Parallel Restore-Optimized Snapshot Compression for Cost-Efficient Inference Serving

Radostin Stoyanov*
University of Oxford
radostin@stoyanov.io

Fergus Finn
Doubleword
fergus.finn@doubleword.ai

Dan Feigin
NVIDIA
dfeigin@nvidia.com

Andrei Vagin
CRIU
avagin@gmail.com

Rodrigo Bruno
INESC-ID, IST, University of Lisbon
rodrigo.bruno@tecnico.ulisboa.pt

***Abstract*—Inference services incur cold-start costs from KV-cache profiling and CUDA-graph construction. Container checkpoints avoid repeated initialization, but large memory images consume storage and serial reconstruction delays restoration. We present PROCS, a parallel, restore-optimized compression method for CRIU snapshots. PROCS stores pages in VMA-bounded, page-addressable blocks, supports partial and incremental reads, and reconstructs bounded batches using reusable workers. It coordinates parallel restoration of private mappings, memory-backed files, and anonymous shared memory under restore-wide CPU and buffer limits. We implement PROCS in CRIUgpu and evaluate five SGLang inference models. With 256 KiB blocks and serial restoration, PROCS reduces snapshot size by 12.5–45.5% and restore-to-first-token latency by 7.2–44.7% relative to uncompressed CRIUgpu. At fixed parallelism, 1 MiB blocks reduce size by 6.2%, checkpoint time by 12.4%, and restore-to-first-token latency by 22.5% relative to 4 KiB encoding. For Qwen3.5 27B FP8, increasing decompression parallelism to 16 threads reduces CRIU's restore latency by 25.8%.**

***Index Terms*—Containers, GPU checkpointing, Memory compression, LLM Inference, CRIU.**

I. Introduction

Transparent checkpointing allows to capture the execution state of a running container and to restore it without requiring application-specific changes [1]–[3]. For GPU-accelerated workloads, system-level checkpointing combines CPU process state with device state, allowing an initialized application to resume without repeating the runtime and accelerator initialization [4], [5]. The startup performance therefore depends on the cost of loading the snapshot data from storage and reconstructing the captured execution state. Modern inference platforms serve large language models (LLMs) with billions or trillions of parameters, whose model artifacts can reach hundreds of gigabytes or even terabytes [6]. Once loaded, their weights and runtime data occupy CPU and GPU memory, increasing the amount of state included in a snapshot. Compressing these snapshots reduces storage requirements and the amount of data read during restore, but decompression adds performance overhead to the latency-critical restore path.

*Also with Doubleword.

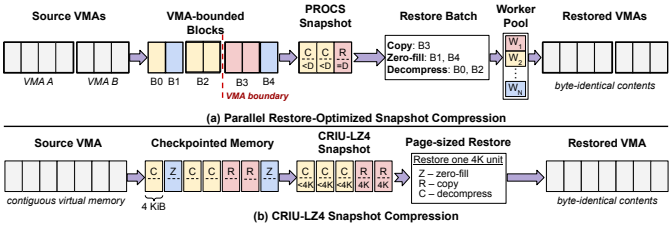


Fig. 1. Comparison of memory snapshot formats and restore paths. **(a)** PROCS groups pages into VMA-bounded blocks, combines block data into restore batches, and restores them in parallel using worker pools. **(b)** CRIU-LZ4 compresses and restores each 4 KiB page independently. Labels `C`, `R`, and `Z` denote *compressed*, *raw*, and *zero* representations, respectively.

To address this challenge, prior work introduced CRIU-LZ4 [7], which integrates compression directly into the checkpoint and restore data paths. This approach compresses each 4 KiB memory page as it is written to the snapshot and decompresses it inline during restore, eliminating a separate pre-restore decompression phase while preserving the random-access addressability required for the restore operation, incremental checkpointing, and page deduplication. However, the per-page granularity limits compression efficiency, while serial decompression limits restore throughput. Our key observation is that the unit used to compress memory need not be the unit scheduled during restoration.

We therefore present PROCS, a parallel, restore-optimized compression method for container snapshots, integrated into CRIU's memory-image path. PROCS groups pages into VMA-bounded blocks while preserving page-level addressability for partial and incremental reads. During restoration, it combines block data into bounded batches and decompresses them in parallel using reusable, caller-assisted worker pools, as illustrated in Figure 1.

