Hybrid Adaptive Checkpointing for Virtual Machine Fault Tolerance

Abel Souza*, Alessandro Vittorio Papadopoulos†, Luis Tomás Bolivar*‡, David Gilbert‡, Johan Tordsson*
*Department of Computing Science, Umeå University, Sweden.
†IDT, Mälardalen University, Sweden.
‡Red Hat Inc.
Email: *{abel,tordsson}@cs.umu.se, †alessandro.papadopoulos@mdh.se, ‡{ltomasbo,dgilbert}@redhat.com.

***Abstract*—Active Virtual Machine (VM) replication is an application independent and cost-efficient mechanism for high availability and fault tolerance, with several recently proposed implementations based on checkpointing. However, these methods may suffer from large impacts on application latency, excessive resource usage overheads, and/or unpredictable behavior for varying workloads. To address these problems, we propose a hybrid approach through a Proportional-Integral (PI) controller to dynamically switch between periodic and on-demand checkpointing. Our mechanism automatically selects the method that minimizes application downtime by adapting itself to changes in workload characteristics. The implementation is based on modifications to QEMU, LibVirt, and OpenStack, to seamlessly provide fault tolerant VM provisioning and to enable the controller to dynamically select the best checkpointing mode. Our evaluation is based on experiments with a video streaming application, an e-commerce benchmark, and a software development tool. The experiments demonstrate that our adaptive hybrid approach improves both application availability and resource usage compared to static selection of a checkpointing method, with application performance gains and neglectable overheads.**

***Index Terms*—Fault Tolerance, Resource Management, Checkpoint, COLO, Control Theory**

I. Introduction

Fault tolerance is the ability of a system to keep working even in case of failures. It is mainly used to increase system uptime and to improve its dependability, reliability and availability [13]. Fault tolerance is a desirable feature especially for mission critical applications residing in data centers [27]. One method to achieve fault tolerance is through *replication*, where critical parts of a system are duplicated using additional hardware, software and/or network resources, to provide extra guarantees in case any of these resources fail [13]. Most fault-tolerance systems have an asymmetry in the sense that a *primary* host executes the application, and a *secondary* host takes over the workload in case the former fails. A key requirement for any replicated system is that failures in a host must be recovered without human intervention no matter when they occur or what the host is doing at the time of failure.

A. Existing Solutions

Fault tolerance can be achieved at different levels of a system stack. Existing solutions range from (i) implementing fault tolerance capabilities inside applications [3]; to (ii) using fault tolerant hardware to provide component redundancy [1]; to (iii) using application agnostic solutions based on virtualization, where applications are wrapped in Virtual Machines (VMs) and the state of a Primary VM (PVM) is replicated to a secondary VM (SVM) running on a different physical server [6]. Solutions of type (i) may not be feasible since applications may not be easily modified to support specific replication strategies and fault tolerance hardware. Type (ii) solutions tend to be very expensive. Type (iii) approaches, also known as general-purpose methods, aim to detect and handle hardware faults at the virtualization layer of the software stack. All types hide system errors from end-users, who can access applications without interruptions even in the presence of faults.

For interactive applications handling network requests, replication approaches are usually based on active/active (A/A) and active/passive (A/P) [9]. In A/A mode, all replicas actively receive and process the same requests concurrently. In A/P mode, only the primary host receives and processes requests, whereas the secondary only receives memory state synchronizations. These two approaches mainly differ in resource consumption tradeoffs, as A/A requires additional CPU and A/P more network bandwidth.

One replication method is *Checkpointing* [11], a technique to enable a system or application to store its state, and to handle failures by recovering from previously stored states. Checkpoint based techniques can be classified as *periodic* and *on-demand* [9]. The former performs checkpoints at fixed time intervals. For interactive applications, periodic checkpointing can affect system performance (e.g., latency) depending on the network bandwidth and checkpoint size. On-demand checkpointing tries to address these problems by reducing the frequency of synchronizations in order to maximize the amount of time an application runs (also known as application *uptime*). One on-demand checkpoint approach is the COarse-grained LOck-stepping Virtual Machines for Non-stop Service (COLO) [10]. COLO simultaneously runs two identical VMs containing the application. They are synchronized through a checkpoint only when it is detected that their executions diverged, defined as when they generate different responses to the same network request. Notably, due to symmetric multiprocessing, thread scheduling, etc., the actual execution inside the PVM and SVM frequently diverges, but this is ignored in COLO unless the deviation is noticeable from the client perspective.

0-7695-6371-6/18/31.00 ©2018 IEEE
DOI 10.1109/IC2E.2018.00023

In contrast, passive checkpointing no comparison of VM outputs are required as a synchronization (checkpoint) is triggered at fixed-length interval times, e.g., at every 250ms as illustrated in Figure 1 (a). This guarantees the consistency of both replicas, as the application is not executed in the SVM. As client requests are withheld until both replicas are synchronized, application performance, as perceived by the clients, may be impacted.

As mentioned, COLO (Figure 1 (b)) requires twice the amount of memory and additional CPU resources in the secondary host: for the running process and for the state synchronization mechanism. Every client request is forwarded to both replicas, and their responses are compared by a proxy module located in the PVM host (the diamond in Figure 1 (b)). If VM responses are identical, the request is released from the proxy to the client. Otherwise, the SVM memory state is synchronized with the PVM by checkpointing, before the response is released to the client.

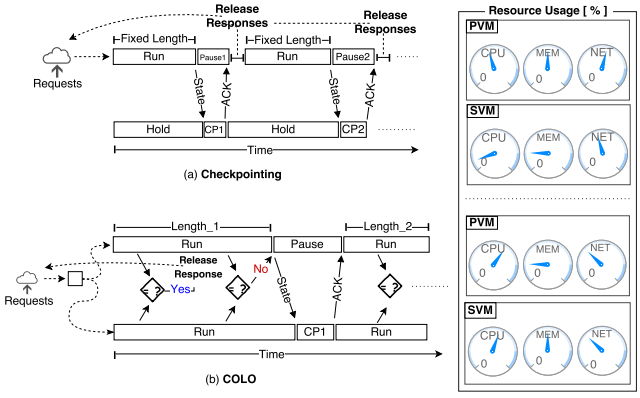

Fig. 1: Overview of operation in PVM (shown above) and SVM (shown below) for checkpointing (a) and COLO (b). The dials in the right hand side illustrate common resource usage patterns for the two methods.

B. Problem Statement and Objectives

COLO and checkpointing modes alter the frequency of state synchronizations between the PVM and SVM. They also highlight an important resource utilization tradeoff: given the non-deterministic nature of (multi-threaded) applications and workload variations, the frequency of COLO checkpoints is itself non-deterministic and can in some cases even exceed that of periodic checkpointing, which results in both performance degradation and inefficient use of resources.

