# Deterministic Fuzzy Checkpoints

Michael Eischer, Markus Büttner, and Tobias Distler
Friedrich-Alexander University Erlangen-Nürnberg (FAU)
Email: {eischer,markus.buettner,distler}@cs.fau.de

***Abstract*—Replicated systems tolerating arbitrary (Byzantine) faults require periodic and deterministic application-state checkpoints to perform essential tasks such as initializing new replicas, enabling faulty replicas to recover, and garbage-collecting old agreement-protocol messages. Existing techniques to create checkpoints in these systems make it necessary to temporarily suspend request execution in order to capture a consistent checkpoint, causing significant service disruptions for applications with large states. Unfortunately, state-of-the-art approaches from the domain of crash-tolerant systems also are not directly applicable, because the checkpoints they produce are not comparable across replicas and therefore cannot be validated in an environment in which replicas may fail arbitrarily and do not trust each other.**

**In this paper, we address these problems by proposing *deterministic fuzzy checkpoints (*DFC*)*, a novel technique that enables all correct replicas in a system to create consistent and matching checkpoints in parallel to processing requests. As a consequence, DFC increases service availability while still allowing replicas to verify the correctness of a checkpoint before applying it to their local states. In addition to our general approach, we present different alternatives to implement DFC within a replication library and furthermore discuss support for the creation of differential checkpoints. Experiments with a key-value store show that DFC is able to snapshot states of 3 GB while sustaining high performance throughout the entire checkpointing process.**

## I. Introduction

Byzantine fault-tolerant state-machine replication [1], [2] enables systems to keep their services available even in scenarios in which an unknown but limited subset of the participating servers may behave in arbitrary ways and therefore cannot be trusted. To offer resilience under these circumstances, in such systems decisions, for example on the result of an application request, are usually not based on the information provided by a single replica alone, but instead are made after comparing and voting on the opinions of multiple replicas. For correctness, it is consequently crucial that all non-faulty replicas in the system maintain a consistent view of the application and are able to communicate this view to others in the form of a checkpoint, that is, a snapshot reflecting the application state at the point in time at which the checkpoint was taken. Such checkpoints are an essential building block of replication protocols as they, for example, allow a protocol to update new replicas joining the system or old ones that have fallen behind [2], [3], [4], to proactively or reactively rejuvenate faulty replicas [2], [5], [6], [7], and to discard agreement-protocol messages that no longer need to be stored [2], [4], [8], [9].

The traditional approach for a replica to ensure that a checkpoint represents a consistent application state is to suspend the execution of requests while copying the state, thereby preventing intermediate modifications from introducing inconsistencies [2], [3]. However, this solution comes with the drawback of lowering service availability and increasing tail latency [10] during the checkpoint-capture phase and therefore is not suitable for periodically checkpointing applications with large states. Relying on differential checkpoints (i.e., snapshots that only contain the state changes since the previous checkpoint [2], [11]) in such cases often is an effective means to mitigate this problem, but without additional measures can still cause significant disruptions, as our evaluation shows.

Unlike Byzantine fault-tolerant systems, many crash-tolerant in-memory databases support the creation of checkpoints in parallel with the processing of transactions by implementing fuzzy checkpoints [12], [13], [14]. Using this technique, a database checkpoint consists of both (1) a possibly inconsistent state snapshot and (2) a collection of transactions that were active during and after the capturing process, and which later can be used to "repair" the fuzzy snapshot. Although efficient, this technique unfortunately cannot be directly applied to replicated systems tolerating arbitrary faults, because the resulting checkpoints are non-deterministic and therefore not comparable across replicas. Comparability, as discussed above, however, is an important requirement for systems in which replicas do not trust each other, as it enables the verification of a checkpoint based on the opinions of multiple replicas.

In this paper, we address these problems with a technique we refer to as *deterministic fuzzy checkpoints (*DFC*)*. Relying on DFC, replicas are able to produce consistent checkpoints without requiring support for database transactions and, more importantly, without the need to suspend request processing. Nevertheless, the resulting checkpoints are comparable across replicas. To achieve this, replicas not only acquire a copy of the application state but also capture intermediate modifications in a deterministic order. If necessary, for example for verifying snapshot contents, based on this information correct replicas are able to retroactively create identical checkpoints for a specific point in time; similar to state capture, this procedure can also be performed in parallel with request execution.

In summary, this paper makes the following contributions: (1) It presents the DFC technique that enables replicas to create deterministic, and therefore comparable, checkpoints in parallel to request processing. (2) It discusses two different variants to integrate DFC with a replication library. (3) It elaborates on how the basic concept of DFC can also be used for differential checkpoints. (4) It uses a key-value store to experimentally evaluate the proposed DFC variants in comparison with state-of-the-art checkpointing approaches.

This is the authors' version of an article to appear in the *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*, Lyon, France, 1–4 October 2019.

## II. SYSTEM MODEL

We consider systems that run multiple replicas of a stateful application on different servers to make the application resilient against faults [1]. If a replica becomes faulty, for example as the result of a software/hardware problem or a malicious attack, the replica may behave in arbitrary (Byzantine) ways [15]. At all times, at most $f$ of the replicas in the system are assumed to be faulty. If supported by the replication infrastructure, replicas may recover from failures, for example, by being rejuvenated using a clean state [2], [5], [6], [7].

The replicas of a system are connected through an unreliable network that might reorder, delay, corrupt, or drop messages. To enable a replica to verify the origin and integrity of a received message, all correct replicas properly authenticate the messages they send. We do not require a system to rely on a specific authentication scheme for this purpose, only that the cryptographic algorithm used is strong enough to prevent an attacker from successfully impersonating a correct replica or manipulating the content of a message without being detected.

To ensure that the application states of correct replicas remain consistent even in the presence of replica and network failures, as shown in Figure 1 the servers in a system use a replication library that executes a Byzantine fault-tolerant agreement protocol [2], [4], [8] to establish a stable total order on incoming client requests. More specifically, the agreement protocol assigns each request a unique, monotonically increasing sequence number and ensures that all correct replicas eventually commit to the same assignment. Requests for which the agreement process is complete are then processed by all correct replicas in the order of their sequence numbers. For consistency, the application logic must be deterministic, that is, starting from the same initial state and processing the same sequence of committed requests, all correct replicas must reach the same follow-up states and produce the same results.

In our target systems, the state of the application can be modeled as a collection of disjoint *objects* [11], [16], [17] that each are identified by a unique *object id* (e.g., a byte string). The set of objects a state consists of may change over the lifetime of a system, for example, due to requests creating new objects and/or deleting existing ones. For most applications partitioning the state into such objects is straightforward as the state already possesses some form of internal structure. In a key-value store, for example, each object may represent a different entry, using the entry's key as object id.

