POLITECNICO DI TORINO
Repository ISTITUZIONALE

Processing-aware Migration Model for Stateful Edge Microservices

Original
Processing-aware Migration Model for Stateful Edge Microservices / Calagna, A., Yu, Y., Giaccone, P., Chiasserini, C.F.. - STAMPA. - (2023). (IEEE ICC 2023 Rome (Italy) 28 May 2023 - 01 June 2023) [10.1109/ICC45041.2023.10278877].

Availability:
This version is available at: 11583/2974824 since: 2023-06-02T08:59:25Z

Publisher:
IEEE

Published
DOI:10.1109/ICC45041.2023.10278877

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

26 November 2026

Processing-aware Migration Model for Stateful Edge Microservices

Antonio Calagna
Politecnico di Torino
Torino, Italy

Yenchia Yu
Politecnico di Torino
Torino, Italy

Paolo Giaccone
Politecnico di Torino
Torino, Italy

Carla Fabiana Chiasserini
Politecnico di Torino
Torino, Italy

***Abstract*—To support latency sensitive microservices at the edge, stateful container migration has gathered momentum as a key solution to ensure a satisfying experience to mobile users. In this paper, we first investigate experimentally the stateful migration process, by using state-of-the-art tools, namely, Podman and CRIU. We then characterize the main migration KPIs, i.e., migration duration and downtime, and develop an analytical model that can effectively assess whether stateful migration is feasible while meeting the user's QoE requirements. Importantly, our model is validated using real-world microservices and, by accounting for all relevant real-world aspects of stateful migration, significantly outperforms state-of-the-art models.**

***Index Terms*—Migration, Microservices, Modeling**

I. Introduction

Network Function Virtualization (NFV) has been acknowledged as the pivotal technology to meet the challenges of placement, management, chaining, and orchestration of network services. According to NFV, network services are represented by service function chains, composed of a set of Virtual Network Functions (VNFs). Along with NFV, the concept of microservices has emerged with the aim to make VNFs cloud-oriented by design, thus being implemented into lightweight, scalable, general purpose containers [1]. In this context, live migration has gathered momentum as a means to enable container migration and, hence, ensure continuous proximity of latency-sensitive or bandwidth-consuming microservices with mobile end users. Additionally, live migration can be used as dynamic resource management tool for, e.g., resource rescheduling, load balancing, and fault tolerance.

In this paper, we focus on stateful migration, which is used whenever keeping track of the service state is essential to guaranteeing service continuity. In other words, in stateful migration, beside the service template image, the following pieces of information are made available at the destination host: (i) the CPU-context state, e.g., registers, processes' tree structure, and namespaces, (ii) the memory content, i.e., pages allocated in the main memory, (iii) the network sockets, and (iv) the open file descriptors.

Unlike stateless migration, which has already been investigated thoroughly and implemented in relevant orchestration systems like Kubernetes, stateful migration is less straightforward and still exhibits several open issues. In particular, none of the existing works has modeled the container-based live migration process in a sufficiently accurate manner. We thus fill this gap by proposing a Processing-Aware Migration (PAM) model that captures all the relevant real-world aspects of stateful migration. Unlike state-of-the-art models [2], PAM accounts for the processing time overhead introduced by the migration tool and its impact on both the migration and the downtime duration. Our work demonstrates that such component, neglected in prior art, is often a dominant contribution to the latency of the migration process. Furthermore, we design the PAM model starting from experimental observations made through our testbed, and we validate it using such popular microservices as MQTT Broker and Memcached.

Our main contributions are therefore as follows: (i) we assess experimentally the performance of container stateful migration controlled through off-the-shelf tools; (ii) leveraging our experiments, we devise a new analytical model to accurately estimate the migration key performance indicators (KPIs), accounting for the processing time overhead neglected in prior art; (iii) we validate the model in a realistic scenario and show that it greatly outperforms existing models.

In the following, Sec. II introduces stateful migration, Sec. III describes the testbed we developed to get experimental results reported and used in Sec. IV to derive the PAM model. The model is validated in Sec. V, while Sec. VI discusses some relevant related work and Sec. VII draws our conclusions.

II. Overview of Container Migration

This section provides an overview of microservice stateful migration, along with its KPIs. Further, it introduces CRIU, as the primary enabling tool to effectively implement microservice stateful migration.

Microservice migration. We consider microservices running on containers, whose internal state must be migrated. Since stateful migration involves transferring microservice's memory content, multiple strategies have been defined to minimize the time to perform such transfer. All of them are based on the concept of *dirtiness*, which refers to the amount of memory pages that the microservice has modified. Each microservice is characterized by the value of dirty page rate, R, i.e., the number of memory pages it modifies per time unit.

