Achieving Enhanced Performance Combining Checkpointing and Dynamic State Partitioning

Henrique S. Goulart, João Trombeta, Álvaro Franco and Odorico M. Mendizabal
Programa de Pós-Graduação em Computação PPGCC
Departamento de Informática e Estatística, Universidade Federal de Santa Catarina UFSC
Florianópolis, Santa Catarina, Brazil
sgoulart.henrique@gmail.com, joaog.trombeta@gmail.com, alvaro.junio@ufsc.br and odorico.mendizabal@ufsc.br

***Abstract*—Fault-tolerant systems rely on recovery techniques to enhance system resilience. In this regard, checkpointing procedures periodically take snapshots of the system state during failure-free operation, enabling recovery processes to resume from a previously saved, consistent state. Saving checkpoints, however, is costly, as it must synchronize snapshots with the processing of incoming requests to avoid inconsistency. One way to speed up checkpointing is to partition the service state, allowing a parallel checkpoint procedure to operate independently on each partition. State partitioning can also improve throughput by increasing parallelism in request processing. However, variations in the data access pattern over time can result in unbalanced partitions, posing a challenge to achieving optimal performance. In this paper, aiming to improve both checkpointing and overall system performance, we combine parallel checkpointing with a dynamic graph-based repartitioning algorithm. This work formalizes the optimization problem and presents a detailed performance assessment of the proposed approach. The experimental evaluation highlights the benefits of parallel checkpointing and emphasizes the performance gains achieved with repartitioning under realistic workloads. Comparing a cost-effective round-robin partitioning approach with our dynamic method, we examine the degree of execution parallelism achieved by checkpointing threads and the influence of repartitioning strategies on checkpoint performance. Although the rebalancing of state partitions incurs a cost, it comes for free in our technique since it takes advantage of processing idleness during the snapshot-taking process.**

***Index Terms*—fault tolerance, checkpoint/restore, state partitioning, graph partitioning algorithms**

I. Introduction

Fault-tolerant systems employ checkpoint/recovery to keep services running uninterrupted. These techniques combine durability strategies to maintain system availability and resilience, even in the presence of failures or potential threats. Checkpoint procedures periodically capture snapshots of the system's state, facilitating recovery from faults and restoring system states not too far behind the rest of the system [1], [2]. However, creating checkpoints can be resource-intensive as it requires synchronization with incoming requests to establish a consistent and recoverable snapshot of the system's state. As the system's state grows, the I/O-intensive operations of creating checkpoints can become a bottleneck.

This work received financial support of the Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina - FAPESC and Federal University of Santa Catarina - UFSC.

State partitioning approaches can reduce the checkpointing overhead, as different portions of the system state can be handled individually. Especially in multi-core architectures, the state partitions may be saved in parallel, speeding up the checkpoint procedure [3], [4]. Other specific hardware technologies and system-level support for parallelism, such as parallel I/O [5], NVM [6], and transactional memory [7] can contribute to speeding up the checkpoint-related durability operations.

Besides its potential to enhance checkpointing, the state partition technique may also increase the overall system throughput. By dividing the application state into smaller, manageable pieces, partitioning enables the parallelization of work, leading to faster execution compared to sequential processing. When the workload is evenly balanced, services may greatly benefit from multi-core architectures. However, establishing a good partitioning of the service state is not trivial since it requires information about the workload in advance. Even if some information is available, the data access pattern may change over time, causing unbalance.

In this paper, we propose a practical approach that combines dynamic repartitioning with parallel checkpoints to reduce checkpointing overhead and simultaneously rebalance state partitions. Rebalancing state partitions resembles the balanced graph partitioning problem, where the goal is to divide a graph into disjoint subsets while balancing the weights of vertices in each subset. In this arrangement, edges belonging to different subsets are called cut-edges. Our approach not only aims to balance the weights among subsets but also to minimize the total weight of cut-edges. Although rebalancing state partitions is costly, it comes for free in our approach. This is because it occupies the processing cores for rebalancing while the processing of incoming requests is interrupted by checkpoints.

This paper makes the following contributions. First, it presents a parallel checkpointing procedure based on state partitioning. Although the state partitioning approach resembles other state partitioning strategies found in literature [8], [9], the novelty lies in the seamless integration of checkpointing with repartitioning, incurring minimal additional costs. Second, the paper formalizes the optimization problem to be solved by the dynamic repartition approach. Finally, an experimental evaluation carefully demonstrates the pros and cons of the proposed technique. The evaluation exercises realistic workloads

provided by the YCSB benchmark [10] and compares our dynamic approach with a cost-effective partitioning approach that evenly distributes state variables across partitions using a round-robin fashion.

The experiments indicate that the time required to rebalance partitions can be effectively concealed by the checkpoint duration, except in cases where the checkpoint sizes are small. In addition, the repartitioning enables an increase in overall throughput in workloads with cross-partitioning requests (e.g., range scans and multi-variable operations). Results shown a throughput improvement of 2.5 times in such workloads.

The structure of this paper is as follows. Section II explores related work, while Section III presents our approach, which involves parallel checkpointing with dynamic repartitioning. The performance evaluation of the proposed approach is detailed in Section IV. The paper concludes with Section V.

II. Related work

The growing demand for high-performance and scalable applications has increased interest in optimizing application state management. Partitioning the state of an application is a well-established technique for enhancing performance in parallel and distributed systems. As the hardware landscape evolved to include multi-core processors and many-core architectures, partitioning has become increasingly relevant, with recent research exploring new ways to apply it effectively [11]–[16]. Several studies aim to enhance partitioning schemes to increase application robustness, throughput, and response time. While not explicitly focused on checkpointing, these studies offer valuable insights into how repartitioning can boost overall application performance.

The research of [8] presents Schism, a workload-aware method for database partitioning. The study highlights the high cost of distributed transactions that spawn multiple partitions. It emphasizes that automatic partitioning schemes, such as round-robin, range, and hash-partitioning, are the most used in modern distributed systems. While these approaches excel in delivering efficient large-range scan performance, they exhibit limitations in effectively scanning shorter ranges. The suggested approach employs a graph partitioning heuristic that aims to reduce the cost of graph cuts while balancing the weight of each partition. The method represents database tuples as nodes within a graph, and edges represent the tuples concurrently accessed by transactions. Each time a tuple is accessed, the weight of the corresponding node is incremented. If multiple tuples are accessed simultaneously by a single transaction, the weight of the edges linking the nodes representing these tuples is incremented. The research utilizes two distinct heuristics: the first considers the partition size as the node's weight, while the second considers the number of accesses as the node's weight.