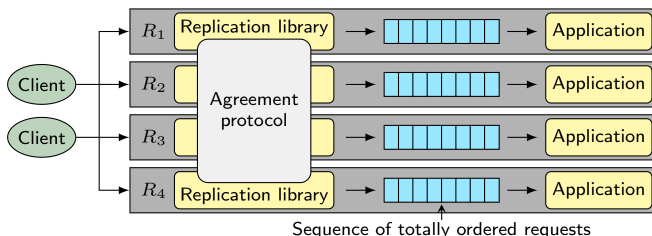



Figure 1. Overview of a system with four replicas ($R_1$,...,$R_4$) that are kept consistent by a replication library establishing a total order on all requests.

## III. BACKGROUND & PROBLEM STATEMENT

In this section, we provide background on the purpose and use of checkpoints in replicated applications that fit our system model presented in Section II. Furthermore, we discuss and analyze state-of-the-art checkpointing approaches in order to identify their individual strengths and weaknesses. In a final step, these insights then allow us to formulate requirements for the design of improved checkpointing mechanisms.

### A. Checkpoints

A checkpoint $C_s$ is a representation of the state a replicated application is in after having processed all requests up to (and including) sequence number $s$. With correct replicas being deterministic and executing all requests in the same order, the checkpoints created by any two correct replicas $i$ and $j$ for the same sequence number are equal (i.e., $C_{s,i} = C_{s,j}$). This means that the checkpoints include the same state objects $O_1, ..., O_x$ and that the values of these objects are pairwise equal (i.e., $\forall o \in \{1, ..., x\} : O_{o,i} = O_{o,j}$).

Checkpoints constitute an essential building block of fault-tolerant replicated systems and are typically used to solve a variety of problems. First and foremost, they often play an important role in state-transfer mechanisms that allow new replicas to join an already running system. Specifically, by fetching a checkpoint and accordingly updating its local application, a new replica is able to reach a consistent state without having to process all requests since system start. In a similar way, checkpoints also provide a means for recovering replicas to catch up [2], [5], [6], [7]. Apart from these use cases, many replication architectures also rely on checkpoints to determine the point in time at which they can safely garbage-collect information on completed agreement-protocol instances [2], [4], [8], [9]. The rationale in this context is that once a checkpoint covers the effects a request had on the application state, the agreement-protocol information of the request is no longer required and therefore can be discarded.

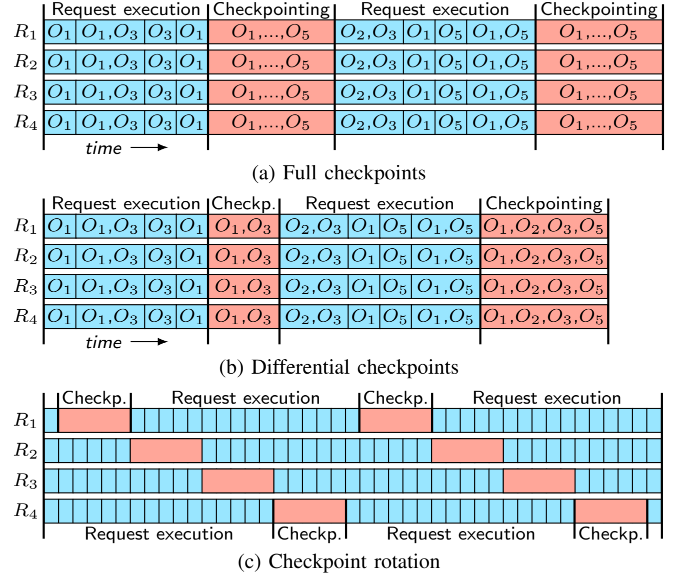
In environments where replicas are assumed to possibly fail in arbitrary ways and therefore do not trust each other, a correct replica must never update its state solely based on a checkpoint provided by a single other replica. Otherwise, if the other replica is faulty it might transmit a corrupted checkpoint that does not represent the actual application state at the specified sequence number, which in turn would result in the correct replica becoming inconsistent. To address this issue, Byzantine fault-tolerant agreement protocols usually enable a replica to verify the content of a received checkpoint based on information gathered from multiple replicas [2], [8]. For this purpose, these protocols ensure that all correct replicas periodically checkpoint their local states at predefined sequence numbers (e.g., in intervals of 100,000). Having created a checkpoint, each correct replica then computes a hash over the checkpoint's content, which another replica can later use to verify the checkpoint by comparing the hashes obtained from multiple replicas. In the presence of at most $f$ faulty replicas, a checkpoint is correct if there are at least $f + 1$ matching hashes from different replicas confirming its content.

## B. State of the Art

The straightforward approach to guarantee that a checkpoint correctly reflects the current state of the application is to suspend request processing while the checkpoint is created, as illustrated in Figure 2a. Using this method, when a checkpoint $C_s$ is due, a replica first waits until the execution of the request with sequence number $s$ is complete. Next, it copies all state objects to a separate memory location reserved for the checkpoint. Finally, the replica resumes request execution at sequence number $s+1$. With no request being processed during checkpoint creation, all objects remain unchanged and consequently the checkpoint reflects the application state between sequence numbers $s$ and $s+1$. On the downside, with all objects being copied this approach often incurs a significant performance penalty, especially for large application states.

One possibility to mitigate the performance penalty associated with checkpointing is to create hybrid checkpoints [3]. Using this technique, an application-state snapshot is only taken in comparably large sequence-number intervals. In between, the captured snapshot is subsequently extended by frequently adding deltas, that is, sequences of the client requests that a replica has executed in the meantime. With the replication library already having these requests available, obtaining such deltas is straightforward. However, this comes at the cost of an increased overhead for applying hybrid checkpoints, because in addition to loading the (outdated) snapshot into the application a replica then also needs to process all the requests that are contained in the succeeding deltas. In general, the approach of bringing an application state up to date via request execution has two drawbacks: (1) Processing a request usually requires more resources than reproducing only the changes the request made to the application state [7]. (2) While verifying a snapshot allows a replica to confirm that certain state parts are correct, successfully verifying a delta solely tells the replica that the inputs it is going to process are valid. However, the same does not hold for the effects the requests have on the application. If, for example, the execution of a request causes a replica to fail and the replica then tries to recover using a hybrid checkpoint containing the same request in its deltas, the request will result in the replica failing again.

An alternative way to speed up checkpointing is to rely on differential checkpoints [2], [11]. In contrast to regular (full) checkpoints, differential checkpoints do not capture the entire application state but only comprise information on the state objects that have been created, modified, or deleted since the last full checkpoint (see Figure 2b). The information in the differential checkpoint is afterwards used to update the previous full checkpoint in parallel with request execution to create an up-to-date full checkpoint. As a main benefit of the differential-checkpoint approach, the duration of the checkpoint-capturing process no longer depends on the overall size of the application state, but instead on the number and size of the objects that changed during the latest checkpoint interval, which in many use cases is significantly smaller. However, as we show in Section VI, for applications with large objects and/or expensive state-retrieval operations differential




(a) Full checkpoints

(b) Differential checkpoints

(c) Checkpoint rotation

Figure 2. Comparison of state-of-the-art checkpointing techniques for an example system with four replicas ($R_1$,...,$R_4$) and an application state comprising five objects ($O_1$,...,$O_5$). Request and checkpoint labels indicate the state object(s) accessed by the corresponding operation.

checkpointing may still lead to considerable service outages, a problem that in most Byzantine fault-tolerant systems is intensified by the fact that all correct replicas create checkpoints at the same sequence numbers (see Section III-A).

