A Multi-level Caching Architecture for Stateful Stream Computation

Muhammed Tawfiqul Islam
tawfiqul.islam@unimelb.edu.au
The University of Melbourne
Melbourne, Victoria, Australia

Renata Borovica-Gajic
renata.borovica@unimelb.edu.au
The University of Melbourne
Melbourne, Victoria, Australia

Shanika Karunasekera
karus@unimelb.edu.au
The University of Melbourne
Melbourne, Victoria, Australia

Abstract

Stream processing is used for real-time applications that deal with large volumes, velocities, and varieties of data. Stream processing frameworks discretize continuous data streams to apply computations on smaller batches. For real-time stream-based data analytics algorithms, the intermediate states of computations might need to be retained in memory until the query is complete. Thus, a massive surge in memory demand needs to be satisfied to run these algorithms successfully. However, a worker/server node in a computing cluster may have limited memory capacity. In addition, multiple parallel processes might be running concurrently, sharing the primary memory. As a result, a streaming application might fail to run or complete due to a memory shortage. Although spilling state information to the disk can alleviate the problem by allowing the query to finish, it will cause significant performance overhead. An in-memory-based object store as the state backend will also perform poorly due to the added communications with the external object store and serializing/deserializing the objects. This paper proposes a multi-level caching architecture to mitigate the surge of memory demand from the processes running complex stateful streaming applications. The multiple levels of the cache span across the process heap space, in-memory distributed object store, and secondary storage. The objects/states required in the computation are always served from the fastest level of the cache to boost the application performance. We also provide a multi-level caching library in Java which can be used to implement scalable streaming algorithms. The underlying cache management completely abstracts the multi-level cache implementation from the application and handles seamless migration of states/objects across different levels of the cache. In addition, the multi-layer caching architecture is configurable (i.e., an application can choose to leave out a cache level.) The experimental results demonstrate that our proposed multi-level caching approach for state management can manage large computational windows and improve the performance of an actual streaming application up to three times compared to the in-memory object store-based state backend.

DEBS '22, June 27–30, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539803

CCS Concepts

• **Computer systems organization** → **Real-time system architecture**; • **Information systems** → **Data management systems**.

Keywords

Stream Computing, Performance Improvement, Cache management, JVM, Caffeine, Redis

ACM Reference Format:
Muhammed Tawfiqul Islam, Renata Borovica-Gajic, and Shanika Karunasekera. 2022. A Multi-level Caching Architecture for Stateful Stream Computation. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524860.3539803

1 Introduction

Real-time data analytics algorithms are used in many application areas such as fraud detection, stock trades, business analytics, customer/user activity tracking, and IT systems monitoring. Although traditional batch-oriented systems can process large chunks of static data, they can not be used to handle such modern applications where data is generated continuously. Thus, stream computing frameworks such as Storm[1], Spark[2], Flink[3], and Heron[4] are used with these applications to process real-time streaming data. In stream computing, continuous data streams are generally discretized to apply computations on subsets of data. The intermediate data and results of computations are called the *state* of an application that can be used in subsequent operations. Researchers have proposed novel ways[5, 6], such as windowing and key-value pairs, to represent, manage, and use the computational state of an application in data stream processing.

Stream processing frameworks can be deployed in a set of Cloud Virtual Machines (VMs) or physical servers. Multiple processes are generally created to run in parallel in a particular host. For example, a window-based ML or data analytics algorithm might accumulate a large amount of data in a specific time frame to be processed as a sequence. In addition, if the outputs from one operation are needed in the subsequent operations, then the states of computations might need to be retained in memory to reduce the performance overhead of the application. The memory allocated to a process can be dynamically increased in size depending on the primary memory availability of the host. However, as the demand for memory increases, the total available memory might run out, especially if multiple processes are co-located in the same host. Although it is possible to allocate extensive memory to a process if the OS supports virtual memory, it might lead to severe performance degradation due to increased page swapping by the OS. Thus, it is often infeasible to satisfy the increasing memory demand of complex stream processing tasks in a scalable manner. There are

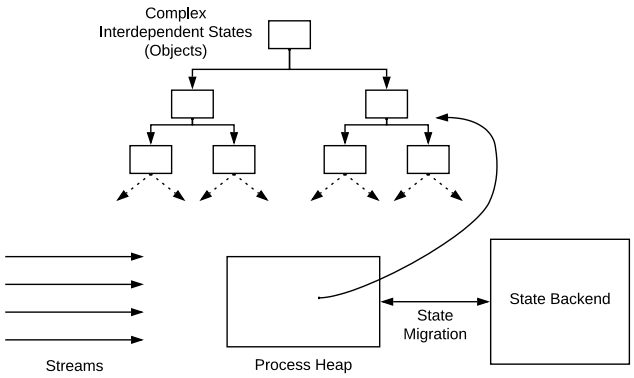


Figure 1: An example of complex interdependent states of objects in a data analytics algorithm.

some works from academia and industry to achieve transparent state management. However, there is limited support for the diverse representation of a process's state[7]. In most existing state management frameworks, the abstraction of process states is limited to key-value pairs. Computational states of a process may, however, exist in other forms such as graphs, trees, and hashes that can hardly be represented with key-value mappings[8]. Thus, there is a need to support arbitrary data structures for state representation while keeping the entire state management process transparent to the applications[5, 9]. For example, data structures offered by the programming languages such as Arrays, HashMaps, Lists need to be cacheable without the need to manage the key-value mappings associated with both the program caches and the in-memory object stores. In addition, there is a need to support nested cacheable data structures such as graphs and trees, which will enable fast and transparent implementation of complex streaming applications. Lastly, reducing the excessive overhead of state migration between the process heap and the state backend is also a key problem that needs to be addressed.

As a motivating example, consider a streaming application that collects real-time social media data streams to build a network graph of users to detect the evolution of communities over time. As shown in Fig. 1, if the application collects real-time streams over a vast geographical location, the size of the graph will increase rapidly, and it would be infeasible to retain the complete graph in the main memory for analysis. Although some parts of the graph can be kept in the process heap while the rest can be migrated to a state backend such as an in-memory object store or disk, this could result in a severe degradation in performance and may not meet the needs of the stream processing paradigm due to increased latency. Thus, based on the usage of the accumulated data/objects, there is a need for the application to access frequently used data/objects quickly, possibly from a closer/faster memory location (e.g., process heap).

To address the problems mentioned above, we propose a multi-level caching architecture to improve the performance of the stateful and memory-hungry streaming applications. This is the first approach to integrating a multi-level cache with stream-based applications to the best of our knowledge. The multiple levels of the cache span across the process heap space, in-memory distributed data structure store, and secondary storage. The states/objects required in the computation are always served from the fastest level of the cache to boost the application performance. If the fastest level of the cache runs out of memory, the objects from the fastest level of cache are seamlessly migrated to the slower levels of the cache by following a caching policy. We also provide an Application Programming Interface (API) written in Java, which provides a high level of abstraction to the programmers while developing complex stateful stream processing applications. The API implements various types of cacheable data structures to obtain any complex state representation. The underlying multi-level caching framework efficiently handles the memory requirement of any application and provides a transparent interface to the applications for using the multi-level cache. In addition, the multi-layer caching framework is configurable (i.e., an application can choose to leave out a cache level.)

