

Phoning Home: Repatriating output from cloud jobs with a user-level, provider-agnostic data transfer tool

Fabio Souza^{*†}, Daniel M.B. Sodré[†], João Paulo^{*†}, Ricardo Macedo^{*†}, Cristina Boeres[†],

Vinod E.F. Rebello[†], Felipe A. Portella[†], Paulo J.B. Estrela[†], Renzo Q. Malini[†]

^{*} INESC TEC & University of Minho – Braga – Portugal {fabio.m.souza, joao.t.paulo, ricardo.g.macedo}@inesctec.pt

[†] Instituto de Computação, Universidade Federal Fluminense (UFF) Niterói – RJ – Brazil
danielbouglex@id.uff.br, {boeres, vinod}@ic.uff.br

[‡] Petróleo Brasileiro S.A. (Petrobras) – RJ – Brazil {fabiomsouza, felipeportella, paulo.estrela, renzo}@petrobras.com.br

Abstract—HPC users have traditionally viewed renting cloud resources as being more expensive than owning on-premises resources. However, changing economics and relaxed data sovereignty laws are prompting them to explore cloud bursting. While many solutions exist for moving data to the cloud, repatriating results cost-effectively and promptly has received less attention. HPC applications generate large output data, which can lead to inefficient use of expensive cloud instances during I/O operations. This issue is exacerbated when using Spot instances in two ways: checkpoints are required to safeguard against excessive recomputation if the instance is terminated prematurely, and all pending outputs must be securely transferred within the eviction notice window. This paper presents SYNC ENGINE, a user-space, file-specific, storage-agnostic data mirroring solution loaded at job submission time that does not require modifying the application. SYNC ENGINE synchronizes outputs to on-premises storage using two mechanisms: primarily (i) LIBSYNC, a novel middleware that transparently mirrors POSIX writes to a secondary target asynchronously, overlapping I/O with computation, and (ii) RSYNC, a widely used UNIX file synchronization tool that employs a delta-transfer algorithm to minimize network traffic, being used if LIBSYNC is unable to complete its operations within the eviction window. LIBSYNC also supports policy-driven synchronization, bounded close behavior for Spot resilience, and optional streaming compression. Evaluations with production-scale reservoir simulations demonstrate that LIBSYNC with compression persists 21 GB of data to on-premises storage (5,000 miles away, from AWS US to Brazil) with only a 3% time overhead, relative to an optimal baseline. In relation to the current solution in use, it is 10% faster, 10% cheaper, and reliably writes the final checkpoint within AWS’s 2-minute spot eviction window.

Index Terms—Storage systems, Data synchronization, Spot instances, Large-scale simulations, High-performance computing

I. INTRODUCTION

The cloud computing market continues to grow, characterized by a compound annual growth rate (CAGR) of 20% [1]. The revenue and global reach of cloud providers, such as Amazon Web Services [2], Microsoft Azure [3], and Google Cloud Platform [4], give them greater negotiating power when purchasing hardware and infrastructure components, which, in turn, allows them to offer compute resources at aggressively low hourly rates. The pay-for-what-you-use cost models being offered appear well-suited to users with sporadic or light-to-medium computational demands. Intense compute users, such

as the HPC community, however, have been more reluctant to migrate entirely to the cloud for several reasons. First, the need for cloud providers (CPs) to address different user demands simultaneously can mean that a given configuration is not highly optimized for one particular user. Second, even their cheaper long-term lease contracts are considered expensive compared with operating on-premises compute facilities. Third, an unwillingness to depend on the services of a single cloud provider (*i.e.*, vendor lock-in) and instead have to harness different providers by adopting CP-agnostic solutions. Finally, while each vendor offers multiple regional data centers for resilience, the resource types, their respective availabilities and costs, all vary from region to region, adding scheduling complexity and the need to address data transfers effectively [5] [6].

Cloud providers offer significant hourly rate discounts on unused compute capacity through Spot instances. While this can be an enticing opportunity, long-running, stateful applications need to be resilient to interruptions with little warning. To safely deploy complex HPC simulations in these volatile environments, modern solutions must incorporate highly adaptive runtime mechanisms to guarantee continuous application progress without losing intermediate operational states, including automated Spot fleet scheduling across multi-region domains, predictive node eviction forecasting, and checkpointing frameworks [7], [8], [9], [10].