A key challenge the study presents is how to efficiently handle large databases as graph representations grow with increasing database size and transaction complexity. Graph partitioners scale effectively regarding partition count, but their runtime escalates with graph size. In this regard, sampling heuristics have been developed to reduce graph size while maintaining a decent partitioning quality. The study affirms that by minimizing distributed transactions and maintaining a balanced workload among the nodes, the service can significantly enhance transaction throughput for databases. These findings align with the results presented in Section IV. We show that repartitioning can effectively reduce the synchronization costs with cross-partition requests. Differently from [8], the costs related to partitions' rebalancing are negligible with our approach.

In [9], authors introduce SWORD, a framework designed to address various challenges in database partitioning. It minimizes book-keeping overhead by employing a two-phase strategy. The first phase compresses the hypergraph, while the second partition the compressed hypergraph. This approach reduces the size of mapping tables, which are essential in directing transactions to the correct partition. The framework manages workload alterations by monitoring and detecting substantial changes and adjusting the partitioning in incremental steps. Incremental repartitioning helps alleviate performance decline caused by workload fluctuations without a complete repartitioning process. The framework also integrates a workload-aware replication mechanism to resolve transaction conflicts, enhancing data placement.

While the research presented in [9] proposes incremental repartitioning, we claim that utilizing the idle time resulting from I/O during the checkpointing process may be sufficient to execute a full repartitioning. This alternative approach simplifies the implementation process by avoiding the need for incremental repartitioning, monitoring, and other SWORD auxiliary components.

DYPART [17] is a framework designed for dynamic state partitioning in Byzantine Fault Tolerance (BFT) protocols that leverage parallel execution of requests [18], [19]. Its primary function is gathering and segmenting application states into partitions, which are then reconfigured in real-time to optimize performance. Each replica continuously observes the dependencies among requests and constructs a graph that represents the state objects' relationships. DYPART handles operations that involve multiple partitions, known as cross-border requests, while ensuring deterministic execution. The state partitioning update is linked with the checkpoint mechanism to maintain determinism and minimize overhead. Upon reaching the checkpoint threshold, a new checkpoint is created, and each replica executes the graph partitioning algorithm, aiming to maintain a low cross-border request rate.

Similarly to DYPART, our approach also benefits from CPU idleness during checkpoints to execute the state repartition. However, DYPART lacks a comprehensive analysis of the impacts of combining checkpointing and repartitioning. The study utilized only one workload for experimental evaluation and discussion, primarily focused on representing relationships between accessed objects that depict the interaction between characters in *Les Misérables* provided by the Stanford Graph Base [20]. In contrast, our study formalizes the optimization problem and conducts experimental evaluations using realistic

workloads to assess the effects of combining checkpointing and repartitioning in a broader context.

Related research on checkpoint techniques has a vast literature ranging from operating systems, databases, high-performance computing, cloud computing, and dependable and distributed systems. Although we adopt a partitioned parallel checkpoint strategy, other approaches, such as fuzzy or incremental checkpoints, could be applied without limiting the approach generality. Thus, we consider improvements in the checkpoint strategy orthogonal to this work. In [1], the authors present the background and the main strategies for checkpointing in message-passing systems. Checkpointing in HPC is addressed in [2], and in [21], authors discuss the evolution of checkpoint techniques over the past decades.

III. Parallel Checkpointing with Dynamic Repartitioning

This section presents a partitioned checkpointing approach combined with dynamic state repartition. The objective of the proposed approach is twofold: to speed up checkpoint execution and to take advantage of processing idleness while taking checkpoints to improve the balance of state partitions.

In the following, we describe the state partitioning strategy adopted, the parallel checkpointing, and the optimization problem definition for state repartitioning.

A. State Partitioning Execution Model

In this work, the application state S is a set partitioned into n disjoint sets, each one called *partitions*. The set of partitions is $\{p_1, p_2, ..., p_n\}$ and the union of the elements of each p_i is equal to S. The system executes n worker threads, where thread t_i is responsible for the execution of requests involving the partition p_i. An arbitrary request r executes over a set of keys. Let $K(r)$ be the set of keys accessed by r, and let $P(k)$ be the partition to which a key k is designated. Every thread t_i has its own queue q_i, with requests that access its designated partition p_i. Thus, when a request r arrives, a scheduler places it in the queues of every thread whose keys are accessed by r. Formally, r is placed in queue q_i for all i such that $p_i \in \bigcup_{k \in K(r)} P(k)$.

As an example, assume a system with state $S = \{x, y, w, z\}$ and a set of threads $T = \{t_1, t_2\}$. Fig. 1 represents a possible state partitioning with threads t_1 and t_2 being responsible for updates in partitions $p_1 = \{x, y\}$ and $p_2 = \{w, z\}$, respectively. The figure also shows threads' queues after receiving requests r_1 to r_7. As can be observed, r_1 accesses x, and once $x \in p_1$, r_1 is dispatched to t_1, and so on for every other request. Requests are dispatched respecting the delivery order to the appropriate queues. In this example, requests semantics are defined by commands $write(k, v)$ and $swap(k_i, k_j)$, where $write$ updates the variable given by the key k with a value v, and $swap$ mutually changes the values of keys k_i and k_j.

In Fig. 1, requests r_1, r_3, r_5, r_6, and r_7 are single-variable requests, and they can be executed in order, one by one, without synchronization. *Single-partition multi-variable* requests access multiple keys that belong to the same partition, so the same thread t executes them. In the example, request r_2 is a single-partition multi-variable. *Cross-partition multi-variable* requests access two or more variables, and at least two of them belong to different partitions. Request r_4 is a cross-partition multi-variable request, and it accesses variables y that belongs to thread t_1's partition, and z that belongs to thread t_2's partition. This kind of request requires threads synchronization to implement atomicity and, consequently, maintain consistency.

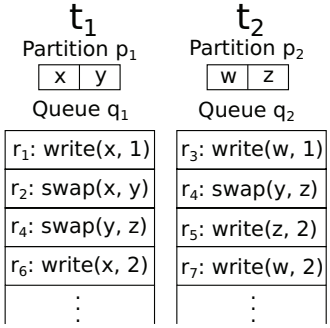

Fig. 1. Example of requests scheduling across thread queues.

Fig. 2 illustrates an execution trace for the threads t_1 and t_2 exemplified by Fig. 1. All threads run in parallel, r_1, r_3, r_5, r_6, and r_7 are single-variable requests, so they are executed as soon as they are in the head of their threads' queue. Request r_2 is a multi-variable single partition request. It involves x and y, both in p_1, so it can be executed by t_1 without synchronization. Request r_4 is a cross-partition multi-variable request. Then, it is present in both t_1 and t_2 queues. When t_2 retrieves r_4 from its queue, it waits for t_1 in a barrier, as represented by the dotted curve. When t_1 retrieves r_4, all involved threads are ready, and r_4 is executed by one arbitrary thread only, for instance, the one with the lowest id (in this example, thread t_1). After that, both threads t_1 and t_2 are allowed to continue. In the time between t_2 reaching r_4 and r_4's execution by t_1, t_2 remains completely idle.