In summary, the **contributions** of this work are as follows:

- We propose a scalable multi-level caching architecture to support the state management of complex streaming applications.
- We implement a prototype system by following the multi-level caching architecture. We also provide a prototype API in Java for the proposed multi-level caching architecture as a caching library for developing stateful stream applications that benefit from in-memory caching.
- We incorporate various data structures into the caching API to represent complex states within any stream application. In addition, the caching API manages the application state and cache levels transparently.
- We implement real-time streaming and synthetic applications by utilizing the prototype API to showcase the performance benefits.

The rest of the paper is organized as follows. Section 2 discusses the related literature and highlights the gaps in the current works. Section 3 presents the proposed system architecture. Section 4 provides the details of the implemented prototype system. Section 5 depicts the proposed multi-level caching library. Section 6 discusses experimental settings, benchmark applications, and various state management approaches used in the evaluation. Section 7 demonstrates the performance evaluation of various state management approaches while running the synthetic application as the benchmark. Section 8 shows the comparison of various state management approaches while running the real benchmark application. Finally, section 9 concludes the paper and highlights future work.

2 Related Work

Iterative Machine Learning (ML) algorithms such as PageRank, K-Means, and its variants[10] work in a step-by-step manner to converge towards a final solution. To deal with big state sizes, some algorithms either try to approximate smaller states[11] or use fewer iterative steps[12–14]. However, these approximate algorithms sacrifice accuracy for performance. Furthermore, for emerging application scenarios, such as the Internet of Things (IoT), social media data analytics, real-time business analytics, and fraud detection, continuous data streams must be processed with minimum delays[8]. In addition, real-time updates and maintenance of big states for these

applications become more challenging due to limited primary memory availability. Lastly, the state management framework needs to be transparent and support a variety of data structures to represent complex states[7] so that it is easier to create and deploy real-time stream-based analytics algorithms.

2.1 State Management in Big Data Frameworks

Big data and stream computing frameworks lack support for transparent and real-time state management[7]. For example, Apache Storm only supports stateless processing. Storm implements state management at the application level to ensure fault tolerance and scalability of the stateful applications. However, there is no native mechanism to manage the state within the framework. To overcome this limitation, an extension of Storm called Trident[15] has been proposed to provide an abstract layer for state management. Heron[4] uses stateful topologies consisting of spouts and bolts. In the Heron topologies, spouts and bolts automatically store their states when processing tuples. Heron also uses tuple acknowledgments for fault tolerance. Samza[16] manages large states (e.g., multiple GBs in each partition) by snapshotting states in local storage while using Kafka[17] to carry out state changes. Spark[2] implements state management by updating operations via transformations in the DStreams (i.e., a discretized stream). Fault tolerance in Spark is achieved with immutable collections or RDDs (resilient distributed datasets)[18]. During stream processing, Spark uses micro-batches, where an old state is used to generate another micro-batch result and a new state. Flink[19] can keep the states in memory (e.g., internal states as objects on the JAVA heap), can back up states in a file system (e.g., HDFS), and can also persist states in RocksDB[20, 21]. However, state management in these frameworks is mainly done with periodic checkpointing, which is insufficient for dealing with dynamic state updates in a real-time application. In addition, these frameworks provide key-value-based state management, suitable primarily for global snapshot and checkpointing for fault tolerance purposes. Furthermore, the state backend utilizes persistent storage, which is not fast enough to cope with dynamic state updates and migrations from real-time applications. Although some frameworks support in-memory object stores such as Redis as the state backend, the performance of real-time applications will degrade significantly due to continuous communication with the external store (e.g., Redis), object serialization/deserialization, and state migrations. Lastly, there is no support for creating different cacheable data structures to represent the state.

2.2 State Management with In-memory Object Store Databases

The performance and size of both RAM and faster persistent storage, such as solid-state drives (SSDs), have become more common these days. Thus, in-memory object store databases have become popular to provide millisecond latency to the applications. When data or objects are kept in such databases, primary memory is used by default to store the objects for getting a fast query response. Thus, the application will perform significantly faster if data/objects are fetched from memory. However, as data/objects become too large to fit into the memory, they will be automatically persisted on the disk by the in-memory object-store. Redis[22] is one such in-memory data structure store that can be used as a database, cache, and message broker. To achieve a better performance, Redis works with an in-memory dataset. Redis offers replication and, depending on the application scenario, can persist data periodically to disk or by appending each command to a disk-based log. If only an in-memory cache is required, persistence can be disabled. In Redis, data/objects can be stored as key-value pairs. There is also limited support to map data structures such as Lists, Sets, Hashes. RocksDB[20, 21] also offers a key-value interface where keys and values are arbitrary byte streams. RocksDB uses persistent storage for the state backend and log-structured merge trees to obtain significant space efficiency and better write throughput while achieving acceptable read performance. HazelCast[23] offers millisecond latency with its in-memory dataset and utilizes persistent storage for replication. However, all these aforementioned in-memory object stores only provide key-value-based interfaces, so it is not possible to store complex states/objects for a streaming application[7]. In addition, compared to the sub-nanosecond performance of the process heap, the sub-millisecond latency incurred by the communication with the in-memory object store would significantly reduce the performance of a streaming application if the backend is utilized for creating, storing, and updating states in real-time. Redisson Live Objects (RLO)[24] uses Redis as its data storage, where all changes to the object are translated to Redis commands and operated on a given Redis hash. The local JVM does not hold any value in the object's fields except for the field representing the hash's key. If a field of an object is updated, only those changes are pushed to Redis. If an object is created as an RLO, the object exists entirely in the Redis and can save critical JVM memory. However, there is a considerable performance overhead as updating a single field of a complex object triggers a Redis communication to synchronize the updated object in Redis. Thus, in-memory object stores are suitable for fault tolerance, but they are not well-suited to deal with the challenge of updating dynamic states of real-time streaming applications.

2.3 State Management with Program Caches

There are some preliminary works to achieve transparent state management [5, 9, 25, 26] for stream computing. ChronoStream[27] is one such framework that treats the internal state of an operator as a first-class citizen and provides state elasticity to cope with workload fluctuation and dynamic resource allocation. ChronoStream enables transparent dynamic scaling and failure recovery by eliminating any network I/O and state synchronization overhead. However, there is still limited support for the diverse representation of the state. In addition, most of the abstraction and presentation of operator states in stream computing is limited to key-value mapping for ease of implementation. However, for complex stream-based ML or data analytics applications, computational states may need to be stored in other forms such as graphs, hashes, and trees that cannot be stored using a key-value mapping. Program caches such as Caffeine[28–30], JCache[31], Guava Cache[32], and Ehcache[33] offer great performance to an application as these caches reside within the heap space of a process. However, these caches also work with key-value-based data structures and do not offer automatic state migration for diverse data structures when the cache size limit is reached.