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

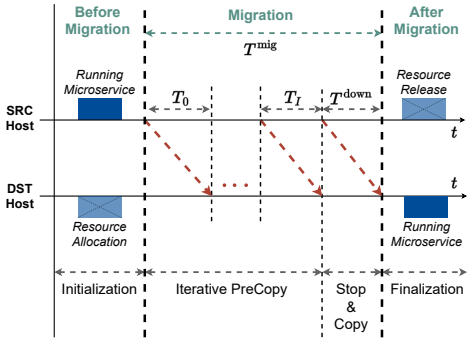


Fig. 1: Live migration diagram under the Iterative PreCopy strategy

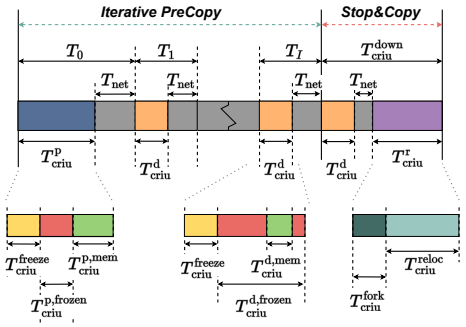


Fig. 2: Live migration diagram: CRIU implementation

Live migration can then be categorized depending upon which strategy is adopted to cope with the microservice dirtiness, namely, PreCopy, PostCopy, or HybridCopy. Since PostCopy and HybridCopy do not yet support container migration and are still at an early experimental stage [3], we focus on PreCopy. In particular, we tackle an extension of the PreCopy strategy, named *Iterative PreCopy*, which, to minimize the microservice disruption time, transfers the dirty pages to the destination host iteratively while the microservice is still running at the source and till, e.g., the new user connection is established or a deadline set by the orchestrator is reached. As depicted in Fig. 1, this approach enables the system orchestrator to set-up the destination host in advance and keep it continuously up-to-date before the final microservice migration is scheduled and executed. Such final migration procedure is known as *Stop&Copy* stage, during which the microservice is stopped at the source node, and its state is transferred to the destination host where the service will be eventually resumed. After migration, the source host is notified about the successful restoration, and the resources reserved therein are released.

We remark that the duration of the Stop&Copy phase determines the service disruption experienced by the final user, which is commonly referred to as *downtime* (T^{down}). The total migration duration consists of the duration of both the Iterative PreCopy and the Stop&Copy stage, i.e.,

$$T^{\text{mig}} = \sum_{i=0}^{I} T_i + T^{\text{down}}, \tag{1}$$

where T_i is the generic iteration duration and $I+1$ denotes the number of iterations required for migration. Given that our study aims at characterizing the migration cost for the network operator as well as the user's QoE, we take both the overall migration duration and the downtime as migration KPIs.

Furthermore, we express the amount of data to be transmitted from source to destination host during iteration i as: $V_i = \rho \cdot \tau_1 \cdot M$ for $i = 0$, and $V_i = \rho \cdot \tau_2 \cdot N_i \cdot \sigma$ for $i > 0$, where M is the microservice state size, N_i is the number of dirty memory pages at iteration i, and σ is the size of each page, which depends on the considered architecture and kernel settings. During the first iteration ($i = 0$), the data volume consists of the whole memory content of the microservice, while for $i > 0$, only the dirty memory pages, i.e., those that have been modified with respect to the previous iteration, are considered. Coefficients τ_1 and τ_2 account for the amount of transferred data, including the encapsulation overhead introduced by a migration tool (which, for any $i > 0$, depends upon the dirty page rate). Parameter ρ indicates the ratio of the compressed data volume to the uncompressed one, while the additive volume contribution due to the CPU-context state is negligible and has been omitted.

Migration tool: CRIU. It is considered the key tool to effectively implement stateful migration. It defines: (i) a *checkpoint procedure*, which seizes a running process, collects its state and encapsulates it into an image, and (ii) a *restore procedure* that leverages a previously created checkpoint image to create a process and resume its state, on a host machine. To successfully retrieve the microservice state, CRIU requires to temporarily freeze the microservice at the source at every iteration during the Iterative PreCopy stage, thus producing a service disruption period, named *frozen time*, that adds to the aforementioned downtime. Our aim is to characterize both such sources of service disruption.