Some companies prefer to store their data and applications in-house due to greater control and flexibility, data sovereignty, improved security, licensing issues, reduced downtime, and potential cost savings [11]. Therefore, implementing cloud bursting [12] with on-premises storage poses a challenge to data locality optimization [13].

This study focuses on an HPC production use case that leverages a commercial oil reservoir simulator from Computer Modeling Group Ltd. (CMG) [14]. In this field, as simulation models grow in size and complexity, computational demands of simulations become unpredictable, long-running, and write-intensive, producing large output and checkpoint files that must be persistently stored on-premises to meet company and government data sovereignty policies [15]. CMG has its own vertical solution for cloud-based simulations, called CMG Cloud [16], but it focuses on ease of use through managed executions. Both versions assume that data is written locally

to the location where the computation takes place and do not directly address latency and on-premises data persistence concerns, nor the time-sensitive I/O constraints that arise when using transient resources, such as Cloud Spot instances.

This paper presents SYNC ENGINE, a storage-agnostic tool that decouples the simulator’s hot-write path from remote on-premises storage. Instead of writing directly to on-premises storage, the output and checkpoint files are redirected to a local cloud storage service and simultaneously mirrored on-premises, with the mirroring being overlapped with the application’s computation, all without any application-side changes and transparent to the user. A critical issue is not only the I/O slowdown, but also the cost of remote flush and close operations, particularly when an application is about to be preempted (from a Spot instance). Another concern is restoring execution without rolling back and recomputing a significant portion of the application, which depends on recording a consistent checkpoint on-premises within the short eviction notification window, typically 2 minutes [17].

Considering the best configurations for each on-premises persistence approach, a set of experiments shows the benefits of SYNC ENGINE and highlights that the proposed middleware LIBSYNC with compression (*LIBSYNC-C*) incurs only 3% more time to make data available on-premises in addition to in the cloud. It also reduces the monetary cost by at least 6 to 10% compared to the current version and provides resilience to Spot evictions by persisting a single final checkpoint.

We present and evaluate various write-back approaches for returning data to on-premises storage while running oil reservoir simulation jobs on production-sized geological models using AWS EC2 Spot instances. The main contributions are summarized as follows:

- A characterization of the I/O bottlenecks that arise when cloud-bursting write-intensive reservoir simulations and their outputs have to be saved to on-premises storage, including the sensitivity of close/flush behavior under preemption.
- SYNC ENGINE, a tool for efficient, resilient data transfer between cloud environments and on-premises storage systems that mirrors outputs to on-premises storage using two mechanisms:
 - **I-Rsync**, a periodic mirroring of the output files during the simulation job using *rsync*.
 - **LIBSYNC**, a new middleware that provides a design that overlaps I/O and computation.

The remainder of this paper is organized as follows. Section II introduces the problem statement and presents the challenges being pursued here. Section III introduces the SYNC ENGINE design, discussing in Subsection III-A a tool commonly used for a similar problem based on *rsync*, while Subsection III-B presents the design of the proposed mechanism LIBSYNC. Section IV discusses the experimental evaluation considering the persistence approaches specified in Subsection IV-B. Section V reviews related work and establishes connections to our work. Finally, Section VI presents the main conclusions and outlines future work.

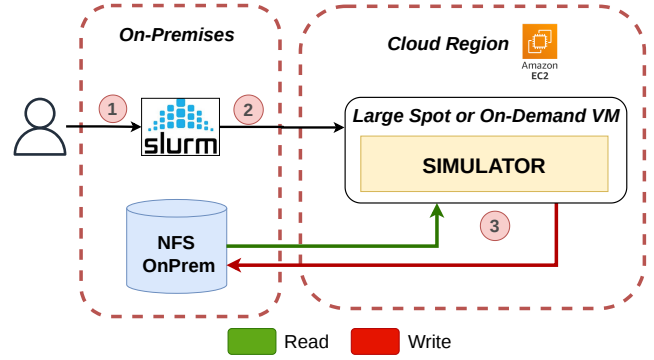


Fig. 1: Current use of HPC in the cloud.

II. PROBLEM STATEMENT

Figure 1 depicts a typical cloud-bursting workflow with its data persistence requirements, where jobs originally submitted to an on-premises HPC environment are executed in a public cloud. In our use case scenario, step-① illustrates the user submitting a simulation job to the local scheduler, in this case, Slurm [18], running on the front-end node of the on-premises HPC cluster. Instead of allocating the job to a cluster node, it can opt to provision a cloud virtual machine (VM) instance in a chosen cloud region at step-②. The reservoir simulator then runs on the allocated (Spot or On-Demand) VM, reading input data and writing simulation output files back to the shared on-premises NFS file storage system during step-③.