To improve system availability during checkpointing procedures, Dura-SMaRt [18] applies a strategy in which different replicas checkpoint their local states at different sequence numbers (see Figure 2c). As a result, the individual checkpoints are not directly comparable, making it impossible for the receiver of a checkpoint (e.g., a new replica) to directly verify the correctness of the checkpoint's content in advance. For this reason, Dura-SMaRt relies on a different approach which in a nutshell involves the following steps: Having received a checkpoint for sequence number $s$, a replica immediately updates its local state accordingly, without performing any checks. In addition to the checkpoint, the replica also obtains and executes all subsequent committed requests with sequence numbers higher than $s$, thereby further updating its state. During this process, every time the replica reaches a sequence number $t > s$ for which another replica has previously created a checkpoint $C_t$, the replica retroactively compares its state at sequence number $t$ to the state announced in checkpoint $C_t$. If at some point the replica fails to verify the correctness of its local state, it retries the state transfer based on a checkpoint and/or requests provided by a different replica. Although this approach ensures that correct replicas eventually will end up with a correct up-to-date state, the method also has one important drawback. It requires a replica to load an unverified checkpoint and therefore entails the risk of a correct replica being compromised by a manipulated checkpoint. That is, if an attacker, for example, manages to introduce a virus via the provided checkpoint, the replica might become faulty even before it is able to detect that the checkpoint has been invalid.

## C. Requirements

Having analyzed the advantages and disadvantages of existing solutions, we identify a set of requirements a resilient, efficient, and flexible checkpointing mechanism should fulfill:

- **Resilience:** In order to prevent a correct replica from becoming faulty as the result of a state transfer with a corrupted checkpoint, a replica must be able to validate the content of a checkpoint prior to applying it locally.
- **Efficiency:** The procedure of creating a checkpoint should impact application performance as little as possible. In the ideal case, request execution does not have to be suspended at all while the checkpoint is captured.
- **Flexibility:** The checkpointing mechanism should support both full and differential checkpoints, this way offering the possibility to exploit the advantages of each technique (i.e., no need to continuously track modifications for full checkpoints vs. efficient creation of differential checkpoints) depending on the application scenario.

In the remainder of this paper we present our approach to meet these requirements, which enables replicas to produce consistent checkpoints in parallel with request execution. Providing comparable checkpoints that represent the application state at deterministic sequence numbers, it can be applied to systems in which the replicas do not trust each other. We first discuss the base version of our approach that supports full checkpoints (Section IV) and then elaborate on how this version can be extended to differential checkpoints (Section V).

# IV. Deterministic Fuzzy Checkpoints

In this section, we provide details on *deterministic fuzzy checkpoints* (DFC), a technique for the efficient creation of consistent checkpoints in systems that assume arbitrary replica failures. Apart from outlining our general approach (Section IV-A), we describe two possible alternatives to handle the necessary interaction with the application (Section IV-B), and furthermore highlight important optimizations (Section IV-C).

## A. General Approach

As discussed in Section III-C, a resilient checkpointing mechanism allows replicas to verify the content of a checkpoint without having to load the checkpoint first. DFC achieves this by ensuring that all correct replicas in a system create consistent and comparable checkpoints in deterministic sequence-number intervals. To do this with minimal impact on service availability, DFC performs state capturing in parallel with request execution, as illustrated in Figure 3. With the application continuously processing client requests during this procedure, in contrast to traditional checkpointing approaches a captured DFC snapshot is fuzzy, that is, in itself the snapshot usually is not a consistent representation of the application state at a specific point in time. Therefore, to make checkpoint contents comparable across replicas, DFC includes a second phase in which the captured snapshot is later adjusted to fit to a predefined sequence number. In the following, we present each of the two DFC checkpointing steps (i.e., state capture and checkpoint completion) in more detail.

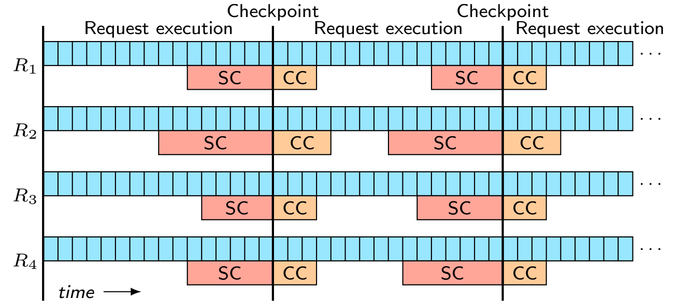



Figure 3. Deterministic fuzzy checkpointing with four replicas ($R_1$,...,$R_4$): Both state capture (SC) and checkpoint completion (CC) are coordinated by each replica individually and performed in parallel to request execution.

***State Capture.*** To create a checkpoint $C_s$ for a sequence number $s$ using DFC, a replica $i$ starts to capture the application state at an earlier sequence number $p_i < s$. As further discussed below, the sequence number $p_i$ is selected by each replica individually without any coordination with other replicas in the system. Once the replica has processed the request with sequence number $p_i$, it starts a dedicated checkpointer thread that, for a full checkpoint, iterates over the entire application state and creates a copy of each state object. In addition, starting with sequence number $p_i + 1$ the replica also records all subsequent modifications to the application state up to and including sequence number $s$. The particular method used to identify and store these modifications, as well as the properties they need to provide, depend on the interface between replication library and application (see Section IV-B). However, in all cases, when the state-capturing process is complete the checkpoint $C_{s,i} = (S_{[p_i,s]}, M_{p_i+1,\ldots,s})$ consists of two parts: (1) a snapshot $S_{[p_i,s]}$ comprising a set of state-object copies that each represent the state of their respective object at some point between sequence numbers $p_i$ and $s$ as well as (2) a list $M_{p_i+1,\ldots,s}$ of all state modifications by requests with sequence numbers between $p_i + 1$ and $s$.

The goal of a replica is to select the starting sequence number $p_i$ in such a way that the checkpointer thread finishes its work shortly before the application reaches the sequence number $s$ for which the checkpoint should be created. If the checkpointer thread takes longer than expected, the replica must temporarily suspend request execution after sequence number $s$ in order to ensure that no effects of later requests are introduced into the checkpoint for $s$. On the other hand, if the starting sequence number is chosen too low and the checkpointer thread terminates early, the list of state modifications may grow unnecessarily large. To address this problem, a replica using DFC dynamically determines the starting sequence number for a checkpoint interval based on measurements conducted in the previous interval. Specifically, a replica counts the number of sequence numbers $d$ the application has actually processed while the checkpointer thread was running. For the next checkpoint interval ending with sequence number $s'$, the replica then selects $p'_i = s' - min(\lambda \cdot d + \delta, I_{CP})$ with $\delta$ and $\lambda$ being configurable numbers serving as an additional buffer and $I_{CP}$ representing the global checkpoint interval.