More specifically, CRIU provides two kinds of checkpoint procedure: predump and dump, corresponding to, respectively, the first and the generic iteration of the Iterative PreCopy. As depicted in Fig. 2, the predump duration $T^{\text{p}}_{\text{criu}}$ consists of three major contributions: (i) the freezing time $T^{\text{freeze}}_{\text{criu}}$, needed to seize a process, (ii) the frozen time $T^{\text{p,frozen}}_{\text{criu}}$, during which microservice state and memory content are identified, (iii) the memory time contribution $T^{\text{p,mem}}_{\text{criu}}$, related to extracting and encapsulating these memory pages. In the case of the dump duration ($T^{\text{d}}_{\text{criu}}$), instead, the memory time contribution is already part of the frozen time period $T^{\text{d,frozen}}_{\text{criu}}$. In summary,

$$T^{\text{p}}_{\text{criu}} = T^{\text{freeze}}_{\text{criu}} + T^{\text{p,frozen}}_{\text{criu}} + T^{\text{p,mem}}_{\text{criu}} \;;\; T^{\text{d}}_{\text{criu}} = T^{\text{freeze}}_{\text{criu}} + T^{\text{d,frozen}}_{\text{criu}}. \tag{2}$$

It is also worth introducing the time needed to transfer the dirty memory pages at each iteration, denoted with T^{net}. Then, considering that the iterations in (1) correspond to a predump stage for $i = 0$, and to a generic dump iteration for $i > 0$, we can write the iterations duration at CRIU layer, as:

$$T_{\text{criu},i} = \begin{cases} T^{\text{p}}_{\text{criu}} + T^{\text{net}} & \text{if } i = 0 \\ T^{\text{d}}_{\text{criu}} + T^{\text{net}} & \text{if } i > 0\,. \end{cases} \tag{3}$$

Given a generic iteration i, the number of dirty memory pages multiplied by their size σ is given by the product between the

dirty page rate and the time period during which the process was actively running during the previous iteration, i.e.,

$$N_i \cdot \sigma = R_{i-1} \cdot (T_{\text{criu},i-1} - T^{\text{x,frozen}}_{\text{criu},i-1}) \,, \tag{4}$$

with $T_{\text{criu},i-1}$ being the duration of the previous iteration and $T^{\text{x,frozen}}_{\text{criu},i-1}$ the corresponding frozen time.

Finally, CRIU performs restoration by forking a new process tree for each microservice. Thus, the restore time, $T^{\text{r}}_{\text{criu}}$, consists of relocating the microservice state in terms of CPU state and memory content, i.e.,

$$T^{\text{r}}_{\text{criu}} = T^{\text{fork}}_{\text{criu}} + T^{\text{reloc}}_{\text{criu}} \,. \tag{5}$$

Finally, the Stop&Copy stage at the CRIU layer consists of (i) one last dump execution, which also stops the microservice at the source host; (ii) the transfer of this final checkpoint image to the destination host, and (iii) the restoration of the microservice state at the destination host. Thus, the overall downtime during Stop&Copy is given by:

$$T^{\text{down}}_{\text{criu}} = T^{\text{d}}_{\text{criu}} + T^{\text{net}} + T^{\text{r}}_{\text{criu}} \,. \tag{6}$$

III. Our Testbed

In this section, we briefly describe the testbed we developed to analyze the migration process of containerized microservices. While the testbed uses CRIU as de facto standard for migration, it supports the creation, running, and management of containerized microservices through the runC and Podman tools, which operate on top of CRIU.

Creation, running, and management of containerized microservices. To extend the process layer perspective offered by CRIU, we leverage runC as container runtime, and Podman as container engine. *runC* is at the basis of most container engines and orchestration systems, including Podman. One of the main perks of runC is its integration with CRIU. Although directly experimenting with runC is possible, our aim is to analyze the migration duration and the downtime experienced at the microservice layer, thus assessing the impact on the user's QoE in terms of additional latency. For this reason, our experimental setup takes a higher layer perspective and focuses on the Podman container engine, in order to evaluate live migration performance in a realistic microservice deployment scenario. *Podman* is an open-source product, designed to develop, manage, and run containers and pods. Podman directly leverages runC APIs, thus leading to better performance than Docker. Also, Podman has been designed to organize containers in pods and allowing their definition to be exported to a Kubernetes-compatible file. These features, along with CRIU integration, strongly motivate the use of Podman as container engine. As for the migration latency, similarly to (3), we can write:

$$T_{\text{podman},i} = \begin{cases} T^{\text{p}}_{\text{podman}} + T^{\text{net}} & \text{if } i = 0 \\ T^{\text{d}}_{\text{podman}} + T^{\text{net}} & \text{if } i > 0 \,. \end{cases} \tag{7}$$

Likewise, the downtime, corresponding to the Stop&Copy stage duration in (6), can be expressed at Podman layer as:

$$T^{\text{down}}_{\text{podman}} = T^{\text{d}}_{\text{podman}} + T^{\text{net}} + T^{\text{r}}_{\text{podman}} \,. \tag{8}$$

As mentioned, our study also characterizes experimentally the processing time overhead introduced by runC and Podman, with respect to the underlying CRIU layer.