PROCS further enables parallelism across the memory-restore paths by asynchronously populating private VMAs and batching shared-memory reads. It also restores memory-backed files and anonymous shared memory concurrently. A restore-wide execution budget coordinates parallel decompres-

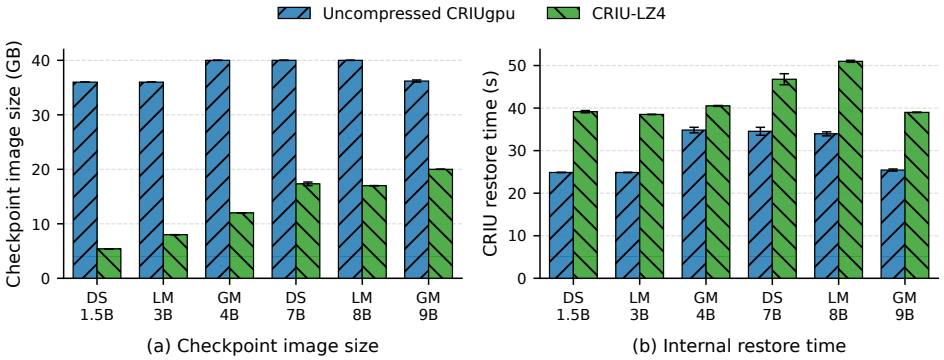


Fig. 2. Comparison between CRIU-LZ4 and uncompressed CRIUgpu for six vLLM [8] inference workloads on an NVIDIA A100 (SXM4 40GB): **(a)** checkpoint size and **(b)** restore latency. Compared to uncompressed CRIUgpu snapshots, CRIU-LZ4 reduces checkpoint image size by ≈45–85% but increases restore latency with 16.3–57.4%. Bars show means, and error bars show standard errors ($n = 3$ or 5 per model) where checkpoint images resided in memory-backed filesystem. Model labels `DS`, `LM`, and `GM` denote DeepSeek, Llama, and Gemma, respectively with suffix indicating a parameter count in billions.

sion across these paths, limiting CPU contention and buffer growth as the number of restored objects increases. This shared policy prevents individual readers from consuming the full decompression capacity. When parallel capacity is unavailable, readers apply the image semantics with serial decompression.

We evaluate PROCS on five inference models served with SGLang [9] using 256 KiB blocks and observe reductions of 12.5–45.5% in snapshot size and 7.2–44.7% in restore-to-first-token latency relative to uncompressed CRIUgpu. Its effect on checkpoint time is workload-dependent, ranging from an 11.3% increase to a 10.7% reduction. In a matched Qwen3.5-4B experiment with decompression parallelism fixed at sixteen, 1 MiB blocks reduce snapshot size by 6.2%, checkpoint time by 12.4%, and restore-to-first-token latency by 22.5% relative to 4 KiB blocks. For Qwen3.5-27B-FP8, increasing decompression parallelism from one to sixteen reduces the restore time by 25.8%.

For inference platforms, a checkpoint is restored many times when creating replicas, for example during auto-scaling [10]. In these platforms, checkpointing is performed once while its restore cost is incurred repeatedly, making the snapshot size and restore latency important optimization targets.

In summary, we make the following contributions:

- We design a VMA-bounded, page-addressable block format with block-aware access for partial and incremental restoration. Each block is stored in compressed, raw, or zero form based on its compressibility.
- We introduce a parallel memory-restore path for private virtual memory areas (VMAs), memory-backed files, and anonymous shared memory, with restore-wide worker and buffer limits.
- We implement PROCS within CRIU's checkpoint and restore data paths while preserving transparency and incremental-restoration semantics. The proposed implementation has been merged in version 4.3 [11].
- We evaluate PROCS on five inference models using end-to-end measurements and controlled studies of block size and decompression parallelism.

II. Background and Motivation

We first describe the core components of GPU-accelerated container checkpointing and CRIU's memory-image semantics. We then explore the challenges of page-granular compression using CRIU-LZ4.

A. Checkpointing GPU-Accelerated Containers

Checkpointing a GPU-accelerated container requires capturing both CPU-side process state and device state. Container runtimes enable this functionality by invoking CRIU, which allows to save and restore the process tree and operating-system resources, including virtual memory, threads, open file descriptors, and namespaces. To handle the corresponding device state, CRIU utilizes GPU plugins that integrate with vendor-specific checkpointing interfaces such as the GPU driver APIs [4].