In this paper, we propose a flexible caching architecture and a prototype caching library to improve stateful stream computing. As

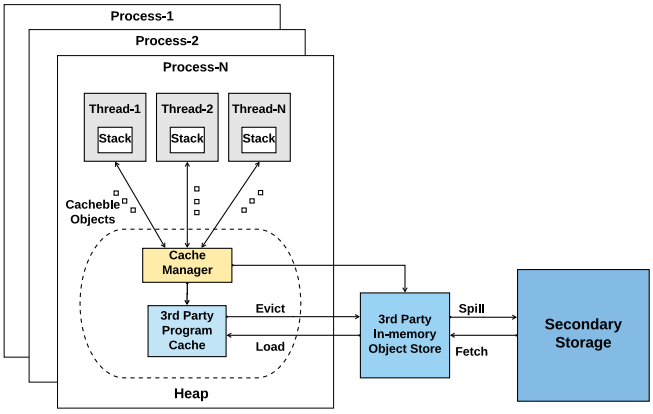


Figure 2: The proposed multi-level caching architecture. Each process has its own singleton program cache, which is shared by multiple threads within that process. Different processes on the same host share the same in-memory object store and secondary storage. The cache manager provides a transparent interface to the application code to enable caching support.

neither a program cache nor an in-memory object store is sufficient to handle this challenging problem, we bridge the gap by combining them to form a multi-level caching architecture to improve performance and scalability. In addition, we also support diverse data structures to represent complex states easily within the application. Lastly, the state management is kept transparent for easy deployment of stateful streaming applications. Thus, the application can use complex cacheable objects without the need to consider any key-value mappings, caching policies, and multi-layer memory.

3 System Architecture

This section proposes a multi-level caching architecture to support scalable memory management of stateful streaming applications. The default heap space of the process is considered the first level of a cache which is also the fastest level. For the other levels, in-memory object store and secondary storage are considered. In a host machine where multiple stream processes are running, each process will only have a limited amount of heap space. Multiple threads will share this heap space within that process. The heap space is used to store objects created dynamically within the process, and it is generally managed automatically. However, the heap space might run out if the state of streaming query is too large to fit in. If some part of the heap space can be managed, it would be easier to understand when the heap memory limit is approaching and, if so, how to deal with the increase in memory demand.

3.1 Objectives

To support scalable stateful computing for a streaming application, we first define the objectives of the system, which are as follows:

(1) Provide a transparent state management system for a streaming application. Thus, the state management system needs to decouple and hide the underlying memory management details from the application code.
(2) Support for multiple cacheable data structures (e.g., Lists, Maps), which can be used by the application to represent the complex state of a streaming query.
(3) Utilize faster memory and caches efficiently to store application states/objects. Thus, efficient caching policies should be utilized to keep frequently accessed objects closer to the application.
(4) Satisfy an increasing memory demand or surge of the streaming application to complete it successfully. Utilize multiple levels of the cache to migrate states/objects as required seamlessly. Provide support for configurable cache levels, so that the application can configure the memory allocated to a particular level, or can entirely leave out a level if needed.

3.2 System components

To achieve the desired objectives, we propose a multi-level caching architecture for stateful stream processing, as shown in Fig. 2. We assume that there can be multiple co-located processes (or operators) sharing the host's resources in a server, such as the CPU, primary memory, and disk. The core components for such a system with multi-level caches are as follows.

(1) **Cacheable Objects:** The first component of the system is the cacheable objects coming from different threads/processes. Generally, objects holding immediate computation results or the state of an application should be cached as these objects may need to be accessed frequently by the application. As we consider a multi-level memory, an object created in the program memory may need to be moved from the program memory to the in-memory object store or the secondary storage. Hence, all the cacheable objects must be *serializable*.
(2) **Cache Manager:** In a typical application/process, objects are created dynamically in the heap space, and there is usually no need to manage this memory. However, a stateful streaming application might run out of program memory, so managing a part of the heap is better to handle the surge in memory demand. The cache manager acts as an interface between the application code and a managed chunk of memory in the heap space and manages all the different memory levels, including caches. The memory management is transparent to the application, which means the cache manager decides where to create objects and keeps track of state/object migration and deletion.
(3) **Memory Levels and Caches:** The available memory spans across different levels, such as a 3rd party cache in the heap space, in-memory object store, and secondary storage. The 3rd party cache located in the heap space is the first and the fastest level of this architecture. It is located inside the process' dedicated heap space, so multiple threads can only share it within that application/process. The benefit of using a 3rd party cache such as Caffeine[28], JCache[31], Guava Cache[32], and Ehcache[33] is that these caching libraries already have efficient caching policies suitable to run for different workload types. Thus, the cache manager can just configure and utilize the program cache to create and store objects. When a surge of memory demand happens (i.e., the cache becomes full), the caching policy of the 3rd party cache automatically determines the victim object(s) for eviction. The cache manager then migrates these objects to the next level of memory. The in-memory object store is the second level that resides in the primary memory (RAM) but is slightly slower than the first level due to the communications involved to create/fetch/delete

objects by the in-memory object store module/software. However, it is still significantly faster than secondary storage. The in-memory object store is dedicated to the host VM/server, and it is shared by multiple applications/processes within that server. The last and slowest level of the architecture is the secondary storage (e.g., SSD or HDD), which is only used when objects are spilled or backed up from in-memory object stores to the disk. The in-memory object store automatically handles the disk spilling following a predefined eviction/snapshotting strategy. Although secondary storage causes a significant overhead, our proposed architecture still utilizes it. This is because the primary memory may not be sufficiently large even after using both program caches and the in-memory object store databases to minimize memory consumption.

(4) **Caching Policy:** The 3rd party program caches run a caching policy to determine which objects to evict from the cache when the cache is full. In the case of an eviction, the cache manager can process the event and fetch the evicted object from the program cache. Then this evicted entry can be migrated to other levels of the memory hierarchy. Depending on the application scenario, the performance of different caching policies might vary. However, for window-based or iterative streaming applications, a caching policy that considers both frequency and recency should perform well because the application tends to work on objects that have been created recently in the current application window.

3.3 Design Choices and Trade-offs

A multi-level cache may introduce some performance degradation to the application due to cache management, objects migration, communication between different memory components. Thus, it might not be suitable for all the streaming applications to use the complete multi-level cache architecture as shown in Fig. 2. The proposed architecture can be varied by deciding whether a particular memory level or cache is needed or not. Here, we discuss the two different design choices and their trade-offs.

(1) **In-memory Object Store Only:** In this approach, the fastest level (program memory) does not incorporate any cache, and all the objects are maintained entirely in the in-memory object store based backend. If the in-memory store runs out of memory (due to heavy memory demand or use of the same store from multiple processes within the same server), it will automatically manage the disk spilling of objects. If an application needs to maintain a low memory footprint in the process heap space, it should only use the in-memory object store-based state backend to store and update states. In this case, the application only uses the state backend for fault tolerance, and it should not carry out updates too often. Otherwise, frequent updates to the states that resided in the in-memory object store can degrade performance.

(2) **Program Cache with In-memory Object Store:** In this approach, the complete multi-level caching setup will be used as shown in the system architecture. A program cache will automatically manage a part of the process heap space while creating/fetching/updating objects. In addition, if any objects are evicted from the program cache, they should be seamlessly migrated to the in-memory object store. There will be a slight overhead on the program cache library in this approach. However, if the application has a large memory footprint and heavily reuses objects, the performance improvement may outweigh the cache management overhead. In addition, the program cache size can also be varied to tune the memory footprint of the application in the process heap space.

