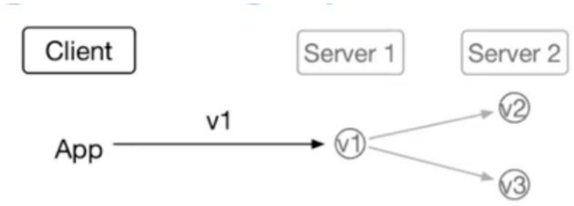


- Massive amount of unused memory in data centers
  - External fragmentation due to bin packing.
  - Internal fragmentation due to overprovisioning.

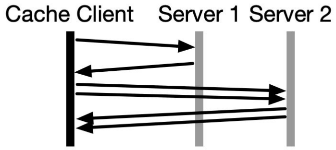
- Spot VM characteristics
  - Pros : Cheap and fast
  - Cons: fragmented and unreliable.

Limitations of Existing Caching Systems

Query example: Graph traversal

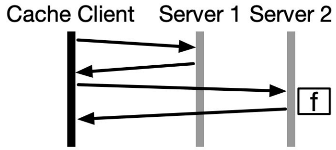


- I/O (r/w) interface
  3 roundtrips : 1 for dereferencing 2 for pointer chasing



(a) I/O-only interface.

- Key-value interface
  2 roundtrips : 1 for dereferencing 1 for pointer chasing

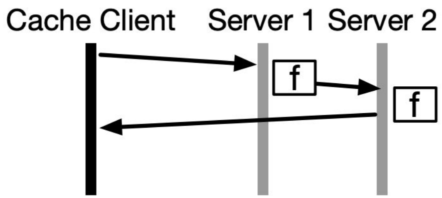


(b) Predefined keys.

Problem : Network is a major bottleneck

CompuCache Approach

- Support the ideal interface for remote caching using Spot VMs
  - Compute offloading with server side pointer chasing.

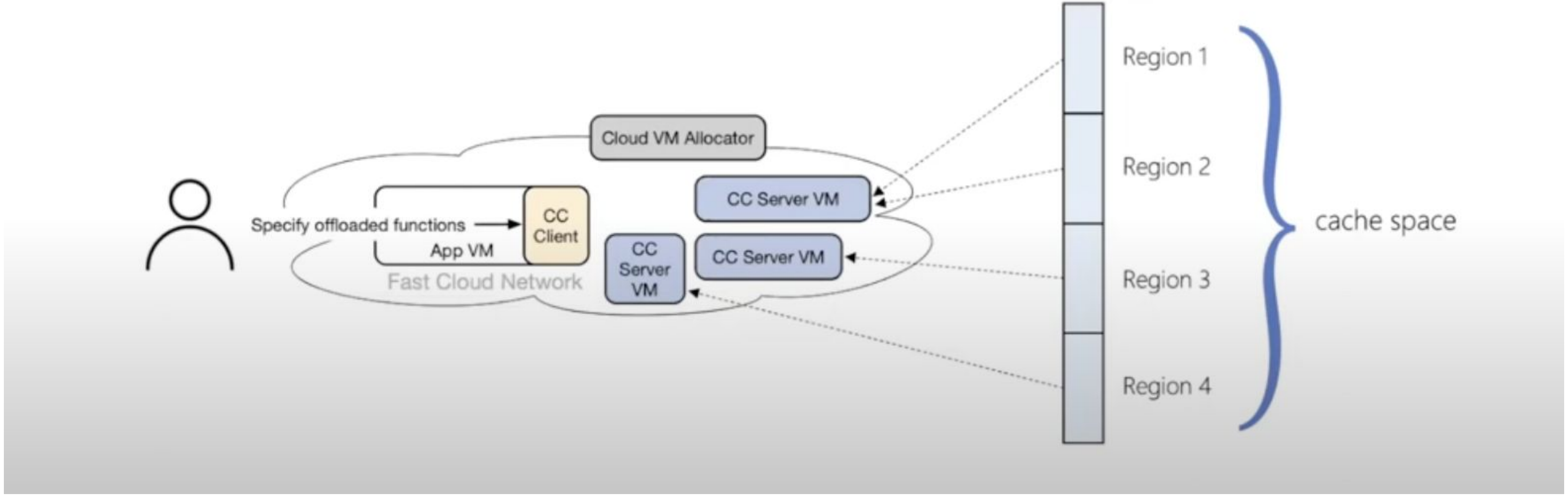


(c) Ideal interface.