Conversely, periodic checkpointing may impose an excessive overhead under different circumstances, which may severely limit its feasibility. For instance, by freezing the PVM during execution and/or by withholding requests from users until its state synchronization is completed, periodic checkpointing mode may increase application latency significantly. Hence, there is no single checkpointing mechanism that achieves acceptable performance tradeoffs in all scenarios.

To tackle these problems, we propose a hybrid approach that dynamically switches between periodic checkpointing and COLO. Such dynamic switching enables system administrators to select the most appropriate checkpoint approach without having to investigate the behavior of the workload in detail. This is particularly useful for data center scenarios with thousands of applications with varied workload behaviors. To optimize the use of this hybrid approach, we design an autonomic mechanism to dynamically select the best checkpointing mode. We provide two implementations of the selection mechanism.

The first is based on a threshold feedback loop that continuously calculates the frequency of checkpoint synchronizations in order to decide the best checkpoint mode. The second is based on a Proportional-Integral (PI) controller that automatically adapts the time spent in periodic (fixed-length checkpoint) and on-demand (COLO) checkpoint modes based on the uptime ratio: the relation of time spent checkpointing over the total run time before a checkpoint is triggered. This ratio is used to dynamically select the best mode for a workload in a given time interval. Our implementation of the hybrid mechanism is open-sourced and is based on modifications to QEMU, LibVirt, and OpenStack. Our experimentation with a mix of applications shows that the hybrid approach is superior to statically selected checkpoint mode, and that our controller further improves system's uptime by up to 10% with neglected performance degradations.

II. Background and Related Work

There is a range of fault tolerance mechanisms to enable the recovery (or the application to keep working) after a failure occurs. Usually, the next actions triggered to resume normal operation [15] are: (i) Reconstruction, where the primary server/site is restored either at its original server/site or another server/site; and (ii) Failover where there exists a secondary or back-up server/site where the primary server/site is being replicated. This secondary server/site becomes the active/working site after the primary site fails.

For both cases redundant hardware is required. However, without replicated data, redundant hardware is of little use [15]. Thus, not only the secondary storage needs to be replicated, but also the in-memory data (i.e., the application state) needs to be replicated so that the failover is fully transparent to users—a failure should not be noticed from clients' viewpoint. To enable this, fault tolerance techniques should minimize both the recovery time and the data loss in case of a failure.

Reconstruction-based solutions, also known as storage-based application fault tolerance solutions, provide a relatively high Recovery Time Objective (RTO) due to the time needed for the application restart on the backup host, and it may also cause data loss or corruption if the running application does not have built-in mechanisms (such as transactional commits in relational databases) to ensure data is not lost, providing a non-zero Recovery Period Objective (RPO). One example of this type of fault tolerance mechanism is VMWare High Availability (HA) [28].

Nevertheless, if a transparent recovery from the clients' point of view is needed, both RPO and RTO need to be minimized, and thus failover techniques are needed.

Mechanisms that provide fault tolerance in an application-agnostic manner can be classified in Server Lockstep and Server State Synchronization Fault Tolerance techniques. The former is based on a technique called Lockstep Processing in which the server is outfitted with Dual Modular Redundancy (DMR) that enables redundant components to process the same instructions simultaneously. Many enterprises that require a high degree of availability for running mission-critical applications use hardware based solutions such as ftServer from Stratus [25] that guarantees 99.999% uptime. Lockstep Processing hardware configuration includes two CPUs, chipsets and memory units, thus, eliminating any single point of failure in the system, as each CPU or memory units can function even if its counterpart is offline due to failure or planned downtime. An equivalent to ftServer is the VMWare FT/vLockstep [29] hypervisor based software-only solution, which relies on deterministic record/replay of non-deterministic inputs.

These application-agnostic approaches have minimal performance degradation and low bandwidth use as only non-deterministic events (e.g., interrupts) are recorded on the primary server and transmitted to the secondary server. However, they require either dedicated hardware (with an integrated lockstep support mechanisms), or cannot support non-SMP (Symmetric Multi-Processing), thus limiting their use.

Server State Synchronization techniques address these problems. They mainly use a modified version of asynchronous replication methods to better handle the trade-offs between keeping the replica as closely synchronized as possible, and minimizing the impact on application performance, as well as other co-located VMs [14], [20], [30].

Examples of active fault tolerant approaches include MicroCheckpointing [12], Remus [7] and Kemari [26]. These all provide high-availability by using efficient 1+1 topology where a single physical server can act as standby host for multiple applications. These approaches differ from the previously mentioned Lockstep Processing as the active VM is the only VM to execute operations. Remus provides a high degree of fault tolerance for a VM by asynchronously propagating changed states to a backup host at frequencies as high as forty times per second. Adaptive Remus quantifies VM metrics as CPU and memory usage to infer the current hosted application load. With this information, the mechanism adjusts the checkpointing frequency between two modes [8].

However, the major drawbacks of such active fault tolerance approaches are degraded performance and increased network consumption, as they depend on heavyweight tracking and synchronization of the entire system state. COLO [10] was designed to mitigate this problem and is also based on server state synchronization but, unlike the previous works, uses active replication. To avoid the excessive overhead of current VM replication techniques, COLO monitors the output response of both VMs and only forces a synchronization (checkpoint) between them when their responses differ, i.e., the client would notice the deviation. If the responses do not match, network is withhold until the PVM's state is synchronized to the SVM.

Although the differences between COLO and periodic checkpoint are clear, it is non-intuitive to assess the impact of each method on a given application [16], [24]. On the downside, periodic checkpointing does not consider application performance. Furthermore COLO would not perform well with highly non-deterministic applications because it may lead to a high number of checkpoints, which in turn would lead to performance degradation due to the continuous VM pauses to perform checkpoints. Based on the same argument, periodic checkpointing may not perform well for similar applications as it assumes static workload behavior, thus unnecessarily pausing the VM at constant times.

In fact, it may happen that COLO performs differently over time even for a single application. Given deviations on application workload, COLO may perform better or worse than periodic checkpointing. In cloud scenarios, COLO may also be affected by other co-located VMs (noisy neighbors) that may cause additional replication actions. These observations suggest that a hybrid mechanism can outperform static selection of either COLO or checkpoint mode for an application.

III. Hybrid Mechanism Architecture

In this paper, we propose a hybrid fault-tolerance mechanism that combines COLO with the periodic checkpointing approach, and dynamically decides which is the most suited mode on the basis of current application downtime. The overall goal of our approach is to improve application availability here defined as $uptime/(uptime+downtime)$, where *uptime* denotes that the application is running and *downtime* that it is not running due to checkpointing.

