Short and Sweet Checkpoints for C-RAN MEC

Saidur Rahman
Montana State University
Bozeman, MT, USA
saidurrahman@montana.edu

Apostolos Kalatzis
Montana State University
Bozeman, MT, USA
apostoloskalatzis@yahoo.com

Mike Wittie
Montana State University
Bozeman, MT, USA
mike.wittie@montana.edu

Ahmed Elmokashfi
SimulaMet
Oslo, Norway
ahmed@simula.no

Laura Stanley
Montana State University
Bozeman, MT, USA
laura.stanley@montana.edu

Stacy Patterson
Rensselaer Polytechnic Institute
Troy, NY, USA
sep@cs.rpi.edu

David L. Millman
Montana State University
Bozeman, MT, USA
david.millman@montana.edu

***Abstract*—Mobile applications can improve battery and application performance by offloading heavy processing tasks to more powerful compute nodes. While Mobile Edge Computing (MEC) provides such nodes in close network proximity, their capacity is limited and may be shared with the Centralized Radio Access Network (C-RAN) in 5G networks. We propose MicroLambda, a framework to partition offloaded computation via dynamic checkpointing to efficiently utilize MEC compute capacity without encroaching on C-RAN operations.**

***Index Terms*—MEC, C-RAN, checkpointing, CRIU**

I. Introduction

Mobile applications can reduce battery usage and speed up response times by offloading computation tasks to the cloud. This design pattern, however, has its drawbacks in that it increases egress traffic for mobile network operators and cloud computing costs for mobile app developers. One solution to mitigate these drawbacks is to provide a first line of defense on mobile edge computing (MEC) nodes that perform some processing on the edge to transform and filter mobile app requests before forwarding them to the cloud. An operational example of this split edge/cloud strategy is Akamai EdgeWorkers, which handle HTTP requests on edge servers before sending them deeper into the Content Delivery Network (CDN). Several research projects have also proposed solutions for splitting the processing of offloaded tasks between MEC and cloud resources [21], [22].

In this paper, we explore the idea of employing hitherto unused MEC resources for offloading mobile application processes. Future 5G networks will deploy Centralized Radio Access Network (C-RAN) designs, where Base-band Units (BBUs) receive analog radio signals over fiber from several nearby Remote Radio Heads (RRHs) located on cellular towers and process these signals in software. BBUs will use general purpose processors, which could be dual-purposed to handle offloaded application requests as well.

The challenge is that C-RAN requests take priority and as network traffic increases BBUs need to shed application processing tasks quickly to handle the C-RAN load. This shedding could be done by checkpointing offloaded application processes, however existing checkpointing methods, discussed in Section II, are relatively slow, which means that a task may be evicted from a BBU before its checkpoint completes. Existing checkpointing methods also produce relatively large checkpoints, which makes it difficult to move them into storage, or onto a less loaded BBU to complete.

In this work we present a new checkpointing method we dub MicroLambda ($\mu\lambda$) that significantly reduces checkpoint time and size. $\mu\lambda$ uses the Python `pyrasite` library [13] to dynamically inject checkpointing code of the Python `python-checkpointing2` library [14] at the current line an offloaded Python process. After the injection, the offloaded process produces and saves a checkpoint before terminating to vacate the BBU. $\mu\lambda$ then restarts the process from the checkpoint at the same, or a different BBU.

$\mu\lambda$ checkpoints are small and fast because $\mu\lambda$ serializes only that internal process state, rather than the enclosing OS process state [4], [24], [29], container [18], [32], or virtual machine (VM) [25], [38], [39]. Although $\mu\lambda$'s code injection approach limits it to tasks implemented in interpreted languages, Python is a popular choice for the kinds of tasks the benefit from offloading, such as image processing and data classification, because of its strong library support and usage across many domains. We demonstrate that $\mu\lambda$ can checkpoint not only custom Python processes, but also those that rely on third-party libraries.

We implement $\mu\lambda$ in a 5G network simulated on MEDICINE [35] and compare the checkpoint size and speed of $\mu\lambda$ with respect to that of CRIU [4] – the lightest of the competing process checkpointing solutions. Our results show that $\mu\lambda$ produces checkpoints faster and consistently smaller than those by CRIU. With small and fast checkpoints $\mu\lambda$ makes it easier for BBUs to shed offloaded processes without causing them to lose progress. Our implementation also include $\mu\lambda$'s process management, which resumes checkpointed processes to support continual execution spanning continual checkpointing and execution cycles.

We evaluate $\mu\lambda$ for three representative applications that transform and filter mobile application requests on their way to the cloud. The face recognition application receives images sent from a camera and reports to the cloud server if a new face is detected. The ride hailing application assigns locally available vehicles and forwards to the cloud ride requests that cannot be satisfied locally. Finally, the stress classification application detects the stress level of human workers in a co-roboting environment [28], [31] and reports to the cloud instances of high stress for possible interventions.

We compare the execution speed and network load of process execution with $\mu\lambda$ and CRIU checkpoints and continual execution without checkpointing. These results show that $\mu\lambda$ checkpointing has low overhead compared to execution without checkpointing and makes efficient use of compute and network resources in a 5G deployment.

The rest of this paper is organized as follows: Section II provides background in on edge computing and computation checkpointing. In Section III, we outline $\mu\lambda$ and in Section IV we present its performance evaluation. We discuss future work in Section V and conclude in Section VI.