- Handle the churn of Spot VMs

CompuCache Overview

Remote Caching + Compute offloading using Stored Procedures (sprocs)



Interface Design

- Challenge #1 : Deciding number of cores when allocating cache space
  - Core count affect runtime of sprocs
- Solution : user specified performance target and runtime CPU adjustment

- Challenge #2 : Server side pointer chasing
- Solution: ***LocalTranslator*** for Sproc implementation

$$\texttt{l\_addr, l\_size} \leftarrow \texttt{Translate(c\_addr, c\_size)}$$

Interface Design

- Challenge #3 : Out of Bounds Exception
  - Compucache is distributed in nature across multiple Spot VMs and a VM may not have the data requested on the server

- Options:
  - Data Shipping : Flow input data from remote VM using **Dflow**

```
l_addr ← DFlow(c_addr, c_size)
```

  - Function Shipping : Ship the execution sproc to remote VM using **FFlow**

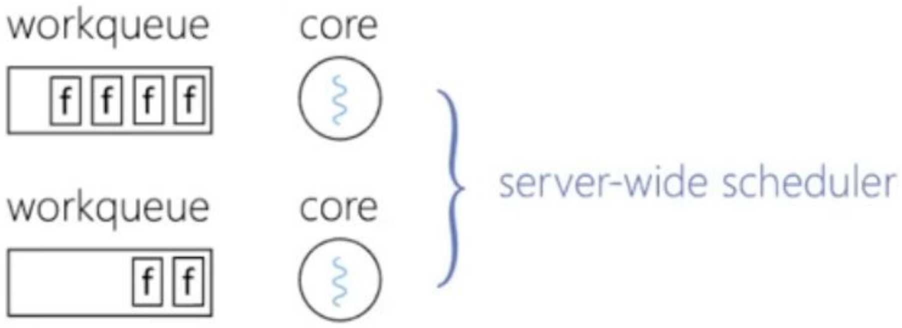
```
FFlow(c_addr, c_size)
```

Execution Design

- Challenge #1 : Fast request / response delivery
  - Traditional networking stack suffer from kernel overheads
  - Sproc requests are small in size but responses may vary in size (aggregate vs scan)
- Solution:
  - eRPC, a user-space RPC library using RDMA
  - Adaptive message batching.

Execution Design

- Challenge #2 :Sproc Scheduling
  - Many sproc requests arrive at the same server
  - Different sprocs have different execution times
  - Sprocs may run into out of bounds exception
- Solution : Have a work queue for each core and a server-wide scheduler



Execution Design

- Challenge #3 : **LocalTranslator, DFlow, FFlow** implementation
  - How to construct LocalTranslator data structure.
  - How to execute DFlow and FFlow requests.
- Solution:
  - Client side mapping : Cache regions -> Server VMs
  - Client broadcasts this mapping to all server which constructs LocalTranslator and route DFlow and FFlow requests.
  - Implement DFlow as read requests and FFlow as sprocs.

