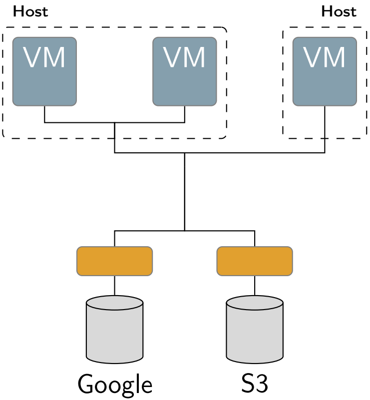
Distributed caching for cloud computing

Maxime Lorrillere, Julien Sopena, Sébastien Monnet et Pierre Sens

February 11, 2013

Cloud computing



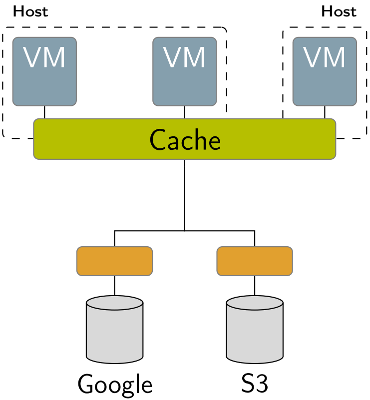
Cloud computing

- Computing resources as a service
- On-demand self-service
- Elastically provisioned
- QoS guarantees

Building a virtual platform

- Deal with different resources
  - From different providers
  - With different properties
- We need a common interface

Cloud computing



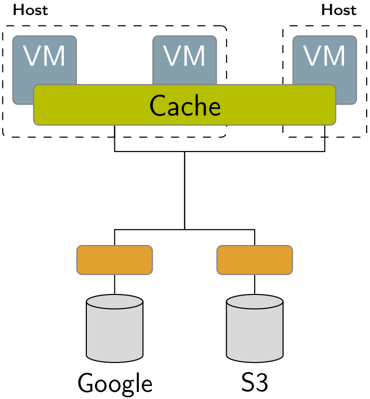
Cloud computing

- Computing resources as a service
- On-demand self-service
- Elastically provisioned
- QoS guarantees

Building a virtual platform

- Deal with different resources
  - From different providers
  - With different properties
- We need a common interface

Cloud computing



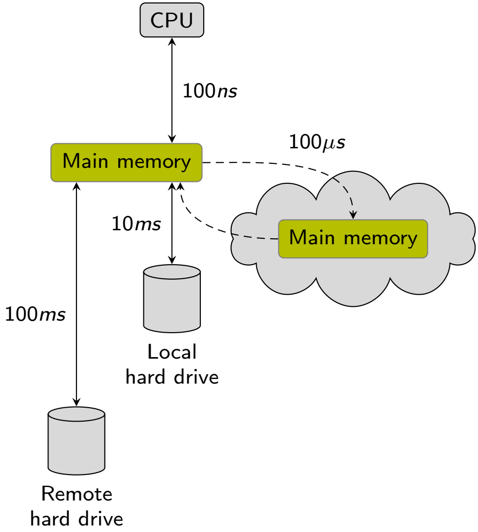
Cloud computing

- Computing resources as a service
- On-demand self-service
- Elastically provisioned
- QoS guarantees

Building a virtual platform

- Deal with different resources
  - From different providers
  - With different properties
- We need a common interface

Distributed caching



Local hard drive

- High capacity
- High latency
- Bottleneck

Network

- Low latency
- Capacity?

Main memory

- Very low latency
- Small capacity
- Principle of locality

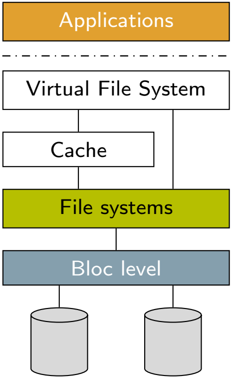
Distributed caching

Related works

Operating system layer:

- Application level [Memcached]
  - Existing applications have to be updated
- Filesystem level [xFS, PAFS, Ceph]
  - Guest operating system have to use a specific file system
- Bloc level [XHive, dm-cache]
  - Incompatible with distributed file systems