II. Related Work

$\mu\lambda$ seeks to increase the flexibility of requests executing on MEC nodes by dynamically splitting computation across multiple function invocations, where function runtime is limited. Our work is similar in some respects to Amazon Step Functions, where developers may compose functions, containers, and machine learning processes into workflow graphs [3]. OpenWhisk also provides similar functionality via Composer [7]. Step Functions and Composer, like $\mu\lambda$, support continuation of computation across function invocations albeit in a limited way by implementing an iterative loop pattern, which reinvokes the same function as long as a condition holds true [6]. This paradigm is not convenient for computation offloading, because it requires developers to manually decompose applications into functions that fit within available runtimes. $\mu\lambda$ on the other hand splits the execution of the functions dynamically during a runtime. Akamai EdgeWorkers allow the deployment of functions on content delivery network (CDN) servers, but limit the runtime of each function to 10 ms [2], [5]. This runtime, in turn, is sufficient to help process web requests, but is too short for more general offloading requests.

$\mu\lambda$ is also related to other methods for computation migration between MEC nodes. In general, migration is accomplished by checkpointing the environment execution the process, which is either a VM, a container, or the OS process.

Researchers proposed the idea of VM migration to load balance computation on available infrastructure. Eunbyung et al. introduced fast and space-efficient checkpointing for VM migration by excluding from checkpoints those memory pages whose data are available on non-volatile storage [34]. Aderholdt et al. developed a VM migration that performs virtual machine introspection (VMI) to find paging data, map the data to find guest users' page memory, and exclude the paging before checkpointing [17]. Saad et al. proposed Differential Checkpointing (DICE) that creates a series of checkpoints based on difference in VM memory since the previous checkpoint [19]. While VM migration is efficient at checkpointing and migrating groups of processes, the overhead of checkpointing the VM OS and memory pages is significant and results in long checkpointing time and large checkpoint size.

Container migration attempts to alleviate these problems by checkpointing a lighter weight execution environment. Ma et al. proposed a container migration technique based on an Ant Colony System (ACS) that considers resource correlation, delay demand, and business relevance as factors [32]. Arif et al. presented a docker deployment technique to checkpoint and restart a swarm of containers using Distributed Multithreaded Checkpointing (DMTCP) and Ceph distributed storage [18]. While lighter weight than VM migration, container migration still introduces significant overheads when the process executing in the container is small.

Finally, process migration aims to save only the memory resources used by a running program. CRIU [4] is the primary technique to take a snapshot of a program and we discuss its details in Section IV-B. Horii et al. proposed a solution to migrate processes using CRIU through inter-process communication channels to communicate between migrated processes [24]. Pekka et al. developed CRIU based checkpointing and suspending for Docker containers with long-running blocking functions in FaaS [29]. $\mu\lambda$ aims to improve on CRIU by injecting checkpointing code into the program to save only program state as opposed to memory pages containing this state.

III. MicroLambda

To allow task offloading onto BBUs we propose $\mu\lambda$, the extension of [36]. Below, we present the $\mu\lambda$ computation model in the abstract and then map it onto a practical implementation.

A. Computation Model

A mobile device may offload a user process by issuing a request for an execution of a function f with input i on an MEC node. In Figure 1 f executes on a compute node with attached memory that stores intermediate program states $s_1, s_2, ..., s_n$ such as variables or other data structures. After r_{actual} seconds of runtime, f completes and returns output o that may be forwarded to the cloud for further processing.

As discussed, this abstraction does not map well to MEC computation because r_{actual} might be greater than the runtime limit r_{limit} allowed by the BBU before it evicts a low priority process. Since r_{limit} is variable in practice as it depends on the BBU's network load fluctuation it may not be possible for an offload program to complete before it is evicted.

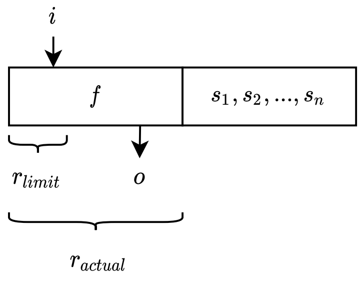


Fig. 1: Computation process on unrestricted runtime.

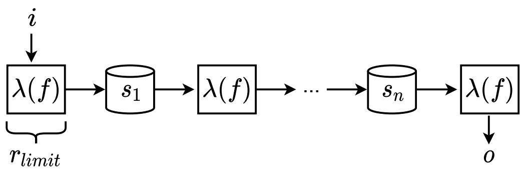


Fig. 2: $\mu\lambda$ computation process.

To complete an execution of f on a limited runtime, $\mu\lambda$ invokes a serverless function λ with f as a parameter, as shown in Figure 2. Before r_{limit} is reached, λ suspends f and saves its state s_1, which includes a program checkpoint, to an external storage. $\mu\lambda$ then continues the execution by invoking $\lambda(f)$ again, which loads s_1 and starts f from its last checkpoint. The process continues until f returns output o. It is important to note that each invocation of λ and storage of the intermediate state may take place either at the same BBU node or at a different ones, which is determined by a computation placement process in a 5G network slice.

The proposed computation model is quite general. $\mu\lambda$ supports user requests processed by a serverless function that would otherwise not complete within r_{limit}. $\mu\lambda$ also supports long-running, stateful processes that traditionally would be hosted on containers or VMs. Different users may also share the intermediate state, subject to consistency guarantees of replicated MEC storage nodes.