The GPU plugins for CRIU reuse most of the existing checkpoint and restore data paths, while invoking device-specific operations through *plugin hooks*. During checkpointing, the CUDA plugin uses the GPU driver API interface to capture and save the device state host memory, which CRIU saves through its memory-image path. During restore, CRIU reconstructs this host-resident state before the plugin restores it back to the device. As GPU data, the largest component of inference snapshots, passes through the memory-image path, PROCS aims to optimize the storage and restoration of this state. gCROP [12] reduces startup latency by overlapping CPU and GPU restoration and fetching memory pages on demand. PROCS focuses on the storage and restoration of GPU data within CRIU's memory-image path, as this data forms the largest component of inference snapshots.

B. CRIU Memory-Image Semantics

To preserve the virtual-address information of a process tree during checkpointing, CRIU saves its virtual-memory ranges in *pagemap* images and stores the corresponding page contents in *pages* image files. During restore, CRIU uses the *pagemap* entries to locate pages by virtual address rather than by the

order in which their contents were written to the *pages* images. As a result, the page contents must remain addressable at page granularity during restore [7], [13].

Page-addressable access is also required across the different memory-restore paths. For instance, private virtual memory area (VMAs) are populated in individual processes, while shared mappings may be restored once and mapped into multiple processes. For incremental checkpoints, pages absent from the current image can be resolved from parent images, while deduplication relies on page-level relationships [7].

These access patterns make a single opaque compressed archive unsuitable for CRIU's restore path. CRIU would need to decompress the entire archive before locating and restoring individual page ranges. Compression within the memory-image path must therefore preserve page-addressable access to both the current and parent images.

C. Challenges of Page-Granular Compression

To preserve these memory-image semantics, CRIU-LZ4 treats each 4 KiB page as an independent compression unit [7]. While preserving page-addressable access, each page requires a separate compression invocation and metadata entry, and restore decompression is serial. Figure 2 shows that while CRIU-LZ4 reduces the snapshot size, the decompression overhead also increases the restore latency. To isolate memory-reconstruction overhead, the checkpoint images were stored in a memory-backed filesystem, minimizing storage latency. These measurements evaluate CRIU-LZ4 end-to-end and do not isolate the individual effects of page granularity and serial decompression. This experiment serves as motivation for PROCS and is not directly comparable to the evaluation in Section IV, which uses different hardware, inference workloads, and experimental configurations. We further evaluate these factors in Section IV. Together, these results highlight the key challenge of reducing snapshot size without sacrificing page-addressable access or restore performance. For instance, Sabre [14] combines hardware-accelerated snapshot compression with memory prefetching for Firecracker microVMs. Snapipeline [15] pipelines snapshot decompression, memory restoration, and function execution to reduce startup latency in FaaS containers. In comparison, PROCS targets CRIU memory images with LZ4 compression and parallel restore.

III. Parallel Restore-Optimized Compression

A. Design Overview

The PROCS design aims to reduce per-page compression and metadata overhead while preserving page-addressable and incremental reads. It must also allow independent restore jobs to run in parallel and limit the number of active decompression workers and their combined buffer memory across concurrent readers. PROCS satisfies these requirements with four core components. A page remains the logical lookup unit, while a block defines the persistent encoding unit. A batch bounds I/O and temporary storage, and a job represents reconstruction work that can be scheduled independently. This separation allows compression granularity and restoration parallelism to be controlled independently, as illustrated in Figure 3.

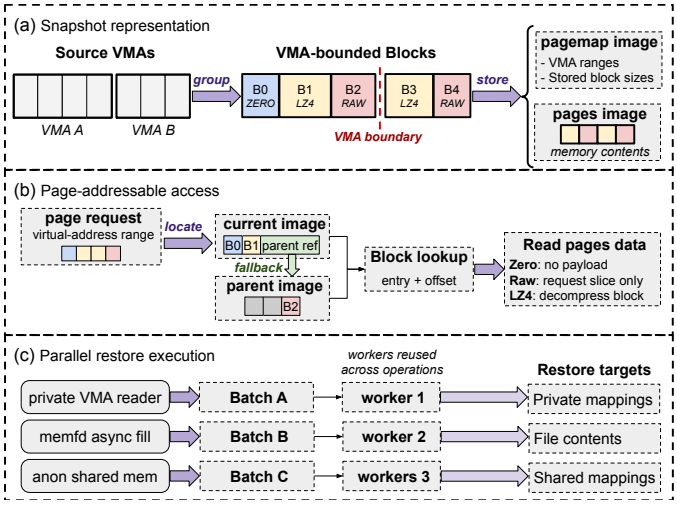


Fig. 3. PROCS separates persistent representation, page-addressable access, and restore execution. **(a)** Pages are grouped into VMA-bounded blocks stored in zero, LZ4, or raw form. **(b)** Block-aware readers translate page-range requests, including parent-image lookups, into bounded batches. **(c)** Reader-owned worker pools execute reconstruction jobs under a shared restore-wide decompression budget. Blocks determine storage, batches bound I/O and buffering, and jobs expose parallel reconstruction.