Fault Tolerance

- Spot VMs can be reclaimed by the cloud provider.
  - Cache migration
  - Cache region mapping update
  - Existing requests routing

Evaluation

- Simple Procs - read and check. Compared to Redis Sproc using **eval**

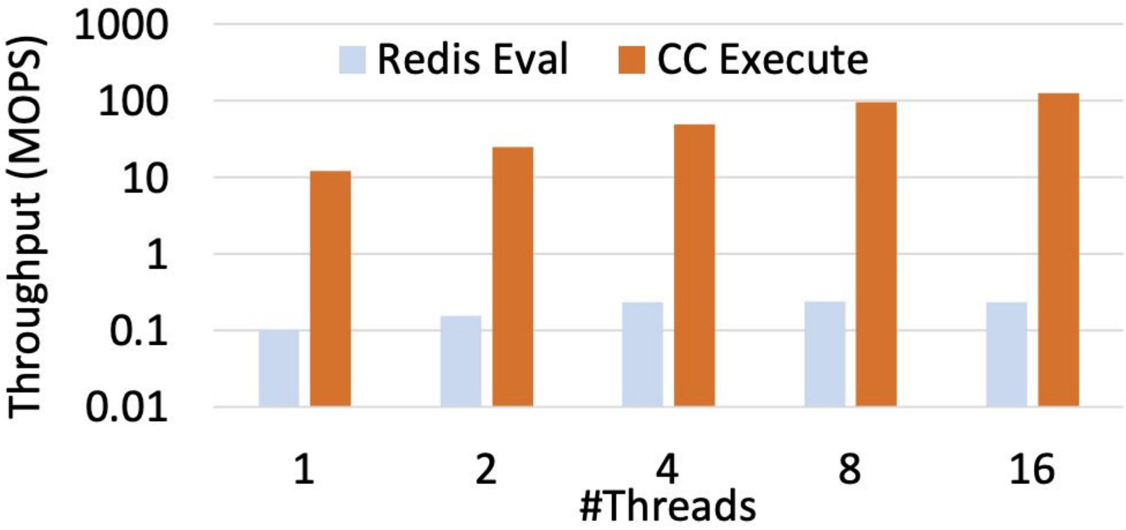


Figure 6: The performance of executing simple sprocs.

Evaluation

- Sproc to aggregate 3 records.

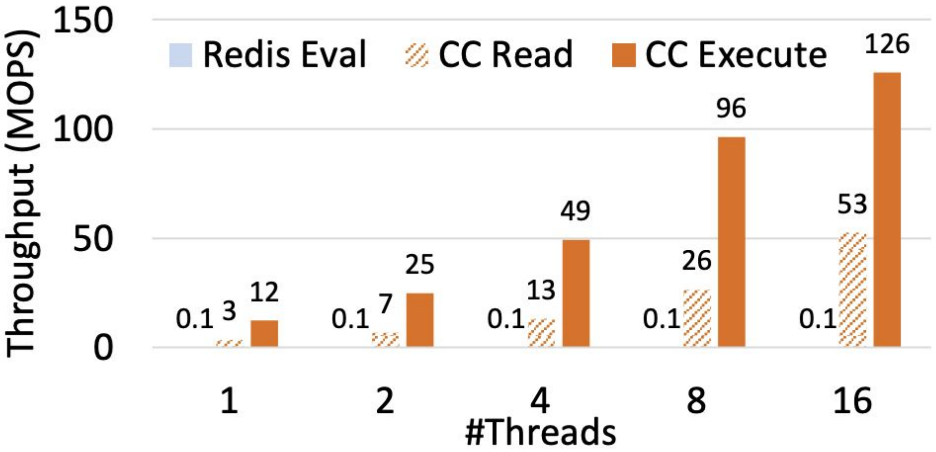


Figure 8: The performance of aggregating three records.

Thoughts

- Interesting Idea to use underutilized resources which is cheap for expensive computation.
- Application Driven design : A lot of onus on the author of the sprocs.
- Performance comparison : Claims to be 200x faster than Redis for even simple I/O. Not much reason mentioned on why Redis is so slow.
- No Key / Value Abstraction
- No discussion on what happens when a server fails unexpectedly. (Not reclaimed)

Questions?

Thank You