The hypothesis is that PVM/SVM resource usage and client-observed performance are optimized the longer the application is available to answer requests. For instance, if synchronizations are too frequent, because the workload is too non-deterministic while in COLO mode, CPU and bandwidth would be poorly utilized and checkpoint mode would save SVM's CPU usage while using the bandwidth more efficiently. On the other hand, if the workload is very deterministic while in checkpoint mode, then the frequency of synchronizations would potentially be low, and COLO would improve application performance, while saving bandwidth and making better usage of SVM's CPU.

Figure 2 shows the overall system architecture of our hybrid mechanism. We target a cloud scenario, with OpenStack [23] being used to manage servers and VMs. We further use a software stack composed of modifications to QEMU [2], LibVirt [21], and the OpenStack Nova components. In our setup, OpenStack is used on top of LibVirt to manage a collection of hosts similarly to a cloud provider managing many VMs. Libvirt is a system orchestrator that interacts with the virtualization capabilities, i.e., wraps hypervisors at a low level to provide an easier interface to both operators and management tools. The hypervisor, QEMU, is the software that performs hardware virtualization. In our modified OpenStack,

a *fault tolerant VM* can be created by the user, causing the VM to run on a pair of hosts instead of on a single server. The VM state is synchronized between these two hosts using the COLO and/or periodic checkpointing mechanisms.

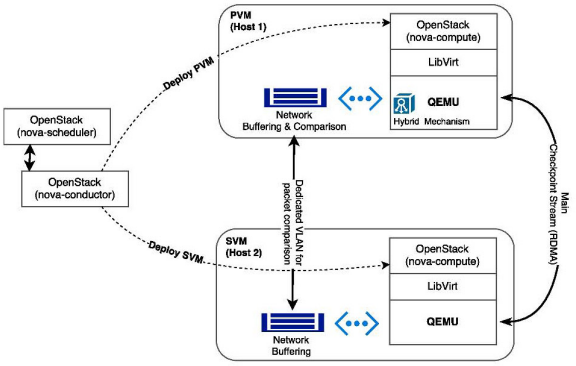

Fig. 2: General Architecture of the hybrid checkpointing approach

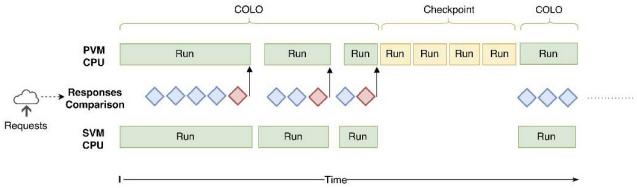

Fig. 3: Overview of the adaptive of the hybrid checkpointing operation. The system runs in COLO mode until a response comparison failure (red diamond) causes a new checkpoint. Frequent failures trigger a transition into checkpointing mode (shown by the PVM running and the SVM not). At a later stage, the system switches mode and retries COLO again.

We made two main modifications: (i) to the standard OpenStack Nova component to allow the creation of VM pairs, where the VMs are guaranteed to run in different hosts for fault tolerance purposes, and (ii) to the QEMU and LibVirt to allow the configuration and setup for this VM pair. Our modifications to QEMU and LibVirt ensure that a VM pair is enabled with one VM in primary mode and the other in secondary mode. The creation of a VM pair also requires setup of extra network connections and devices needed by COLO.

A separate logical network carriers the duplicated input traffic from the PVM to SVM and also responses from the SVM to PVM for output comparison. The existing RDMA (Remote direct memory access) migration code in QEMU has been modified to allow it to live migrate a VM with fault-tolerance in the way COLO and hybrid modes operate. This is created by using a dedicated socket between both servers, ensuring that no other VM can use the dedicated VLAN tag used for the VM-pair.

The hybrid checkpointing scheme is implemented in the QEMU component and its overall operation is illustrated in Figure 3. The pair of VMs is started by LibVirt and when

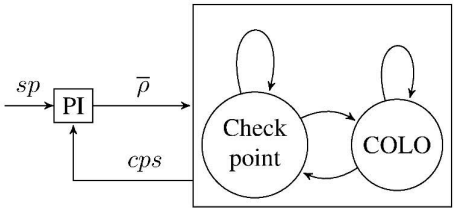

Fig. 4: Structure of the autonomous solution with the PI-controller (left) and a state machine (right). The PI-controller accepts a user-defined set-point (sp, system's aim) and cps (checkpoints per second) as inputs, both used to calculate $\overline{\rho}$, the controller's mode switcher.

requested from OpenStack. The hybrid mode adds a set of migration parameters that allow setting timings of the checkpointing modes and threshold levels (which trigger transition to fixed length checkpoints). Additional monitoring including timing of the various stages of the checkpoint process have been added to allow LibVirt to pass these values up to monitoring layers; this allows administrators to check for bottlenecks caused by lack of bandwidth on the synchronization link between the two VMs.

All of the high level operations are available through the OpenStack interfaces. In fact, from a user point of view, the only change to use the new functionality and create VM in FT (fault-tolerance) mode, is to set a fault-tolerant flag, which is a single flag in the configuration flavor of OpenStack, where compute, memory, and storage capacity for an instance are defined.

IV. Controller Design

As discussed earlier, dynamic switching between COLO and periodic checkpointing mode, henceforth *Checkpoint*, can be very beneficial. However, detecting when to switch from one mode to another cannot be performed by the system administrator, but must be done autonomously by the system as the application workload is unknown. We here present a two layer approach that automates the monitoring and decision making strategy.

In particular, we take decisions based on the measured *Downtime Ratio* ρ, defined as the ratio between *average downtime* and the *average uptime* of the running VM. Downtime is defined as the amount of time taken for the synchronization process to capture, transmit state data and receive an acknowledgment that the checkpoint has been processed in the SVM. If ρ is close to zero, it means that the VM is mostly running, and that the downtime has a limited impact on the actual VM performance. If ρ reaches a value close to 1, or exceeds 1, it means that the operational mode is not providing satisfactory performance, and a change in the mode may be required. This metric is relevant because it simultaneously captures, in a generic manner, key synchronization aspects related to performance of network bandwidth, CPU and memory.

Figure 4 shows the scheme of the proposed autonomous solution. The inner component is in charge of deciding the

actual mode in which the system operates. The outer layer monitors the number of checkpoints per second (here denoted as cps), and it computes what is a good target value $\overline{\rho}$ for ρ, in order to dynamically enforce higher or lower likelihood for the system to be in COLO or checkpoint modes. For example, if cps is low, COLO mode is preferable, and $\overline{\rho}$ should be chosen to be high. The parameter $\overline{\rho}$ determines whether to make the switch to the other mode faster or slower.

A. Inner layer: Monitoring and Actuation level

The inner layer of the proposed decides the mode of operation, i.e., Checkpoint or COLO mode. The behavior can be represented by a discrete-time hybrid automaton [18], i.e., for each discrete state – here called *discrete state* or simply *state* – of a hybrid automaton, a discrete-time dynamic system is defined—here called *continuous component*. All the continuous components share a set of *state variables*, describing the actual state of the system. A *transition* from one mode to another is therefore defined on the basis of conditions on the state variables of the continuous components, and *reset* maps can be executed when a transition occur.