Operational constraints drive this design choice, since simulation artifacts must be persistently stored on-premises to meet data sovereignty requirements and to preserve continuity with existing reservoir engineering workflows (post-processing, history matching, and team-shared data access), which are centered on the on-premises storage namespace.

Although cloud bursting extends compute capacity beyond the on-premises cluster, data persistence policies couple cloud execution to on-premises storage. This coupling can become a dominant performance bottleneck, especially for regions with high RTT latencies relative to the on-premises location. While US regions are often attractive because of the earlier availability of newer instance types, lower pricing, and higher available resource capacities, these benefits can be negated by long data transfers that increase the effective runtime of jobs, in some cases making public cloud use impractical due to higher than expected monetary costs.

One way to keep cloud costs down is to leverage heavily discounted Spot VM instances, though they carry the risk of being interrupted. While cloud providers (CPs) will give prior warning of an impending VM eviction, gracefully terminating the application and returning all output files to on-premises storage within the notice period can be challenging, especially if last-minute checkpoints are used to save the application’s current state and allow it to continue execution on another VM in the future without having to recompute previous states. Given the uncertainty surrounding the frequency of

such evictions, checkpoints may need to be created frequently, increasing the application’s runtime and storage requirements. Furthermore, reservoir simulations typically use geological models with large state spaces, resulting in checkpoints in the gigabyte range that may be difficult to save within the CP’s eviction notification period.

Cloud providers (AWS, Google Cloud, and soon Azure) offer a Spot eviction notification period of up to 2 minutes. To maximize the utilization of the Spot VM, it would therefore be advantageous to use this interval to interrupt and checkpoint the application only at this moment, rather than incur repeated delays from writing checkpoint files at regular intervals during the application’s execution. While such an approach can lead to shorter execution times and lower monetary costs, it is not trivial and requires solving three challenges:

1. **Compute–Storage coupling.** Persisting outputs to a remote on-premises file system couples cloud execution to network and storage latency, turning write-intensive phases into additional wall-clock overheads.
2. **Guarantee successful checkpoint completion within the eviction notification period.** Depending on a single last-minute checkpoint is severely risky, given that the time to persist it to on-premises storage may exceed the notification period, wasting computation and forcing the application to be restarted from an earlier checkpoint, or worse, the beginning.
3. **Transparent implementation.** Solving the previous challenges without any changes to the application.

III. THE SYNC ENGINE DESIGN

This work proposes the SYNC ENGINE framework to support cloud-bursting workflows that require on-premises storage of outputs. SYNC ENGINE is designed to transparently overlap computation with high-latency remote writes, without any application modifications, regardless of the storage vendor, and to be resilient to Spot instance evictions, facilitating data synchronization between the cloud and on-premises environments.

With the SYNC ENGINE framework, the user’s (reservoir simulator) job continues to be submitted to the local job scheduler ①, which then instantiates a cloud instance ② where the job executes, with inputs read from the on-premises namespace as shown earlier in Figure 1. Write operations, however, are redirected to a cloud-local storage target to reduce the overhead of compute-storage coupling. In parallel, SYNC ENGINE propagates selected output files back to the on-premises environment in the background, through one of two unidirectional data synchronization approaches. The first leverages *rsync* [19], [20], a well-established UNIX file transfer utility. The second employs a novel middleware layer that asynchronously forwards specified file writes to a secondary storage location (in our case, on-premises storage via NFS), while the primary output path now points to cloud-local storage. Both approaches use *delta*-transfer-based strategies, one temporal and the other spatial, to make better use of network bandwidth.

To characterize the operational differences between the two approaches, we partition the write activity of a job’s execution into three phases:

Progress phase: The application (*i.e.*, simulation job) is actively executing and periodically updating its output files on cloud-local storage, while SYNC ENGINE asynchronously replicates these files to on-premises storage in the background.

Close phase: This phase refers to the period from the end of normal execution (or a graceful shutdown) until all pending writes are flushed and output files are closed. On Spot VMs, this may include persisting a checkpoint file. When an eviction notice is received, this phase is time-constrained: failure to persist the necessary data on-premises before the instance is reclaimed may prevent the job from resuming on a replacement instance. The required write throughput during this phase can be substantially higher than during the **Progress phase**.