Experimental setting. To run extensive, yet controlled, experiments, we developed a synthetic containerized microservice that mimics an actual microservice but whose behavior in terms of memory allocation is finely controllable. Starting from a scratch Docker image, we developed a testing software, in C language, which was encapsulated along with its library dependencies. It leverages `malloc` API to allocate a circular buffer of size M bytes which is randomly initialized to maximize entropy and avoid compression. The software keeps modifying the content of the buffer with a predetermined rate evaluated within a fixed time period. Two different scenarios are considered, with minimum (best-case scenario) and maximum (worst-case scenario) dirty page rate (denoted with $R_{\min}$ and $R_{\max}$, respectively).

For our experiments, we leverage a cloud computing architecture featuring Intel Xeon CPU E5-2620 v3 and instantiate two identical virtual machines (VMs), one acting as source and the other as destination of the migration process. The two VMs, with Ubuntu 20.4 LTS as operating system, are assigned 4 vCPUs and 16 GB of RAM each. The size of each memory page is set equal to 4,096 B. The results shown in the following have been obtained by averaging over 50 runs, and computing the 99% confidence interval.

IV. Modeling migration and downtime duration

We now present our experimental analysis and leverage it to derive the Processing-Aware Migration (PAM) model. The PAM model accurately describes, regardless of the specific microservice, the fundamental KPIs that characterize stateful container migration and their components. Given (1), (7) and (8), we relate T^{mig} and T^{down} at the Podman layer to CRIU time metrics. The parameter setting used for the proposed model depend upon the specific testbed architecture and its computational capabilities; however, they can be estimated easily for any scenario by running few experiments.

A. Checkpoint duration

As shown by the experimental results in Fig. 4, the overhead introduced by Podman with respect to the underlying runC and CRIU layers can be approximated through multiplicative constant factors (respectively, α_1 and α_2). Then, combining this observation with (2), we get:

$$T^{\text{p}}_{\text{podman}} = \alpha_1\alpha_2 \cdot (T^{\text{freeze}}_{\text{criu}} + T^{\text{p,frozen}}_{\text{criu}} + T^{\text{p,mem}}_{\text{criu}}) \tag{9}$$

$$T^{\text{d}}_{\text{podman}} = \alpha_1\alpha_2 \cdot (T^{\text{freeze}}_{\text{criu}} + T^{\text{d,frozen}}_{\text{criu}}) \,. \tag{10}$$

Additionally, we can replace $T^{\text{freeze}}_{\text{criu}}$ with a constant, β, as demonstrated by the results in Fig. 3(left).

Fig. 3(center) provides experimental evidence that the frozen time has a linear relationship with the microservice state size M and it depends on both the dirty page rate R and the type of phase, i.e., predump or dump. Thus, we can write:

$$T^{\text{p,frozen}}_{\text{criu}}(M) = \varphi^{\text{p}} + \gamma^{\text{p}} \cdot M \;;\; \gamma^{\text{p}} = \Gamma \cdot \zeta \tag{11}$$

$$T^{\text{d,frozen}}_{\text{criu}}(M,R) = \varphi^{\text{d}} + \gamma^{\text{d}}(R) \cdot M \;;\; \gamma^{\text{d}} = \Gamma \cdot \xi(R) \,. \tag{12}$$

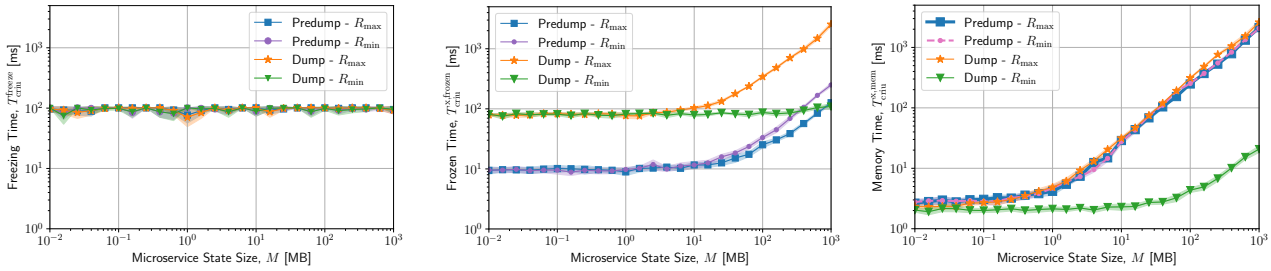


Fig. 3: Checkpoint time contributions at CRIU layer, namely (left) freezing time, (center) frozen time, and (right) memory time