B. Implementation

We assume that a BBU node supports general purpose processing, on which we may deploy Docker containers. We implement $\mu\lambda$ as an integration of the OpenFaaS [8] serverless framework, the Redis [9] real-time database with a publish/subscribe interface. In this section, we describe the interaction between the components of $\mu\lambda$ and their composite functionality.

Figure 3 shows the messages between $\mu\lambda$ components that accomplish computation offloading. In step 1, the Client on user equipment (UE) submits an offloading request to a Redis database container deployed on an edge device (ED), such as a BBU node. The request includes the name of the serverless function f and user input i, such as user ID, location, UE sensor

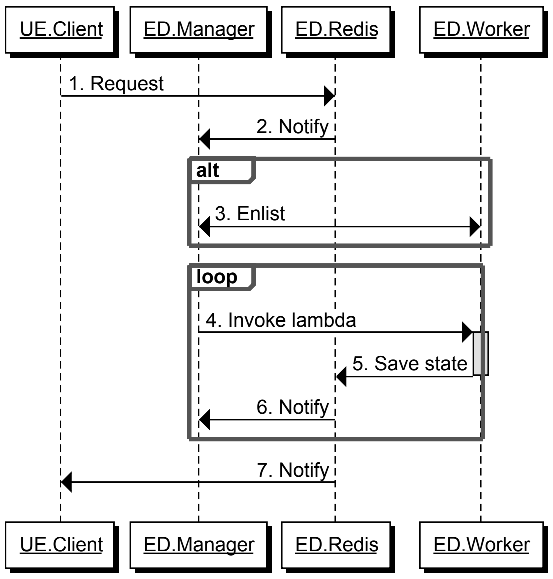


Fig. 3: $\mu\lambda$ messaging and coordination process.

reading, or other parameters. The starting application state s_0 is null. In step 2, Redis notifies the OpenFaaS Manager container of a request to invoke a function by sending it f and i. In step 3, the Manager enlists one or more Worker containers by directing them to install $\lambda(f)$, if they have not installed it already. Once a Worker has been enlisted, the Manager invokes $\lambda(f)$ on the Worker by sending it i and s_0 through an `HTTP POST` request in step 4. The Worker runs $\lambda(f)$ for some time less than r_{limit} and stores the next application state s_1 at the Redis container in step 5. In step 6, Redis notifies the Manager, which continues the execution by resuming the $\lambda(f)$ from the checkpoint saved in step 4, now with s_1 as the starting state. If step 5 was the final invocation of $\lambda(f)$, then, in step 7, Redis notifies the Client of the finished computation with output o, or forwards the request to the cloud for further processing. It is important to note that only steps 1 and 7 suffer the 5G latency; other communication may be done between EDs deployed at the tower over a wired network, or internally, if the Manager, Redis, and Worker components reside on the same ED.

In our implementation, we made several design decisions. First, to make sure that the OpenFaaS Manager does not experience an ACK implosion from multiple Workers, we configure the Workers to save computation state to a replicated Redis server. That way a Manager may pick up the new state when it is ready to invoke a function, or multiple managers may collaborate on invoking functions by making exclusive requests to Redis. Second, in OpenFaaS, the Manager is also a Worker. To force the Manager to farm out the function requests to other Workers in our evaluation, we place a dummy long-running function at the Manager so it counts itself as busy.

C. Dynamic Program Checkpointing

The central innovation behind $\mu\lambda$ as a system is its ability to pause computation within a specific runtime duration and to resume it. Python is interpreted and allows for injection code into a running program. We combine the functionality of `python-checkpointing` [14], a library for checkpointing and continuing the execution Python programs, and of `pyrasite` [13] a Python code injection library.

The `python-checkpointing` library provides a `save_checkpoint()` function, which developers may include in their programs to save partial results of the computation as program state and to restart the program from the checkpoint on the next invocation. The `save_checkpoint()` function generates a collection of files that represent the state of program variables, the Python interpreter stack, and the bytecode instruction pointer at which the `save_checkpoint()` was called.

The problem we needed to solve is how to dynamically place the `save_checkpoint()` function into a running program just before r_{limit} expires. To so we leverage the `pyrasite` library. A $\mu\lambda$ serverless function includes two threads: the offloaded application function f and a companion process `timeout_tracker`. The `timeout_tracker` keeps track of execution time and some time before r_{limit} expires uses `pyrasite` to inject the `save_checkpoint()` function into f. The function f then creates a checkpoint, which the `timeout_tracker` uploads to Redis before r_{limit} actually expires. The following invocations of $\mu\lambda$ invoke a new serverless function, where f restarts and continues from the last saved checkpoint.

We want to make it clear that the above approach can inject checkpointing code into the Python execution, but not inside library calls that may be internally implemented in languages other than Python. For example, the Face Recognition library [23] we use to implement the face recognition application used the `dlib` machine learning toolkit [1] internally. The `dlib` library is implemented in C++ and the Python Face Recognition library provides wrapper functions. So, while we can inject checkpoints into the Face Recognition library code, the checkpoint resolution is limited to the boundary calls to `dlib`. Nevertheless, the limited granularity of checkpoints is not a practical limitation to checkpointing time as discussed in Section IV-C1.

IV. Evaluation

The goal for our evaluation is to understand the practicality of $\mu\lambda$ for execution offloaded processes. We based our evaluation on three prototypical workloads: face recognition, ride hailing, and stress classification. Our results show a performance comparison of these processes with $\mu\lambda$, and CRIU checkpoints as well as an uncheckpointed execution.