B. VMA-Bounded Block Format

PROCS groups consecutive present pages within each pagemap range into blocks of a configured size. A block never crosses a VMA boundary because CRIU prepares and restores each mapping independently. When a pagemap range or VMA ends before a block is full, PROCS stores the remaining pages as a shorter final block. This preserves CRIU's mapping semantics while allowing the compression granularity to differ from the page granularity.

PROCS stores each block in one of three forms. An all-zero block requires no payload. For a nonzero block, PROCS retains CRIU-LZ4's compression threshold [7] and stores a block in compressed form only when doing so reduces its size by more than 12.5%. Otherwise, it stores the block in raw form. This threshold avoids decompression when compression provides only a small storage reduction. Similar to CRIU-LZ4, the saved size metadata identifies the representation. A value of zero refers to a zero-filled block, a value equal to the decompressed block size denotes raw (uncompressed) data, and any smaller positive value indicates LZ4-compressed data.

C. Page-Addressable Block Reads

For each page-range request, PROCS first identifies the pagemap entry containing the requested virtual address. The page offset within that entry determines the block index and the position within the block. Since block payloads are stored consecutively in the pages image, PROCS locates a block using the cumulative stored size of the preceding blocks. As the request advances across block or pagemap boundaries, PROCS

TABLE I
EXPERIMENTAL CONFIGURATIONS. EACH CONDITION CONTAINS FIVE VALIDATED REPETITIONS.

Experiment	Platform	Workload	Configuration	Run order
Cross-model	RTX PRO 6000	Five SGLang models	Uncompressed and 256 KiB; 1 thread	Fixed
Block size	RTX PRO 6000	Qwen3.5-4B	4, 256, 512 KiB and 1 MiB; 16 threads	Fixed
Parallelism	NVIDIA B300	Qwen3.5-27B-FP8	256 KiB; 1, 4, 8, 16, and 30 threads	Cyclic rotation

updates both the virtual-page location and the corresponding offset in the pages image.

The block representation determines how the requested contents are obtained. A zero block requires no payload read and directly clears the destination pages. A raw block permits reading only the requested portion. An LZ4 block must be read and decompressed in full, after which PROCS returns the requested pages. When a request consumes only part of an LZ4 block, PROCS caches the decompressed block for subsequent requests to the same block.

For incremental checkpoints, a pagemap entry may refer to pages stored in a parent image. PROCS follows this reference and interprets the parent using its own pagemap metadata and block layout. Current and parent images may therefore use different block sizes or representations without changing page-addressable incremental restoration. For example, when a requested page is absent from the current image, PROCS resolves the corresponding parent-image reference and locates the page using the parent image's pagemap metadata and block layout. This allows incremental checkpoints to use different block sizes or representations without requiring the parent image to be re-encoded. Queued page requests are then collected into batches, as described below.

D. Bounded Parallel Reconstruction

PROCS collects queued page reads into bounded batches before reconstructing their contents. Each batch limits the payload temporarily held in memory and the number of destination pages processed at once. As a result, PROCS buffers only a bounded portion of the snapshot at a time. This is important because restoring large memory regions while GPU state is staged in host memory already consumes a significant amount of memory, and unbounded buffering would further increase peak host memory usage.

Within each batch, LZ4 blocks and sufficiently large zero-filled ranges form independent jobs that write to non-overlapping destinations. Raw blocks are copied directly by the calling thread because they require no decompression. Each top-level reader maintains a reusable worker pool that also serves its parent-image readers, avoiding repeated thread creation across batches. The calling restore thread also processes jobs alongside the workers.

A restore-wide budget limits decompression concurrency and buffered compressed data across all reader-owned pools. When parallel capacity is unavailable, the batch is processed serially. This design prevents concurrent readers from creating excessive CPU and memory demand.