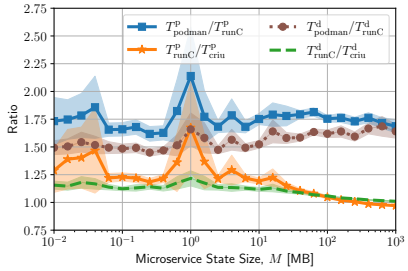


Fig. 4: Comparison between different stage durations at Podman, runC, and CRIU layer

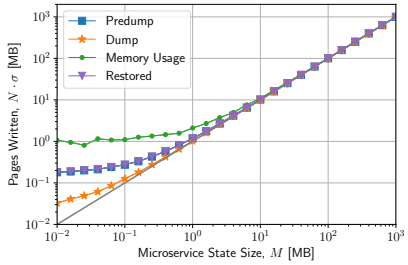


Fig. 5: Amount of memory pages written by CRIU in the final checkpoint image, for $R_{\max}$

Note that φ^{p} and φ^{d} act as a lower bound due to CRIU algorithms and the specific implementation, and they are independent of the microservice state size and the dirty page rate. Coefficients γ^{p} and γ^{d} are sensitivity factors that relate processing time with memory allocation; they consist of a constant Γ scaled by parameters ζ and ξ (resp.), with the latter expressing the relationship with the dirty page rate R.

Next, according to the results in Fig. 3(right), the processing time contribution due to memory operations, i.e., pages selection and extraction, linearly depends upon M. Thus,

$$T^{\mathrm{p,mem}}_{\mathrm{criu}}(M)=\delta+\Lambda\cdot M \;;\; T^{\mathrm{d,mem}}_{\mathrm{criu}}(M,R)=\delta+\Lambda\cdot\eta(R)\cdot M\,, \quad (13)$$

where δ is a lower bound independent of the stage (predump or dump), the microservice state size, or the dirty page rate; $\eta(R) \in (0,1]$, in accordance with the behavior shown in Fig. 3(right), models the impact of the dirtiness tracking system adopted in dump iterations and its relationship with R; Λ is a constant scaling factor.

Finally, T^{net} is approximated as the time needed to transfer V_i data over a link of capacity L_i, i.e., $T^{\mathrm{net}} = V_i/L_i$. According to the experimental behavior depicted as an example in Fig. 5 for $R_{\max}$, the number of memory pages written into the checkpoint image linearly depends upon the microservice state size:

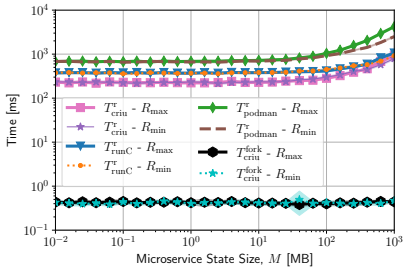


Fig. 6: Restore operations time at CRIU, runC, and Podman layer

$$N^{\mathrm{p}}(M) = \mu^{\mathrm{p}} + \nu^{\mathrm{p}} \cdot M \quad (14)$$

$$N^{\mathrm{d}}(M,R) = \begin{cases} \mu^{\mathrm{d}}(R) \;\text{ if } M \leq 10^5\,\mathrm{B} \\ \mu^{\mathrm{d}}(R) + \nu^{\mathrm{d}}(R)\cdot M \;\text{ if } M > 10^5\,\mathrm{B}\,. \end{cases} \quad (15)$$

In (14) and (15), μ^{p} and μ^{d}, and slopes ν^{p} and ν^{d}, describe, respectively, the minimum number of pages extracted and the overhead with respect to the actual microservice state size.

B. Restore duration

To address the restoration of the microservice state at the destination host, we leverage the experimental evidence that, similarly to what shown for the predump and dump phases, relates the restoration time to the duration at the runC layer and the latter to the restore duration at the CRIU layer through constant values (namely, α_3 and α_4 below). Furthermore, considering (5) and given that the forking time can be neglected and the context relocation time linearly depends upon M only (see the results in Fig. 6), we can write:

$$T^{\mathrm{r}}_{\mathrm{podman}} \approx \alpha_3\alpha_4 T^{\mathrm{reloc}}_{\mathrm{criu}} = \alpha_3\alpha_4(\psi + \omega \cdot M)\,. \quad (16)$$

In (16), ψ is the minimum time needed to accomplish a restore procedure, regardless of the value of M, while ω accounts for the impact of M on the total restore duration.

C. Dirty page rate analysis

We now enhance our model by investigating and characterizing the dependency of its parameters on the dirty page rate. To this end, we extend the experimental setting introduced in Sec. III to consider any value of dirty page rate as input, and we define $\hat{R} = \frac{R-R_{\min}}{R_{\max}-R_{\min}}$ as the normalized dirty page rate.