Selecting the starting sequence number for state capture this way has several benefits: First, unlike the use of static starting points, the adaptation allows a replica to minimize overhead in the presence of varying application-state sizes. Second, the formula on the one hand ensures that the size of the state-capture range can be adjusted in both directions, but on the other hand still prevents two checkpoint intervals from overlapping. Finally, with each replica selecting its starting sequence number individually, DFC is able to handle scenarios in which the costs for state capture are not uniform across replicas, for example, as a result of the system comprising heterogeneous servers with different performance capabilities [9].

***Checkpoint Completion.*** As replicas perform the state capturing for a DFC checkpoint $C = (S, M)$ in parallel with request execution, the contents of the state-object snapshots $S$ created by different replicas can differ. This has two reasons: (1) Different replicas are allowed to choose different starting sequence numbers for the state-capturing process and (2) the interleaving between application and checkpointer threads is likely to vary across replicas, resulting in some replicas capturing earlier states of an object than others in cases where the object is modified after the capturing started. The former reason generally causes the list of state modifications $M$ to differ in length between individual replicas. However, with all correct replicas respecting the agreement protocol's total order of requests, DFC guarantees that the list $M_i$ of a replica $i$ is a suffix of the list $M_j$ captured by a replica $j$ if $p_i > p_j$ is true for their respective starting sequence numbers.

Although the immediate results of the state-capture phase may not be directly comparable across correct replicas, they nevertheless already contain all information necessary for each of these replicas to produce a consistent checkpoint. For this purpose, a replica takes the state-objects snapshot $S$ and applies all intermediate changes in the order in which they appear in the list of modifications $M$. Due to the way both the snapshot and the state-modification list have been constructed, the output of this checkpoint-completion procedure is identical for all correct replicas. Consequently, the resulting checkpoints can be used in the same manner as traditional full checkpoints even though the DFC checkpoints have been created while the application was continuously running. As another benefit, the checkpoint completion can also be performed in parallel to request execution. Furthermore, it may be deferred to a point in time when another replica actually needs the checkpoint, for example, to join the system or recover after a failure.

### *B. Interaction with the Application*

DFC efficiently creates consistent checkpoints based on a combination of state-object snapshots and state modifications. In the following, we discuss two alternatives for retrieving this information from the application through generic interfaces that enable some parts of the processing to be implemented in an application-agnostic way inside the replication library. Our first variant DFC$_{\text{caw}}$ applies copy-after-write and requires only a small amount of application-specific functionality, while our second variant DFC$_{\text{upd}}$ is based on updates provided by the application and therefore offers additional means to improve checkpointing efficiency with a tailored implementation.

```
1  /* Application interface */
2  interface CAW_Application {
3    /* Request execution */
4    RESULT invoke(REQUEST r);
5
6    /* Checkpointing */
7    BYTE[] object(OBJECTID oid);
8    void apply(OBJECTID[] oids, BYTE[][] objects);
9  }
10
11 /* Replication-library callback interface */
12 interface CAW_Callback {
13   void modified(OBJECTID oid);
14 }
```

Figure 4. Interfaces between replication library and application if deterministic fuzzy checkpoints are implemented based on copy-after-write (DFC$_{\text{caw}}$).

***Variant I: Copy after Write (DFC$_{caw}$).*** The main idea behind DFC$_{\text{caw}}$ is to enable the replication library to track each modification to a state object, thereby making it possible for the library to coordinate the checkpoint creation process itself. In particular, this approach allows the library to learn which object changes while the state capture is in progress and later specifically request these objects to be copied again.

As shown in Figure 4, DFC$_{\text{caw}}$ uses an application interface similar to the ones that can already be found in existing Byzantine fault-tolerant systems [11]. To process a request, the replication library hands it to the application via `invoke()` and obtains a result once the execution finished. In order to snapshot an object, the library calls `object()` passing the corresponding object id. This method either returns a copy of the requested object in an application-specific serialized form or nil in case the object currently does not exist. Finally, the application offers means to apply a checkpoint after it has been successfully verified (`apply()`). In addition to these methods provided by the application, the replication library comprises a callback method `modified()` that is invoked by the application before it changes the state of an object.

Using these interfaces, DFC$_{\text{caw}}$ implements deterministic fuzzy checkpoints as follows. At all times, the replication library maintains a set of ids referring to the state objects that currently exist in the application. The library adds new ids when it learns them via `modified()` and removes old ids if `object()` returns nil. When request execution reaches the starting point of the state-capture phase, the library starts a checkpointer thread that iterates over the application state and for each known id copies the corresponding state object using `object()`. At the same time, the library begins to record the ids of newly modified objects in a separate set. As soon as the checkpointer thread has produced the state-objects snapshots $S$ and the application has processed all requests up to the checkpoint sequence number, the replication library triggers the creation of the final state-modification list $M$. For this purpose, the library in a last step temporarily pauses request execution and once again calls `object()` for all objects that have been modified in the meantime.

At the end of the state-capture phase a full $\text{DFC}_{\text{caw}}$ checkpoint thus has the following properties: (1) For all objects that have not been modified during state capture, the set of object snapshots $S$ contains a copy of their current state. (2) For all other objects, $S$ may either include the newest version or an earlier one; however, the modifications list $M$ is guaranteed to include the latest state for these objects. Therefore, creating a deterministic full $\text{DFC}_{\text{caw}}$ checkpoint is straightforward, all a replication library needs to do is to replace the copies of modified state objects in $S$ with their latest version from $M$.

The copy of an object must reflect the object's state between the execution of two requests. Usually the most efficient way to achieve this is to implement object-level locking directly in the application. However, the replication library can also implement a coarse-grained variant by simply alternating between normal request execution and object-state capturing.

Although relying on a similar interface between replication library and application, DFC's copy-after-write significantly differs from the copy-on-write (COW) approach used by existing replication libraries [3], [19] to create full checkpoints. Most importantly, while COW creates a checkpoint for the sequence number marking the start of the state-capture phase, $\text{DFC}_{\text{caw}}$ produces a checkpoint for the sequence number at which state capturing ends. Consequently, if an object that so far has not yet been captured is about to change, COW must delay request execution and immediately copy the object to ensure that it retrieves the object's original version. $\text{DFC}_{\text{caw}}$ on the other hand needs to track the ids of objects that already have been captured and are modified afterwards; copying these objects (a second time) in this case may be done later. Another major difference between the two approaches is the delay with which a checkpoint becomes available. Once a replica reaches the checkpoint sequence number, COW has to perform the entire checkpoint creation process, whereas $\text{DFC}_{\text{caw}}$ at this point typically has already captured the application state and directly can proceed to completing the checkpoint.