A. Applications

We want to investigate the suitability of $\mu\lambda$ to support different types of edge computing workloads. To do so, we have identified and implemented three workloads representing different classes of computation suitable for offloading to edge computing nodes.

1) Face Recognition: The face recognition application compares two consecutive images and returns true if a person's face in the second image has been present in the first image. These images might come from a wearable camera (Client) transmitted to an edge compute device (Worker) to detect if the captured scene contains a new face, which can be sent to the cloud for further analysis.

We implement the application using the Face Recognition library [23] based on the `dlib` machine learning toolkit [1]. The Client drives the application traffic by sending a series of images to the Worker, which processes them by recognizing faces in them and returns true if images in consecutive images represent the same person. The application is stateful in that the Worker saves the encoding of the features of the last recognized face for comparison against the next image.

It is important to note that the face recognition application, as well as the others described below, are realistic implementation that rely on third party libraries. This point is important, because it demonstrates the fact that $\mu\lambda$ can inject checkpoints not only into simple academic code, but in fact anywhere within a third party Python library to provide agility of the checkpointing process. To illustrate the size of the face recognition application, used `pprofiler` [15] we analyzed the number of unique lines of code invoked by the application during that its runtime. The face recognition app invokes 24,083 unique lines of Python code in the main process and the imported library code.

2) Ride Hailing: The ride hailing application allows the Client to request the nearest available taxi in a geographical area. If a ride is not available in the area, the request can be sent to the cloud for a search in a wider area. Thus the ride hailing application is representative of a broad class of such workloads.

To implement that application we rely on the New York taxi trip record dataset [11]. The data includes pick-up and drop-off locations and times. When the Client issues a new ride request the application finds the taxi with the nearest drop-off point to the Client's location according to the shortest driving distance calculated with the Python `OSMnx` library [12] based on a real-world street network. To realistically constrain the search space, the application considers taxis whose drop-off times are within 5 min of the request. We also use the `OSMnx` library to calculate driving time based on the Client's destination and update the taxi drop-off point and time in the application's shared state. The shared state also includes a record of which taxis are assigned to users, so that two users do not get assigned the same taxi. The application traffic is driven by Clients submitting ride requests.

The application queues these requests and submits them to a $\mu\lambda$ process, which processes a number of them within the edge computing runtime limit, before saving the current state of driver availability. The ride hailing application invokes 7,084 lines of Python code.

3) Stress Classification: The stress application predicts if a mixed reality (MR) user is stressed given their physiological responses. Stress classification is useful in smart manufacturing [37].

To implement the application, we adapt the system proposed by Kalatzis et al. [27]. In particular, for each participant, their system uses electrocardiogram (ECG) and respiration sensors to produce a time varying signal at a rate of 1,000 samples per second. As the Client continually uploads physiological data to Redis the offloaded application performs the following steps every 30 seconds. Step 1, *feature extraction*, gathers the most recent two minutes of the signal (120,000 samples) and uses Neurokit2 [33] to produce a feature vector consisting of the participants heart rate (HR), respiration rate (RR), heart rate variability (HRV), and respiration rate variability (RRV). Step 2, *classification*, inputs the feature vector into a trained support vector machine that classifies the user as stressed/calm and returns the result to the Client, or forwards it to the cloud. Finally, Step 3, the application saves in memory the features extracted for the last 90 seconds to combine with those it will extract from the next 30 seconds of physiological data in the following iteration. This implementation also allows us to parallelize the feature extraction in Step 1 among multiple Workers.

In summary, the three workloads represent classes of computation that mobile applications might choose offload to edge computing devices. The three application maintained shared state between requests and so typically would be executed on long-running processes. Because $\mu\lambda$ implicitly saves process state at the end of the edge computing runtime, it is able to continue the applications from their current state within the execution of a single client request, but also across requests seamlessly.

B. CRIU

We want to compare $\mu\lambda$'s performance to another relatively light-weight checkpointing solution, the Checkpoint/Restore In Userspace (CRIU) is a Linux tool [4]. CRIU extracts the complete kernel information from the OS, from which the process may be restarted and its execution continued [16]. Docker uses CRIU migrate containers (as processes) [20] and using CRIU to checkpoint processes allows us a reasonable comparison of $\mu\lambda$ to container migration approaches [18].

To use CRIU checkpointing we invoke the offloaded function f and `timeout_tracker` with the process ID of f. After r_{limit} the `timeout_tracker` runs CRIU to suspend f and upload the saved checkpoint to Redis. As under code injection checkpointing, the following invocations restart and continue f from the last saved checkpoint.

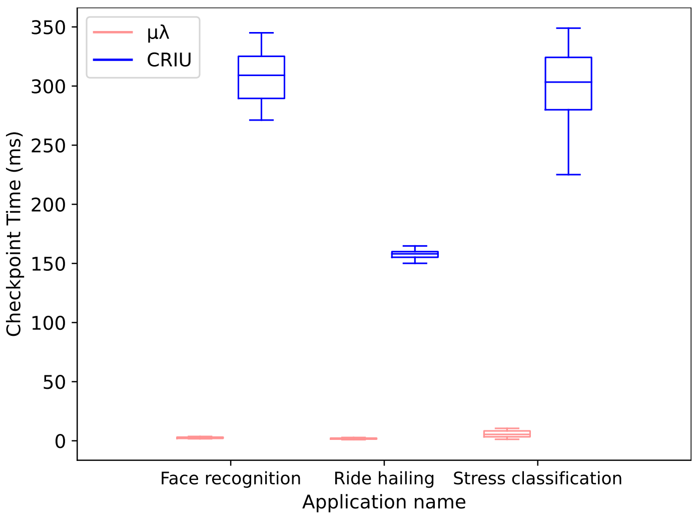


Fig. 4: Checkpoint time.

