# Smart-Cache: Prefetching Using Markov Chains in Distributed Caching

Meia Alsup*
Massachusetts Institute of Technology
Cambridge, MA

Amir Farhat*
Massachusetts Institute of Technology
Cambridge, MA

Alexander J. Root*
Massachusetts Institute of Technology
Cambridge, MA

## ABSTRACT

Caching methods for distributed key-value stores do not detect patterns in data accesses. Many types of data accesses, such as web page accesses and machine learning workloads have simple patterns that can be detected via a Hidden Markov Model (HMM). This work presents Smart-Cache, a distributed caching model that uses a HMM to predict future data accesses and prefetch data according to some learned patterns. We have implemented this cache and compared it to a `memcached`-like cache that does no data prefetching. Smart-Cache provides significant speed up on workloads with patterns. The experiments conducted show execution time reduction of [**TODO: some percentage range**]% for various testing scenarios.

**TODO: PUT SUMMARY OF RESULTS HERE**

## KEYWORDS

distributed systems, caching, Markov chain, distributed learning

## 1 INTRODUCTION AND RELATED WORK

Efficient, distributed caching is a well-known solution to alleviating load on underlying and expensive-to-query data stores [7]. Some caching systems predict future data accesses and pre-fetch accordingly. Some previous efforts to optimize pre-fetch mechanisms have relied on assumptions around data locality [1]; few methods relax the locality assumption for truly-unopinionated cache pre-fetch prediction.

Markov chains have been shown to learn data access patterns in many other settings including file I/O, hard disks access, and in smaller scale buffers or pages [3, 5, 6]. Smart-cache is motivated in particular by the deep learning workload scenario, but is generalizable and usable by a much wider array of scenarios with patterned workload accesses.

Machine leaning workloads, which are traditionally bottlenecked by compute power and GPU cycles, are benefiting from new hardware so as to put increasing load on underlying datastores. With state-of-the-art hardware, these datastores become a significant bottleneck [8]. In such situations, even a simple caching layer in front of the datastore gives huge reductions in datastore egress requirements and significantly lowers round-trip request latency [8]. When training large deep neural networks, most hyper paramter optimization procedures require that a large set of training runs are started, each with different hyper-paramater configurations. While each run has a different configuration, all runs access data in the same order. These runs, which are often in the same cluster of GPUs, all make queries to data with the same access pattern.

Our contributions are three fold. First, we introduce Smart-Cache, a fault-tolerant and distributed cache based on memcached [2] that can dynamically learn data access patterns. Second, we provide a simple patterned workload generator to proxy web workloads. Third, we characterize the performance of Smart-Cache on three workloads with varying degrees of contained patterns, and demonstrate significant improvement with respect to both latency measurements and queries to the underlying datastore.

## 2 SYSTEM AND SETUP

### 2.1 Design

*2.1.1 Assumptions.* One assumption with respect to the underlying datastore is that it is static. The original intent of Smart-Cache is to support machine learning workloads. Machine learning workloads are short-lived workloads on a static underlying dataset.

We also assume similar file popularity amongst files in the datastore. This version of Smart-Cache is not resilient to hot files and nodes, though we discuss a simple extension in our future work section that would make it resilient.

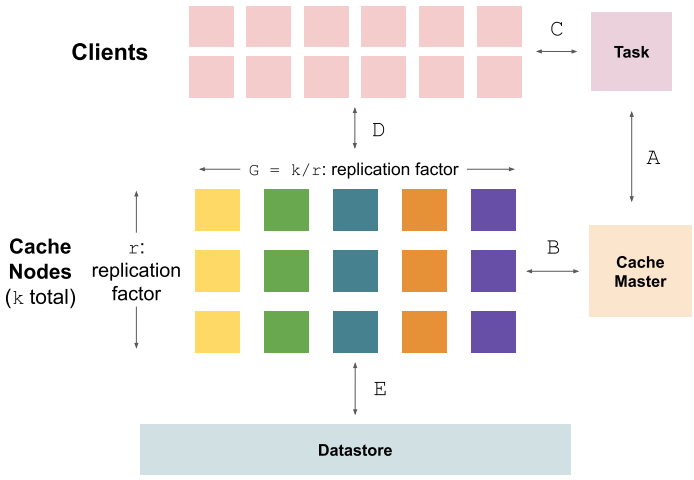


**Figure 1: Components of Smart-Cache and communication**

*2.1.2 System Design.* Figure 1 shows the system components. Smart-Cache supports a single task at a time, which coordinates many clients. The task interacts with *Cache master* which coordinates the behavior of the individual caching nodes. *Caching nodes* cache and store data, and are the only nodes connected to the underlying datastore.

*Denotes equal contribution.

