

On Effects of Applying Predictive Caching for State Machines

James Ryan Perry Akyüz and Tolga Ovatman^a

Department of Computer Engineering, Istanbul Technical University, 34469 İstanbul, Turkey

Keywords: State Machines, Predictive Caching, Execution Path Prediction.

Abstract: State machines are frequently used in software development, in many different contexts, ranging from modeling control software to distributed applications that operate in cloud environments. We have implemented and experimented on basic execution path-based predictive caching approaches for state machines to show that due to the limited number of paths that can be taken during a state machine run better pre-fetching can be achieved for state machine caches. We have applied our predictive approaches over least frequently used (LFU) and least recently used (LRU) replacement on two different state machine instances run with real-world execution traces.

1 INTRODUCTION

State machines are powerful formal models that are being used in a wide area of applications to capture the stateful behaviour of developed software. By their description, state machines contain execution paths that are repeatedly executed for different kinds of input to perform the predetermined sequence of operations. This repetitive behaviour may be taken advantage of to perform predictive caching decisions based on the anticipated path that may be executed through the use of historical execution information.

In this paper we investigate the effects of execution path-based predictive caching in state machine implementations. Basically, on top of employing a conventional cache replacement approach like Least Frequently Used (LFU), we observe the recent execution of the state machine to predict which execution path is most likely to be taken in the future. Using statistics of the paths taken in the past, we direct the cache pre-fetching decisions according to the most likely path to be taken next.

We have used a data set based on actions performed in a limited amount of time among a group of Twitter users and implemented a state machine that performs these actions on a simple database. We have also used another state machine that has been derived from GitHub interactions provided by a commercial database vendor. We have applied a write back policy over an in memory cache that is capable of holding only a few objects from the database. Our results

show that applying a cumulative path probability calculation approach in path prediction provides the best results in pre-fetching.


Directing caching decisions on state machines may specifically be important for environments where relatively smaller state machines run simplistic functionality with a limited amount of memory, such as Internet of Things applications. Increasing the cache hit ratio based on historical data can significantly improve energy consumption and run-time performance of those devices as well.

Section 2 presents related work on predictive caching in state machines. Section 3 presents our experimental data, state machines that are used in experiments, and the approaches that we implemented. Later in Section 4 we present the results of our experiments and evaluate the results. In Section 5, we conclude by mentioning possible future work.

2 RELATED WORK AND LITERATURE REVIEW

Even though the use of caching to improve performance has been a greatly investigated topic, there are relatively a limited number of recent studies present in the literature in the domain of state machines. For a recent literature survey on proactive caching (we use the more restrictive term predictive caching in this paper), readers may refer to Sun Guolin et al.'s work (Anokye et al., 2020).

In (Santos and Schiper, 2013), Santos et al. make

^a  <https://orcid.org/0000-0001-5918-3145>

use of State Machine Replication (SMR) where replicated state machines have a cache of requests that have been previously executed. This cache is consulted by the replicas when a request is received. If the request has already been executed, its previously computed reply is sent by the replica to the client.

A service manager is used to manage events in sequence, executing them on the service and sending a reply back to the client. Replies are cached to ensure at-most-once execution of requests. A *ConcurrentHashMap* is used to reduce contention and improve multi-threaded performance when under high load from many threads.

For Scalable State Machine Replication (S-SMR) in (Bezerra et al., 2014), when a command is executed, the executing client multicasts the command to all partitions that contain variables that the command reads or updates. An oracle is assumed to exist that handles informing the client about which partitions the command should be sent to. A server in a partition is able to cache variables that are from other partitions in multiple ways. They consider conservative caching and speculative caching.

With conservative caching, a server waits for a message from a variable's partition that says if the cached value is still valid or not before executing a read operation for that variable. If it's been invalidated, the server discards the cached copy and requests the value from the partition. With speculative caching, it is assumed that the cached values are up to date. A server will immediately get the cached value from a read operation without using any validation messages. If the variable is invalidated by a message from a server in the variable's partition, the server with the invalid value will be rolled back to before the operation, the cached value will be updated to the new value, and execution will be resumed from the earlier state. In (Le et al., 2016), Dynamic Scalable State Machine Replication (DS-SMR) improves on (Bezerra et al., 2014)'s S-SMR by giving each client a local cache of the oracle's partition information. If the cache contains information about the variables involved in a particular command, the oracle is not consulted, improving scalability.