E. Parallel Restoration Across Memory Objects

Private VMAs are restored within each task because their destination mappings belong to that process. This makes parallel restoration difficult because each task follows a separate memory-read path, and uncoordinated parallelism could multiply CPU and buffer demand when restoring large process trees. To expose parallelism while controlling these costs, PROCS groups compatible page-range reads into bounded batches and reconstructs independent blocks in parallel while preserving VMA and parent-image dependencies.

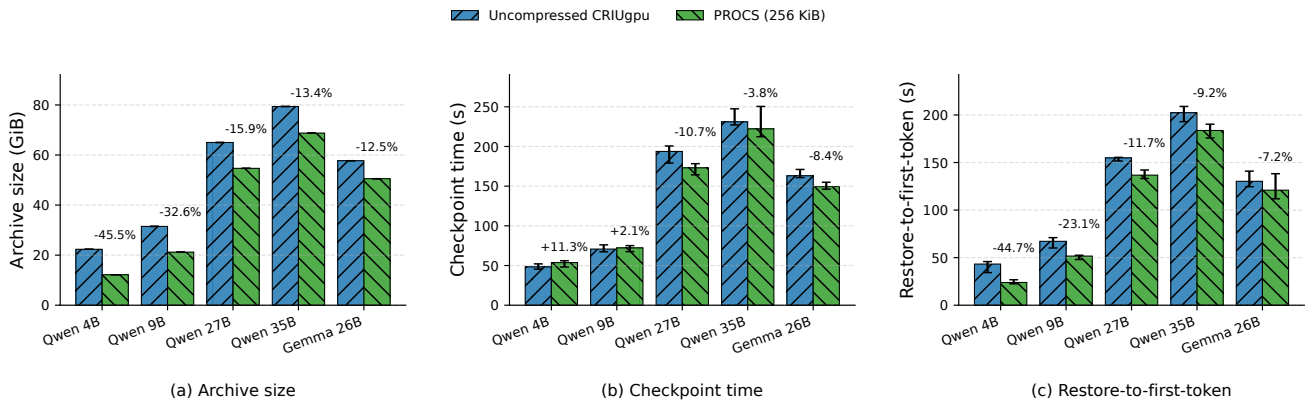
Memory-backed files and anonymous shared-memory objects follow a different path. Each object is populated once and may then be mapped by multiple processes. CRIU submits these object fills to an asynchronous daemon, allowing independent objects to be populated concurrently and to overlap with private-memory restoration.

The asynchronous daemon controls object-level concurrency independently of PROCS's decompression workers. Page readers running in the daemon still acquire capacity from the restore-wide decompression budget, so concurrent object fills do not exceed the configured decompression and buffering limits. This separates parallelism across memory objects from parallel reconstruction within each object.

F. Implementation

We implement PROCS directly in CRIU's page writer, pagemap reader, and memory-restore planner. The implementation extends CRIU's pagemap metadata and pages-image path rather than compressing a completed snapshot. This approach preserves CRIU's standard checkpoint and restore interfaces and requires no changes to container runtimes. PROCS is enabled during checkpointing with `--compress-block size`, where `size` specifies the compression block size (e.g., `256K` or `1M`). During restoration, it automatically detects the compression configuration stored in the checkpoint image, while `--decompress-threads N` controls the maximum decompression concurrency. A value of one performs serial decompression, while zero allows CRIU to select the concurrency automatically.

In the current implementation, multi-page blocks are supported only for seekable checkpoint images stored locally. The page-server and image-streaming paths continue to use page-sized compression. During restore, PROCS validates block counts, sizes, offsets, and payload bounds before allocating buffers or invoking LZ4. It supports zero, raw, and LZ4 representations across partial reads, short final blocks, incremental parent images, and as well as private and shared memory.



(a) Archive size (b) Checkpoint time (c) Restore-to-first-token

Fig. 4. End-to-end comparison between uncompressed CRIUgpu and PROCS with 256 KiB blocks across five SGLang inference models. Both configurations use one decompression thread. Bars show medians over five repetitions, error bars show deterministic bootstrap 95% confidence intervals, and annotations report changes relative to uncompressed CRIUgpu.

IV. Evaluation

We consider three questions. First, how does PROCS affect archive size, checkpoint time, and restore-to-first-token latency across inference models? Second, how does compression granularity affect these metrics when decompression parallelism is held fixed? Third, how does decompression parallelism affect CRIU's internal restore time and application-visible latency?