4 Prototype System Implementation

Most existing stream computing platforms are implemented in Java or use the Java Virtual Memory (JVM) to create stream operators. Hence, we have implemented both of the caching approaches in Java to offer diverse cacheable data structures for representing the state of complex streaming applications. Furthermore, we have extended the RAPID[34] system to enable caching support for stateful complex stream-based data analytics applications. RAPID is a complete stream computing platform where users can perform interactive social media data analytics. The users can submit their queries through web portals or Java GUIs, and the queries are collected by using Apache Kafka as the message broker. In addition, the system uses Apache Storm as the stream compute engine, where Twitter streaming APIs collect real-time streaming data for any chosen topic, and Storm spouts are used to pre-process and store the streaming data in MongoDB. Finally, the system creates bolts (compute units) to execute the user queries.

4.1 In-memory Object Store Only

In this approach, we use an in-memory object store (i.e., Redis) as the first level of the cache. In Redis, *Serializable* objects can be stored as a key-value pair. Thus, for the applications that need caching support, the classes for which many objects are created and used frequently must be serialized so that these objects can be stored and fetched from in the in-memory object at any time. Note that, for in-memory object stores such as Redis, the most outer level is the secondary storage managed by Redis and used for the periodic snapshot of states for fault tolerance or spilling of states in case of memory shortage. We implement a cache management library that provides flexible APIs to create cacheable objects for a streaming application. The cache manager is responsible for managing the objects entirely without any intervention from the application logic. Thus, the entire caching logic is abstracted from the application. We employ *Redis Buckets* to create objects in the multi-level caching architecture. In this type of Redis object, the serialized objects can be created in Redis memory as a key-value pair. If the object is fetched by the JVM and modified, the changes will not be reflected automatically in the Redis cache. Thus, the outdated Redis buckets are updated after they have been fetched and modified in the JVM.

4.2 Program Cache with In-memory Object Store

To mitigate the need to write custom caching policies tailored to a particular application, existing caching libraries can be used to create program caches in the JVM. These program caches can provide near-optimal hit-rate across various application scenarios. In this architecture, we use Caffeine[28] as the program cache. Thus, instead of creating objects directly into the JVM, we utilize the cache manager to create the Serialized objects in the Caffeine cache. We employ a centralized singleton Caffeine cache for a particular JVM so that multiple threads within the same application can use

the cache without overwhelming the JVM memory capacity. A predefined size of the Caffeine cache can be defined to occupy the JVM memory, and the rest of the JVM memory is kept free to create any other objects within the application. The size of the cache can be represented either as a total weight for all the objects in the cache (e.g., total memory consumption in MB) or a total number of objects that the cache can hold before eviction.

The Caffeine cache uses the TinyWindowLFU[35] caching policy, which considers both frequency and recency. Thus the caching policy works well across various application scenarios with different cache access patterns. As window-based streaming algorithms use the most recent and frequently occurring states, the caching policy can help to keep most of the required objects by the algorithm in the cache. Only when the size of the cache is full the caching policy will automatically decide which objects need to be evicted. As the evicted objects/states might be needed in subsequent operations, we store the evicted objects in the secondary caching level (Redis). If the object is needed in the future (in case of a cache miss), the object is loaded from the Redis cache to the Caffeine cache. Note that, in this approach, objects are always used/updated in the JVM to improve the performance of the application. Thus, if an application does not require a large memory capacity, all the objects will reside in the JVM (Caffeine cache), and the application will not be impacted by the performance degradation caused by the slower memory (Redis). Note that, using any data structure or caching such as Caffeine will introduce some performance overhead due to the cache management compared to the use of bare JVM memory. Note that, this paper focuses on the architectural perspective of the memory limitation of streaming applications. We have utilized the existing available technological tools (e.g., Caffeine, Redis) to implement the prototype system. However, the proposed architecture can be extended to support other technological tools and design choices. In fact, while developing the prototype system, we have considered alternatives such as Memcached, Hazelcast, etc. These alternatives did however showcased subpar performance as compared to the Caffeine and Redis combination.

5 Multi-level Caching Library

In this section, we provide a brief discussion on the various core components of the multi-level caching library[1] implemented in Java. Fig. 3 provides a class diagram for the implemented multi-level caching library, which can be used in a streaming application to provide caching support with flexible data structures, so that complex states of the application can be represented easily. Depending on the application scenario, the complete multi-level setup can be used where both the program cache (i.e., Caffeine) and the in-memory object store (i.e., Redis) can be used as different levels of the cache. In addition, a particular level of the cache can also be disabled. For example, if an application requires cacheable data structures with only Redis as the cache, the program cache can be disabled. Now, we briefly discuss the core classes of the caching library as follows.

- **Cacheable:** This is an interface that must be implemented by a class for which the created objects can be cached if required. When an object is created for a class that inherits this interface, the object automatically gets a unique ID. This unique id is important because the cache works as a map where each value/object has to be associated with a key. For example, if we require to cache the objects from an array, there is no key associated with these objects. Hence, providing an interface to manage the unique keys used with the various objects addresses this problem.
- **ApplicationDriver:** This is the main application code which runs a particular streaming algorithm. The application can choose to create any number of cacheable objects or data structures holding cacheable objects. Whenever creating any object, the application driver should use the CaffeineObjectFactory. Depending on the caching approach, the underlying memory management API decides whether to create the object in the Caffeine cache or Redis.
- **CacheableAppClass:** There can be one or more Cacheable classes for which the objects can be cached if required. Note that all these classes must implement a Serializable interface to ensure that cached objects can be serialized/deserialized seamlessly across different memory levels.
- **CaffeineCacheManager:** This is the primary cache management class that manages the underlying memory levels. This class can be configured to choose whether to use a primary program cache such as Caffeine and set the size of the Caffeine cache if used. In addition, it can also be used to configure the in-memory Redis store. Data structures that hold cacheable objects such as CaffeineHashMap, CaffeineArrayList, and CaffeineHashset are also managed from this class. If the Caffeine cache is used, this class creates and holds the data structure for a Caffeine cache shared across multiple threads within the same application/process. Thus, objects from different data structures (e.g., CaffeineHashMap) or different threads are maintained by using the unique keys. When a particular object is created, migrated, or deleted, the cache manager performs the necessary memory management tasks associated with the particular operation. For example, if an object is accessed, the cache manager can check whether it is available in the primary cache (Caffeine) and passes a valid reference of this Caffeine object to the application. If the object does not exist in the primary cache, the object is loaded from the other levels to the primary level, and then a valid reference is passed on to the application code. Thus, from an application's perspective, the objects are always accessible from the object reference it holds, but underneath, the cache manager resolves the valid reference to the object.
- **CaffeineObjectFactory:** This factory class is used to create objects in the Caffeine cache. An application can create regular JVM objects of a cache-enabled class if no caching support is required. However, if caching support is required, the cacheable objects must be created through the CaffeineObjectFactory, so that the object reference can always be resolved from the cache by the CacheManager.
- **RedisObjectFactory:** This factory class is used to create, fetch or delete Redis objects. The application code does not have access to this class and is exclusively used by the CaffeineCacheManager as required.