We may also mention Robbert van Renesse et al.'s work (Marian et al., 2008), where they use a middle tier state machine-like model to alleviate replication of services, that may include caching. Even though this study is quite out of scope of our study we still mention it as a possible approach to handle caching services on the cloud.

For Mobile Edge Caching in (Yao et al., 2019), caching in the context of mobile edge networks is covered, including the issues of where, how, and what

to cache. They discuss caching locations, including caching at user equipment, base stations, and relays, as well as caching criteria such as cache hit probability, network throughput, content retrieval delay, and energy efficiency. Several caching schemes are mentioned, for example, distributed caching, in which nodes use info from neighboring nodes to increase the perceived cache size to the user.

In (Le et al., 2019), Le Hoang et al. propose DynaStar, based on DS-SMR from (Le et al., 2016). Their improvements over DS-SMR involve multicasting of commands, caching of location information from the oracle on the client, and re-partitioning of a command's variables to a target partition that executes the command, replies, and returns the updated variables to their partitions.

Last but not least we should also mention recent studies of Changchuan Yin et al. (Chen et al., 2017; Chen et al., 2019) where they utilized a liquid state machine learning model to manage caching in an LTE-U unmanned Aerial Vehicle Network. Compared to our approach this study uses network level information and the state machines come into action as a learning model rather than the executed model itself.

3 STATE MACHINE MODEL AND EXPERIMENTAL DATA

3.1 Caching Approaches Used in Experiments

Figure 1 presents the overall architecture we have used to perform our experiments. As our state machine instance receives events, a cache managing component accesses data from the database, caching it locally. The cache manager also has a collection of statistics that is used to determine likelihoods of what event will be sent next. These statistics consist of subsets of the full state machine history, with each subset path containing the frequencies of the potential next events. Based on these frequencies of previous events, the most likely upcoming event can be predicted, and the data that would be used by the resultant state can be cached in advance.

We propose two approaches for predicting the next set of objects to be fetched to the cache as follows. We use the notations in Table 1 during our definitions.

- Path based prediction: Based on the current state of the state machine and the possible predecessors paths, we choose the next most likely path and the

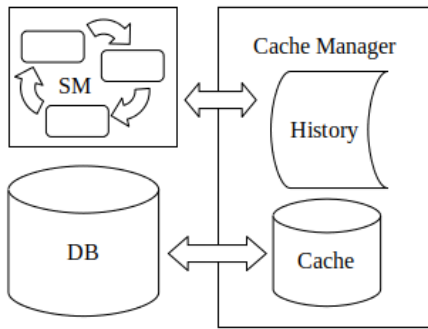


Figure 1: Overall system diagram, consisting of the state machine, cache manager, and database.

Table 1: Notation used for state machines.

Type	Symbol
State Machine	$SM = S \cup T$
States	$S : \{S_0, S_1, \dots, S_n\}$
Transitions	$T : S \times E \times S$
Events	$E : \{E_0, E_1, \dots, E_n\}$
Paths	$P : \{P_0, P_1, \dots, P_n\}$
Path	$P_i : (T_0, T_1, \dots) \mid T_i \in T$
Path fragment of length n	$P'_i : (T_0, T_1, \dots, T_{n-1}) \mid T_i \in T$

second most likely path to fetch their data. During the execution of the state machine, after each state change from P_i to P_j we fetch the possible set of path fragments of a particular length of size n , that may emerge from the current state P_j and choose the most likely two based on execution data. To obtain path frequencies we analyze the execution data of the state machine to count the number of occurrences for possible future path fragments of length n .

- **Event based prediction:** In this approach, based on the current state, we determine the set of possible predecessor events and assign a weight to each of them based on the frequency of each event calculated by the execution data. Out of the weighted events, we perform a weighted random selection, repeat this process n times for each selected event and pre-fetch the data of the selected event or path.