We here consider a set of three modes $Q = \{q_1, q_2, q_3\}$, associated with Checkpoint mode (q_1) and with COLO mode (q_2 and q_3). The overall status of the automaton is characterized by the current mode $q(k)$ at the k-th time interval, and two counter variables $b(k)$ (a *time budget* for staying in Checkpoint mode), and $\sigma(k)$ (the *switching-to-Checkpoint likelihood*, that is high when the system should switch back to checkpoint and low, when it should stay in COLO mode). In addition, we monitor the current value of $\rho(k)$, which works as a mode switcher. The continuous components are defined as follows:

- if $q(k) = q_1$, the time budget $b(k)$ is linearly decreased over time, while $\sigma(k)$ is kept constant, i.e.

$$\begin{cases} b(k+1) = b(k) - 1 \\ \sigma(k+1) = \sigma(k), \end{cases} \tag{1}$$

- if $q(k) = q_2$, the time budget $b(k)$ is linearly increased over time, while the variable $\sigma(k)$ is kept constant, i.e.

$$\begin{cases} b(k+1) = b(k) + 1 \\ \sigma(k+1) = \sigma(k) \end{cases} \tag{2}$$

- if $q(k) = q_3$, the counter variable $\sigma(k)$ is decreased over time by a fixed value $d > 0$ down to a minimum value $\sigma_{\min}$, while the time budget $b(k)$ is kept constant, i.e.

$$\begin{cases} b(k+1) = b(k) \\ \sigma(k+1) = \max\left(\sigma(k) - d, \sigma_{\min}\right) \end{cases}. \tag{3}$$

The transitions from one node to the others and the corresponding reset actions are defined as follows:

- A transition from q_1 to q_2 happens if and only if $b(k) \leq 0$. When the transition occurs, the time budget is reset as:

$$b(k) = 0 \tag{4}$$

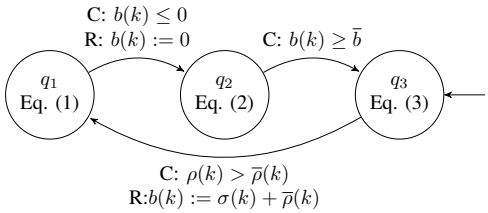

Fig. 5: Scheme of the hybrid state automaton in the inner layer. The conditions for the transition to occur, and the reset map to apply are indicated with "C" and "R" respectively in the figure.

- A transition from q_2 to q_3 happens if and only if $b(k) \geq \overline{b}$, with $\overline{b}$ being a prescribed maximum threshold. When the transition occurs, no reset occurs.
- A transition from q_3 to q_1 happens if and only if $\rho(k) > \overline{\rho}$. When the transition occurs, the time budget is reset as:

$$b(k) = \sigma(k) + \overline{\rho}(k) \tag{5}$$

Figure 5 summarizes the functionality of the inner layer of the automated solution. The layer is initialized to be in mode $q(0) = q_3$, i.e., in COLO mode, and the variables are initialized to be:

$$\begin{cases} b(0) = 0 \\ \sigma(0) = 1 \end{cases}$$

The overall idea is that when the VM is run in checkpoint mode (mode q_1), there are regular checkpoints periodically. Therefore, if we assign an initial budget $b(0) > 0$ where the system is operated in such a mode. When the budget is elapsed, the system tries to switch to COLO mode for a fixed amount of time $\overline{b}$ (mode q_2), and we measure with a moving average mechanism the average pause ratio ρ. After $\overline{b}$ checkpoints, ρ is considered to be a good estimate of the actual average interarrival time, and the system switches to mode q_3, where ρ is still computed on the basis of the measured frequency of checkpoints and pause time. At the same time a variable σ decreases over time.

B. Outer layer: External controller

The outer layer is in charge of adapting the value of the threshold $\overline{\rho}$ on the basis of the actual behavior of the system. While the inner layer is mainly enforcing a transition from one mode to the other, an outer controller is used to decide how fast this should happen. In particular, the value of $\overline{\rho}$ affects the likelihood to stay in Checkpoint or COLO mode.

In order to compute dynamically the value of $\overline{\rho}$ we use a saturated PI controller. In particular, the controller is implemented as shown in Algorithm 1, where K and T_i are parameters of the controller, and T_s is the time elapsed from the last measurement. The controller measures the checkpoints per second $cps(k)$, and, on the basis of this value, it decides what is a good value for the Downtime ratio threshold $\overline{\rho}(k)$. More specifically, $cps(k)$ is compared with the setpoint $sp(k)$, and, on the basis

Algorithm 1 Computation of the saturated PI controller.

function COMPUTECONTROL(sp,ρ)
- $\Delta sp \leftarrow sp - sp_{old}$
- $\Delta\rho \leftarrow \rho - \rho_{old}$
- $\Delta P \leftarrow K \cdot (\Delta sp - \Delta\rho)$ ▷ Computing proportional
- $\Delta I \leftarrow K \cdot T_s/T_i \cdot (sp - \rho)$ ▷ Computing integral
- $\Delta\overline{\rho} \leftarrow \Delta P + \Delta I$ ▷ Computing control action
- $\overline{\rho} \leftarrow \overline{\rho}_{old} + \Delta\overline{\rho}$
- **if** $\overline{\rho} > \overline{\rho}_{max}$ **then** ▷ Saturation
  - $\overline{\rho} \leftarrow \overline{\rho}_{max}$
- **if** $\overline{\rho} < \overline{\rho}_{min}$ **then** ▷ Saturation
  - $\overline{\rho} \leftarrow \overline{\rho}_{min}$
- $\rho_{old} \leftarrow \rho$ ▷ Storing state variables
- $\overline{\rho}_{old} \leftarrow \overline{\rho}$
- **return** $\overline{\rho}$

of the PI control strategy, $\overline{\rho}$ is dynamically adapted, so that the error between the desired amount of checkpoints per second $sp(k)$ and the measured $cps(k)$ decreases over time. The value for the setpoint is chosen empirically to be $sp(k) = 1.5$.

The values $K = 0.01$ and $T_i = 3$ were selected and used in all the following experiments. The lack of a dynamical model hinders the possibility of a model-based tuning of the PI controller, therefore we tuned the controller, with a grid of values for the parameters $K \in [10^{-3}, 5]$ and $T_i \in [10^{-3}, 5]$, and optimizing the obtained performance with respect to the experimental results obtained with the BugZilla workload considered in this paper (see the next section).

V. EVALUATION

In this section, we evaluate the performance, resource usage and downtime ratio for different applications with three different characteristics. We compare our results against running the different applications with four different fault tolerant mechanisms, explained in the following sub-sections.