[1] https://github.com/tawfiqul-islam/MemoryManagerJVMRedis

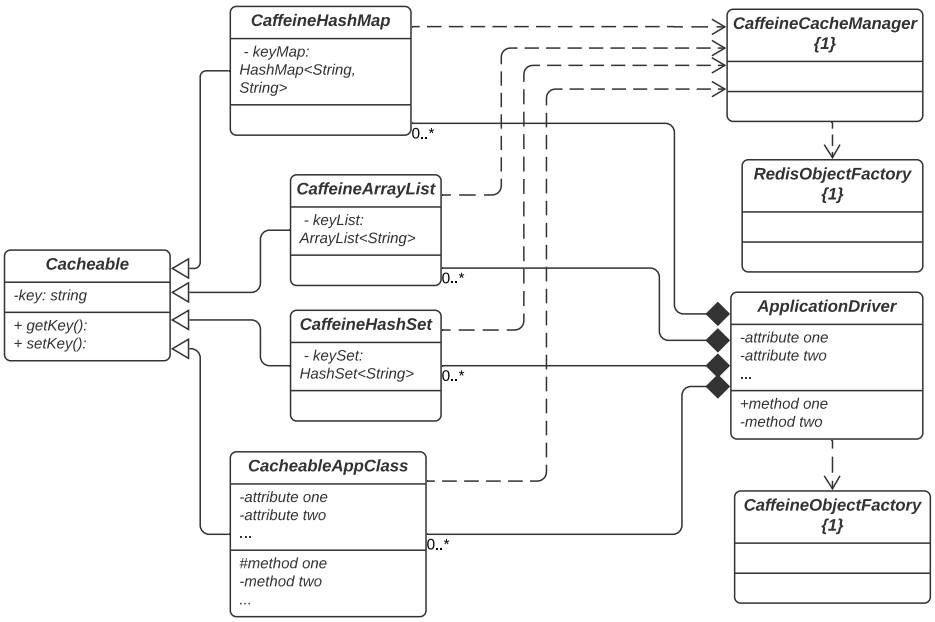


Figure 3: Class diagram for the multi-level caching library. Application driver is a sample application code that can use any type of cacheable data structure to represent a complex state. The cache manager provides a transparent interface and automatically manages the state updates/migrations. The application code can also utilize the cacheable interface to create cacheable objects.

How to use the Cacheable API: The caching library can be imported in Java-based stream application development. As an example, consider an application that requires an ArrayList where any element of the ArrayList can be cached to provide faster access. The application developer needs to inherit the Cacheable interface for the class objects which will be stored in the CaffeineArrayList. In addition, all the array objects need to be created through the CaffeineObjectFactory. Now, the CaffeineArrayList supports normal array operations that the application developer can use. The underlying memory management layer will only keep the associated unique keys for each object in its internal data structure, but the actual objects will be kept in one of the memory levels. Each access to an individual object may cause a cache hit/miss for the Caffeine cache. If the Caffeine cache becomes full, the memory manager will automatically migrate evicted objects from the Caffeine cache to the Redis cache. Thus, in case of a Caffeine cache miss, the desired object will be resolved from the Redis cache. Similarly, in case of Caffeine cache hit, the object will be resolved from the cache for faster access.

Note that the multi-level caching library can be extended to add more data structures. Although we have chosen Caffeine as the program cache and Redis as the secondary cache, the proposed library can be extended to support any other program caches and in-memory object store databases. The implemented caching library can be used with JVM-based stream computing frameworks such as Apache Storm, Apache Spark, and Apache Flink, as it is written in Java. Thus, the streaming application needs to be written in Java, and should import the caching library to use cacheable data structures instead of using typical non-cacheable data structures provided by the programming language.

6 Experiment Setup

In this section, we first discuss the benchmark applications. Then we discuss the state management approaches implemented to be compared in the performance evaluation.

To conduct the experiments, the RAPID[34] system was deployed in a Virtual Machine (VM) located in the Nectar Research Cloud[2]. The VM has 8 CPU cores, 32 GB of memory, and 100 GB of disk storage. In addition, the caching library is plugged into the RAPID system to provide caching support for the benchmark applications. The caching library uses Redisson 3.12.0 for communicating with Redis for state migration and Caffeine 2.9.2 as a program cache. Note that, the RAPID system utilizes Apache Storm as its compute engine. Hence, both the synthetic and the real applications run as storm bolts (processes) in the system.

6.1 Benchmark Applications

(1) **Synthetic Application:** We have developed a synthetic application in Java to evaluate the application performance and memory footprint for different state management approaches. The application collects data from a file stored in the disk and dynamically creates objects for these files in real-time to mimic a sizeable increasing state. Hence, each object contains a large file (approximately 50MB), which the application may require access to make changes. The JVM heap space may not be sufficiently large to hold all these data objects that need to be processed, and the access request to any data object can be repeated. The motivation behind the design of this application is to simulate the out-of-heap-space problem that occurs with memory-hungry streaming applications.

[2]https://ardc.edu.au/services/nectar-research-cloud/

To present a realistic object access pattern to different state management approaches, we have generated the access pattern by following a Zipf distribution. There are 102 unique objects, with 1000 accesses to these objects in the experiment. The JVM heap space memory is capped at 3GB (3000MB) by using VM options in the JVM (-Xmx 3000MB) so that we can simulate the limited heap space memory scenario. For the Caffeine-Redis-based multi-level cache, the Caffeine cache size is varied from 10% to 30% of the total JVM heap space with a 5% increment. In addition, we have utilized a weighted loading caching for Caffeine so that the cache eviction triggers when the total memory footprint of the cached objects exceeds the predefined maximum weight of the cache. All the experiments are conducted three times, and the average results are taken.

(2) **Real application:** We have chosen the online Spatio-temporal event detection algorithm[36] as our real benchmark application. This algorithm uses social media data streams to detect events at different time and space resolutions. The algorithm utilizes a quadtree-based method to split the geographical space into multi-scale regions based on the density of social media data. Then, an unsupervised statistical approach is performed to identify regions with an unexpected density of social posts. The algorithm also estimates the event duration by merging events in the same region at consecutive time intervals. A post-processing stage is used to filter fake or incorrect events. This application uses rolling windows to process Tweets in a particular time interval where the time interval can be multiple days. Thus, as the application progresses, the size of the quadtree increases. In addition, the application needs to update states (quadtree join, merge, and prune) in real-time. This algorithm requires representing and updating complex computation states. We have implemented it with our proposed multi-level caching library to enable caching support for the quadtree structure. The motivation behind the design of this application is to observe the applicability of the proposed caching library in a real-time application that requires caching of complex object states.

We collected Twitter streaming data for four different time intervals from December 1, 2020, to December 8, 2020, to be used with the event detection algorithm. The four intervals have an increasing number of tweets, where the number of collected tweets are 120k, 131k, 157k, and 172k, respectively. We have varied the size of the Caffeine cache from 10% to 30% of the total collected tweets with a 5% increment. In addition, we have utilized a size-based loading caching for Caffeine so that the cache eviction triggers when the total number of cached objects exceeds the predefined maximum size of the cache. All the experiments are conducted three times, and the average results are taken.

6.2 State Management Approaches:

We have implemented the benchmark applications in the proposed multi-level caching approach, the Redis-only state management backend, and as a native JVM-based application (the best baseline regarding performance).

(1) **JVM-based (native application):** This is a native implementation of the benchmark applications, where no caching support is provided. However, the algorithm only uses the JVM heap space to maintain application states. Thus, the JVM-only approach serves as the fastest baseline, which can be used to compare the proposed approaches for streaming windows that fit into the memory.

(2) **Redis-only Cache:** In this approach, the Redis cache is used to store all the objects of an application to save up critical heap space. Whenever access to an object is required, the object state is migrated from the Redis to the JVM for the application, and then the updated state is propagated back to Redis.