The experimental results in Fig. 7(left) highlight that ξ, η, μ^{d}, and ν^{d} have a linear relationship with

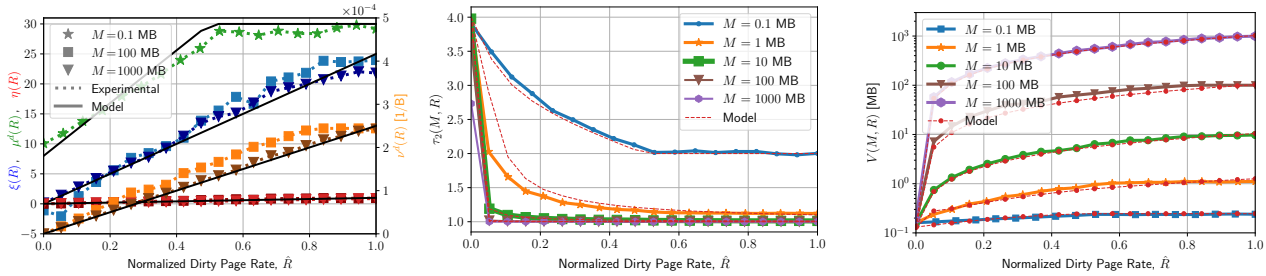


Fig. 7: Experimental behavior of model parameters vs dirty page rate, and analytical function that best fit them

the dirty page rate, which can be modeled as follows: $\xi(R)$=$25\hat{R}$; $\eta(R)$=0.01+$0.99\hat{R}$; $\nu^{\mathrm{d}}(R)$=$2.5{\cdot}10^{-4}\hat{R}$; $\mu^{\mathrm{d}}(R) = 8$+$44\hat{R}$ if $\hat{R} < 0.5$, and $\mu^{\mathrm{d}}(R) = 30$ otherwise.

On the other hand, as depicted in Fig. 7(center), for small values of microservice state size M (i.e., $M < 10^5$ B), τ_2 can be approximated by a logarithmic relation with $\hat{R}$ (see the blue curve), while for $M \geq 10^5$ B such relationship is hyperbolic.

Finally, Fig. 7(right) validates our model by comparing analytical and experimental results for the volume of data V_i to be transmitted, as given in Sec. II and combined with (14)-(15). The results highlight (i) how the data volume depends upon $\hat{R}$ and the state size M, as well as (ii) the excellent match between our model and the experimental results.

D. Migration KPIs

We can now derive the PAM model for the fundamental migration KPIs, i.e., migration duration and downtime. Combining (7), (9), and (10), the duration of the Iterative PreCopy stage for iteration i can be written as:

$$T_0 = \alpha_1\alpha_2 \cdot (T_{\mathrm{criu}}^{\mathrm{freeze}} + T_{\mathrm{criu}}^{\mathrm{p,frozen}} + T_{\mathrm{criu}}^{\mathrm{p,mem}}) + T^{\mathrm{net}} \quad (17)$$

$$T_i = \alpha_1\alpha_2 \cdot (T_{\mathrm{criu}}^{\mathrm{freeze}} + T_{\mathrm{criu}}^{\mathrm{d,frozen}}) + T^{\mathrm{net}}. \quad (18)$$

Then, using (8), (10), and (16), the downtime is given by:

$$T^{\mathrm{down}} = \alpha_1\alpha_2 \cdot (T_{\mathrm{criu}}^{\mathrm{freeze}} + T_{\mathrm{criu}}^{\mathrm{d,frozen}}) + T^{\mathrm{net}} + \alpha_3\alpha_4 T_{\mathrm{criu}}^{\mathrm{reloc}}. \quad (19)$$

Finally, combining (1), (17), (18) and (19), we get the total migration duration:

$$T^{\mathrm{mig}} = \alpha_1\alpha_2 \left(T_{\mathrm{criu}}^{\mathrm{freeze}} + T_{\mathrm{criu}}^{\mathrm{p,frozen}} + T_{\mathrm{criu}}^{\mathrm{p,mem}}\right) + T^{\mathrm{net}} + (I+1)\cdot \left(\alpha_1\alpha_2 \cdot (T_{\mathrm{criu}}^{\mathrm{freeze}} + T_{\mathrm{criu}}^{\mathrm{d,frozen}}) + T^{\mathrm{net}}\right) + \alpha_3\alpha_4 T_{\mathrm{criu}}^{\mathrm{reloc}}. \quad (20)$$

V. Model Validation

We now validate the PAM model by using popular, real-world microservices, namely, *MQTT Broker* and *Memcached*. As shown below, our results suggest that the PAM model accurately describes stateful migration performance and remarkably outperforms the state-of-the-art (SotA) model in [2].