A. System Environment

For evaluating the proposed mechanism, we built a fault tolerant environment with the same characteristics as explained in Section III with one OpenStack controller and two workers. We configured three different VMs with three applications, described in the following, all with the fault tolerance option enabled. This option invokes the SVM on a different server, while the PVM is put in paused mode. It allows the SVM to be ready to start simultaneously with the PVM in a consistent state. The hosts are assumed to fail in a non-byzantine manner - i.e. they either fail or they keep on working perfectly. Cyclic Redundancy Check (CRC) hardware systems must be used to guard against memory/bus/cache corruptions, and such are present in most modern server systems.

Our tests and measurements were performed on a testbed formed of 3 servers, each with 24 Intel Xeon (E5-2620 v2 @ 2.10GHz - Core i7) cores, 64 GB of memory, managed by OpenStack (Juno version): one server for generating client web requests, and the other two servers to host the PVM and SVM, respectively. All hosts are interconnected through a 1 Gbps Network (for client requests), but the checkpoint stream between the PVM and SVM is carried over a 56 Gbps Infiniband connection (using a Mellanox ConnectX-3 card).

All tests were done using a modified OpenStack VM Flavor to enable the guest fault tolerance support - with 4 vCPUs - using the same version of our modified hypervisor (QEMU 2.5.50). One OpenStack (Nova) controller is used to submit network requests for both PVM and SVM, and two workers for package comparison (See Figure 2). All codes for the OpenStack, LibVirt and QEMU modifications are available through the ORBIT (Business Continuity as a Service) Project [17] at a GitHub repository [1].

B. Experiment Overview

We executed 3 different applications (described in the next sub-section) in four configurations with fault tolerance mechanisms: forced checkpoint mode, COLO mode and the two herein proposed Hybrid modes.

1) Checkpoint Mode: When a VM is set into checkpoint mode, a synchronization happens at every 250ms regardless of what is going on inside the PVM. The 250 number was used as a (empirical) tradeoff between application latency and performance overheads due to the time to migrate the PVM's state to the SVM. Depending on the intensity of the workload, a higher number would change more of PVM's (memory)state, negatively affecting the time to complete state synchronizations between replicas.

2) COLO: In COLO, a synchronization occurs only when a miscompare is detected between the PVM and SVM (Figure 1(b)). COLO can provide better latency performance than traditional checkpointing by lengthening the checkpoint period if the behavior of the two VMs are consistent as seen from the network traffic they generate.

3) Hybrid Modes: We include two different methods for evaluating our proposed Hybrid approach: a Threshold and a Controlled mechanism. Both methods switches between the two aforementioned fault tolerant methods - checkpoint and COLO -, but they differ on when and for how long a mode should be used and probed. Additionally, both hybrid methods were designed to increase PVM's uptime.

Threshold. It is a threshold-based hybrid heuristic that combines checkpoint and COLO modes. The threshold method rationale is as follows. If the moving average of the time between two checkpoints falls below a configurable minimum limit (set to 400ms), the system starts using checkpoint (passive) mode. The 400ms limit was empirically set for switching modes as many applications would timeout in case checkpoints were to be performed more frequently than so. This limit indicates to the SVM that, once crossed, the checkpoint mode is to be used. On reception of this signal, the SVM receives the checkpoint as normal, but does not start to execute the application. The primary triggers a

[1] http://www.orbitproject.eu/portfolio/github/

new passive checkpoint after a small, configurable time (set to 250ms). The system next stays in checkpoint mode for 100 checkpoints, and then retries with COLO mode again. If the calculated average downtime is lower than 400ms, the Threshold approach keeps in COLO mode. Otherwise, it switches back to checkpoint mode, and repeats this cycle until the application completes execution. We note that the moving average downtime used by the Threshold mode to switch mechanisms (COLO or checkpoint) is calculated as the average of total downtime since the last 10 seconds. This is weighted so that a few short checkpoints do not trigger a transition, and a previous long history of long checkpoints does not inhibit a transition when the workload behavior changes.

Controlled. As explained in Section IV, the controlled hybrid mode is configured to dynamically adapt the downtime ratio threshold at run-time depending on the frequency of checkpoints per second and the moving average of the Uptime Ratio from the last 10 seconds. Every time the moving average for the downtime ratio is greater than the controlled ratio, the mode is switched from COLO to Checkpoint. It stays in Checkpoint mode for a minimum of $\overline{b}$ ($= 25$) checkpoints, before switching to COLO. If COLO is still worse, it switches back to Checkpoint and updates the likelihood σ of switching back to Checkpoint (or to COLO). Overall, the adaptation strategy tries to adapt to the current workload by increasing or decreasing the switching likelihood σ, and thus changing the time budget to stay in checkpoint.

The time horizon of the moving average was set to 10 seconds because QEMU performs a checkpoint after 10 seconds in case no miscompare is detected while running in COLO mode. This is made to avoid large downtimes due to large VM (memory) state changes.

C. Applications

Three applications were used in order to evaluate the proposed hybrid checkpointing approach: the RUBiS online auction benchmark, the BugZilla Tracking System, and a video streaming application.

1) RUBiS online auction benchmark: RUBiS (Rice University Bidding System) is an auction site prototype modeled after eBay.com that is used to evaluate application design pattern and server performance scalability [4], [22]. The number of requests for this workload follows a rectangular step function. It is initiated with a load of 5 parallel web requests per second for 5 minutes, suddenly increasing it to 50 parallel requests per second for another 5 minutes. This pattern is repeated three times and was chosen to illustrate a variable high and low workload.

2) BugZilla: The BugZilla bug tracking system [19] is used here as a benchmark to represent a multi-threaded application with random behavior, and a common use-case in data centers. For this test, a BugZilla server was installed in a VM and populated with random bugs and a set of users. An external test harness (running on a 3rd host) interacts with the BugZilla database server via its API using the Python BugZilla package.

The VM was configured to reduce randomness, thus increasing the chance of benefiting from COLO; in particular Perl's hash randomization was disabled. For this test, a multi-threaded BugZilla `create` method was used, which spawns a variable number (up to 16) of parallel `create new bug` invocations at every second, making it very nondeterministic and thus not suitable to any static checkpointing fault-tolerant mode selection.

3) Video Streaming: For this workload, we used VLC (VideoLan [5]) in order to do two concurrent things. The first is to encode a 13 minutes video with a high resolution: A H264 video and resolution of 1920x1080 pixels with a ratio of 30 frames per second (fps). The second is to stream (in loop-back) its content over an UDP connection. This workload has high randomness due to high CPU and thread usage. This workload should thus result in a high number of miscompares and the use of checkpoint mode should generally be beneficial.