It is worth noting that while CRIU may suspend arbitrary processes, it does have its limitations. First, CRIU requires kernel level permissions [10] and while it can checkpoint a container, it cannot be run inside a container to checkpoint processes executing therein. Thus, while the code injection checkpointing of $\mu\lambda$ may run in an OpenFaaS container, the CRIU-based system must run natively. Second, CRIU checkpoints are much larger that $\mu\lambda$ checkpoints as discussed in the following section.

C. Results

To understand the performance of $\mu\lambda$ we compare it to CRIU along several dimensions. Specifically, we measure the efficiency of the checkpointing approaches in terms of the time to create a checkpoint and the size of the checkpoint for different applications. We also compare the execution speed and network load for applications checkpointed using $\mu\lambda$ and CRIU with respect to executions without checkpointing.

1) Checkpoint time: Figure 4 shows the time in milliseconds to create a checkpoint on the y-axis and the evaluated applications on the x-axis. We measure checkpoint time by subtracting the time at which we start the checkpointing process to when a checkpoint is produced on the file system. The light red and dark blue series show checkpoint time for $\mu\lambda$ and CRIU respectively. Each box plot illustrates the distribution of 50 checkpoints made after random application execution duration to produce checkpoints at different points in application execution.

In general, we observe that for all applications, $\mu\lambda$ takes less time to create a checkpoint. While $\mu\lambda$ creates a checkpoint for all applications in under 10 ms, CRIU takes at least 150 ms in the case of the ride hailing application and much longer for the other applications. Since CRIU checkpoints the OS process

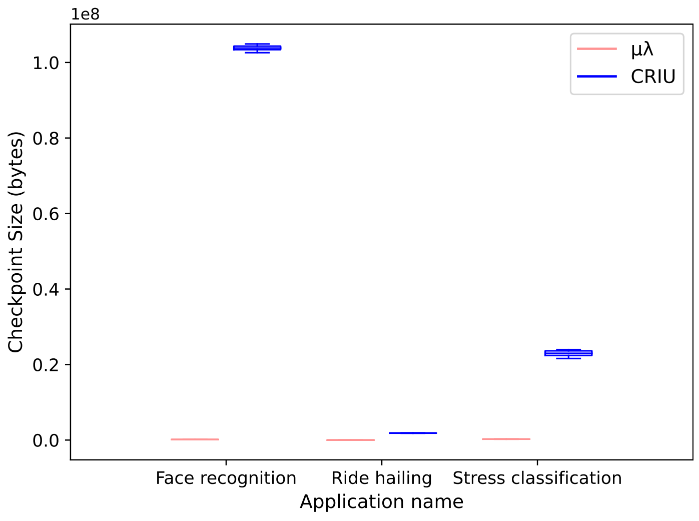
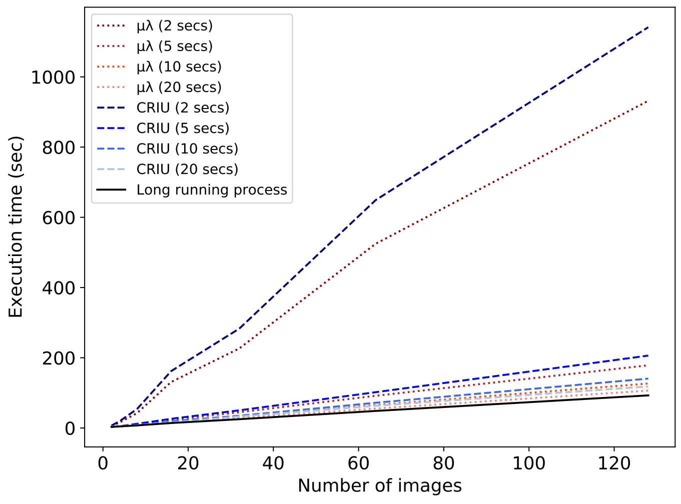


Fig. 5: Checkpoint size.

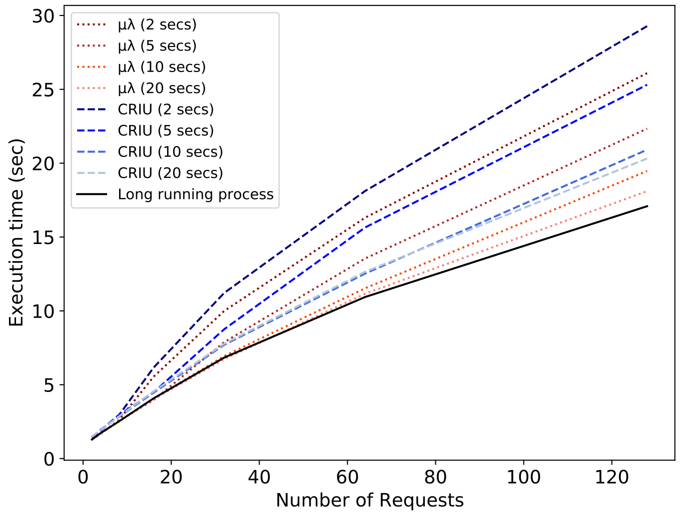
state it includes memory pages for the process and serializing them takes more time than for $\mu\lambda$ to serialize program state. We also observe that there is far less variability in checkpoint size for $\mu\lambda$ both within and across applications.