We also use two different approaches during the frequency calculations during the prediction approaches.

- **On-the-fly calculation:** We only use the past execution data that occurred up to a certain history window size to calculate path/event frequencies. Every time the state machine executes events that match the width of window size we re-calculate the path execution statistics from scratch, disre-

garding former event executions and taking the most recent history window into account.

- **Cumulative calculation:** Like on-the-fly calculation we use data from a certain history window. However, instead of calculating probabilities from the scratch every time, we cumulatively update the probabilities every time the history window is about to be updated with recent events.

We compare our proposed approaches against by applying them over a well-known cache replacement approaches: LFU and LRU. We compare our approach among each other and with random pre-fetching over varying cache sizes and history window sizes to get an idea about how pre-fetching behaves under different circumstances.

We have used a write back policy through all our experiments where we write the updated value in the cache whenever it is to be replaced. Since we only experiment with a single instance, there weren't any observed side effect in terms of consistency.

3.2 Data Sets and State Machines Used in Experiments

In our experiments we have chosen to use two different data sets to synthesize and experiment on artificial state machines. The first data set we used is the activity time graph of the Higgs Twitter data set¹. The Higgs Twitter data set contains action data from 300K users, with 170K tweets containing mentions, 36K replies to tweets, and 350K retweets, along with follower data consisting of almost 15M connections between users. In our experiments we have built a simple state machine that contains the events "tweet", "reply", "retweet" and "follow" to process a subset of the events available in this data set. We extracted the relevant information from the data sets and pre-processed them to filter out the unrelated columns and rows.

All users are kept in a single set, where an individual user consists of an ID, a follower list, a following list, and a timeline. The follower and following lists are sets of users which make up those who the user follows and those who follow the user. A timeline consists of a collection of posts, where a post contains a poster ID, mentioned ID, and a timestamp. For all posts, which can be tweets (TW), retweets (RT), and replies (RE), each row from the data set consists of the user ID who posted the tweet, the user ID who was mentioned, the timestamp of the post, and an identifier for what type of tweet it is, either TW, RT, or RE.

¹<https://snap.stanford.edu/data/higgs-twitter.html>

Table 2: Notation used for Twitter data set.

Category	Set
Users	$U = \{u_0, u_1, \dots, u_n\}$
Followers	$F = \{f_{u_0}, f_{u_1}, \dots, f_{u_n}\}$ $f_{u_i} = \{u_j, u_k, \dots, u_m\}$
Posts	$P = \{p_{ts_0}, p_{ts_1}, \dots, p_{ts_n}\}$
Timelines	$T = \{t_{u_0}, t_{u_1}, \dots, t_{u_n}\}$ $t_{u_i} = \{p_{ts_j}, p_{ts_k}, \dots, p_{ts_m}\}$

Table 3: Key for Twitter data set notation.

Symbol	Meaning
U	set of all users
u_i	an individual user
F	set of sets of each user’s followers
f_{u_i}	a set of the followers for user u_i
ts_i	a timestamp
P	set of all posts
p_{ts_i}	an individual post that was posted at timestamp ts_i
T	set of all timelines
t_{u_i}	ordered set of posts on the timeline of user u_i

For additional clarification we present additional formal definitions of the notation we have made use of in our experiments in Tables 2 and 3.

Due to the relative simplicity of the Twitter data set, our experiments were performed on a simple state machine as illustrated in Figure 3. There exists basically four different states which navigate to a specific state upon receiving an event such as tweet (TW), reply (RE), retweet (RT) or follow (FL).

After receiving the event the state machine fetches related data to the cache and performs necessary actions indicated in the figure. The tweet, retweet, and reply paths update the data structures containing the tweet timelines of the poster and their followers, while the follow path updates the structures containing the following and follower lists of the involved users. Simply each tweet, retweet and reply adds the current post($p_{ts_{cur}}$) to the specific users timeline while each follow activity adds the follower (u_y) to the followed user’s (u_x) followers list (f_{u_x}).