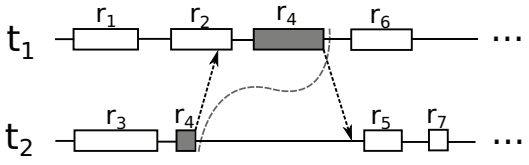

Fig. 2. Execution of system in Fig. 1.

B. The parallel checkpoint

This approach aims to benefit from the parallelism in modern hardware, both in processing and I/O. The goal is to parallelize the checkpointing operation, potentially speeding up saving and restoring the checkpoint state.

The checkpointing approach features multiple threads, each tasked with saving a designated partition of the entire state.

The execution scheduler is responsible for initiating a checkpoint. The scheduler periodically adds a *checkpoint request* to all the partition's queues. The *checkpoint request* is a *cross-partition multi-variable* involving all partitions, so it synchronizes all checkpoint threads to execute the complete checkpoint in parallel. The checkpointing procedure awaits the completion of the process by each partition thread.

For example, partitions p_1 and p_2 in Fig. 1 can be saved in parallel, speeding up the checkpoint. Computers with multiple storage devices can even benefit from parallel I/O to improve throughput while snapshotting the application state. The time to accomplish a checkpoint is given by the time taken by the slowest thread to finish. The local checkpoint is now composed of multiple files, each representing a snapshot of a single partition. The recovery procedure also benefits from the partitioned schema as it can restore multiple files in parallel. We do not discuss recovery in this paper, as it is straightforward. In this case, the only difference is that a checkpoint is composed of multiple files, and restoring a consistent state requires the set of files to be completely restored before starting the process execution.

C. Augmenting checkpoint with repartitioning

Despite the potential parallelism observed by state partitioning, the execution performance of incoming requests highly depends on how the scheduler distributes requests among worker threads since poor scheduling results in expensive synchronizations [13]. Establishing a good partitioning of the service state is not trivial since it requires information about the workload in advance. Moreover, even if some information is available, the data access pattern may change over time, causing a local good data partitioning to perform poorly under workload global changes. Therefore, dynamic repartitioning running at run-time might enhance load balancing across the partitions.

Graph algorithms may support repartitioning strategies, and they are widely used to solve problems in various fields, such as road mapping, computer networking, and database representation. In our case, a graph may represent the access in the multi-partitioned state, and we define the repartitioning optimization problem as follows.

1) Problem Definition: Suppose that a system maintains the use of each state variable as the following. The system registers how often the variables were triggered by read and write requests, possibly by requests involving many variables (we call multi-variable requests). This historic can be organized in a *vertex-edge weighted graph* where vertices represent the variables, the number of requests that use a variable is the weight of the corresponding vertex (c_i for vertex i), the edges represent the multi-variable requests, and the weight of the corresponding edge gives the number of times that the two variables appear in multi-variable requests (w_{ij} for edge ij). Now, the system can use this graph to suggest, in certain periods, a state partition $P = \{p_1, \ldots, p_k\}$ such that it meets the interest of the application. Here, each p_i in P must be "balanced", *i.e.*, the sum of the weights of all vertices belonging to it is not so different when compared with the weight of another p_j in P; another important property of the application is that the sum of the edges that have vertices in different parts of P is minimum. From the application's point of view, the first objective helps to accelerate the parallel saving of the state, and the second helps to reduce the online synchronization of requests. This problem is known as the *balanced partition minimum cut problem*. It is NP-hard [22], and many heuristic methods trying to solve it, e.g., [11], [12], [23], [24]. The Metis [23] is a heuristic much used in this case.

Our approach proposes a two-phase model to treat the balanced partition minimum cut problem. The first phase finds a balanced partition. The second phase finds a minimum cut considering the balanced partition given by the first phase. The number of partitions does not vary and is informed at initialization. Next, it is given a formulation of the problem in two phases.

a) Phase 1: Consider an integer variable y, which means the slack to pack all vertices into partitions. This variable should be as small as possible to ensure some balance. We also have binary variables x_{ip} defined as the following.

$$x_{ip} = \begin{cases} 1, & \text{if vertex } i \text{ belongs to partition } p, \\ 0, & \text{otherwise.} \end{cases}$$

For the first phase problem, we assume a fixed k. Consider a set of k partitions $P = \{p_1, \ldots, p_k\}$ and $C = \sum_{i \in V} c_i$. We want to pack all vertices in partitions from P so that each partition is not too "heavy". The integer programming model is

$$\begin{aligned} &\min\ y \\ &\text{subject to:} \quad \textstyle\sum_{p \in P} x_{ip} = 1, \forall i \in V && (1) \\ &\qquad\qquad \textstyle\sum_{i \in V} c_i x_{ip} \leq \frac{C}{k} + y, \forall p \in P && (2) \\ &\qquad\qquad x_{ip} \in \{0,1\}, \forall i \in V \text{ and } \forall p \in P && (3) \\ &\qquad\qquad y \in \mathbb{Z}^+. && (4) \end{aligned}$$

Constraint (1) ensures that each vertex is in (exactly) one partition. Constraint (2) ensures that each partition is not too "heavy" in an optimal solution since we want to minimize y. Constraints (3) and (4) mean that the variables x_{ip} are binary and y is an integer, respectively.

b) Phase 2: Since the value of the variable y is already known from *Phase 1*, set $M = \frac{C}{k} + y$. Now, it is desired to find the minimum cut considering such value to y. The variables x_{ip} are kept, and new binary variables z_{ijp} are defined as the following.

$$z_{ijp} = \begin{cases} 1, & \text{if edge } i-j \text{ has the both vertices } i \text{ and } j \text{ in} \\ & \text{partition } p, \\ 0, & \text{otherwise.} \end{cases}$$

$$
\begin{aligned}
\min \quad & \textstyle\sum_{i-j\in E} w_{ij}(1-\sum_{p\in P} z_{ijp}) \\
\text{subject to:} \quad & \textstyle\sum_{p\in P} x_{ip} = 1, \forall i \in V && (5)\\
& \textstyle\sum_{i\in V} c_i x_{ip} \le M, \forall p \in P && (6)\\
& z_{ijp} \le x_{ip}, \forall i-j \in E \text{ and } \forall p \in P && (7)\\
& z_{ijp} \le x_{jp}, \forall i-j \in E \text{ and } \forall p \in P && (8)\\
& z_{ijp} \ge \tfrac{1}{2}(x_{ip}+x_{jp}) - \tfrac{1}{2}, \forall i-j \in E, \\
& \qquad \forall p \in P && (9)\\
& x_{ip} \in \{0,1\}, \forall i \in V \text{ and } \forall p \in P && (10)\\
& z_{ijp} \in \{0,1\}, \forall i-j \in E. && (11)
\end{aligned}
$$

Constraints (5) and (6) ensure a balanced partition. Constraints (7), (8) and (9) find the edges inside partitions. Constraints (10) and (11) mean that the variables x_{ip} and z_{ijp} are binaries. Since constraint (5) ensures each vertex i in exactly one partition p, for an edge $i-j$ or both of its vertices are in one partition p (when i and j are in p) or at most one of them is in p. When both vertices are in p, z_{ijp} is forced to be 1 by constraint (9). Otherwise, z_{ijp} is forced to be 0 by constraints (7) or (8). Therefore, given a fixed edge $i-j$, the $\sum_{p\in P} z_{ijp}$ is equal to 1 only when both vertices i and j are in the same partition p. So, the sum of the weight of edges between partitions can be done as described in the objective function of the *Phase 2* problem.