Post-execution phase: A final verification pass ensures that all required output files have been correctly and completely mirrored on-premises. This step is necessary because expensive HPC or transient Spot cloud instances may be terminated before all write operations to remote storage have been acknowledged, leaving the on-premises file system in a potentially inconsistent state.

A. Remote synchronization with *rsync*

System administrators often need to synchronize or back up files and directories between local and remote locations without transferring entire files each time. *rsync* is a highly effective tool for this purpose since it uses a *delta synchronization* scheme to transfer only the modified portions of a file. To achieve this, *rsync* divides the two files being synchronized into corresponding pairs of chunks, computes and compares their respective checksums, and transfers only the chunks whose checksums differ. To further optimize this process, *rsync* pipelines the operations of reading files, calculating checksums, and transmitting distinct chunks to minimize the time resources, such as CPU, disk bandwidth, and network capacity, are idle. This approach has recently received renewed attention for improvements [21], particularly given its adoption for large-scale synchronization across multiple regions, similar to file transfers between cloud and on-premises infrastructures.

rsync supports two mechanisms for establishing a connection to a remote host: through a remote shell (*e.g.*, SSH, RSH), or via a direct TCP connection to an *rsync daemon*. While SSH provides a secure, encrypted channel, it also introduces computational overhead from cryptographic operations. On the other hand, using the *rsync daemon* mode requires launching a daemon process on the cloud instance that listens for incoming *rsync* requests on a designated port, potentially exposing sensitive data if not properly secured. Thus, when using daemon mode, it may be necessary to protect the connection through a VPN. Both *rsync* mechanisms were evaluated in this study. Furthermore, the following *rsync* options are enabled: archive mode, which preserves permissions, timestamps, symbolic links, and device files; and compression during data transfer to save bandwidth when necessary.

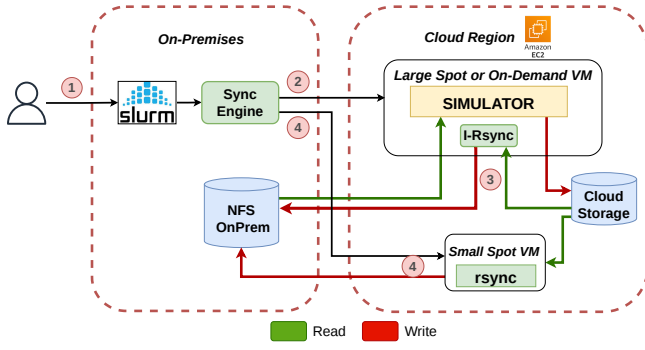


Fig. 2: SYNC ENGINE operation with I-RSYNC.

When configured to use the `rsync`-based approach, SYNC ENGINE employs two `rsync` instances at distinct points in the job lifecycle. As illustrated in Figure 2, when the job scheduler opts to create a Cloud VM (②), SYNC ENGINE configures the chosen approach to be initialized with the application on the VM. When the application begins, it enters the **Progress phase** (③), where it is configured to write or update its files directly to a cloud storage that is accessible after the VM is shut down. Concurrently with the application, SYNC ENGINE periodically runs `rsync` on these output files, synchronizing their current states with on-premises storage in the directories specified in the original job submission. In this approach, named I-RSYNC, `rsync` is executed over a TCP or SSH connection at runtime-configurable time intervals of I_{rsync} , ensuring that each invocation completes before the next is triggered.

When the application completes, *i.e.*, reaches the **Close phase**, and all output files have been closed on the local cloud storage, the VM will be shut down. If the application has not been completed and an eviction notice is received, a checkpoint is recorded during the **Close phase**. Once the application and the VM have been terminated, in the **Post-execution phase** (④), a final `rsync` synchronization is performed on a new, but smaller VM, to ensure file persistence between the on-premises and local cloud storage.

B. LIBSYNC design

LIBSYNC is a middleware that, similar to I-RSYNC, transparently mirrors file writes to secondary storage locations asynchronously during the **Progress phase**, overlapping these operations with application computation. As shown in Figure 3, the key distinction from I-RSYNC (③) is that LIBSYNC is loaded through `LD_PRELOAD` at VM startup, enabling it to intercept POSIX I/O calls issued by the application and manage both write paths simultaneously (cloud-local storage and on-premises storage) without any application modification. LIBSYNC supports a wide range of synchronization policies for the remote mirroring path, allowing the user to configure which files to synchronize, the size of each buffer, the number of buffers (*i.e.*, the size of the circular ring buffer), and the frequency of `fdatasync` operations. It additionally supports on-the-fly flush buffer compression and a **Close phase** workflow strategy that aims to minimize file synchronization