2) Checkpoint size: Figure 5 shows the size in bytes to create a checkpoint on the y-axis and the evaluated applications on the x-axis. We measure the checkpoint size as the size of the compressed checkpoint byte data. In general we observe that $\mu\lambda$ creates much smaller checkpoints than CRIU. This difference is pronounced for the face recognition and stress classification applications, where $\mu\lambda$ checkpoints are under 128 KB, while CRIU checkpoints above 22 MB. The difference between checkpoints sizes is less pronounced for the ride hailing application, where $\mu\lambda$ checkpoints are under 281 KB, while CRIU checkpoints above 1.8 MB. These differences are due to the fact that different applications use different amounts of memory and lay out that memory differently, which makes it easier for CRIU in the case of the ride hailing application to serialized the relevant memory pages. When compared with checkpoint time graph in Figure 4 we observe that CRIU has a significant checkpointing time overhead for the ride hailing application even though the checkpoint size itself is small. In combination, Figure 4 and Figure 5 show that checkpointing with $\mu\lambda$ can shed load from BBU nodes much more quickly than checkpointing with CRIU. Since the time to vacate a BBU depends on both the time to create a checkpoint and the time to offload it to the file system, or another storage node, the shorter checkpoint time and smaller checkpoint size of $\mu\lambda$ give it a clear advantage.

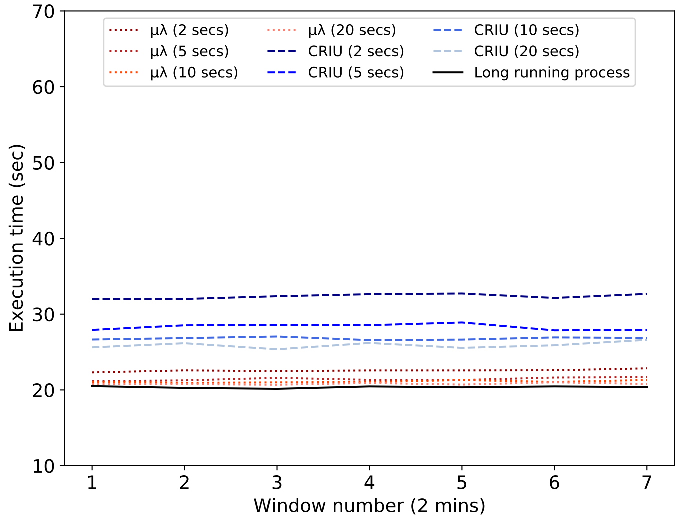
3) Execution Time: Figure 6 shows on the y-axis the mean offloaded request execution time in seconds for $\mu\lambda$, CRIU, and a long-running process without checkpointing. We measure execution time by calculating the difference between the time



(a) Face Recognition



(b) Find A Taxi



(c) Stress Classification

Fig. 6: Execution Time

the Client submits an offloading request to Redis and the time Redis notifies the Client of the finished computation with its output.

The x-axis in Figure 6a shows the number of images that comprise a bulk request of the face recognition application. The long-running execution, shown as the black solid line, processes the set of images in its entirety. On the other hand, $\mu\lambda$, red dotted lines, and CRIU, blue dashed lines, process some number of images within the r_{limit} of 2, 5, 10, and 20 sec before checkpointing. The x-axis in Figure 6b marks the number of batched ride requests in the find a taxi application. Finally, the x-axis in Figure 6c marks the overlapping window number for the stress classification workload. $\mu\lambda$ performs the feature extractions with multiple workers in parallel where r_{limit} is from 2 to 20 s.

We observe that the execution time of $\mu\lambda$ and CRIU nearly tracks the execution of the long-running process at longer runtime limits in the face recognition application. As we move to the ride hailing and the stress classification applications $\mu\lambda$ outperforms CRIU, but remains close to the long-running process execution time. This result shows that the overhead of multiple $\mu\lambda$ invocations is negligible and that workloads executed in $\mu\lambda$ and as a long-running process achieve comparable performance. As the runtime limits get shorter the more frequent saving of intermediate state in $\mu\lambda$ and CRIU increase execution time. We observe, however, that the effect is less pronounced for $\mu\lambda$, which produces smaller checkpoints.

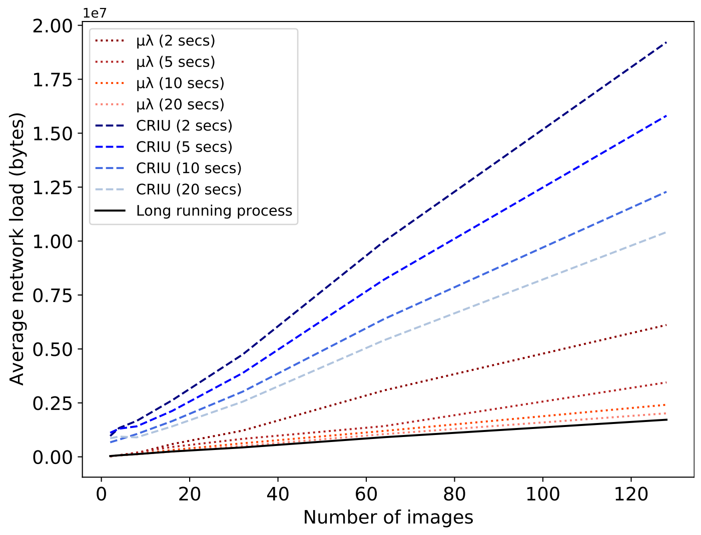
4) Network Load: Figure 7 shows on the y-axis the average network load in bytes for the different applications under $\mu\lambda$, CRIU, and long-running execution. We measure the network load using `tshark` by calculating the summation of all the packets' frame lengths. `tshark` tracks the network packets that are carrying protocol messages in Figure 3.