Note that any feasible solution of *Phase 1* problem can be used to construct a feasible solution of *Phase 2* problem. Given a feasible solution y and x_{ip} to the first problem, the x variables can be maintained in the second problem and set $z_{ijp} = 1$ for any edge ij only if $x_{ip} = 1$ and $x_{jp} = 1$ ($z_{ijp} = 0$, otherwise).

2) Implementation: The checkpoint process incurs significant costs due to intensive I/O operations, yet it leaves room for additional processing usage during its execution. Recognizing this underutilization of CPU, we propose masking the cost of repartitioning by executing it concurrently with the checkpointing.

When a checkpoint process starts, the system simultaneously begins repartitioning. The system returns to its regular operation after both processes are complete. For effective repartitioning, the scheduler tracks requests and their respective partitions. Additionally, it upholds a data structure linking each variable to its specific partition, which aids in understanding the workload behavior.

The scheduler annotates partition accesses using a graph structure: each command key access is tied to a vertex. Each partition access increments its respective vertex value. For commands like $swap(a, b)$, which access multiple variables, an edge is formed between the vertices of a and b. Subsequent similar accesses increase the value of this edge. In this model, it is worth noting that each edge denotes a connection between two variables. Thus, a multi-variable request accessing three variables, for example, is represented by three distinct edges connecting each variable pair.

The repartitioning function utilizes the scheduler's graph, capturing the entire workload context. With the vertex and edge values, the algorithm reallocates the graph to ensure nearly equal weight for all partitions. It also weighs the *cut cost* for each partition, which refers to the edges connecting different partitions. The primary aim is to lower the cut cost while maintaining balanced partition weights.

3) Repartitioning algorithm: We utilized the METIS [23] for the balanced partition minimum cut problem because of its popularity and academic significance. This algorithm has three stages:

Coarsening: The graph is simplified by collapsing vertices and edges, creating a reduced version. A method called Heavy Edge Matching (HEM) merges vertices, focusing on those with high edge weights to minimize cut-edges between different subsets.

Partitioning: The reduced graph is divided into balanced sections. Initially, a 2-way partition is made, further split in subsequent iterations until the desired k-way partitions are reached. A spectral bisection algorithm, influenced by the Laplacian matrix values, handles the partitioning [25], [26].

Uncoarsening: The original graph is reconstructed, keeping the partition details from the previous stages. The algorithm translates the partition data from the simplified graph back to the original while undoing the vertex merges from the coarsening stage.

This approach ensures quicker processing times, especially with larger graphs.

IV. Evaluation

In this section, we present an experimental evaluation of our proposed approach. A key-value store prototype was developed in C++, integrating our parallel checkpointing strategy. The prototype consists of the following components: an in-memory key-value map storage; partition threads, each with individual request queues; a scheduler task that dispatches requests to each partition queue; checkpointing threads that activate the checkpoint operation after a given number n of requests have been dispatched, and all partition threads received a checkpointing request from the dispatcher.

Our analysis primarily focuses on the execution of requests, with the client-side and network layer abstracted for simplicity and to avoid interference. All requests are pre-loaded into memory before the start of the experiment. The scheduler then dispatches these requests for execution. Consequently, the cost metrics under examination are scheduling, request execution, thread synchronization, and repartitioning.

The experiment was conducted on a machine equipped with an Intel i5-11400F processor, 32GB of RAM, and a 1GB Samsung NVMe (980 PRO m2) storage unit.

This study analyzes the effects of checkpointing and repartitioning on comprehensive service performance. Upon receiving a request, data is divided into eight fixed partitions, with each key initially assigned to these partitions using a round-robin algorithm. The experiment maintains eight worker threads to execute regular operations but varies the number of checkpoint threads. We selected 8 worker threads, a choice influenced by the processor's capabilities encompassing 12 threads spread over 6 cores. This allocation considers the

additional load from an active scheduler and the operational demands of the system.

The study explores four distinct partitioning configurations: (i) Round-robin partitioning without checkpointing or repartitioning, representing a scenario devoid of checkpoint interference; (ii) A singular checkpointing thread handling the entire state, exemplifying a conventional checkpoint procedure devoid of parallelism; (iii) A parallel checkpoint involving eight checkpoint threads, but devoid of repartitioning; and (iv) A parallel checkpoint with eight threads and repartitioning activated, where repartitioning is initiated whenever a checkpoint is triggered.

We call attention to the version using the repartitioning algorithm, as this study aims to assess the feasibility of leveraging idle processing resources during the checkpointing process.

A. Workload

We used the Yahoo! Cloud Serving Benchmark (YCSB) [10] to generate diverse workloads simulating real-world scenarios. In particular, we evaluate the impact of the different prototype configurations under load profiles that experience single-variable requests distributed over the state space (YCSB-A), distributions aiming to intensify the access to variables recently inserted by the previous requests (YCSB-D), and multi-variable requests implemented by range scan requests (YCSB-E). Based on our expectations that the YCSB-B and YCSB-C workloads would produce results analogous to YCSB-A, we excluded them from this evaluation.

More precisely, YCSB-A comprises single-variable requests, 50% reads, and 50% writes. In YCSB-D, 5% of requests are insertions, and 95% are reads. Those reads are performed mainly on the newly inserted keys. In YCSB-E, 95% of the requests are range scans, *i.e.*, multi-variable requests, while 5% are insertions. Scans access at most 8 keys together. It is worth noting that workload E distinguishes itself from the others as it uniquely involves operations accessing multiple keys, which may reside within the same partition or across different partitions, all within a single operation.

Regarding workload YCSB-A, 94 million requests were executed involving 8 million unique keys. Similarly, for YCSB-D, there were 110 million requests with 10 million unique keys; for YCSB-E, 6 million requests were performed with 2 million unique keys.

For the performance analysis, we analyzed the throughput, *makespan*, time spent on checkpointing and repartitioning, checkpointing sizes, and request distribution.