In the presented state machine a brief formal definition to represent the performed action is presented in the transitions; in the real implementation many other pieces of data needs to be fetched to perform the action. For instance, retweeting action needs to fetch the specific user’s timeline to obtain information on the tweet that is being retweeted.

The second data set we use is the GitHub activity data served publicly by a commercial database ven-

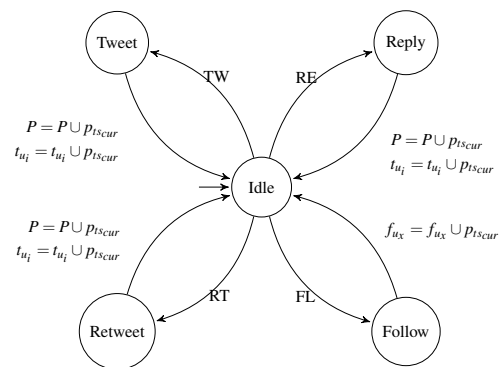


Figure 2: Model of the simple GitHub state machine used in experiments.

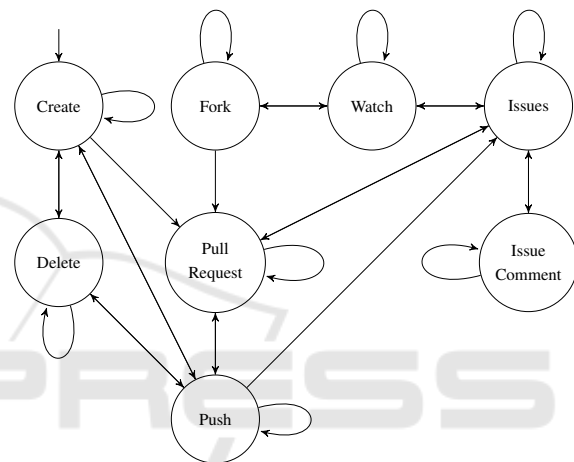


Figure 3: Model of the simple Github state machine used in experiments.

dor². This data set contains 3.5GB of 1.2M git activities for a set of 310K repositories. To construct an artificial state machine based on this data we have analyzed the sequence of events that has been issued per repository basis and build a common state machine, presented in Figure 2, that can properly handle 95% of the events that occur for any repository in the data set. During the execution of state machine instances in our experiments, we properly drop out the infrequent activity (less than 5% of the whole data) that doesn’t correspond to any transitions in the state machine. For brevity, we do not present the specific activities that corresponds to each transition in Figure 2.

²<https://www.citusdata.com/blog/2017/01/27/getting-started-with-github-events-data/>

4 EXPERIMENTS AND EVALUATION

Experiments were performed on a system running Kubuntu with 12 Intel i7-9750H 2.60 GHz cores, 16 GB of RAM, and a 1 TB SSD. The client and state machine are Java-based and making use of Spring for state machines. They were kept in a Docker container and communicated directly as to not induce any timing penalties from communicating over the network.

Trials were repeated over a variety of cache sizes for prediction with cumulative and on-the-fly statistics calculation. With the on-the-fly approach, statistics were updated on-the-fly as events were sent to the state machine, while the cumulative approach updates a set of statistics periodically. Each experiment was conducted with varying amounts of space in the cache 1,000 to 50,000 objects, and varying history window sizes from 10/15 to 50. The term history window size in the experiments correspond to the term used in our explanations in Section 3.1; it simply describes the amount of past events to be considered when calculating path execution statistics.

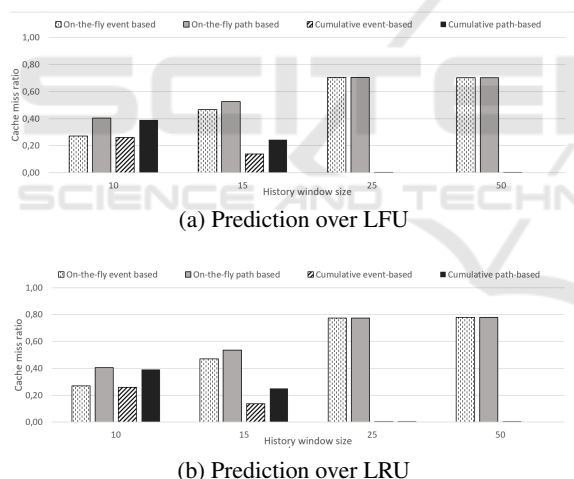


Figure 4: Cache misses comparison for twitter state machine for a 5000 object cache and varying history window sizes.