The number of cache nodes, $k$, and the replication factor $r$, are configurable. These two parameters determine the number of file groups, $G = \frac{k}{r}$. In the event $r$ is not divisible by $k$, the floor is used and extra caches are allocated to file groups at random. These parameters allow the user to scale the caches along two axes. In Figure 1, in which color denotes file group, $k$ is 15, $r$ is 3, and the number of file groups $G$ is 5. The cache master (CM) handles distributing files across the $G$ file groups. Consider the underlying data-set $\mathcal{D}$ and $\{d_1, ..., d_g\}$ as disjoint subsets of $\mathcal{D}$ such that $\mathcal{D} = \bigcup_{i=1}^{n} d_i$. For each subset of $d_i \subseteq \mathcal{D}$, a file group is responsible for that subset and the subset is replicated across $r$ caching nodes.

CM assigns subsets $D_i$ to each file group. CM also dictates which caches a client fetches data from. CM creates a mapping that takes a client ID and filename, and outputs an ordering on the cache IDs in the file group which contain that filename. This mapping handles spreading the load for files in a file group across the replicated caches in that group. Each client gets back a list of all the cache IDs in the file group of interest. This is the order the client tries to fetch the file in. On failure, the client will try the fetch from the next client ID.

*2.1.3 Execution Flow and Communication.* A task is initiated with a call to CM over Channel A in 1. The task tells CM the client IDs, the files in the datastore, and the configuration parameters for the cache. The cache master spins up cache nodes, and assigns each cache responsibility for a subset of the datastore - this happens over Channel B. The cache nodes are responsible for fetching files from the datastore over Channel E. CM shares the mapping function from client IDs and files to cache lists with the task over channel A, which distributes this information to each client over channel C. The clients do not directly interact with the cache master nor datastore, but instead utilize the mapping to go fetch files from the correct cache nodes over Channel D.

*2.1.4 Fault Tolerance.* Increasing $r$ facilitates better redundancy as more caching nodes tasked with caching files from $d_i$ can fail before the system is unable to serve assets from $d_i$. While our implementation doesn't do so, nodes serving data from another subset of $\mathcal{D}$ can be re-purposed to serve files from $d_i$ if enough nodes fail such that the number of nodes serving data from each subset remains balanced.

While we assume that the load on each subset of $\mathcal{D}$ is uniform, the number of nodes serving a particular subset can also be modified to balance the load placed on that subset by clients. As such, the replication factor for each subset may not be uniform, but can be proportional to the required load.

This setup provides fault-tolerance configurable to the user, through the replication parameter. Without the optimizations to re-balance caches mentioned above, if all caches fail in a given file group, the whole task fails. This is an acceptable trade-off as replication is configurable and the probability that all caches in a file group fail concurrently

## 2.2 Experiments and Workload Types

Our experimental setup consists of a *task* for a given workload type, which is run on an instance of Smart-Cache. The task spins up several clients, each of which must complete the workload assigned by the task.

Our terminology follows:

- *Task*: Over-arching "workload master", which starts many clients each of which have their own workload.
- *Client*: A machine which is owned by the task and dedicated to its workload. The client handles fetches data from the data-store (via the caching layer), in the order specified by its given workload.
- *Workload*: This term is overloaded. In the context of a client, "workload" refers to the specific order of file accesses it needs to make. More generally, "workload" refers to the type of work and file access pattern the task represents (ie Machine Learning, Web, etc).

We analyze three workload types (these are defined in 2.2.1). For each workload we run the task to completion twice, once using Smart-Cache with Markov based pre-fetching, and once without pre-fetch. The no-prefetch scenario is used to benchmark the performance improvement attributable to the Markov chain based approach. In total we ran 6 tasks, the cross product between our three workload types with two prefetch mechanisms: none, and Markov chain-based.

*2.2.1 Workload Types.*

- **Deterministic**: Deterministic workloads exhibit the same file access pattern across all clients within a single task. Each client is given the same workload. An example of this is machine learning workloads.
- **Patterned**: Patterned workloads share some patterns in data accesses across client workloads. These workloads are generated by creating some short patterns that are shared amongst all clients. Each client's workload is a random concatenation of these patterns. Concretely, for example, if the patterns generated are ["A", "B", "C"] and ["X", "Y"], then client 1 might have workload ["A", "B", "C", "A", "B", "C", "X", "Y"] and client 2 might have workload ["X", "Y", "A", "B", "C", "X", "Y", "A", "B", "C"]. These patterned workloads, while not exact replications of a scenario in particular, are used to assess Smart-Cache's ability to learn patterns in file access workloads. This workload is similar to what a website workload could like. Web workload transitions are modeled with a large markov chain (ie in PageRank) which represent the probability of going to page A from page B.
- **Random**: Random workloads exhibit no patterns. This is important to help benchmark the improvements seen in Deterministic and Patterned workloads, and to show that Smart-Cache does not slow down performance even if there is no pattern to learn.