Validation setup. *MQTT* is a publish/subscribe protocol, commonly used for IoT applications, which involves three main logical entities: broker, publisher, and subscriber. An MQTT broker is a microservice acting as an intermediate between publisher and subscriber. Since the MQTT broker manages the connections and preserves the messages that must be delivered in its internal queue, a stateful approach is fundamental to prevent information loss during migration. *Memcached* is an in-memory, key-value store intended as user-defined and high-performance caching system. Other than speeding up applications by alleviating database load, Memcached is widely exploited to define distributed virtual pools of memory. Clearly, due to its memory-related nature, Memcached migration must be stateful to prevent information loss. To thoroughly evaluate the migration performance, we define a validation setup that allows for a fine tuning of the microservice state size and of the dirty page rate.

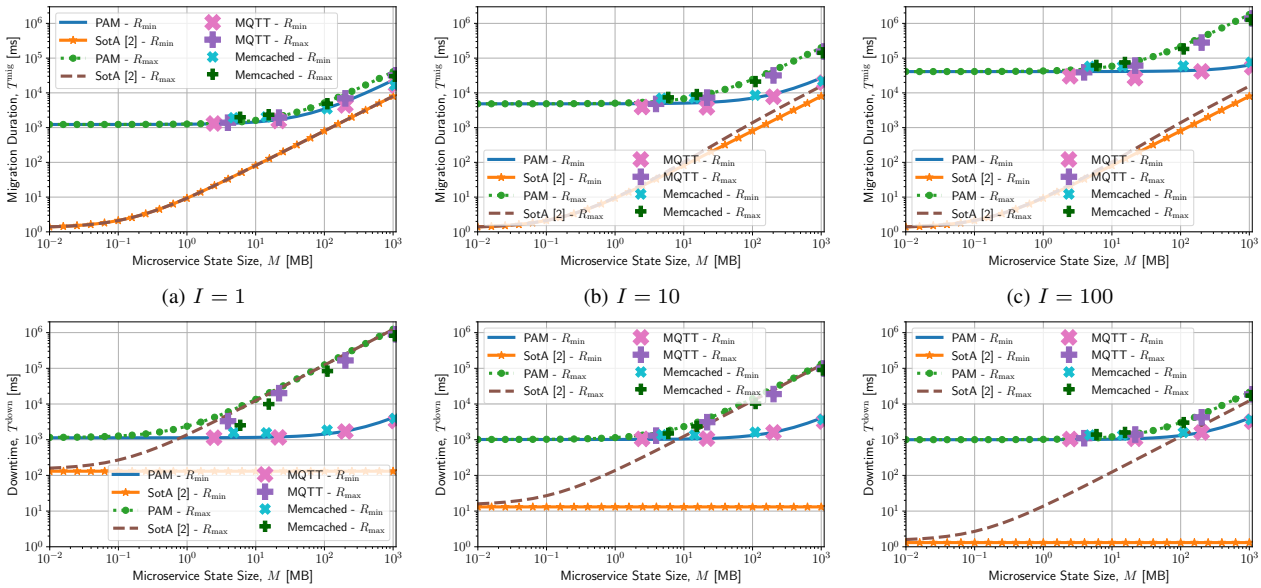
Results. Figures 8a–8c present the total migration duration as a function of the number of iterations I. Observe how the PAM model (blue and green curves, respectively, for $R_{\min}$ and $R_{\max}$) matches the experimental results obtained with real-world microservices ("x" and "+" markers) very closely in all cases, while the state-of-the-art, ideal model in [2] (orange and brown curves) is unable to do so. The reason for this behavior is that, under ideal conditions (i.e., not accounting for the processing contribution), the number of pages to be transmitted decreases at each iteration, and, hence, so does the iteration duration. Instead, combining (3) and (4), it can be seen that, according to the PAM model, the number of memory pages written during the i-th dump iteration depends upon both the processing overhead and the network transfer, with the processing time being the dominant component.

Figures 8d–8f further investigate the downtime, versus the state size M, for varying values of bandwidth L. Again, notice how our model well approximates the migration performance, and the gap with respect to the SotA model dramatically increases with L. Indeed, consistently with (19), the larger L, the more significant the processing contribution to the downtime, due to the dump and the restore phases.

Finally, by looking at Figures 8c and 8d, we observe that dirtiness is best leveraged for large values of M, while, for lower values, the KPIs are practically independent of R.