***Variant II: Updates (DFC$_{upd}$).*** In contrast to $\text{DFC}_{\text{caw}}$, in our second variant for implementing deterministic fuzzy checkpoints, $\text{DFC}_{\text{upd}}$, the replication library does not know about the identities and contents of the objects an application state consists of. Instead, during the state-capture phase the library receives and maintains a list of application-specific updates which (if necessary) it can later apply to the captured object snapshots in order to make the checkpoint deterministic.

Figure 5 presents the interface between replication library and application in $\text{DFC}_{\text{upd}}$. Compared with $\text{DFC}_{\text{caw}}$, the `invoke()` method offers a second parameter to initiate the creation of an update that represents the state modifications triggered by a request during execution. If a request changes multiple objects, the corresponding update comprises information on each of these modifications. Using `fuzzy()`, the library obtains a fuzzy application-specific snapshot of all current state objects. Upon request the library uses this snapshot as basis for a complete checkpoint (`complete()`), which then can be used to initialize another replica (`apply()`). As all required information is exchanged through snapshots and updates, in $\text{DFC}_{\text{upd}}$ there is no need for the application to call back the replication library when a state object is modified.

```
1 interface Upd_Application {
2   /* Request execution */
3   [RESULT, UPDATE] invoke(REQUEST r, boolean createUpd);
4
5   /* Checkpointing */
6   SNAPSHOT fuzzy();
7   SNAPSHOT complete(SNAPSHOT s, UPDATE[] u);
8   void apply(SNAPSHOT s);
9 }
```

Figure 5. Interface between replication library and application if deterministic fuzzy checkpoints are implemented based on updates ($\text{DFC}_{\text{upd}}$).

To avoid redundant work the replication library only instructs the application to produce updates during the state-capture phase. Once created, the library adds these updates to its local list of state modifications, thereby preserving the request order. In parallel, the library executes a checkpointer thread that copies the application state by calling `fuzzy()`. As a result, at the end of the state-capture phase a full $\text{DFC}_{\text{upd}}$ checkpoint either comprises a copy of the latest state of each object or it contains an earlier state accompanied by a list of updates that transform the previous state into the latest state.

$\text{DFC}_{\text{upd}}$ does not require updates to possess a specific format as long as it is guaranteed that applying all updates to the fuzzy snapshot in the determined order results in identical checkpoint contents on all correct replicas. In particular, this must be true independent of whether the obtained object copy initially represents the state of the object at the beginning, middle, or end of the state-capture phase. One solution to achieve this, for example, is to maintain per-object version counters that the application increments on each modification. Adding these version numbers to the corresponding updates later enables an application to skip updates which are already reflected in the object state when completing the checkpoint.

Requiring consistency of individual state objects, $\text{DFC}_{\text{upd}}$ updates impose comparably weak restrictions. In contrast, the request sequences included in hybrid checkpoints [3], for example, are only executable on a consistent application state and therefore could not be used to complete a fuzzy snapshot.

***Comparison.*** Even though both $\text{DFC}_{\text{caw}}$ and $\text{DFC}_{\text{upd}}$ produce deterministic fuzzy checkpoints in parallel with request execution, the two approaches differ in two main aspects: the degree to which the replication library has knowledge about state objects as well as the way of representing state modifications. Using $\text{DFC}_{\text{caw}}$, a replication library sees the individual objects the application state consists of and therefore is able to handle large parts of state capture and checkpoint completion in an application-agnostic manner. While this generalizability requires $\text{DFC}_{\text{caw}}$ to track and model modifications at the granularity of entire objects, $\text{DFC}_{\text{upd}}$, by leaving object handling and modification tracking to the application, allows a replica to manage state changes at a much finer level. Therefore, $\text{DFC}_{\text{caw}}$ and $\text{DFC}_{\text{upd}}$ represent the two ends of the spectrum of feasible ways to implement deterministic fuzzy checkpoints; additional variants combining ideas from both approaches are possible.

### C. Optimizations

In the following, we present optimizations that further improve the performance and/or efficiency of DFC.

***Object-specific Modification Tracking.*** As discussed in Section IV-A, to produce a comparable checkpoint a replica uses the previously captured snapshot and applies all recorded state modifications. This way, the replica can ensure that the state of each object in the final checkpoint is up to date, even though for some objects the fuzzy snapshot might comprise an earlier version. Building on this insight, it is possible to speed up checkpoint completion by only recording and applying the modifications that actually result in newer object versions. In particular, a replica can ignore modifications for an object that occur after the beginning of the state-capture phase but before the checkpointer thread creates a copy of the object. That is, a replica may select for each object individually the point in time at which it starts to track modifications for the object, depending on the progress made by the checkpointer thread.

***Throttled Checkpointing.*** Using DFC, a replica is able to create checkpoints while continuously processing requests. To further minimize the impact checkpointing procedures have on performance, a replica may deliberately throttle the state-capture process by introducing short pauses between copying two state objects. As a consequence of the prolonged state capture, the costs for obtaining an application-state snapshot do not have to be payed at once, but instead are distributed over an extended period of time. This approach is especially effective in cases in which the checkpointer thread and the application contend for shared resources such as the CPU.

## V. Deterministic Differential Fuzzy Checkpoints

In this section, we present details on how our approach can be extended to create deterministic differential fuzzy checkpoints (DDFC), that is, checkpoints that only cover the application-state changes made since the previous checkpoint.

***General Approach.*** To create a differential checkpoint, a replica needs to identify recently modified parts of the application state. For DDFC, we solve this problem by introducing an additional monitoring phase ("state observation"), which a replica executes prior to performing the two main checkpointing steps (i.e., state capture and checkpoint completion).

***State Observation.*** As shown in Figure 6, when a replica has completed the state capture for a checkpoint it immediately starts the state-observation phase for the next checkpoint. During this phase, the replica does not yet record the actual contents of state modifications, but it already collects knowledge on the state parts that changed. How exactly this state-observation-phase information is maintained by a replica depends on the specific implementation of DDFC. In case of the copy-after-write variant $\text{DDFC}_{\text{caw}}$, the replication library accumulates the ids of modified state objects. For the update-based variant $\text{DDFC}_{\text{upd}}$, on the other hand, the library asks the application to generate a *meta update* for every processed write request. In contrast to regular updates, a meta update solely contains meta-data information on the state parts affected, but lacks the newly written contents. Consequently, meta updates are typically much smaller than regular updates and can be created more efficiently. This significantly minimizes redundant work in scenarios where the same state parts are repeatedly modified within the same state-observation phase.

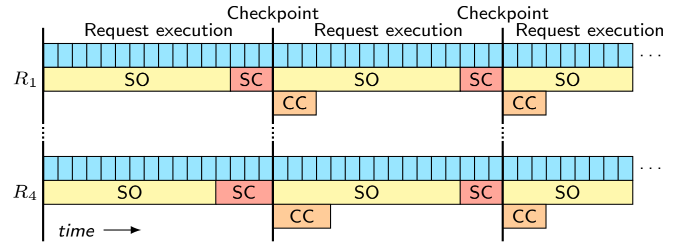



Figure 6. Three phases of deterministic differential fuzzy checkpointing: state observation (SO), state capture (SC), and checkpoint completion (CC).