(3) **Caffeine-Redis Cache:** This is a multi-level caching approach that spans from the JVM heap space to the Redis cache. The cache manager manages a part of the JVM heap space with the Caffeine cache. Thus, objects are created and stored in the Caffeine cache for faster performance. However, when the cache is full (either cache size limit reached or the total cache weight limit is reached), the caching policy in Caffeine starts evicting objects. Therefore, we seamlessly migrate these objects to Redis so that they can be fetched later if required.

7 Performance Evaluation - Synthetic Application

We first evaluate the performance of various state management approaches while running the synthetic application as the benchmark. Then, we present the application performance across different alternatives coupled with the memory footprint incurred by these approaches. Finally, we present the effect of cache sizes on cache hit rates for the multi-level cache-based state management approach.

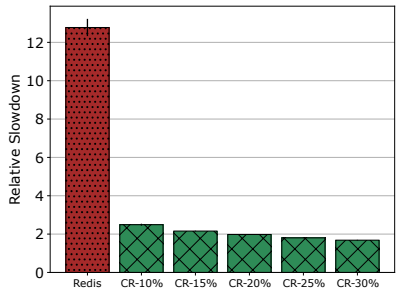
7.1 Evaluation of Application Performance

As the synthetic application demonstrates the characteristics of a real-time application that is memory-hungry and makes real-time access to a large amount of data, the JVM-based baseline runs out of memory after completing the application only partially. Thus, to conduct the relative performance analysis between the proposed approaches, we collect the application completion times up to a point when the JVM approach fails. In addition, we record the completion times for an entire run of the synthetic application for the Redis-based approach and the proposed multi-level approach with varying cache sizes. The relative slowdown for an approach as compared to the native JVM is calculated by:

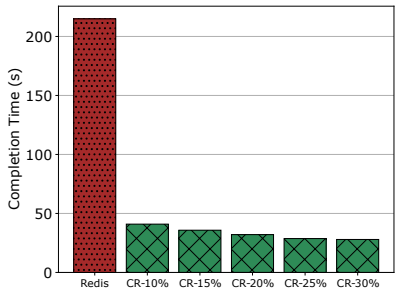
$$Relative\_Slowdown = \frac{T_{proposed}}{T_{JVM}}$$

where $T_{proposed}$ is the time for the application to complete in a chosen interval by a proposed approach. Thus, the lower the relative slowdown, the better (and closer) an approach performs to the native JVM.

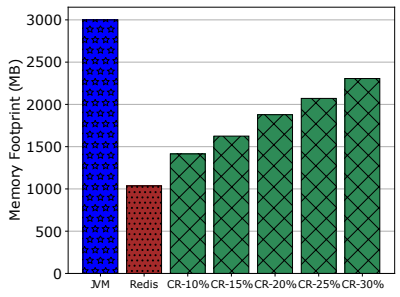
Fig. 4a depicts the relative slowdowns incurred by various state management approaches as compared to the native JVM implementation. Note that these results are collected from a partial run of the application (a sequence of 382 accesses to different objects by the application), as the JVM fails after this point. The results indicate that the Redis-based state backend performs the worst as it creates and manages all the objects in Redis, resulting in a 12.3x performance slowdown. However, the relative slowdown is lower for the multi-level cahing-based approach and varies between (2.2x to 1.8x) for different cache sizes. As shown in Fig. 4b, the Redis-based approach



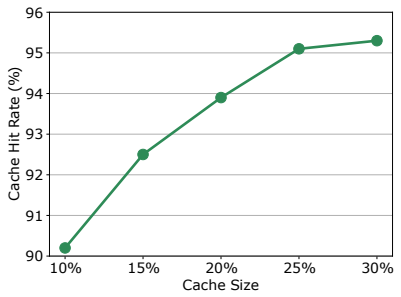
(a) Relative Slowdown from partial run (lower is better)



(b) Application Completion Time (lower is better)



(c) Memory Footprints (lower is better)



(d) Caffeine Hit Rates (higher is better)

Figure 4: Experimental results for different state management approaches while running the synthetic application. (a) shows the relative slowdowns of different state management approaches as compared to the native JVM-based approach for a partial run of the application, (b) shows the completion times of different approaches for a complete run of the synthetic application, (c) shows the comparison of the memory footprints for different approaches, and (d) shows cache hit rates for the multi-level caching based state management approach with varying cache sizes.

takes 212 seconds on average for completing the execution of the application. The multi-level caching-based approach showcases stable performance, and the application completion time reduces linearly with the increase of the cache size. It can be observed that a 30% cache size of the total JVM memory performs the best and completes the application within 25 seconds.

7.2 Evaluation of Memory Footprint

In this evaluation, we investigate the memory usage of all the approaches. As depicted in Fig. 4c, the JVM memory footprint is highest (3GB) as it tries to fit all the data into the heap space. Even when using the highest amount of heap space, the JVM-based approach still fails to complete the execution of the application. All the other compared approaches have lower memory footprints than the JVM and can still complete the execution of the application. The Redis-based approach consumes the smallest amount of memory as all the objects are always kept in the Redis memory, outside the heap space. Note that, the memory footprint incurred by Redis includes only the memory utilized in the JVM, but excludes the total Redis memory used in a host node. For the multi-level caching approach, the memory footprint increases with cache size. Even though the Redis-based approach has a low memory footprint, the application performance is degraded significantly (the approach sacrifices performance to save memory). On the other hand, the multi-level caching approach reduces the memory footprint and showcases stable performance (achieving a delicate balance between memory usage and performance overhead). Lastly, depending on the application scenario, an appropriate cache size can be selected to tune the application to be either memory-efficient or performance-efficient.

7.3 Effects of Cache Sizes on Cache Hit Rate

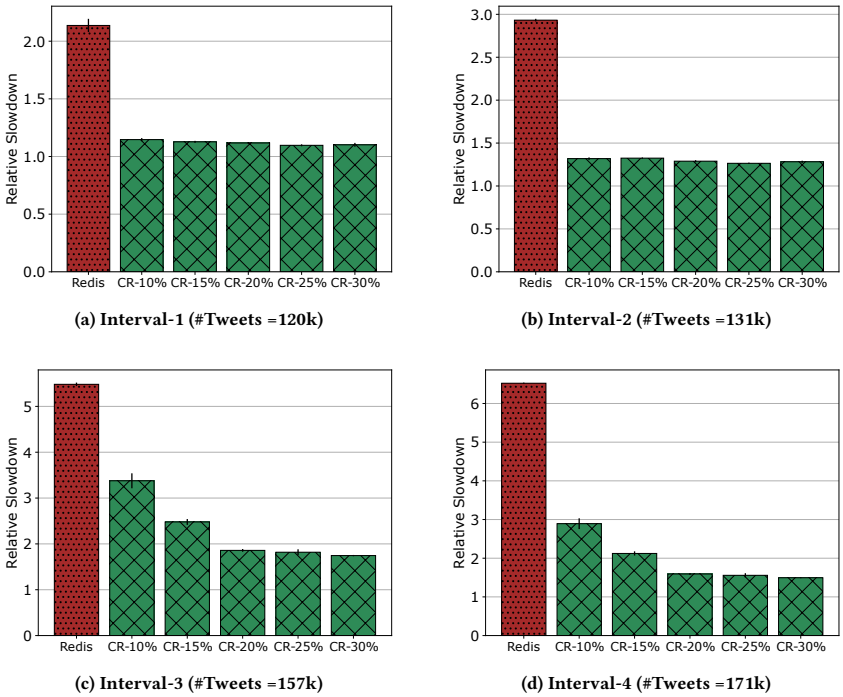
This evaluation observes the effects of cache sizes on the cache hit rate percentage produced by the multi-level caching approach. In a complex stream-based data analytics algorithm, it is crucial to produce high hit rates since a cache miss implies that the objects need to be fetched from the secondary level (in-memory object store such as Redis), which will negatively impact the application performance. As highlighted in Fig. 4d, even for a small cache with only 10% of the JVM heap, the cache hit is above 90%. Furthermore, as the size of the cache increases, the cache hit rates also increase.