B. Throughput impact

Fig. 3 illustrates the time-based throughput, denoting the number of requests executed per second. The applied workload is the YCSB-A, characterized by its update-heavy nature. The graph encompasses four distinct configurations: *no-ckp*, which serves as the baseline with no checkpoints performed; *1p-ckp*, wherein a single, non-partitioned checkpoint is performed by one thread; *8p-ckp*, which involves the execution of parallel checkpoints by eight threads; and *8p-ckp-metis*, where parallel checkpoints are executed by eight threads while simultaneously implementing the METIS algorithm for repartitioning.

In this workload, the first checkpoint's execution begins just before the 20-second mark, evident from a decreased throughput due to checkpoint processing. The drop is especially noticeable in the *1p-ckp* configuration, which sustains zero throughput for about $\approx 50s$. In contrast, the parallel configurations, *8p-ckp*, and *8p-ckp-metis*, display a more efficient checkpointing, evidenced by only about $\approx 20s$ of no throughput. This efficiency underscores the advantages of parallel configurations in managing checkpointing states.

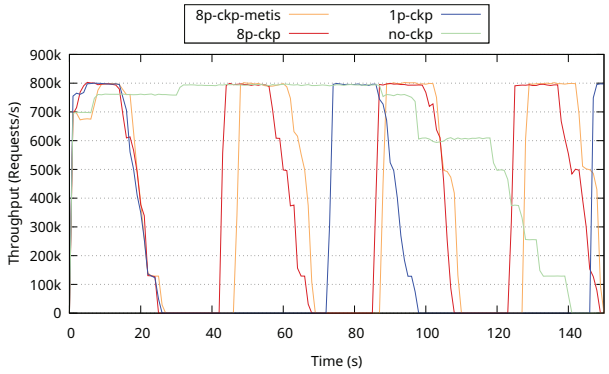

Fig. 3. Throughput over time for YCSB-A.

Fig. 4 presents the same metrics as before but with the YCSB-D workload instead. The YCSB-D workload is characterized as a "read latest" workload, where the most recently inserted data is accessed the most frequently. The results are similar to those of the workload YCSB-A but with a higher throughput attributed to the predominance of read operations.

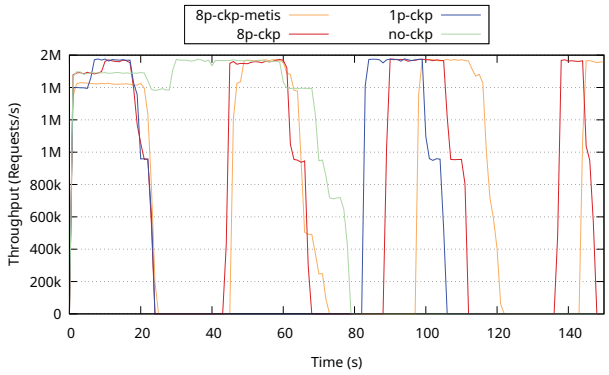

Fig. 4. Throughput over time for YCSB-D.

The most intriguing findings from the experiments emerged from the YCSB-E workload. Unlike other workloads, its operations access multiple data variables. Characterized by short-range scan operations, the YCSB-E workload simultaneously accesses more than one variable. These scan operations target

key ranges from 1 to 8, indicating the presence of operations scanning as few as one key (single access) and extending up to those scanning eight keys. Furthermore, the YCSB-E workload is predominantly read-based (95% of reads and 5% of writes).

The throughput observed for this workload is considerably lower than that of the previously mentioned workloads, as shown in Fig. 5. This decrease in performance is attributed to the increased cost associated with reading multiple variables that may be distributed across different partitions and the synchronization required among the threads assigned to the involved partitions. The figure demonstrates an increase in throughput after each execution of our checkpoint technique. The main reason for this behavior is the repartitioning algorithm, which moves variables accessed simultaneously to the same partition. While the experiment indicates that partitioned checkpoints take a similar amount of time to execute, the throughput achieved with repartitioning (*8p-ckp-metis*) experiences noticeable gains compared to the versions without repartitioning (*8p-ckp*), reaching ≈ 2.5 times more throughput.

Although *no-ckp*, *1p-ckp*, and *8p-ckp* configurations also operate with eight execution threads for partitions, they are limited to a very similar throughput level. This stems from the fact that range scans force many threads to synchronize as the variables involved in the operation are in different partitions.

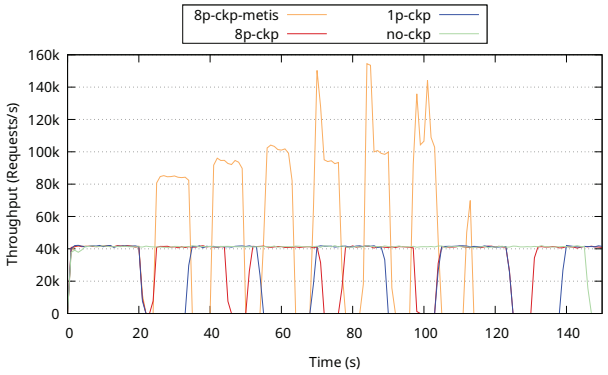

Fig. 5. Throughput over time for YCSB-E.

C. Cross-border access

In this study, we use the term "cross-border access" to denote multi-variable requests that interact with variables located in separate partitions. The experiment tracks the number of worker threads involved in the execution of the request. The number ranges from one, when all variables are in the same partition, to eight, when the eight variables accessed are in different partitions. Fig. 6 illustrates the cross-border access distribution of the YCSB-E workload performed by the scan operations.

The *8p-ckp-metis* configuration stands out as significantly different from the others. It demonstrates that most scan operations access keys within the same partition, which justifies the increased throughput for this workload as it does not need to synchronize partitions as much as the others need.

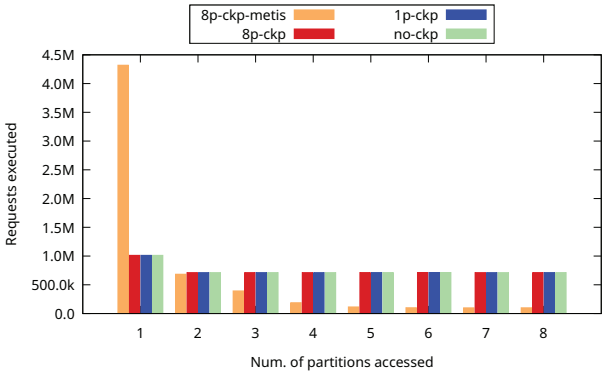

Fig. 6. Cross-border access for YCSB-E.

D. Balancing distribution

In the following results, *distribution* denotes the number of requests each partition executes, with the x-axis corresponding to the thread. Notably, the *8p-ckp-metis* configuration outshines the others in achieving a more balanced distribution for workload YCSB-A and YCSB-E, as shown in Fig. 7 and Fig. 9. Conversely, this configuration failed to balance workload YCSB-D efficiently (see Fig. 8) predominantly due to the novelty of most of the requests' keys.