- Existing solutions are not *"cloud aware"*



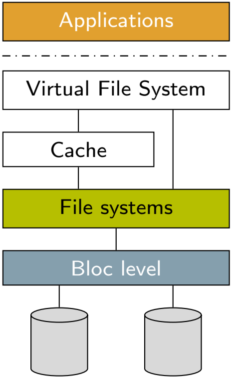
Distributed caching

Related works

Operating system layer:

- Application level [Memcached]
    - Existing applications have to be updated
- Filesystem level [xFS, PAFS, Ceph]
    - Guest operating system have to use a specific file system
- Bloc level [XHive, dm-cache]
    - Incompatible with distributed file systems

- Existing solutions are not *"cloud aware"*



Our contribution: a generic approach to develop ditributed caches for cloud computing

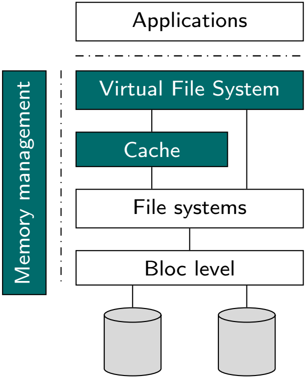
Development of a distributed cache

Implementation constraints

- Ensure genericity
  $\Rightarrow$ Integration into the Linux kernel
- Be non-intrusive

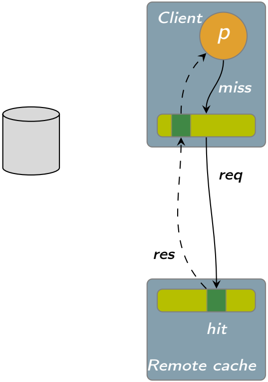
Performance constraints

- Limit overhead
- Minimise memory footprint



Remote cache

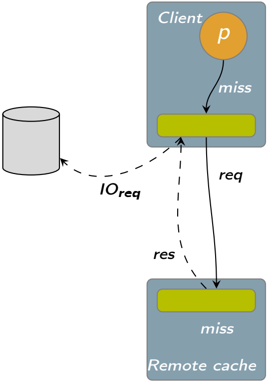
Direct client cooperation



- Remote memory extends local memory
- Easy localisation of data
- No data sharing

Remote cache

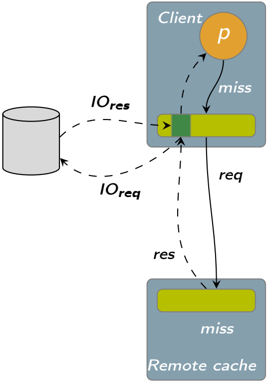
Direct client cooperation



- Remote memory extends local memory
- Easy localisation of data
- No data sharing

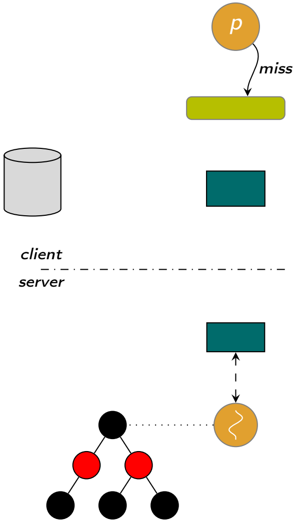
Remote cache

Direct client cooperation



- Remote memory extends local memory
- Easy localisation of data
- No data sharing

Architecture



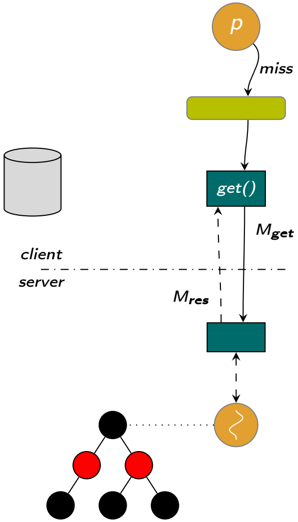
Client

- Basic operations: *get* and *put*
- Blocking *get*
- Executed by the process in kernel-space