D. Measurements

For all experiments, we collected the following metrics: *downtime* (percentage of how long the VM was paused during the whole execution), *checkpoints per second* (percentage of how long the VM was frozen due to a checkpoint in each second), SVM *CPU* and *Network Usage*, *average response time* (frames per second (fps) for the Video Streaming workload), *throughput* (total number of requests, bugs created, and elapsed-time in seconds for the RUBiS, BugZilla and Video Streaming workloads respectively). The average downtime ratio was calculated based on the averages downtimes and uptimes. Each test was performed for 30 minutes and repeated for 10 times. The times presented are based on the hardware wallclock measured by QEMU. This decreases the possibility of inaccuracies in the guest clock due to checkpointing.

E. Results

Table I shows the averages for all the experiments. Unless shown, the standard deviation is lower than 0.1% from the average shown. We highlighted in bold the best performance obtained with the considered methods, for each metric and for each application. We also report the standard deviation for the average downtime, since it is the main metric that the herein proposed algorithm aims to optimize.

Figure 6 shows the evolution over time of some of the most relevant application metrics considered in our evaluation: CPU, bandwidth utilization, and downtime ratio. In comparison, Figure 7 shows the evolution of response time for the three workloads.

Figure 6 shows that the hybrid approach adapts its behavior to the current workload. For example, in the Video Streaming case, it can be noticed in the CPU Usage column that it adapts over time using increasingly longer intervals in COLO mode between spikes. This learning gives an improvement in the downtime over the Threshold and Checkpoint approaches. The Threshold approach stays in checkpoint mode for a fixed

TABLE I: Experiment Results showing averages for 120 experiments, 10 for each application in each mode. Bold columns indicate the best value for each metric.
* Throughput is the total number of requests, bugs created and elapsed-time (in seconds) for RUBiS, BugZilla and Video Streaming workloads respectively.

Application	Mode	Average Downtime (%)	Average Downtime Ratio (%)	Average Checkpoint Time (ms)	Average SVM CPU Utilization (%/s)	Average SVM Network Usage (MB/s)	Average Response Time (ms/s)	Average Throughput*
RUBiS	Checkpoint	7.6 ± 0.7	8.3	**118**	**4.1**	34	337	45055
	COLO	50.5 ± 0.8	10.2	823	10.8	54.8	**163**	38008
	Threshold	8.0 ± 0.7	8.7	131	4.3	34.5	230	**46532**
	Hybrid	**7.3 ± 0.6**	**8.2**	122	4.8	**33.5**	244	46319
BugZilla	Checkpoint	39.1 ± 5	64.03 ± 13	**634**	**0.3**	312	4460	3289
	COLO	42.1 ± 3	73.52 ± 10	737	20.3	**251**	3870	3505
	Threshold	40.2 ± 6	68.93 ± 17	682	16.4	281	4478	2626
	Hybrid	**37.1 ± 5**	**61.16 ± 11**	647	10.7	289	**3720**	**3882**
Video Streaming	Checkpoint	42.2 ± 4.6	64.6	636	**4.1**	264	15.85	**907**
	COLO	45.4 ± 4.2	83.3	749	19.8	**251**	7.46	408
	Threshold	40.0 ± 5	61.5	615	5.6	271	15.72	885
	Hybrid	**39.0 ± 4.6**	**56.3**	**581**	4.5	278	**15.90**	906

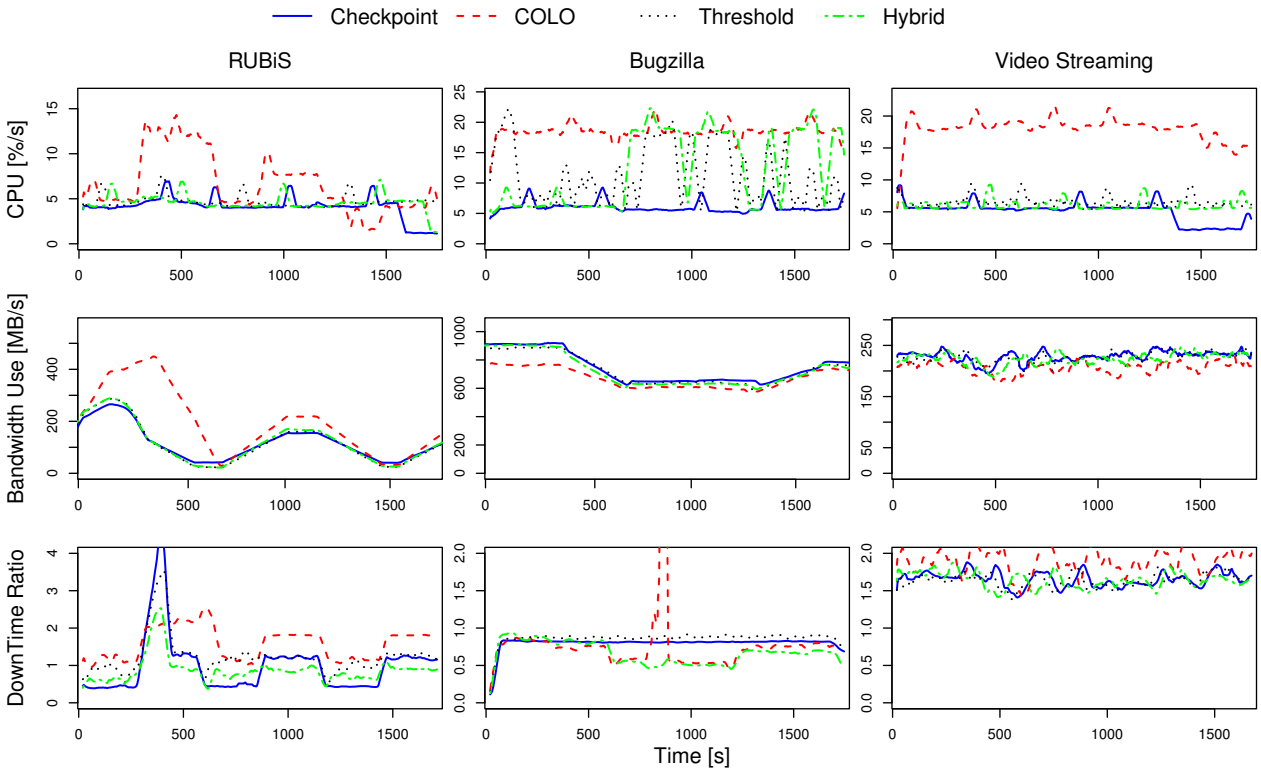

Fig. 6: Results for three example experiments, detailing over for the duration of the experiment the average measurements of three main metrics: SVM CPU, bandwidth, and downtime ratio.

amount of time, and then trying COLO once again, i.e., it did not learn COLO was not good.