In general, the network load for $\mu\lambda$ is comparable with the long-running execution at longer runtime limits. The CRIU network load is higher than that of $\mu\lambda$ because of the larger checkpoints; the effect is especially pronounced at shorter runtime limits with more frequent checkpoint creation.

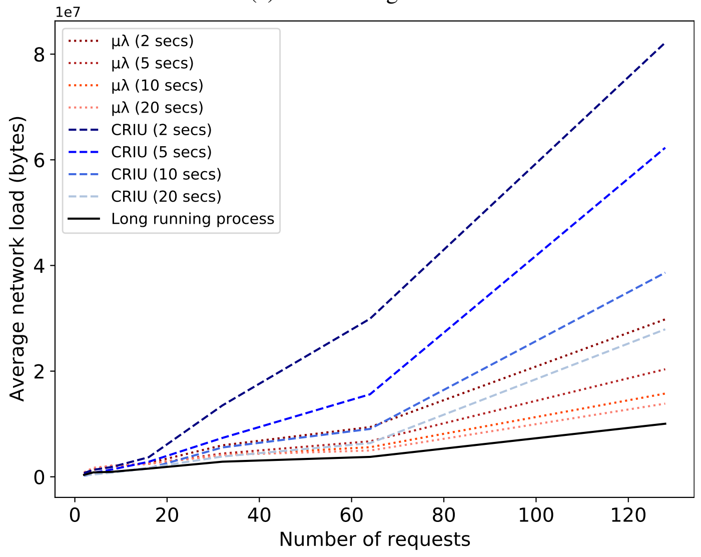
These results indicate that the overhead of saving and loading the intermediate application state under $\mu\lambda$ is not significant, because intermediate state in the application is small compared to the workload data that must be delivered to function invocations as a part of the application offloading process for both $\mu\lambda$ and the long-running process.

V. Discussion

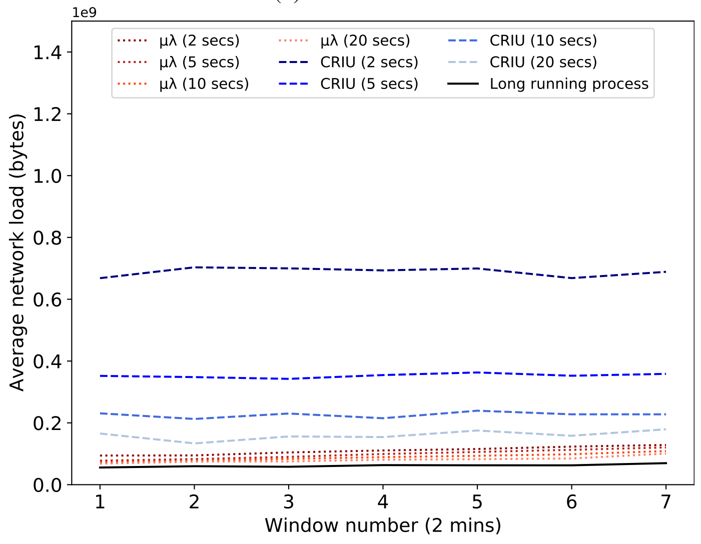
We note that for some applications, $\mu\lambda$ presents a security concern. In particular, because we use code injection, the function that is supplied by the user has a different hash than what is executed in the run time. Moreover, as the injection of the checkpoints are non-deterministic, analyzing that the function was executed faithfully could be problematic. CRIU avoids this security concern at the cost of a heavy-weight



(a) Face Recognition



(b) Find A Taxi



(c) Stress Prediction

Fig. 7: Network Load

serialization. One avenue to address the concern is to develop a Python extension that can invoke, pause, resume, and serialize a Python function. We believe that such an extension would have the security benefits of CRIU while maintaining a light-weight serialization.

In its current version, we have considered workloads on a homogeneous CPU based architecture. MEC, in general, supports heterogeneous architectures. By integrating GPUs and TPUs, $\mu\lambda$ would enable the development of more sophisticated graphics and machine learning workloads. One of the major challenges with such integrations, however, is the state serialization from a GPU or TPU. The check pointing mechanism required to enable the serialization is an active area of research [26], [30].

VI. Conclusions

In this paper, we presented MicroLambda ($\mu\lambda$) – a framework to partition offloaded application execution across runtimes. $\mu\lambda$ allows dynamic splitting of stateful and long-running computation across multiple serverless function invocations. The presented results show that the execution efficiency of $\mu\lambda$ is comparable to that of computation offloaded to long-running functions, which may however not fit within runtime caps of edge computing nodes. We hope that these results will lead to new solutions similar to Akamai's EdgeWorkers [5] being deployed on 5G BBUs and that increasing the availability of edge computing resources will lead to more flexibility in mobile application architectures.

References

[1] Dlib C++ library. http://dlib.net/, December 2019.

[2] Introducting serverless computing at the edge with Akamai EdgeWorkers. http://bit.ly/2P6JCxW, October 2019.

[3] AWS Step Functions. https://go.aws/2uZixpJ, February 2020.