Serveur

- Dedicated kernel thread
- *Request-response*
- Red-black tree

Architecture



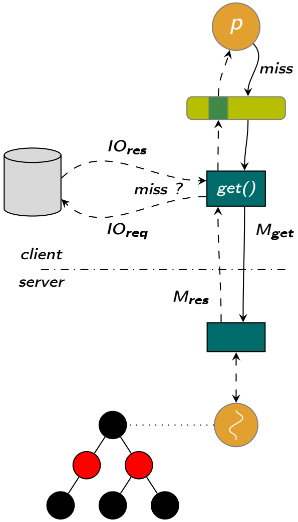
Client

- Basic operations: *get* and *put*
- Blocking *get*
- Executed by the process in kernel-space

Serveur

- Dedicated kernel thread
- *Request-response*
- Red-black tree

Architecture



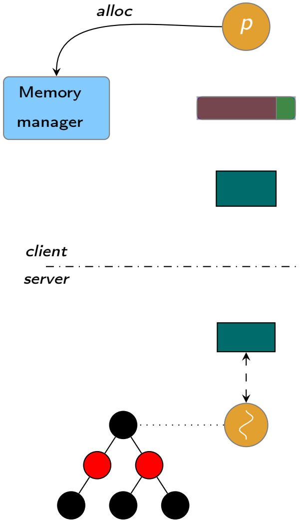
Client

- Basic operations: *get* and *put*
- Blocking *get*
- Executed by the process in kernel-space

Serveur

- Dedicated kernel thread
- *Request-response*
- Red-black tree

Architecture



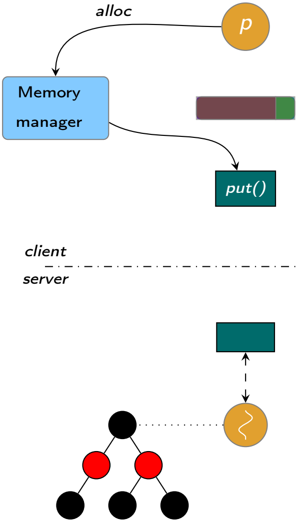
Client

- Basic operations: *get* and *put*
- *put()* called inside critical section

Serveur

- Dedicated kernel thread
- *Request-response*
- Red-black tree

Architecture



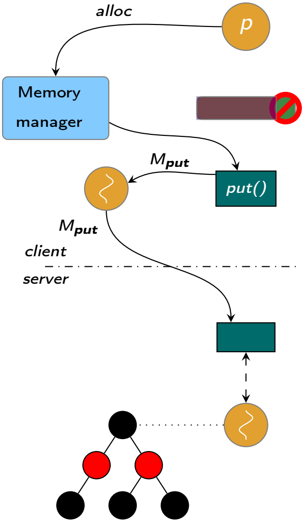
Client

- Basic operations: *get* and *put*
- *put()* called inside critical section

Serveur

- Dedicated kernel thread
- *Request-response*
- Red-black tree

Architecture



Client

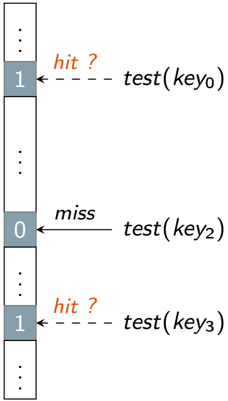
- Basic operations: *get* and *put*
- *put()* called inside critical section
- Executed in a dedicated thread

Serveur

- Dedicated kernel thread
- *Request-response*
- Red-black tree

Metadata management

Problem: metadata management efficiency

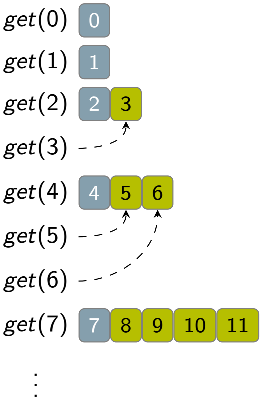


Solution: Bloom filter [Bloom'1970]

- Probabilistic data structure
- Compact
- No false negative
- False positive possible

Cache accesses management

Problem: sequential access detection

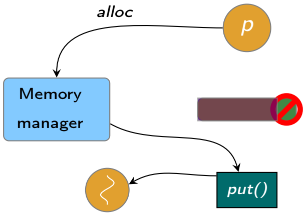


Solution: prefetching

- Sequential read detection
- Read prediction
- Read ahead of data
    - Amortized network latency