A. Experimental Methodology

We evaluate PROCS using three experiments across two systems, summarized in Table I. The RTX PRO 6000 experiments evaluate the end-to-end effect across models and isolate compression granularity. The B300 experiment evaluates decompression parallelism.

The RTX PRO 6000 system has 96 GB of GPU memory, 16 AMD EPYC 9335 vCPUs, and 151 GB host memory, with Linux 6.8, CUDA 13.0.1, and runc 1.5.1. The B300 system has 288 GB of GPU memory, 30 AMD EPYC 9575F vCPUs, and 269 GB host memory, with Linux 5.15, CUDA 13.3, and runc 1.3.0. Both systems use Podman 6.0.2 and CRIU 4.2.1.

The uncompressed baseline uses the same CRIUgpu revision with page compression disabled. Page-sized encoding uses 4 KiB blocks, while multi-page encoding uses blocks of 256 KiB or larger. Serial restoration uses one decompression thread, including the calling thread. Podman archive compression is disabled in every condition so that archive size reflects CRIU's online memory-image compression.

Checkpoint time begins immediately before the Podman checkpoint command and ends when the command returns. Restore-to-first-token begins immediately before Podman restore and ends when the restored server produces its first nonempty output token. This interval includes runtime restoration, server readiness, and request processing. The parallelism experiment additionally reports CRIU's internal restore time.

All experiments use tensor parallelism one, an 8,192-token context, and deterministic requests that generate at most 32 tokens. Model artifacts and checkpoint images reside on local storage, and host caches remain warm.

TABLE II
Block-size results for Qwen3.5-4B with sixteen decompression threads. Values are medians of five repetitions.

Block	Archive (GiB)	Checkpoint (s)	Restore (s)
4 KiB	12.96	58.57	27.14
256 KiB	12.17	54.19	23.62
512 KiB	12.16	52.49	24.29
1 MiB	12.15	51.30	21.05

B. End-to-End Impact Across Inference Models

We compare uncompressed CRIUgpu with PROCS using 256 KiB blocks across five SGLang models on the RTX PRO 6000. Both configurations use one decompression thread, so this experiment evaluates the effect of multi-page compression independently of parallel decompression.

As shown in Figure 4, PROCS reduces archive size by 12.5–45.5% and restore-to-first-token latency by 7.2–44.7% across the five models. Its effect on checkpoint time is workload-dependent, ranging from an 11.3% increase to a 10.7% reduction. Smaller snapshots therefore consistently improve restoration in these workloads, whereas the checkpoint-time depends on whether reduced output data size offsets the performance overhead of compression.

C. Effect of Compression Granularity

We isolate compression granularity using Qwen3.5-4B on the RTX PRO 6000. We compare 4 KiB, 256 KiB, 512 KiB, and 1 MiB blocks while holding decompression parallelism fixed at sixteen threads. Each condition contains five repetitions. The 4 KiB condition is the page-sized baseline from the same implementation.

As shown in Table II, moving from 4 KiB pages to 256 KiB blocks reduces median archive size from 12.96 to 12.17 GiB (6.1%), checkpoint time from 58.57 to 54.19 s (7.5%), and restore-to-first-token latency from 27.14 to 23.62 s (13.0%). Increasing the block size further provides little additional space reduction: the 512 KiB and 1 MiB archives are 12.16 and 12.15 GiB. The 1 MiB condition has the lowest median

TABLE III
DECOMPRESSION SCALING FOR QWEN3.5-27B-FP8 WITH 256 KIB BLOCKS. CRIU TIMES ARE MEDIANS; REDUCTIONS ARE PAIRED MEDIANS RELATIVE TO ONE THREAD.

Threads	CRIU (s)	CRIU reduction	First-token reduction
1	7.25	–	–
4	5.84	20.9%	3.3%
8	5.47	24.5%	5.2%
16	5.40	25.8%	0.9%
30	5.47	25.0%	1.1%

checkpoint time (51.30 s) and restore-to-first-token latency (21.05 s), respectively 12.4% and 22.5% below the page-sized baseline. These results show diminishing space returns beyond 256 KiB for the evaluated workloads.

D. Effect of Decompression Parallelism

We evaluate decompression parallelism using Qwen3.5-27B-FP8 on the B300 system. Within each repetition, we restore the same immutable 256 KiB-block checkpoint with limits of 1, 4, 8, 16, and 30 threads. We rotate the condition order across five repetitions. The configured limit includes the calling thread; the number of active threads may be lower when the batch contains insufficient work.