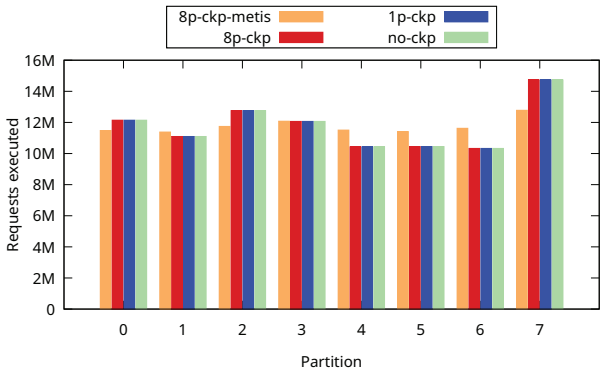

Fig. 7. Access Distribution for YCSB-A.

The unbalancing causes one partition thread to take longer to execute while other checkpoint threads are idle, waiting for that one to complete. Given that the heavier partition thread has to save more data, it negatively impacts the parallel checkpoint duration. Despite the repartitioning algorithm's efforts to distribute the requests equally, it has yet to produce a more efficient partitioning scheme. Workload YCSB-D changes the range of keys more likely to be accessed over time, representing, thus, a challenging workload for the repartitioning algorithm.

An adaptive repartitioning strategy could be employed for dealing with dynamic workloads like the YCSB-D. The system should assess the current scheme's effectiveness and only repartition if performance improvements are observed, opti-

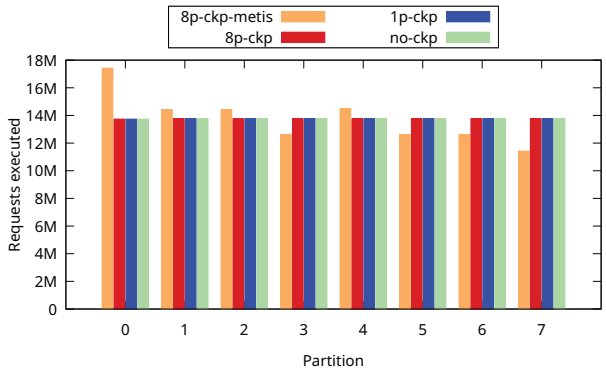

Fig. 8. Access Distribution for YCSB-D.

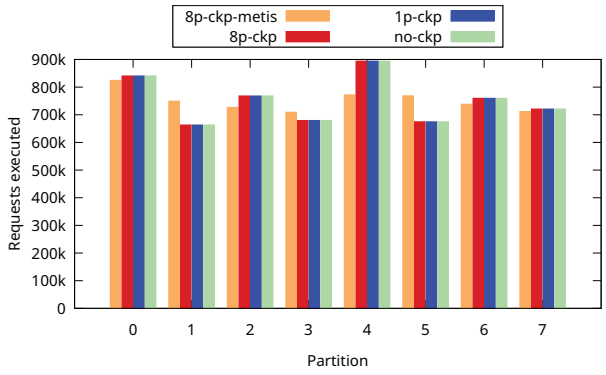

Fig. 9. Access Distribution for YCSB-E.

mizing efficiency and reducing unnecessary overhead. This approach makes the system better suited for evolving workloads.

E. Checkpoint sizes and duration

The checkpoint sizes refer to the size of each checkpoint partition, representing the average size, in GB, of all checkpoints executed by that partition during the test executions. Fig. 10, 11, and 12 present a comparison between two configurations in terms of partition sizes. The *8p-ckp* configuration uses the round-robin approach to distribute the keys among the partitions, which is the best partition scheme for state distribution and serves as a baseline. On the other hand, the *8p-ckp-metis* approach is dynamic and alternates variables from partitions according to workload.

Fig. 10 and 12 showcase the most disparate partition size distributions under the *8p-ckp-metis* configuration. This inequity in partition sizes is an unintended consequence of our partitioning approach. Our strategy prioritizes balancing variable access across partitions and minimizing synchronization disruptions from multi-variable requests while concurrently striving to maintain approximate partition balance. The horizontal lines indicate the larger partitions for *8p-ckp-metis* (orange color) and *8p-ckp* (red color). As observed, a perfectly distributed partitioning achieved by the round-robin scheme results in state partitions of around 4 GB for the workload YCSB-A, while with our approach, one partition accounts for 5GB. When considering the YCSB-E workload, we notice a partition of 1.5 GB compared to 1.1 GB from the perfect-balanced partitions.

Fig. 11 shows a better state distribution for the *8p-ckp-metis*, in light of the previously mentioned characteristics inherent to this specific workload. In order to improve further the state distribution for checkpoint threads, the repartitioning algorithm would have to consider the state size and not just the number of requests and synchronization costs, as implemented in our approach.

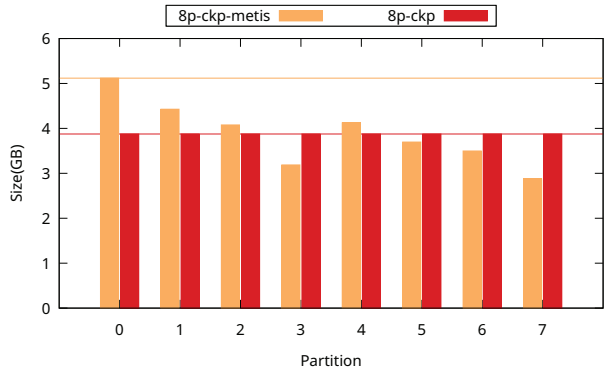

Fig. 10. Checkpoint sizes for YCSB-A.

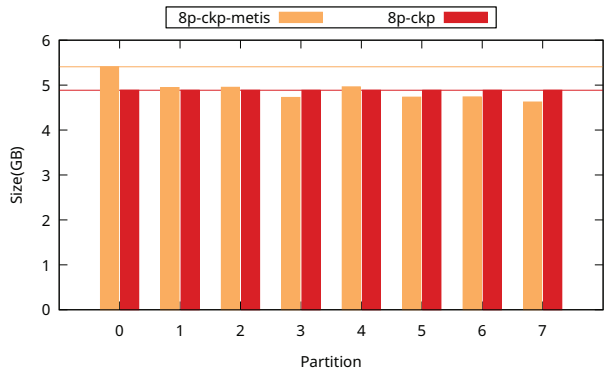

Fig. 11. Checkpoint sizes for YCSB-D.

As expected, the time taken to execute the checkpoint varies proportionally to the largest partition state size. Fig. 13 illustrates the average time required to take the checkpoint in the three workloads (YCSB-A, YCSB-D, and YCSB-E). As demonstrated in the figure, the time required for the checkpoint procedure with repartition (*8p-ckp-metis*) and without repartition (*8p-ckp*) is comparable.