***State Capture.*** Similar to the approach for full checkpoints (cf. Section IV-A), to create a checkpoint for sequence number $s$ using DDFC a replica $i$ starts the state-capture phase at an individually selected sequence number $p_i$ and records all state modifications $M_{p_i+1,\ldots,s}$ between $p_i$ and $s$. However, in this case the captured checkpoint $D_{s,i} = (\Delta_{[p_i,s]}, M_{p_i+1,\ldots,s})$ contains a fuzzy snapshot $\Delta_{[p_i,s]}$ that only comprises copies of objects that have been modified since the previous checkpoint. To identify these objects, a replica relies on the information obtained during state observation, that is, object ids and meta updates for $\text{DDFC}_{\text{caw}}$ and $\text{DDFC}_{\text{upd}}$, respectively.

***Checkpoint Completion.*** Except for being limited to a subset of state objects, at the end of the state-capture phase a differential fuzzy checkpoint possesses the same properties as a full fuzzy checkpoint. In particular, for each modified object the captured checkpoint either includes (1) a copy of the latest object state in the snapshot $\Delta$ or (2) one or more changes in the modification list $M$ that bring the captured object copy up to date. As a result, by applying the modifications to the fuzzy snapshot a replica is able to produce a differential checkpoint that is comparable across replicas. The resulting differential checkpoint is then merged with the last full checkpoint in order to yield an up-to-date full checkpoint.

## VI. Evaluation

In this section, we experimentally evaluate DFC in comparison to state-of-the-art approaches for creating checkpoints in replicated systems that tolerate arbitrary faults.

***Environment.*** Our prototype implementation is based on the REFIT replication library [8], [9], which we configure to rely on the PBFT [2] agreement protocol to provide Byzantine fault tolerance. For this paper, we extend the library to support the $\text{DFC}_{\text{caw}}$ and $\text{DFC}_{\text{upd}}$ interfaces as well as differential checkpoints. All experiments run on a cluster of five servers that are connected via switched Gigabit Ethernet. While one of the servers (Intel Xeon E5645, 2.4 GHz, 32 GB RAM, Ubuntu 18.04.2 LTS, Java 11) runs 100 client instances, the other four (Intel Xeon CPU E3-1275, 3.6 GHz, 16 GB RAM, Ubuntu 18.04.2 LTS, Java 11) each host a replica.

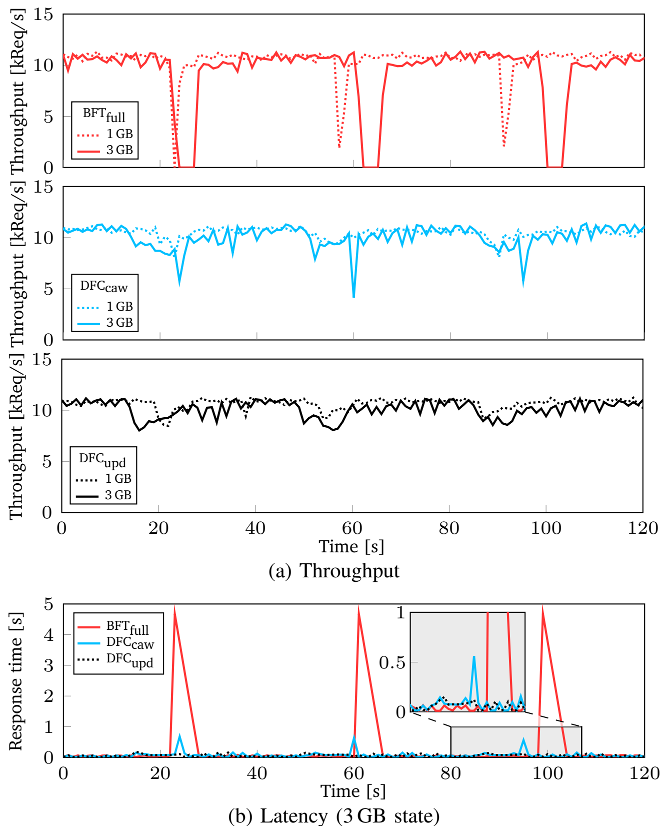



Figure 7. Full checkpoints: Comparison of throughput and latency results for $\text{BFT}_{\text{full}}$, $\text{DFC}_{\text{caw}}$, and $\text{DFC}_{\text{upd}}$ for checkpointing states of 1 GB and 3 GB.

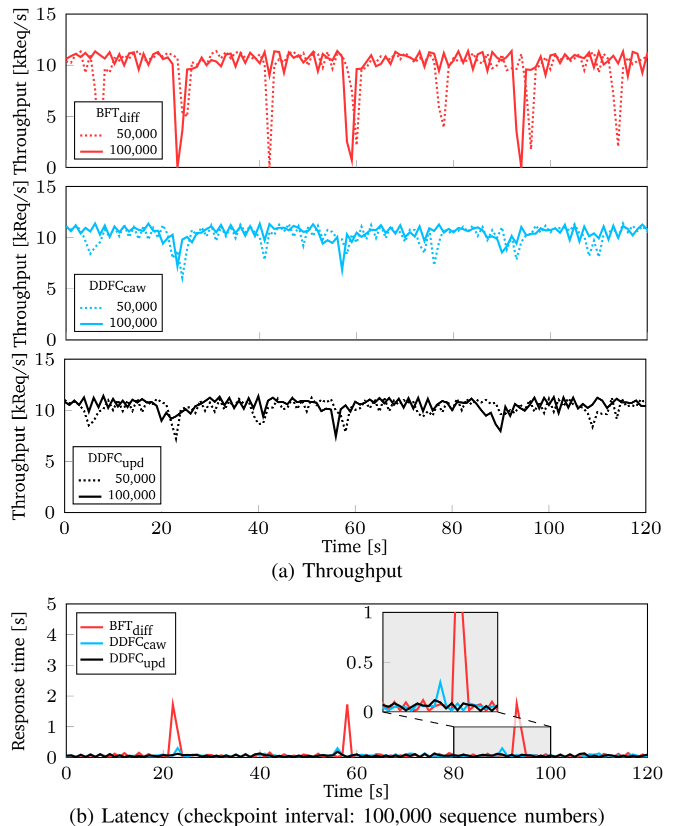



Figure 8. Differential checkpoints: Comparison of $\text{BFT}_{\text{diff}}$, $\text{DDFC}_{\text{caw}}$, and $\text{DDFC}_{\text{upd}}$ for checkpoint intervals of 50,000 and 100,000 sequence numbers.