Table III shows that median CRIU internal restore time falls from 7.25 s with one thread to 5.47 s with eight and 5.40 s with sixteen. Sixteen threads produce a paired median reduction of 25.8%, while thirty provide no further improvement. The useful parallelism for this checkpoint therefore saturates between 8 and 16 threads. The phase-level improvement does not transfer proportionally to application-visible latency. Relative to one thread, restore-to-first-token improves by 3.3% with four threads and 5.2% with eight, but by only 0.9% with sixteen and 1.1% with thirty. With faster decompression, work outside the restore phase dominates the first-token latency.

V. DISCUSSION

A. Compression Benefits Are Workload-Dependent

Across the five inference models, 256 KiB blocks consistently reduce archive size and restore-to-first-token latency, but their effect on checkpoint time varies by workload. Compression adds encoding work while reducing the amount of data written, so its checkpoint-time effect depends on the memory image and storage path. Our evaluation uses local storage with warm host caches. Storage backends with different bandwidth and latency characteristics may change the trade-off between reduced I/O and compression overhead. Evaluating these effects across storage backends is left for future work.

Increasing the block size to 1 MiB reduces the archive by only a further 0.2%, although it produces the lowest checkpoint and restore-to-first-token times for Qwen3.5-4B. Larger blocks amortize the compression and metadata overhead, but they also increase the amount of data that must be decoded for a partial request and require larger temporary buffers. Thus, 256 KiB provides a practical default without requiring workload-specific tuning.

B. Phase-Level Parallelism and End-to-End Scaling

Increasing decompression parallelism from one to sixteen threads reduces CRIU's internal restore time by 25.8%. The improvement saturates between eight and sixteen threads, with no further improvement at thirty. Thus, the optimal thread count depends on the available jobs, CPU affinity, memory bandwidth, and the amount of work assigned to each thread.

Eight threads produce the largest paired median restore-to-first-token reduction, at 5.2%, while sixteen improve the interval by only 0.9%. These end-to-end changes are also less consistent across repetitions than the CRIU-phase reductions. Once decompression becomes faster, container-runtime restoration, server readiness, and request processing dominate the remaining interval. Parallel decompression therefore improves a specific restore phase but does not by itself provide proportional application-visible acceleration.

C. Restore-Path Parallelism and Resource Bounds

The implementation applies batching and parallel reconstruction to private VMAs, memory-backed files, and anonymous shared memory. It also enforces restore-wide limits on decompression threads and buffered encoded data. This separates parallelism within and across restore paths while coordinating their use of shared CPU and memory resources. The results demonstrate block-granularity and decompression-parallelism effects, but do not establish that asynchronous object restoration improves end-to-end latency as the number of memory objects grows. Likewise, the configured decompression concurrency limit does not measure CPU utilization, and the implementation's working-set bound does not measure peak host-memory consumption. Evaluating these properties requires varying the number of private mappings, memory-backed files, and shared-memory objects while recording internal restore time, CPU utilization, active threads, and peak buffered memory.

VI. CONCLUSION

We presented PROCS, a parallel, restore-optimized compression method for CRIU memory images that separates page-level lookup, block-level encoding, bounded batching, and parallel reconstruction. Its VMA-bounded format preserves partial and incremental reads, while restore-wide limits coordinate decompression across private and shared memory paths. Across five SGLang models, 256 KiB blocks reduce archive size by 12.5–45.5% and restore-to-first-token latency by 7.2–44.7% relative to uncompressed CRIUgpu, although checkpoint-time changes remain workload-dependent. At fixed decompression parallelism, 1 MiB blocks reduce archive size by 6.2%, checkpoint time by 12.4%, and restore-to-first-token latency by 22.5% relative to 4 KiB page-sized encoding. Increasing decompression parallelism from one to sixteen threads reduces CRIU's internal restore time by 25.8%, but does not provide proportional end-to-end improvement.

Acknowledgment

The authors gratefully acknowledge Viktória Spišaková, Adrian Reber, and Alexander Mikhalitsyn for their invaluable feedback on the compression functionality, and Steven Gurfinkel and Jesus Ramos for their help with introducing support for CUDA checkpointing. This work was supported in part by Doubleword and FCT under projects UID/PRR/50021/2025, LISBOA2030-FEDER-00748300, UID/50021/2025, and by the EU's Horizon Europe research and innovation programme under Grant Agreement No. 101189689.