The BugZilla Workload is the most complex workload to adapt. The high and variable number of threads increase the out-of-order instructions in parallel, increasing the number of comparisons resulting in a checkpoint triggering event. Given the CPUs assigned to the guest, a maximum of 16 threads with the `create_new_bug_test()` method, all running in similar times without COLO and only increase as the load increases past the number of CPUs available. In COLO mode the performance degrades as the amount of randomness increases and the size of each checkpoint increases. This can be seen from a comparison of the bandwidth and downtime. The hybrid approach identified some better scenarios for COLO

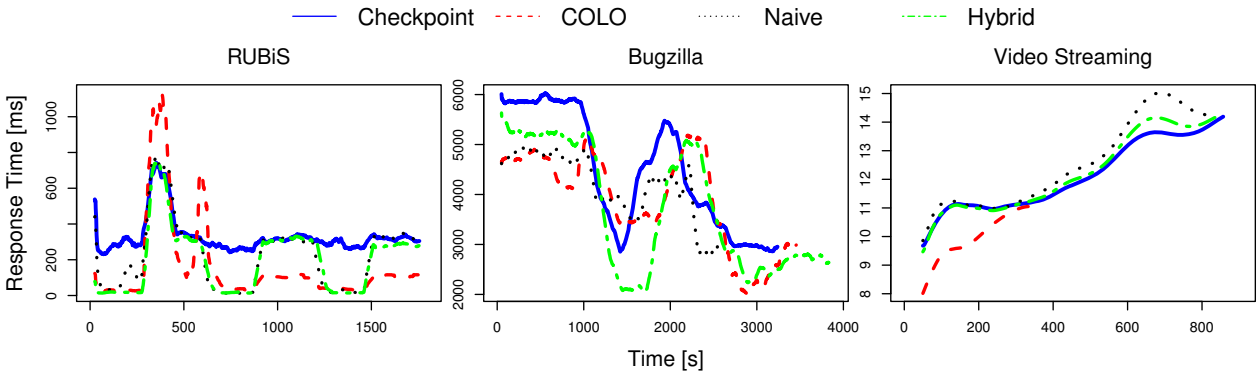

Fig. 7: Results for three example experiments, detailing over time the average measurements for response time for RUBiS and Video Streaming as well as throughput for Bugzilla.

during execution and thus was able to switch modes, turning the workload into a less resource hungry one and with better performance (throughput). Overall, the hybrid mode had the lowest average downtime ratio, checkpoint had low downtime, though many pauses, and used the least CPU (due to its characteristics). The throughput was very similar between all methods.

VI. Discussion

The experimental results highlight the advantages of the proposed hybrid mechanism. By rapidly adapting to current workload, the proposed approach improves the application run-time over the other methods by at least 10%, with negligible performance overheads in CPU and network bandwidth. The comparisons per second and the downtime ratio metrics seem to provide a good indication how to select checkpointing mode depending on the workload.

Moreover, we can also conclude that the threshold approach cannot outperform checkpoint mode, because when the system was set in Checkpoint and often tried COLO, the moving average of the time between checkpoints was still lower than the threshold (400ms) due to randomness in the workload. This pushed the threshold solution to switch to checkpoint mode again. The hybrid approach, conversely, adapts the threshold based on the actual behavior of the system, avoiding unnecessary switches. The hybrid approach is designed to try COLO sporadically, so to understand if COLO mode can achieve better performance.

In fact, the dynamic switching to checkpoint mode provides a means for users of a fault-tolerant system to select the operation mode appropriate to their applications without having to investigate the behavior of the workload in detail. This phenomenon is particularly relevant when considering the BugZilla application: Workloads that are primarily CPU bound get the full advantage of COLO, whereas those with random network traffic get a more traditional checkpointing system, that uses little additional CPU on the secondary host.

The adaptation abilities of the proposed approach can be relevant in many applications, as it is fully rare to find very predictable behavior in cloud applications. Also, by checking the average response times and throughput, identify a few noticeable tradeoffs: the Hybrid mechanism has not underperformed in any of the metrics for performance, and only by small overheads for CPU and Network usages. In particular, unpredictable behavior can be caused by many different factors. Examples of some factors that we managed to identify while conducting the experimental evaluation include:

- **Timestamping**: many applications include a timestamp in the output message or as part of an internal ID. The higher precision of time used, the higher the chances are that the timestamp timestamp by the other hosts differ, thus causing a checkpoint.
- **Unique IDs**: Applications that allocate unique IDs to requests based on a sequence of incoming requests incorporate randomness due to interactions between different clients.
- **Hashes**: Some applications that produce apparently consistent output may internally use a hashed list whose output order is non-deterministic. One example that we observed was HTTP headers from BugZilla; the headers were always the same, but their order varied between requests.
- **Intentional** – but non-obvious – **randomness**: e.g., web based applications that appear to give consistent output may hide variability in the HTML or Javascript delivered to the client.

Whenever network data is non-deterministic the advantage of COLO is reduced, depending on how much variability there is in the data; e.g. a workload that is compute-intensive but produces a non-deterministic output every 3 seconds, does still benefit from a reduced checkpoint frequency compared to a simple checkpointing mode. The reality for many applications is that they contain a mix of different behaviors, and it is difficult for a user or administrator to predict whether their application will benefit from COLO.

A. Application Performance

Simultaneously and efficiently managing a data-center from a dual perspective - operators who seek to save costs and improve resources' utilization, and users, who want their service to run at full performance and to be highly available - is a hard problem. Particularly, if performance is taken into account and one does not know how workload and application resource profiles look.

COLO aims at improving application availability for improving responsiveness. Based on this, COLO aims to decrease the amount of checkpoints an interactive application triggers at the expense of doubling usage of CPU and Memory, but generally lowering network usage on deterministic workloads. So does our controller, which targets optimizing application availability in data-centers by using a active replication in a second site. Notable, COLO does not aim to improve application performance and neither does our hybrid approach. However, as Table I shows, our approach performed with neglected performance losses.

Table I also shows that the performance of an application is not directly correlated with its increased availability, as can be seen in the Video-Streaming experiment: the throughput for Checkpoint mode, which stopped the PVM at every 250ms was higher than for COLO, which performed very poorly. This experiment shows that checkpoint mode is to be used: periodically checkpointing the PVM results in better performance (Figure 7) than dynamically checkpointing it, as checkpoint mode reduces synchronization frequency and improves frame continuity in the Video-Streaming experiment. As the video streaming workload is very unpredictable - common scenario in data-centers - checkpoint mode performed better. Our proposed controller dynamically learned that Checkpoint was a better mode for such workload.