8 Performance Evaluation - Real Application

This section evaluates the performance of various state management approaches while running the real event detection application as a benchmark. We compare the performance of these approaches in terms of application performance. We also demonstrate the effects of cache management overhead and cache sizes.

8.1 Evaluation of Application Performance

We first evaluate the performance of different caching approaches for the event detection algorithm. As the JVM-based native approach serves as the fastest baseline, we have used the application completion time for the JVM to calculate the relative slowdown



(a) Interval-1 (#Tweets =120k)

(b) Interval-2 (#Tweets =131k)

(c) Interval-3 (#Tweets =157k)

(d) Interval-4 (#Tweets =171k)

Figure 5: Relative performance comparison of the Redis-only and the Caffeine-Redis based multi-level caching approach relative to the JVM-based implementation of the event detection algorithm. (a)-(d) show the relative slowdowns observed for different time intervals: e.g., (a) has the smallest interval, whereas (d) has the largest interval in terms of the number of collected Tweets.

incurred by the other state management approaches. Note that, for comparing with the best baseline, we assume that the application's data perfectly fit in the JVM memory for the JVM approach so that the application can be completed within the shortest time. Next, we experimented with varying cache sizes for the proposed multi-level caching approach to investigate whether the application performance is comparable to the native JVM when we have a limited memory capacity due to smaller cache sizes. This evaluation will help us understand whether the process heap size reduction can be tolerated with managed caches that take limited space in a heap.

Fig. 5 shows the relative slowdown comparison of the native JVM-based implementation of the event detection algorithm with both Redis-based and the Caffeine-Redis multi-level cache-based state management implementations for four different time intervals. There is no memory, object size, or cache size limit on the JVM. Thus, its performance is always the same. For the Redis-based approach, the state of objects is directly saved into Redis. Thus, the objects are fetched from Redis whenever the application needs to change/update states. When the total number of tweets increases, many states need to be saved in the Redis backend, resulting in a performance slowdown for the application. As shown in Fig. 5a, for a smaller interval, the performance slowdown is 2x for the Redis-based approach as compared to the native JVM implementation. However, as the interval size increases, performance slowdown also increases (Fig. 5b, 5c, 5d for up to 6x). The Caffeine-Redis implementation shows varying performance with different cache sizes. For small and medium intervals (Fig. 5a-5b), the slowdowns incurred for different cache sizes range between only 1.1 (CR30% for interval-1) to 1.25 times (CR10% for interval-2). For large intervals (Fig. 5c-5d), when the cache size is smaller (CR10% to CR15%), the performance slowdown observed is up to 3 times. However, as the cache size increases, the performance improves, and the application completion time decreases as most of the application states/objects can be accessed directly from the Caffeine cache due to an increased number of hits. As an example, when the cache size is 30% in interval-4 (Fig. 5d), it only produces a 1.4 times slowdown as compared to the native JVM. Thus, when the number of objects exceeds the size of the cache, the caching policy automatically selects the victim objects for eviction, which are migrated to the Redis cache. If these objects are accessed again, they are automatically loaded from the Redis cache due to a cache miss event in the Caffeine cache.

8.2 Effects of Cache Management Overhead

In this evaluation, we investigate whether there is an overhead associated with using the multi-level Caffeine-Redis based caching approach. This experiment is done with varying cache sizes, but all the caches are large enough to fit all the application data. Thus, the performance slowdown incurred in this experiment will result from cache management only. Fig. 6 shows the relative performance slowdown of the Caffeine-Redis approach for varying cache sizes. When a JVM based application is launched, the first requests it receives are generally significantly slower than the average response time. This warm-up effect is due to class loading and bytecode interpretation at startup. After 10k iterations, the main code path

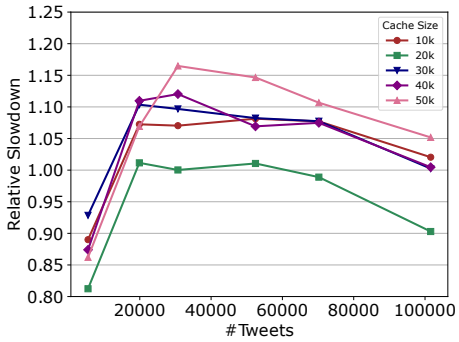


Figure 6: Effects of cache management overhead on the application performance for varying cache sizes of the proposed multi-level cache. The relative slowdowns are calculated in terms of the native JVM performance of the application.

is compiled and 'hot'. Thus, when there are only a small number of tweets (<10k), the native JVM performance is worse than the Caffeine cache. For smaller cache sizes (around 10k to 20k), the Caffeine cache management exhibits warm-up delays, which result in slight performance slowdowns (around 10-15%). However, a surprising trend can be observed for huge caches with many data (tweets). For these cases, the native JVM application experiences slowdowns due to many object creations and deletions, which result in Garbage Collection (GC) overheads. This overhead increases with interval size. For large intervals, the Caffeine-Redis cache overcomes the warm-up delay and performs better than the native JVM as it does not suffer from the GC overhead.

8.3 Effects of Cache Size

We now discuss the cache statistics recorded from the Caffeine cache while running the event detection algorithm for various cache sizes over a large interval (interval-4 with 171k objects). When the cache size is smaller, many cache evictions trigger the migration of objects to the Redis cache. As the application needs many of these objects in the subsequent operations, the Caffeine cache misses are resolved from the Redis cache. As communicating and fetching objects from Redis is slower than the process heap memory, Redis load time is the main performance bottleneck of the application. When the cache size is around 30% of the total objects, the cache is big enough to hold most of the objects so that only a few eviction happens (Fig. 7a), and a very high rate (99.99%) is achieved. The Caffeine cache policy performs well even for smaller cache sizes as the cache hit rate is above 99.5% for a cache size equal to 10% of the JVM heap. Fig. 7b presents the cache load time statistics with increasing cache size. When the cache size is small, many cache evictions trigger the migration of objects to the Redis cache. As the application needs many of these objects in the subsequent operations, the Caffeine cache misses are resolved from the Redis cache, so the Redis load time increases. Table 1 and Table 2 provide the raw numbers for hits/misses, and the Redis load times, respectively.