Fig. 14 provides a time breakdown consisting of the time the repartition algorithm takes to execute and the time required for the checkpoint process, which includes reading the in-memory map and writing it to stable storage. It is notable that

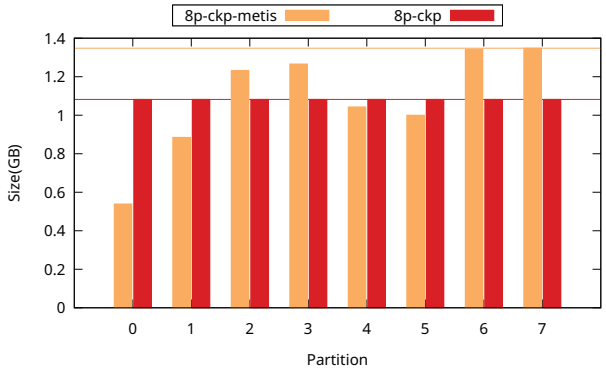

Fig. 12. Checkpoint sizes for YCSB-E.

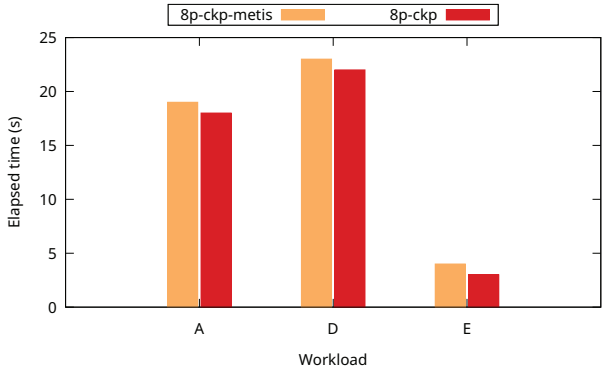

Fig. 13. Checkpoint duration for workloads YCSB-A, D, and E.

writing the checkpoint to stable storage incurs a significant I/O cost, as corroborated by the substantial sizes of the checkpoint files written, as shown in Fig. 10, 11, and 12. Furthermore, the figure supports our intuition that executing a repartition algorithm within a reasonable timeframe is feasible, given a processor that is not fully utilized while performing I/O operations. This figure is related to the workload YCSB-A but a similar behavior occurs for YCSB-D and YCSB-E.

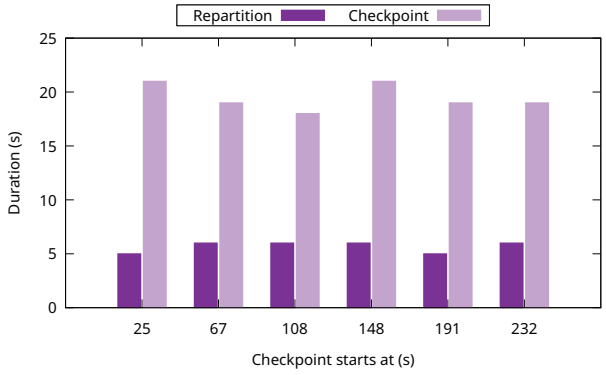

Fig. 14. Checkpoint and repartition duration for YCSB-A.

It is important to note that the duration of the checkpoints in this experiment directly correlates with the size of the key-value pairs in memory. The experiment was conducted using a key size of 4 bytes and a value size of 4kB, accounting for snapshot sizes of 32GB and individual partition sizes of around 4GB.

Another experiment was conducted, but we used 1kB values in this iteration to ascertain the potential durations of the checkpoint and repartition processes. Consequently, the sizes of the checkpoint files were smaller, which implied the short time required to read the memory and write the checkpoint file. Moreover, Fig.15 reveals that the repartition process could incur a higher cost than the checkpoint process for this configuration. At instant 71, the repartition took 1s more than checkpointing. This consideration should be factored in when deciding whether to employ a repartitioning algorithm.

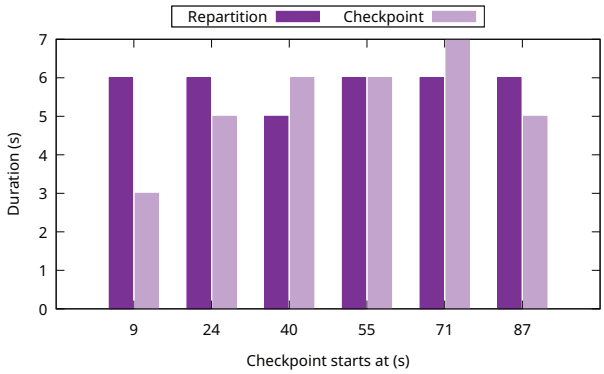

Fig. 15. Checkpoint and repartition duration for YCSB-A with keys of 1kB.

F. Makespan

This study evaluates the impact of checkpointing configurations on the *makespan*. This metric measures the time required to complete the entire execution. Thus, this analysis aims to assess the effectiveness of different checkpointing configurations in minimizing *makespan* time.

We set up all experiment configurations with the same number of requests, and the results are presented in Fig. 16 using the workload YCSB-A.

A notable observation from the figure is that the *8p-ckp-metis* configuration achieved a similar *makespan* compared to the other partitioned configuration without repartitioning. Interestingly, while Fig. 3 indicates that the *8p-ckp-metis* and *8p-ckp* configurations had similar throughput and Fig. 10 depicts an unbalancing the partition sizes when using variable partitioning, the better distribution of requests across threads compensates in the overall performance. Observe that the makespan of *8p-ckp-metis* and *8p-ckp* are practically the same in Fig. 16. Furthermore, the results demonstrate that partitioned checkpointing offers a substantial advantage over non-partitioned checkpointing regarding the time required to complete the checkpoint, directly impacting the overall application execution duration.

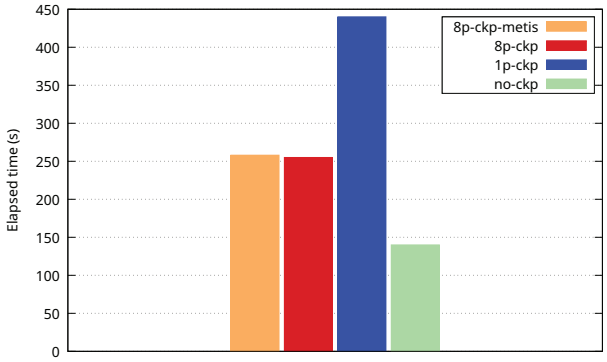

Fig. 16. Makespan for YCSB-A.