This is why our proposed hybrid mechanism fits very well these applications, though other tradeoffs related to performance metrics like CPU Usage and memory footprint could also extend the controller model. There is potential for further performance improvements in terms of CPU usage in the passive checkpoint mode, by a leaner implementation when compared to the more complex execution paths required for COLO. For example, in checkpoint mode (A/P) the SVM does not write to memory or disk during a checkpoint period. Also, note that when low-latency is the critical metric, COLO mode (A/A) behaves generally better than any other modes. This happens simply due to fast response release, lowering the latency experienced by users. This cannot happen in any of the other modes as they delay responses by choosing a mode which makes the VM to pause less frequently.

VII. Conclusion

VM replication through checkpointing is a well-known approach to achieve high availability and to enable disaster recovery. A set of mechanisms have been proposed for this, based either on periodic or on-demand checkpointing. These have different benefits and drawbacks in terms of performance (application response time and unavailability due to VM freeze during migration) and resource usage (the overhead imposed by replication and checkpointing in terms of CPU, memory usage, and/or network transfers). We combine the best of the two checkpointing schemes by proposing a hybrid approach that dynamically switches between periodic and on-demand checkpointing depending on how long the application executes in relation to how long it performs checkpoint synchronizations. A PI controller is designed to dynamically select the optimal checkpointing method over time, to maximize application availability based on the Downtime Ratio metric, that considers the performance of takes into consideration network bandwidth, CPU and memory. The proposed approach was implemented through modifications to QEMU and was integrated in OpenStack. An experimental evaluation based on three application benchmarks – RUBiS, a streaming video server, and a BugZilla server – demonstrates that the hybrid approach performs better than other checkpointing approaches by at least 10% reduction in downtime. A threshold-base mechanism (naive) was also used for the evaluation as a baseline hybrid approach. In all three use-cases, our controllers followed the best possible mode selection in regards to performance and resource utilization, simultaneously, autonomously and with no prior application knowledge. In large-scale data-center operations, this has a practical impact when dealing with fault-tolerance.

The developed mechanism can be used in infrastructures where fault tolerance is of critical importance but operators are not able to timely and dynamically select the correct mode based on the application workload, while trying to optimize different performance indicators. The benefits are particularly noticeable on long running and non-deterministic applications, where the aggregated downtime will be reduced over and over. Finally, with learning mechanisms, like Statistical and Machine Learning techniques, workload pattern variations can be matched with resource utilizations and the switch operation between modes can be optimized for various applications.

Acknowlegments

We thank the anonymous reviewers whose constructive feedback improved the quality of this work. The research leading to the results presented in this paper has received funding from the European Union's seventh framework programme (FP7) Project ORBIT under grant agreement number 609828. In addition, this work was partially supported by the Swedish Research Council (VR) for the projects "Cloud Control" and by the Brazilian National Council for Scientific and Technological Development (CNPq) under project no. 207555/2014-1. The authors also thank Simon Kollberg and the ORBIT team for technical contributions.

References

[1] N. Ayari, D. Barbaron, L. Lefevre, and P. Primet. Fault tolerance for highly available internet services: concepts, approaches, and issues. *IEEE Comm. Surveys & Tutorials*, 10(2):34–46, 2008.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[4] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *ACM Sigplan Notices*, volume 37, pages 246–261. ACM, 2002.

[5] A. Cellerier et al. VideoLAN-VLC media player, 1998. Last accessed on 2018, http://www.videolan.org.

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.

[7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 161–174, 2008.

[8] M. P. da Silva, R. R. Obelheiro, and G. P. Koslovski. Adaptive Remus: adaptive checkpointing for Xen-based virtual machine replication. *International Journal of Parallel, Emergent and Distributed Systems*, pages 1–20, 2016.

[9] X. Defago, A. Schiper, and N. Sergent. Semi-passive replication. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 43–50. IEEE, 1998.

[10] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *4th Annual Symposium on Cloud Computing*, SOCC '13, pages 3:1–3:16, New York, NY, USA, 2013. ACM.

[11] J. Dongarra, T. Herault, and Y. Robert. Fault tolerance techniques for high-performance computing. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer, 2015.

[12] M. Hines. Qemu features - microcheckpointing, 2015. Last accessed on 2017, http://wiki.qemu.org/Features/MicroCheckpointing.

[13] R. Jhawar and V. Piuri. Fault tolerance and resilience in cloud computing environments. *Computer and Information Security Handbook,*, pages 125–141, 2013.

[14] M. Ji, A. C. Veitch, and J. Wilkes. Seneca: remote mirroring done write. In *USENIX Annual Technical Conference, General Track*, pages 253–268, 2003.

[15] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *3rd USENIX Conference on File and Storage Technologies (FAST)*, pages 59–62, 2004.

[16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[17] D. Kyriazis, V. Anagnostopoulos, A. Arcangeli, D. Gilbert, D. Kalogeras, R. Kat, C. Klein, P. Kokkinos, Y. Kuperman, J. Nider, et al. High performance fault-tolerance for clouds. In *Computers and Communication (ISCC), 2015 IEEE Symposium on*, pages 251–257. IEEE, 2015.

[18] H. Lin and P. J. Antsaklis. *Hybrid Dynamical Systems: An Introduction to Control and Verification*, volume 1 of *Foundations and Trends in Systems and Control*. Now publisher, 2014.

[19] Mozilla Foundation, Bugzilla Team. Bugzilla documentation-5.0. 2+ release, 2015. Last accessed on 2016, https://www.bugzilla.org/docs/5.0/en/html/integrating/api/.

[20] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File-system-based asynchronous mirroring for disaster recovery. In *1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[21] Red Hat. libvirt: The virtualization API, 2012. http://libvirt.org.

[22] Rice University. RUBiS: Rice University Bidding System, Last Accessed on October, 2016. http://rubis.ow2.org.

[23] O. Sefraoui, M. Aissaoui, and M. Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[24] A. Sîrbu and O. Babaoglu. Towards data-driven autonomics in data centers. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, pages 45–56, 2015.

[25] Stratus. Stratus ftserver architecture, 2008. Last accessed on 2018, http://www.stratus.com/assets/StratusUptimeAssuranceArchitectureForWindows.pdf.

[26] Y. Tamura. Kemari: Fault Tolerance VM Synchronization based on KVM. *KVM Forum 2010*, 2010.

[27] T. Thanakornworakij, R. F. Nassar, C. Leangsuksun, and M. Păun. A reliability model for cloud computing for high performance computing applications. In *Euro-Par 2012: Parallel Processing Workshops*, pages 474–483. Springer, 2012.

[28] VMWare. VMWare High Availability (HA), 2003. Last accessed on 2018, https://www.vmware.com/pdf/ha_datasheet.pdf.

[29] VMWare. VMWare vSpere 4 Fault Tolerance: Architecture and Performance, 2009. Last accessed on 2018, https://vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-vsphere-fault-tolerance-architecture-and-performance.pdf.

[30] T. Wood, H. A. Lagar-Cavilla, K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. Pipecloud: Using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 17:1–17:13, New York, NY, USA, 2011. ACM.