To evaluate the efficiency of our DFC variants, we compare them against two baselines: (1) *$BFT_{full}$* represents the approach used in traditional Byzantine fault-tolerant systems [4], [8], [9] by suspending request execution while creating a full checkpoint. (2) *$BFT_{diff}$* also pauses the application during checkpointing, but in contrast to $\text{BFT}_{\text{full}}$ generates differential checkpoints [2], [11]. To obtain meaningful and comparable results, both $\text{BFT}_{\text{full}}$ and $\text{BFT}_{\text{diff}}$ share the same code base as our DFC implementations. We do not experimentally evaluate Dura-SMaRt's approach of configuring replicas to create checkpoints for different sequence numbers. Experiments in the Dura-SMaRt paper [18] indicate that this method is efficient with regard to performance, however, as discussed in Section III-B, it also introduces a vulnerability due to requiring replicas to load unverified checkpoints.

As application scenario for our experiments we rely on a key-value store, which is used by clients to save and retrieve data in chunks of 4 kilobytes. Each request accesses a single, randomly selected key-value pair. Unless stated otherwise, the clients issue equal shares of save and retrieve operations. Besides managing the actual data, the store for each key-value pair also maintains a small set of metadata such as a last-accessed timestamp. For DFC, each key-value pair represents an individual state object that is uniquely identified by its key. To capture the state of an object, the replication library serializes both data and metadata into a byte buffer. Using $\text{DFC}_{\text{upd}}$, updates (in serialized form) only include the pieces of data or metadata that actually changed. In all cases, the key-value pairs currently retained by the application are stored in a SQLite database that keeps them in memory. As the state already resides in memory and does not have to be fetched from disk, creating a checkpoint is comparably efficient. That is, our configuration minimizes the time it takes to capture the state, which is favorable for the baselines $\text{BFT}_{\text{full}}$ and $\text{BFT}_{\text{diff}}$.

***Full Checkpoints.*** In our first experiment, we evaluate the impact of creating full checkpoints on performance depending on application-state size. For this purpose, we vary the number of stored key-value pairs, creating a 1 GB setting that maintains 250,000 objects and a 3 GB setting with 750,000 objects. Figure 7 shows the results recorded after an initial warm-up phase. In this experiment, replicas produce a checkpoint every 100,000 sequence numbers, which due to the replication library applying request batching [2] translates to about 400,000 requests and a time interval of about 40 seconds.

As illustrated in Figure 7, although all the data is stored in memory it takes $\text{BFT}_{\text{full}}$ about 1.3 seconds to serialize the 1 GB state. During this time, $\text{BFT}_{\text{full}}$ suspends request processing in order to create a consistent checkpoint, causing the application to become unavailable. For the 3 GB state, the checkpointing duration increases to about 4.7 seconds and results in a significant service disruption. In contrast to $\text{BFT}_{\text{full}}$, both $\text{DFC}_{\text{caw}}$ and $\text{DFC}_{\text{upd}}$ capture the state in parallel with request execution and thus maintain high throughput even during the creation of full checkpoints for large states.

Producing the fuzzy snapshot while the application is running temporarily affects throughput due to the synchronization required between checkpointer thread and key-value store. In addition, for $DFC_{caw}$ our measurements indicate small latency spikes of less than 700 milliseconds that are the result of $DFC_{caw}$'s last state-capture step which once again copies all state objects that have been modified during the capture of the fuzzy snapshot (see Section IV-B). Including only the most recently changed objects, this step typically consumes a limited amount of time. However, as we discuss in Section VII, if necessary it is possible to mitigate the effects of the step by modifying $DFC_{caw}$ to execute additional rounds of object copying. With $DFC_{upd}$ collecting state updates during snapshot capture, unlike $DFC_{caw}$, $DFC_{upd}$ does not need a corresponding step of additional object copies, enabling low latency throughout the entire checkpointing process.

***Differential Checkpoints.*** In the second part of our evaluation, we focus on differential checkpoints and therefore rely on $BFT_{diff}$ as baseline. With differential checkpoints only capturing the changes since the previous checkpoint, the duration of the checkpointing process does not depend on the total size of the application state, but instead on the checkpoint interval and the extent to which the request workload modifies the state. Figure 8 presents the measurement results for an experiment with a 3 GB state in which we vary the checkpoint interval between 50,000 and 100,000 sequence numbers. In both scenarios, $BFT_{diff}$ is able to produce a checkpoint with less overhead than $BFT_{full}$ in the previous 3 GB experiment. Nevertheless, if more than 200,000 objects change since the last checkpoint it still takes $BFT_{diff}$ more than a second to capture them, which shows that even differential checkpointing can cause considerable service disruptions. Operating $BFT_{diff}$ with tiny checkpoint intervals in general does not solve this issue, because such an approach would reduce state-capture duration at the expense of a decreased overall throughput that is the result of a replica copying state objects at high frequency. Using $DDFC_{caw}$ and $DDFC_{upd}$, in contrast, the performance impact of creating a differential checkpoint is limited to the short state-capture phase that is executed in parallel with request processing. As our results from additional experiments confirm (see Figure 9), the maximum response times of $DDFC_{caw}$ and $DDFC_{upd}$ vary with the number of modified objects, however, the absolute values are significantly smaller than the maximum response times induced by $BFT_{diff}$.

To further examine the differences between $DDFC_{caw}$ and $DDFC_{upd}$, we conduct an experiment with two distinct workloads consisting of (1) requests that retrieve values and thus only modify metadata (i.e., a key-value pair's last-accessed timestamp) and (2) requests that store new values and change both metadata and data. As the results in Figure 10 illustrate, the fact that $DDFC_{upd}$ relies on application-specific updates enables it to only capture the actual changes, thereby improving state-capture efficiency. This is especially beneficial for scenarios such as the metadata-only workload (more than 25 % speedup) where there is a notable difference between the size of an object and the parts that have been modified.

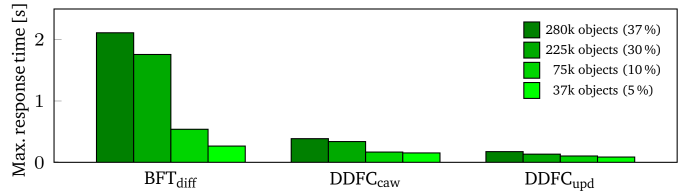



Figure 9. Observed maximum response times depending on the number of modified state objects that need to be captured for a differential checkpoint.

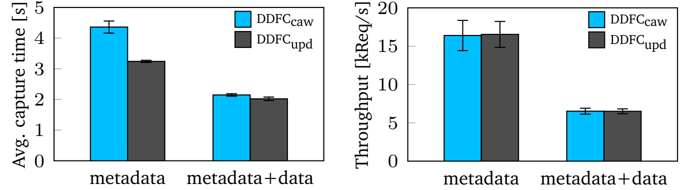



Figure 10. State-capture durations (left) and throughputs (right) of $DDFC_{caw}$ and $DDFC_{upd}$ for different workloads modifying different parts of an object. The metadata workload has a higher throughput and capture time as its requests are smaller than those of the metadata+data workload, which results in larger request batches and thus more object accesses between checkpoints.