9 Conclusions

Stream processing is used in many real-time applications that deal with large volumes, velocities, and varieties of data. For complex data analytics algorithms, the intermediate states of computations might need to be retained in memory, triggering a considerable surge in memory demand to run the application successfully. This paper proposes a multi-level caching architecture to mitigate the surge of memory demand from the processes running complex streaming data analytics and ML algorithms. We have implemented two approaches with the caching architecture to support scalable state management. The first approach uses an in-memory object store as a state backend and provides diverse data structures to represent complex states. The second approach uses a program cache in the heap space to reduce the communication overhead with the in-memory object store for faster performance. We have provided a prototype multi-level caching library in Java that can be used to implement scalable streaming applications. The underlying cache management completely abstracts the multi-level cache implementation from the application and handles seamless migration of states/objects across different levels of the cache. We have implemented a real-time streaming algorithm in Apache Storm and extended the algorithm implementation with the proposed caching library. We have also implemented a synthetic application to demonstrate the limited memory problem in the process heap and showcased how the implemented approaches tackle the issue. We have also conducted extensive experiments with real workloads to compare the performance of the proposed approach against the fastest implementation. The experimental results showed that our proposed multi-level caching architecture could handle large computational windows.

9.1 Limitations and Future Work

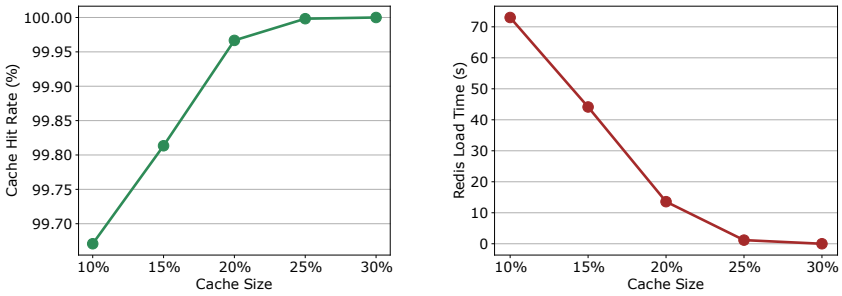
In the future, we plan to extend this work by exploring custom caching policies to be used for migrating objects across various caching levels instead of using the default policy of the Caffeine cache. We plan to investigate the problem of optimal allocation of memory to each layer of the cache. Currently, we use a fixed amount of memory for each layer, but the implementation supports configurable memory allocation for each layer. Thus, we plan to extend this work by making the allocation workload agnostic. We also want to extend more memory-intensive ML algorithms with the proposed caching architecture to investigate the performance in various application and workload scenarios. In addition, we would like to implement more cacheable data structures into the caching library so that it would be easier to support diverse and complex states. We plan to explore the proposed caching architecture with different types of program caches and in-memory object stores, and deploy our solution to different stream computing platforms. This will allow us to conduct evaluation among the native frameworks with and without the cache-support.

Acknowledgments

This research is funded by the Defence Science and Technology Group (DSTG), Edinburgh, South Australia, under contract MyIP: 7293.

References

[1] Apache storm. https://storm.apache.org/. Accessed: 2022-05-18.
[2] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine.



(a) Hit Rate **(b) Redis Load Times.**

Figure 7: Effects of cache sizes for the multi-level Caffeine-Redis cache based approach. (a) shows the hit rates for different cache sizes, (b) shows the Redis load times caused from the cache misses by different cache sizes.

Table 1: Caffeine Cache Hits and Misses Statistics

5%		10%		15%		20%		25%	
hits	misses	hits	misses	hits	misses	hits	misses	hits	misses
145268136	512481	142365460	470148	130905414	244735	131106445	43704	131147852	2297

Table 2: Redis Load Time Statistics

Cache Size	5%	10%	15%	20%	25%
Redis Load Time (s)	139.62	73.01	44.13	13.58	1.17

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4), 2015.

[4] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[5] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 725–736, 2013.

[6] Tiziano Matteis and Gabriele Mencagli. Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach. *Int. J. Parallel Program.*, 45(2):382–401, apr 2017.

[7] Xunyun Liu and Rajkumar Buyya. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Computing Surveys*, 53(3), may 2020.

[8] Quoc-Cuong To, Juan Soto, and Volker Markl. A survey of state management in big data processing systems. *The VLDB Journal*, 27, 12 2018.

[9] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. Elastic stateful stream processing in storm. In *2016 International Conference on High Performance Computing and Simulation (HPCS)*, pages 583–590. IEEE, jul 2016.

[10] *Top 10 algorithms in data mining*, volume 14. 2008.

[11] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 1–12, New York, NY, USA, 2005. Association for Computing Machinery.

[12] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G. Dimakis, and Constantine Caramanis. Frog wild! - Fast PageRank approximations on graph engines. *Proceedings of the VLDB Endowment*, 8(8):874–885, 2015.

[13] Wenting Liu, Guangxia Li, and James Cheng. Fast PageRank approximation by adaptive sampling. *Knowledge and Information Systems*, 42(1):127–146, 2015.

[14] Ziv Bar-Yossef and Li Tal Mashiach. Local approximation of pagerank and reverse pagerank. *International Conference on Information and Knowledge Management, Proceedings*, pages 279–288, 2008.

[15] Trident state. https://storm.apache.org/releases/current/Trident-state.htm. Accessed: 2022-03-18.

[16] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, aug 2017.

[17] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

[19] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, aug 2017.

[20] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[21] Rocksdb: A persistent key-value store for fast storage environments. http://rocksdb.org/. Accessed: 2022-03-18.

[22] Redis. https://redis.io. Accessed: 2022-05-18.

[23] Hazelcast: The real-time intelligent applications platform. https://hazelcast.com/. Accessed: 2022-03-18.

[24] Introduction to redisson live objects. https://gist.github.com/jackygurui/f889744539fe2e7e2152a318f90153b3. Accessed: 2022-03-18.

[25] E-Storm: Replication-Based State Management in Distributed Stream Processing Systems. *Proceedings of the International Conference on Parallel Processing*, pages 571–580, 2017.

[26] Kasper Grud Skat Madsen, Philip Thyssen, and Yongluan Zhou. Integrating fault-tolerance and elasticity in a distributed data stream processing system. *In Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM'14)*, 2014.

[27] Yingjun Wu and Kian-Lee Tan. Chronostream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering*, pages 723–734, 2015.

[28] Caffeine: A high-performance caching library for java. https://github.com/ben-manes/caffeine. Accessed: 2022-03-18.

[29] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 94–106, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Lightweight robust size aware cache management. *ArXiv*, abs/2105.08770, 2021.

[31] Jianyu Fu, Yubo Liu, and Guangming Liu. Jcache: Journaling-aware flash caching. *IEEE Access*, 8:61289–61298, 2020.

[32] Guava cache. https://github.com/google/guava/wiki/CachesExplained/. Accessed: 2022-03-18.

[33] Ehcache. https://www.ehcache.org/. Accessed: 2022-03-18.

[34] Kwan Hui Lim, Sachini Jayasekara, Shanika Karunasekera, Aaron Harwood, Lucia Falzon, John Dunn, and Glenn Burgess. Rapid: Real-time analytics platform for interactive data mining. In *Machine Learning and Knowledge Discovery in Databases*, pages 649–653, Cham, 2019. Springer International Publishing.

[35] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage*, 13(4), nov 2017.

[36] Yasmeen George, Shanika Karunasekera, Aaron Harwood, and Kwan Hui Lim. Real-time spatio-temporal event detection on geotagged social media. *Journal of Big Data*, 8(1):1–28, 2021.