The workloads used to generate our results can be found in the code we released [1].

[1] https://github.com/rootjalex/smart-cache

### 2.3 Markov Prefetching Model

## 3 RESULTS

The results from our runs are shown in Table 1. As of now, our patterned workloads are broken and as such these results are omitted pending a fix.

A limitation of the results is that these workloads were run locally on our machines, instead of on a dedicated compute cluster. These results are also for a single run rather than an average across several runs. As such, the difference observed in the Random task between LRU and Markov Chain is likely not statistically significant. However, the result on the Deterministic task is likely significant.

**Table 1: Total time in seconds to perform tasks.**

| | LRU | Markov Chain |
|---|---|---|
| Random | 91 | 86 |
| Patterned | 0 | 0 |
| Deterministic | 2125 | 359 |

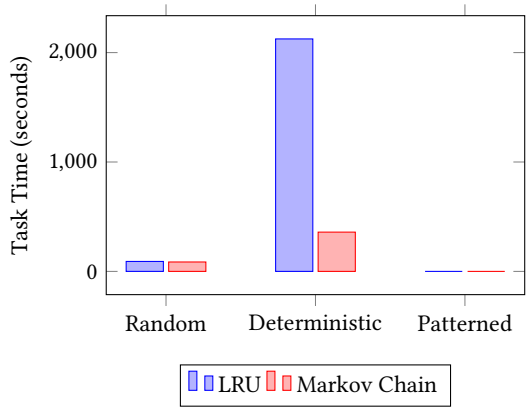


**Figure 2: Total time to perform tasks, in seconds, across workloads and caching strategies.**

## 4 CONCLUSION AND FUTURE WORK

In this paper, we demonstrate the utility of Smart-Cache to speed up workloads that exhibit patterns. This is immediately useful for machine learning applications and in any other application where workloads can be characterized by repeat data access patterns.

This can be expanded upon on a few axes. The cache master can dynamically allocate cache nodes across file groups to prevent hot files from taking the system down and make the system more resilient to caches going down in a particular file group. The system currently uses a simple hashing scheme, but more complex hashing schemes such as consistent hashing [4] can be leveraged to make the system more tolerant to caches going off and online.

Further, the cache master could implement periodic markov chain syncing across caches in the same file group. This would enable learning to happen quicker, since patterns observed can be learned at double or triple the speed for a replication factor of 2 or 3 respectively.

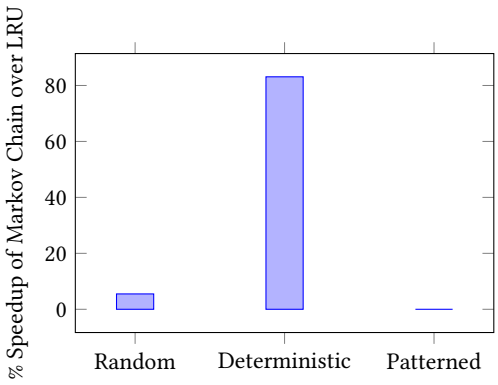


**Figure 3: Total time to perform tasks, in seconds, across workloads and caching strategies.**

## ACKNOWLEDGMENTS

We'd like to thank Professor Robert Morris, Anish Athalye, and the entire 6.824 staff for guidance and helpful conversations. We would also like to thank Jacob Kahn for providing insight into machine learning workloads at Facebook AI Research, and inspiring this work.

## REFERENCES

[1] Swapnil Bhatia, Elizabeth Varki, and Arif Merchant. 2010. Sequential Prefetch Cache Sizing for Maximal Hit Rate. 89–98. https://doi.org/10.1109/MASCOTS.2010.18

[2] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[3] Doug Joseph and Dirk Grunwald. 1997. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 252–263. https://doi.org/10.1145/264107.264207

[4] Leighton Levine Lewin Panigrahy Karger, Lehman. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. ACM, USA.

[5] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff. 2016. Lynx: a learning linux prefetching mechanism for SSD performance model. , 6 pages.

[6] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. 2008. TaP: Table-Based Prefetching for Storage Caches. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (San Jose, California) *(FAST'08)*. USENIX Association, USA, Article 6, 16 pages.

[7] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(nsdi'13)*. USENIX Association, USA, 385–398.

[8] Vineel Pratap Vitaliy Liptchinsky, Jacob Kahn. 2020. Scaling Deep Learning for Automatic Speech Recognition. (2020). https://developer.nvidia.com/gtc/2020/video/s21838 S21838.