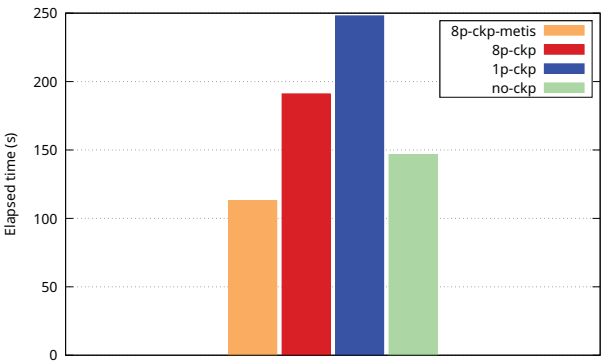

Fig. 18. Makespan for YCSB-E.

On the other hand, the *makespan* for workload YCSB-D in Fig. 17 demonstrated a low performance. As mentioned, this workload is challenging for repartitioning algorithms (see Fig. 8). The imbalance causes one partition to take longer to execute by the designated thread. In contrast, other checkpoint threads remain idle, waiting for the completion of the slower partition. Since the thread responsible for the slower partition has to process more data to be saved, it highlights the impact of unbalanced partitions on parallel checkpoints.

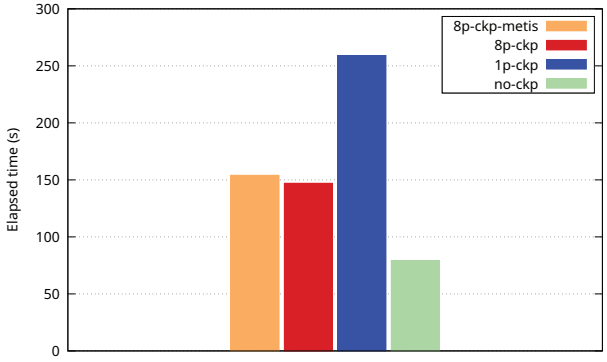

Fig. 17. Makespan for YCSB-D.

Fig. 18 illustrates a significant improvement in the *makespan* for the checkpoint configuration employing the repartitioning algorithm using the workload YCSB-E. The observed enhancement is due to a reduction in the number of synchronization operations, which consequently enables the process to reach completion more rapidly. This figure corroborates the throughput gains previously depicted in Fig. 5. As observed, even with checkpointing and repartitioning costs, our approach overcomes the makespan with a scenario without checkpointing.

V. Conclusion

Parallel checkpointing is particularly advantageous as it efficiently leverages the capabilities of modern hardware. Multiple threads can save state partitions in parallel, reducing the checkpointing execution time. Besides speeding up checkpointing through parallelism, our checkpoint procedure balances the state partitions according to the fluctuations in the workload experienced by the application.

While repartitioning algorithms can be resource-intensive, our method deliberately leverages the processing disruption caused by checkpoints and idle periods during I/O operations. We utilize these periods to tap into underutilized CPU resources for repartitioning the application state. Our studies revealed that repartitioning within this timeframe is feasible without significantly extending the total application execution time. Also, with repartitioning, requests are balanced among the partitions, and multi-variable requests become more likely to involve variables in the same partition, reducing synchronization costs.

We experimentally evaluated the impact on performance with the proposed approach. In particular, we compared our approach with a non-partitioned checkpoint and a static partitioning version. In workloads dominated by multi-variable requests, executing the repartitioning demonstrated a performance improvement that surpassed even the baseline scenario with no checkpoint execution. It means the benefits of repartitioning compensated for the costs of checkpointing. Dynamic repartitioning executed during parallel checkpointing still demonstrated a compelling performance when testing unfavorable workloads. Paper results support the intuition that it is possible to benefit from the idle processing time during checkpointing for more efficient application state redistribution.

Acknowledgments

This work received financial support of the Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina – FAPESC and Federal University of Santa Catarina – UFSC.

References

[1] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[2] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[3] O. M. Mendizabal, F. L. Dotti, and F. Pedone, "Analysis of checkpointing overhead in parallel state machine replication," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 534–537.

[4] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *USENIX Annual Technical Conference (ATC)*, 2013, pp. 169–180.

[5] F. Z. Boito, E. C. Inacio, J. L. Bez, P. O. A. Navaux, M. A. R. Dantas, and Y. Denneulin, "A Checkpoint of Research on Parallel I/O for High-Performance Computing," *ACM Computing Surveys*, vol. 51, no. 2, 2018.

[6] C.-G. Lee, H. Byun, S. Noh, H. Kang, and Y. Kim, "Write Optimization of Log-Structured Flash File System for Parallel I/O on Manycore Servers," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019, p. 21–32.

[7] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas, "ReViveI/O: efficient handling of I/O in highly-available rollback-recovery servers," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, 2006, pp. 200–211.

[8] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, 2010.

[9] A. Quamar, K. A. Kumar, and A. Deshpande, "Sword: Scalable workload-aware data placement for transactional workloads," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 430–441. [Online]. Available: https://doi.org/10.1145/2452376.2452427

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[11] J. Nishimura and J. Ugander, "Restreaming graph partitioning: simple versatile algorithms for advanced balancing," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 1106–1114.

[12] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM international conference on Web search and data mining*, 2014, pp. 333–342.

[13] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone, "Reconfiguring parallel state machine replication," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2017, pp. 104–113.

[14] L. H. Le, E. Fynn, M. Eslahi-Kelorazi, R. Soulé, and F. Pedone, "Dynastar: Optimized dynamic partitioning for scalable state machine replication," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1453–1465.

[15] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, and R. P. Liu, "Survey: Sharding in blockchains," *IEEE Access*, vol. 8, pp. 14 155–14 181, 2020.

[16] T. A. Ayall, H. Liu, C. Zhou, A. M. Seid, F. B. Gereme, H. N. Abishu, and Y. H. Yacob, "Graph computing systems and partitioning techniques: A survey," *IEEE Access*, vol. 10, pp. 118 523–118 550, 2022.

[17] B. Li, W. Xu, and R. Kapitza, "Dynamic state partitioning in parallelized byzantine fault tolerance," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2018, pp. 158–163.

[18] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004, pp. 575–584.

[19] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-verify replication for multi-core servers," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 237–250.

[20] D. E. Knuth, *The Stanford GraphBase: a platform for combinatorial computing*. AcM Press New York, 1993, vol. 1.

[21] H. Goulart, Álvaro Franco, and O. Mendizabal, "Checkpointing techniques in distributed systems: A synopsis of diverse strategies over the last decades," in *XXIV Workshop de Testes e Tolerância a Falhas*, 2023.

[22] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete problems," in *Proceedings of the sixth annual ACM symposium on Theory of computing*, 1974, pp. 47–63.

[23] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[24] P. Sanders and C. Schulz, "Think locally, act globally: Highly balanced graph partitioning," in *International Symposium on Experimental Algorithms*. Springer, 2013, pp. 164–175.

[25] S. T. Barnard and H. D. Simon, "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," *Concurrency: Practice and experience*, vol. 6, no. 2, pp. 101–117, 1994.

[26] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM journal on matrix analysis and applications*, vol. 11, no. 3, pp. 430–452, 1990.