VI. Related Work

A large body of work has investigated container live migration. A taxonomy of the main stateful migration techniques is presented in [4], [5]. Promising applications of stateful migration by using CRIU can instead be found in [6], [7]. As for latency-sensitive applications, [8] proposes migration as a decisive technology to ensure proximity of services to IoT, while [9] presents a migration framework for mobile core network components and demonstrates that container



(a) $I = 1$ (b) $I = 10$ (c) $I = 100$

(d) $L = 10$ Mbps (e) $L = 100$ Mbps (f) $L = 1000$ Mbps

Fig. 8: Model validation: migration duration vs no. of iterations, with network bandwidth set to $L = 1$ Gbps (top); downtime vs L (bottom)

PreCopy outperforms other strategies and virtualization technologies. [10] introduces Teddybear, a Docker based system that enhances live migration by using the user's mobile device as a carrier for the container. [11] presents CloudHopper, a functional live migration system for containerized applications that holds and redirects client connections. Further, [12], [13] investigate QUIC, underlying its validity for stateful migration and extending it to support server-side connection migration.

Few studies however have modeled microservice migration. The recent work in [14] explores container orchestration in a hybrid computing environment and proposes an optimization model to achieve minimal downtime for fault recovery by either re-instantiating or migrating containers. The closest work to ours is [2], which introduces an ideal model that serves as a starting point for planning and scheduling multiple VMs. As mentioned, our goal is to present a model more accurate than the one in [2] by capturing all the relevant real-world aspects of the container migration process.

VII. Conclusions

We proposed a novel processing-aware migration model that effectively characterizes the fundamental stateful migration KPIs. Using state-of-the-art tools, we evaluated their processing overhead and assessed their impact on the migration performance. The results show that our model accurately describes the migration process, substantially outperforming the state-of-the-art. In addition, our study demonstrates that Iterative PreCopy is very effective for microservices with a large state size and low dirty page rate, while alternative solutions are required in case of small state size or high dirty page rate. Future work will exploit the proposed model to optimally configure the system for the migration of various, real-world containerized microservices.

References

[1] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[2] T. He, A. N. Toosi, and R. Buyya, "SLA-aware multiple migration planning and scheduling in SDN-NFV-enabled clouds," *J. of Systems and Software*, vol. 176, p. 110943, 2021.

[3] D. Fernando, J. Terner, K. Gopalan, and P. Yang, "Live migration ate my VM: Recovering a virtual machine after failure of post-copy live migration," in *IEEE INFOCOM*, 2019, pp. 343–351.

[4] M. Terneborg, J. K. Rönnberg, and O. Schelén, "Application agnostic container migration and failover," in *IEEE LCN*, 2021, pp. 565–572.

[5] G. Singh and P. Singh, "A taxonomy and survey on container migration techniques in cloud computing," *Sustainable Development Through Engineering Innovations*, pp. 419–429, 2021.

[6] M. Sindi and J. R. Williams, "Using container migration for HPC workloads resilience," in *IEEE HPEC*, 2019, pp. 1–10.

[7] H. Htet, N. Funabiki, A. Kamoyedji, X. Zhou, and M. Kuribayashi, "An implementation of job migration function using CRIU and podman in docker-based user-pc computing system," in *ACM ICCCM*, 2021.

[8] C. Puliafito, A. Virdis, and E. Mingozzi, "The impact of container migration on fog services as perceived by mobile things," in *IEEE SMARTCOMP*, 2020, pp. 9–16.

[9] S. Ramanathan, K. Kondepu, M. Razo, M. Tacca, L. Valcarenghi, and A. Fumagalli, "Live migration of virtual machine and container based mobile core network components: A comprehensive study," *IEEE Access*, vol. 9, pp. 105 082–105 100, 2021.

[10] A. Elgazar and K. Harras, "Teddybear: Enabling efficient seamless container migration in user-owned edge platforms," in *IEEE CloudCom*, 2019, pp. 70–77.

[11] T. Benjaponpitak, M. Karakate, and K. Sripanidkulchai, "Enabling live migration of containerized applications across clouds," in *IEEE INFOCOM*, 2020, pp. 2529–2538.

[12] L. Conforti, A. Virdis, C. Puliafito, and E. Mingozzi, "Extending the QUIC protocol to support live container migration at the edge," in *IEEE WoWMoM*, 2021, pp. 61–70.

[13] C. Puliafito, L. Conforti, A. Virdis, and E. Mingozzi, "Server-side QUIC connection migration to support microservice deployment at the edge," *Pervasive Mobile Computing*, 2022.

[14] S. Aleyadeh, A. Moubayed, P. Heidari, and A. Shami, "Optimal container migration/re-instantiation in hybrid computing environments," *IEEE Open J. of the Communications Society*, vol. 3, pp. 15–30, 2022.