Communications management

Problem: network buffers memory footprint

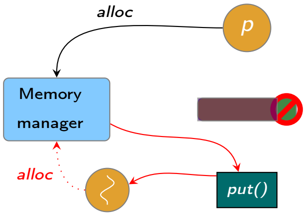


Solution: zero-copy

- Avoid copying into the network stack
- Decrease memory allocations
- Avoid *deadlocks*

Communications management

Problem: network buffers memory footprint



Solution: zero-copy

- Avoid copying into the network stack
- Decrease memory allocations
- Avoid *deadlocks*

Evaluation

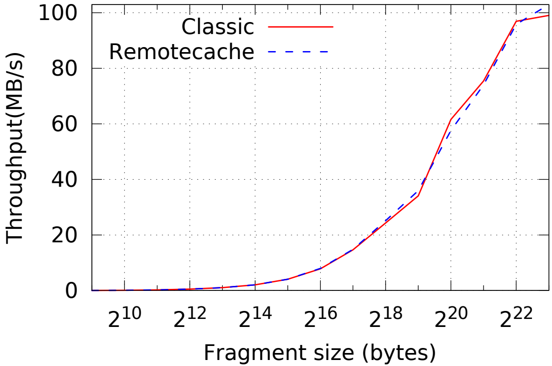
Experiment setup

- Virtualized platform
  - Intel Core i7-2600 (4 hyper-threaded cores), 8GB memory
  - Cache server (2 cores, 4GB)
  - Client (2 cores, 512MB)
    - Reads from local virtual hard drive
  - 1Gbit/s virtual network ($\sim 600\mu$s RTT)

- Micro-benchmark
  - 32MB read
  - Each read is split into fragments from 512 bytes to 8MB
  - Each fragment is read at a random position from a file

Remote *miss* overhead

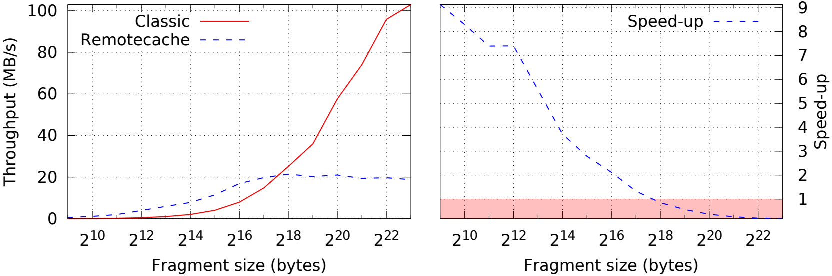
Empty remote cache



- Bloom filter avoids remote *miss*
- Code execution has a negligible

Performance peak

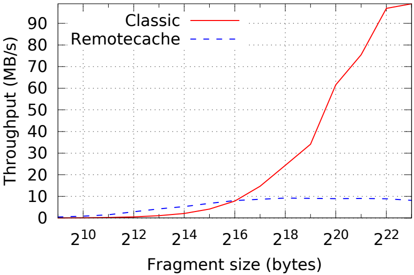
Data preloaded in remote cache

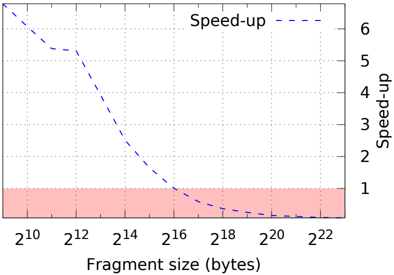


- Up to 8x performance improvement with small fragments (1KB)
- Performance drop above 128KB

Performance with local memory full of data

Data preloaded in remotecache, full local memory





- Up to 6x performance improvement with small fragments (1KB)
- Performance drop above 64K

Conclusion

Summary

- Existing distributed caches are not *"cloud aware"*
- We propose an approach to develop distributed caches for the cloud
- Working non-intrusive prototype
- Promising: up to 8x performance improvement in random read

Future works

- Realistics benchmarks: Memcached, dm-cache, bcache,...
- Sequential read performance improvements
- Consistency guarantees