For the experiments using both of the state machines, every setup was sent events that compromised all activities in the data set, with each of the setups being repeated 10 times, averaging their results.

We present the cache miss rate comparisons counted during the experiments on twitter state machine in Figures 4 and 5 to present results for varying history window and cache sizes respectively. Each subfigure individually presents results for prediction over LFU and LRU separately.

From the experiments, we can see that predictive

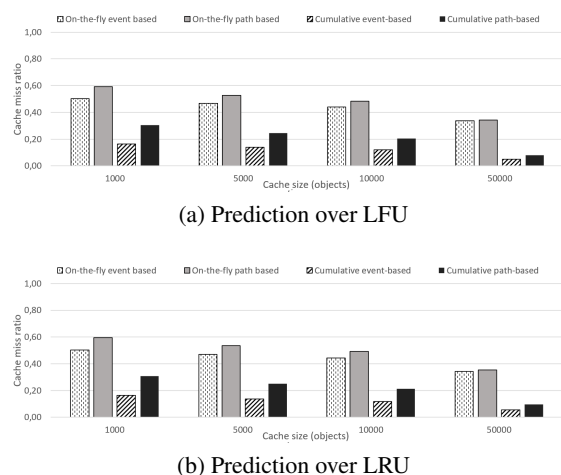


Figure 5: Cache misses comparison for twitter state machine for a 25 object history window size and varying cache sizes.

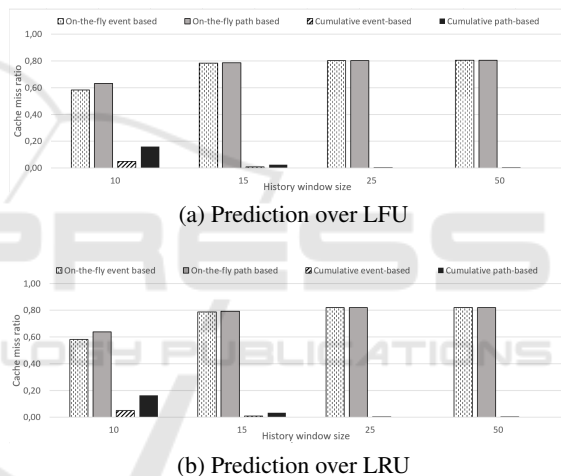


Figure 6: Cache misses comparison for github state machine for a 5000 object cache and varying history window sizes.

caching performs better, in terms of cache miss ratios, for on-the-fly approach especially for larger history window sizes. As expected, the selection of replacement policy doesn't effect the predictive caching performance slightly, only for larger history windows.

Another observation from the results may be that, by the increase of the cache size in terms of number of cached objects, performance of predictive caching deteriorates slower with respect to the cache size.

We present the cache miss rate comparisons counted during the experiments on the GitHub state machine in Figures 6 and 7. We obtain similar results in terms of comparison between prediction approaches for this set of experiments. On the other hand, for this set of experiments, using a larger state machine resulted in an increased performance for on-

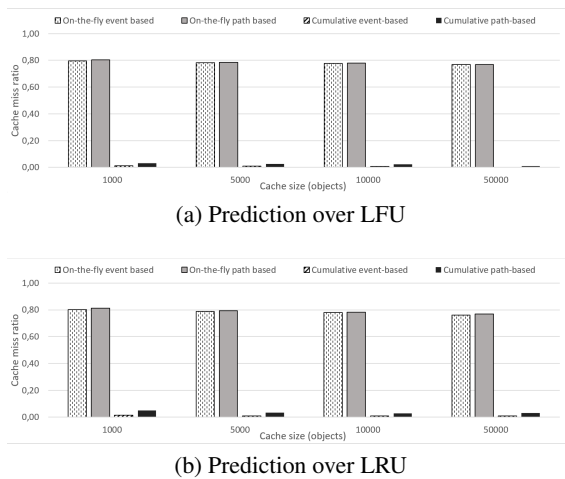


Figure 7: Cache misses comparison for github state machine for a 15 object history window size and varying cache sizes.

the-fly strategies where it greatly decreased the performance of cumulative strategies.