References

[1] S. Kohli, S. Kharbanda, R. Bruno, J. Carreira, and P. Fonseca, "Pronghorn: Effective checkpoint orchestration for serverless hot-starts," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 298–316. [Online]. Available: https://doi.org/10.1145/3627703.3629556

[2] R. Stoyanov and M. J. Kollingbaum, "Efficient live migration of linux containers," in *High Performance Computing*. Cham: Springer International Publishing, 2018, pp. 184–193.

[3] D. Shukla, M. Sivathanu, S. Viswanatha, B. Gulavani, R. Nehme, A. Agrawal, C. Chen, N. Kwatra, R. Ramjee, P. Sharma, A. Katiyar, V. Modi, V. Sharma, A. Singh, S. Singhal, K. Welankar, L. Xun, R. Anupindi, K. Elangovan, H. Rahman, Z. Lin, R. Seetharaman, C. Xu, E. Ailijiang, S. Krishnappa, and M. Russinovich, "Singularity: Planet-scale, preemptive and elastic scheduling of ai workloads," 2022. [Online]. Available: https://arxiv.org/abs/2202.07848

[4] R. Stoyanov, V. Spišáková, J. Ramos, S. Gurfinkel, A. Vagin, A. Reber, W. Armour, and R. Bruno, "CRIUgpu: Transparent checkpointing of GPU-accelerated workloads," 2025. [Online]. Available: https://arxiv.org/abs/2502.16631

[5] S. Saereesitthipitak, D. Feigin, V. S. Mailthody, and M. Khadkevich, "Nvidia dynamo snapshot: Fast startup for inference workloads on kubernetes," NVIDIA Technical Blog, May 2026, accessed: 2026-08-11. [Online]. Available: https://developer.nvidia.com/blog/nvidia-dynamo-snapshot-fast-startup-for-inference-workloads-on-kubernetes/

[6] Y. Liu, H. Li, X. Huang, Y. Wang, H. Guo, H. Chen, Y. Ren, and N. Jia, "Accelerating model loading in LLM inference by programmable page cache," in *24th USENIX Conference on File and Storage Technologies (FAST 26)*. USENIX Association, 2026, pp. 117–132.

[7] R. Stoyanov, V. Spišaková, A. Reber, A. Vagin, and R. Bruno, "Towards on-the-fly snapshot memory compression for low-latency elastic inference serving systems," in *Proceedings of the 6th Workshop on Machine Learning and Systems*, ser. EuroMLSys '26. New York, NY, USA: Association for Computing Machinery, 2026. [Online]. Available: https://doi.org/10.1145/3805621.3807612

[8] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. Association for Computing Machinery, 2023, pp. 611–626. [Online]. Available: https://doi.org/10.1145/3600006.3613165

[9] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "SGLang: Efficient execution of structured language model programs," in *Advances in Neural Information Processing Systems*, vol. 37. Curran Associates, Inc., 2024, pp. 62 557–62 583.

[10] S. Saereesitthipitak, M. Abdulwahhab, H. Zhang, D. Feigin, N. Shah, M. Khadkevich, I. Neeman, V. S. Mailthody, and W.-m. W. Hwu, "Fast recovery for llm serving via decoupled device memory lifetime in dynamo," 2026. [Online]. Available: https://arxiv.org/abs/2609.25451

[11] R. Stoyanov, "Add support for memory pages compression," Jul. 2026. [Online]. Available: https://github.com/checkpoint-restore/criu/pull/2895

[12] Y. Yang, D. Du, H. Song, and Y. Xia, "On-demand and parallel checkpoint/restore for gpu applications," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, ser. SoCC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 415–433. [Online]. Available: https://doi.org/10.1145/3698038.3698510

[13] R. Stoyanov, A. Reber, D. Ueno, M. Clapiński, A. Vagin, and R. Bruno, "Towards efficient end-to-end encryption for container checkpointing systems," in *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 60–66. [Online]. Available: https://doi.org/10.1145/3678015.3680477

[14] N. Lazarev, V. Gohil, J. Tsai, A. Anderson, B. Chitlur, Z. Zhang, and C. Delimitrou, "Sabre: Hardware-Accelerated snapshot compression for serverless MicroVMs," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/lazarev

[15] Y. Lan, X. Peng, and Y. Wang, "Snapipeline: Accelerating snapshot startup for faas containers," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, ser. SoCC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 144–159. [Online]. Available: https://doi.org/10.1145/3698038.3698513