***Discussion.*** A comparison of our results to previously published numbers reveals that the cost of snapshotting the application state in our experimental environment is low and therefore favorable to $BFT_{full}$ and $BFT_{diff}$. Bessani et al. [18], for example, reported service downtimes of several seconds for checkpointing a 1 GB state to memory, instead of 1.3 seconds in our case, and disruptions of more than ten seconds when saving a snapshot to disk or SSD. This indicates that the benefits of deterministic fuzzy checkpointing over traditional techniques may even be larger in other replicated systems.

## VII. Related Work

To our knowledge, DFC is the first approach leveraging fuzzy checkpoints in the context of Byzantine fault tolerance. So far, they have only been used in crash-tolerant systems including in-memory databases [12], [13], [14] and coordination services [20]. Due to replicas in such systems trusting each other, there is no need for synchronizing checkpoint procedures across replicas. Without further interaction, a replica can snapshot its own state at any given time and transfer the checkpoint to another replica. For tolerating Byzantine faults this is not an option as a replica must be able to verify a received checkpoint. DFC allows a system to generate a proof of correctness for a checkpoint by ensuring that all correct replicas produce comparable checkpoints for the same sequence number. In contrast to the crash-tolerant systems mentioned above, DFC for this purpose does not require the application to implement state modifications in the form of transactions.

In high-performance computing (HPC) clusters, checkpointing also plays a vital role in the handling of component failures, however its usage differs in several aspects from that in replicated systems: (1) For a checkpoint, an HPC system commonly saves the full state of each application process [21], whereas a replicated system only captures the application's data. (2) HPC applications and their states are spread across a large number of servers, resulting in each server to contribute

a unique piece to the overall checkpoint; ensuring that the sum of these pieces represents a consistent checkpoint requires cluster-wide coordination [21]. In contrast, in replicated systems each server maintains a consistent copy of the entire application state, making checkpoints directly comparable. (3) With each server providing a unique checkpoint part, to tolerate crashes HPC systems must store each newly created snapshot on several machines, thereby possibly minimizing storage and performance overhead using erasure codes [22] and a hierarchy of heterogeneous storage devices [23]. A replicated system on the other hand only needs to transfer a checkpoint when a new or recovering replica demands it.

The Byzantine fault-tolerant systems VM-FIT [6] and ZZ [24] run each replica inside a virtual machine and exploit file-system snapshots to reduce checkpoint-creation overhead. Unlike DFC, this approach is only effective if an application already maintains large parts of its state on disk. DFC, on the other hand, does not make any assumptions on where the state of an application resides. Furthermore, it does not pose special restrictions on the functionality provided by the underlying file system, thereby offering the opportunity to implement the technique as part of a platform-independent replication library.

Being able to capture the state of a component without suspending execution is not only beneficial in replicated systems, but also for the live migration of virtual machines. Clark et al. [25], for example, propose a multi-round approach to transfer the memory pages used by a running virtual machine to another server. In particular, after an initial round that copies all affected memory pages, their mechanism executes several transmission rounds involving intermediate modifications. Similarly, the last step of $\text{DFC}_{\text{caw}}$'s state-capture phase (i.e., the final copying of recently modified state objects) could be split into multiple rounds. As a result, fewer state objects would have to be copied once the application has reached the sequence number at which the checkpoint is due.

In this paper, we focused on creating verifiable checkpoints, which is usually the first problem to be solved to enable a reliable state transfer between replicas. Other authors aimed at improving subsequent steps limiting transmission to the state parts that are actually required by a replica [2] or enabling a replica to parallelize state transfer by fetching different parts from different replicas [26]. With completed DFC checkpoints being identical to traditionally created checkpoints, such techniques can be directly combined with DFC.

## VIII. Conclusion

DFC enables Byzantine fault-tolerant systems to efficiently create deterministic application checkpoints in parallel with request execution. Supporting the method requires additional logic as well as computing and memory resources compared with the straightforward approach of suspending the application during state capture, but in return offers increased availability. As shown in this paper, DFC is flexibly applicable to both full and differential checkpoints. Depending on the use case, DFC can either be closely integrated with a generic replication library or optimized for a specific application.

## References


[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[2] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[3] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "UpRight cluster services," in *Proc. of SOSP '09*, 2009, pp. 277–290.

[4] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *Proc. of DSN '14*, 2014, pp. 355–362.

[5] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Veríssimo, "Resilient intrusion tolerance through proactive and reactive recovery," in *Proc. of PRDC '07*, 2007, pp. 373–380.

[6] T. Distler, R. Kapitza, and H. P. Reiser, "State transfer for hypervisor-based proactive recovery of heterogeneous replicated services," in *Proc. of SICHERHEIT '10*, 2010, pp. 61–72.

[7] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat, "SPARE: Replicas on hold," in *Proc. of NDSS '11*, 2011, pp. 407–420.

[8] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient Byzantine fault tolerance," *IEEE Trans. on Computers*, vol. 65, no. 9, pp. 2807–2819, 2016.

[9] M. Eischer and T. Distler, "Scalable Byzantine fault-tolerant state-machine replication on heterogeneous servers," *Computing*, vol. 101, no. 2, pp. 97–118, 2019.

[10] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[11] M. Castro, R. Rodrigues, and B. Liskov, "BASE: Using abstraction to improve fault tolerance," *ACM Trans. on Computer Systems*, vol. 21, no. 3, pp. 236–269, 2003.

[12] R. B. Hagmann, "A crash recovery scheme for a memory-resident database system," *IEEE Trans. on Computers*, no. 9, pp. 839–843, 1986.

[13] K. Salem and H. Garcia-Molina, "Checkpointing memory-resident databases," in *Proc. of ICDE '89*, 1989, pp. 452–462.

[14] J.-L. Lin and M. H. Dunham, "Segmented fuzzy checkpointing for main memory databases," in *Proc. of SAC '96*, 1996, pp. 158–165.

[15] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[16] T. Distler and R. Kapitza, "Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency," in *Proc. of EuroSys '11*, 2011, pp. 91–105.

[17] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza, "SAREK: Optimistic parallel ordering in Byzantine fault tolerance," in *Proc. of EDCC '16*, 2016, pp. 77–88.

[18] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Proc. of USENIX ATC '13*, 2013, pp. 169–180.

[19] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proc. of PODC '07*, 2007, pp. 398–407.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. of USENIX ATC '10*, 2010, pp. 145–158.

[21] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Proc. of SRDS '92*, 1992, pp. 39–47.

[22] J. S. Plank and K. Li, "Faster checkpointing with N+1 parity," in *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*, 1994, pp. 288–297.

[23] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proc. of SC '10*, 2010, pp. 1–11.

[24] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *Proc. of EuroSys '11*, 2011, pp. 123–138.

[25] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. of NSDI '05*, 2005, pp. 273–286.

[26] R. Kapitza, T. Zeman, F. J. Hauck, and H. P. Reiser, "Parallel state transfer in object replication systems," in *Proc. of DAIS '07*, 2007, pp. 167–180.



*Acknowledgments:* This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 ("REFIT").