As an overall evaluation of our experiments from different perspectives, following results were obtained:

- Path-based prediction versus event-based prediction: Path based prediction provided better results especially for smaller state machines, smaller window sizes and smaller cache sizes. There were almost no difference for the larger state machine between the two approaches.
- On-the-fly calculation versus cumulative history calculation: From this perspective on-the-fly calculation is the obvious winner. Cumulative calculation was only able to provide comparable but still worse performance under few resources and smaller state machines.
- Effects of state machine size: The GitHub state machine, being a more complicated state machine with more states and transitions, not only presented better results under on-the-fly prediction, it also deteriorated cumulative calculation performance significantly. The Twitter state machine, on the other hand, provided more comparable, but overall worse, results for prediction and history calculation strategies.
- Effects of cache size: Cache didn't make any important difference for the larger GitHub state machine. Larger cache sizes decreased all the strategies' performance slower with respect to the increase in cache size for the Twitter state machine.
- Effects of cache replacement policy: There were almost no difference in using different replacement policies under predictive caching.

5 CONCLUSION

In this study, we examined predictive caching for state machines, where the data to be pre-fetched to cache is predicted using the historical data based on the past execution of the state machine. Since the number of different paths that a state machine can take is relatively limited, execution path-based predictive caching is expected to perform better for state machines. Our experiments showed that, indeed, predictive caching increases the performance of replacement algorithms, in terms of cache miss ratio, significantly when performed by collecting cumulative statistics.

We plan to expand our study to perform more experiments on different state machines and investigate further the relation between cache size, history window size and prediction performance for path-based prediction in state machines. We also plan to experiment on multiple and replicated state machines where write policies may greatly affect system performance. Furthermore we may also adopt more intelligent approaches than simple statistics, such as artificial intelligence based approaches and/or deep learning, to provide even fewer cache misses as future work.

ACKNOWLEDGMENT

This study is supported by the scientific and technological research council of Turkey (TUBITAK), within the project numbered 118E887.

REFERENCES

- Anokye, S., Mohammed, S., and Guolin, S. (2020). A survey on machine learning based proactive caching. *ZTE Communications*, 17(4):46–55.
- Bezerra, C. E., Pedone, F., and Van Renesse, R. (2014). Scalable state-machine replication. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 331–342. IEEE.
- Chen, M., Saad, W., and Yin, C. (2017). Liquid state machine learning for resource allocation in a network of cache-enabled lte-u uavs. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE.
- Chen, M., Saad, W., and Yin, C. (2019). Liquid state machine learning for resource and cache management in lte-u unmanned aerial vehicle (uav) networks. *IEEE Transactions on Wireless Communications*, 18(3):1504–1517.
- Le, L. H., Bezerra, C. E., and Pedone, F. (2016). Dynamic scalable state machine replication. In *2016 46th*

- Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 13–24. IEEE.
- Le, L. H., Fynn, E., Eslahi-Kelorazi, M., Soulé, R., and Pedone, F. (2019). Dynastar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1453–1465. IEEE.
- Marian, T., Balakrishnan, M., Birman, K., and Van Renesse, R. (2008). Tempest: Soft state replication in the service tier. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 227–236. IEEE.
- Santos, N. and Schiper, A. (2013). Achieving high-throughput state machine replication in multi-core systems. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 266–275. Ieee.
- Yao, J., Han, T., and Ansari, N. (2019). On mobile edge caching. *IEEE Communications Surveys & Tutorials*, 21(3):2525–2553.