[4] Checkpoint/Restore In Userspace (CRIU). https://criu.org/Main_Page, February 2020.

[5] EdgeWorkers User Guide: Limitations. http://bit.ly/2V46Bxs, February 2020.

[6] Iterating a Loop Using Lambda. https://amzn.to/328LKe9, February 2020.

[7] Open Whisk Github. https://github.com/apache/openwhisk, February 2020.

[8] OpenFaas. https://openfaas.com/, February 2020.

[9] Redis Database. https://redis.io/, 2020.

[10] CRIU: Check the kernel. https://criu.org/Check_the_kernel, 2021.

[11] New York TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2021.

[12] OSMnx: Python for street networks. https://github.com/gboeing/osmnx, 2021.

[13] Pyrasite: Dynamic Python Code Injection. https://github.com/lmacken/pyrasite, February 2021.

[14] Python Checkpointing. https://github.com/a-rahimi/python-checkpointing2, February 2021.

[15] Python pprofile. https://github.com/vpelletier/pprofile, 2021.

[16] Julien Adam, Maxime Kermarquer, Jean-Baptiste Besnard, Leonardo Bautista-Gomez, Marc Pérache, Patrick Carribault, Julien Jaeger, Allen D. Malony, and Sameer Shende. Checkpoint/restart approaches for a thread-based mpi runtime. *Parallel Computing*, 85, 2019.

[17] Ferrol Aderholdt, Fang Han, Stephen L. Scott, and Thomas Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *CCGRID*, 2014.

[18] Arif Ahmed, Apoorve Mohan, Gene Cooperman, and Guillaume Pierre. Docker container deployment in distributed fog infrastructures with checkpoint/restart. In *MobileCloud*, 2020.

[19] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In *Languages, Compilers, and Tools for Embedded Systems*, 2019.

[20] Yang Chen. Checkpoint and restore of micro-service in docker containers. In *Mechatronics and Industrial Informatics*, 2015.

[21] A. Das, A. Leaf, C. A. Varela, and S. Patterson. Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. In *Cloud Computing (CLOUD)*, October 2020.

[22] Tarek Elgamal, Atul Sandur, et al. Costless: Optimizing cost of serverless computing through function fusion and placement. 2018.

[23] Adam Geitgey. Face recognition. https://github.com/ageitgey/face_recognition, December 2019.

[24] M. Horii, Y. Kojima, and K. Fukuda. Stateful process migration for edge computing applications. In *Wireless Communications and Networking Conference (WCNC)*, April 2018.

[25] Yaohui Hu, Tianlin Li, Ping Yang, and Kartik Gopalan. An application-level approach for privacy-preserving virtual machine checkpointing. In *Cloud Computing*, 2013.

[26] Twinkle Jain and Gene Cooperman. Crac: Checkpoint-restart architecture for cuda with streams and uvm. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[27] Apostolos Kalatzis, Laura Stanley, Rohith Karthikeyan, and Ranjana K Mehta. Mental stress classification during a motor task in older adults using an artificial neural network. In *UbiComp and ACM ISWC*, 2020.

[28] Kari Kallinen. The effects of transparency and task type on trust, stress, quality of work, and co-worker preference during human-autonomous system collaborative work. In *Human-Robot Interaction*, 2017.

[29] Pekka Karhula, Jan Janak, and Henning Schulzrinne. Checkpointing and migration of iot edge functions. In *Workshop on Edge Systems, Analytics and Networking*, 2019.

[30] Kyushick Lee, Michael B. Sullivan, Siva Kumar Sastry Hari, Timothy Tsai, Stephen W. Keckler, and Mattan Erez. Gpu snapshot: Checkpoint offloading for gpu-dense systems. In *ACM Supercomputing*, 2019.

[31] Mike Ligthart, Koen Hindriks, and Mark A. Neerincx. Reducing stress by bonding with a social robot: Towards autonomous long-term child-robot interaction. In *Human-Robot Interaction*, 2018.

[32] Z. Ma, S. Shao, S. Guo, Z. Wang, F. Qi, and A. Xiong. Container migration mechanism for load balancing in edge network under power internet of things. *IEEE Access*, 8, 2020.

[33] Dominique Makowski, Tam Pham, Zen J. Lau, Jan C. Brammer, François Lespinasse, Hung Pham, Christopher Schölzel, and Annabel S H Chen. Neurokit2: A python toolbox for neurophysiological signal processing, 2020.

[34] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space-efficient virtual machine checkpointing. In *Virtual Execution Environments*, 2011.

[35] M. Peuster, H. Karl, and S. van Rossem. Medicine: Rapid prototyping of production-ready network services in multi-pop environments. In *Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2016.

[36] Saidur Rahman, Mike P Wittie, Ahmed Elmokashfi, Laura Stanley, and Stacy Patterson. MicroLambda -Packetized Computation for 5G Mobile Edge Computing. In *Workshop on Hot Topics in Edge Computing (HotEdge)*, 2020.

[37] Ashita Soni and Shriyash Shete. Mixed reality for stress relief. In *Tangible, Embedded, and Embodied Interaction*, 2020.

[38] B V Srinivas, B Shruthi, Kavitha S Patil, and Indrajit Mandal. A framework for efficient virtual machine migration. In *Current Trends towards Converging Technologies (ICCTCT)*, 2018.

[39] Long Wang, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Arun Iyengar. Vm-μcheckpoint: Design, modeling, and assessment of lightweight in-memory vm checkpointing. *IEEE Transactions on Dependable and Secure Computing*, 12(2), 2015.