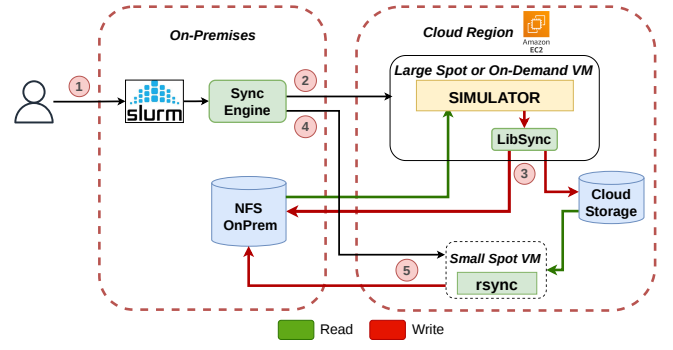


Fig. 3: SYNC ENGINE operation with LIBSYNC.

latency during this time-sensitive phase, thereby increasing resilience to Spot instance evictions. Figure 4 summarizes the end-to-end LIBSYNC workflow.

In Figures 4 and 5, the color scheme is as follows: blue denotes storage components; green denotes read data paths (in Fig. 4) and successful states (in Fig. 5); orange represents LIBSYNC policy-controlled I/O behavior; gray denotes internal LIBSYNC mechanisms; the red arrows in Fig. 4 represent writes data paths to cloud-local and on-premises storage, and; the red box in Fig. 5 indicates the safety fallback mechanism. In both figures, boxes with dashed lines indicate optional features or triggers that may or may not occur. The full set of LIBSYNC environment variables is described in Table I.

TABLE I: LIBSYNC environment variables.

Variable	Description
<code>SYNC_TARGET_PATH</code>	Mirror base directory (<i>string</i>)
<code>SYNC_FILES</code>	Extensions of files to be mirrored (<i>comma-separated</i>)
<code>BUFFER_SIZE</code>	Per-buffer capacity (<i>int in bytes</i>)
<code>BUFFER_NUM</code>	# of buffers per file - size of ring buffer (<i>int</i>)
<code>BUFFER_NUM_SYNC</code>	Sync every N buffer flushes during writes (<i>int</i>)
<code>SYNC_CLOSE</code>	Sync once at file close (<i>set 1 to enable</i>)
<code>SYNC_COMPRESS_FILE</code>	Extensions of files to compress (<i>comma-separated</i>)
<code>DEBUG_SYNC</code>	Verbose diagnostics to stderr (<i>set 1 to enable</i>)

File interception and I/O transparency. LIBSYNC intercepts `write()` and `close()` POSIX calls selectively, targeting only output files whose extensions are listed in `SYNC_FILES`. The SLURM job and its parameters remain unchanged. LIBSYNC transparently reshapes the I/O stack at the middleware layer, redefining how output files are persisted without requiring modifications to the application or to storage appliances.

Asynchronous mirroring. To decouple cloud-local execution from on-premises persistence costs, during the **Progress phase**, LIBSYNC replicates intercepted writes to a configurable on-premises target path (`SYNC_TARGET_PATH`) via a ring buffer and a dedicated per-file worker thread. The main thread first persists data to cloud-local storage and deposits write payloads into per-file ring buffers. The worker thread then asynchronously flushes the filled buffers to the target path. This design avoids blocking the critical foreground I/O path by absorbing output bursts in memory and asynchronously offloading data persistence to a worker thread, preventing high-

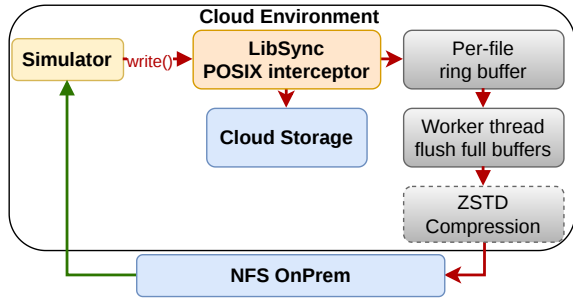


Fig. 4: LIBSYNC design.

latency remote writes from stalling the application’s critical I/O path.

Data flushing and durability. For long-running applications like simulations, LIBSYNC can enforce *periodic durability* (asynchronous safety) during normal execution by setting `BUFFER_NUM_SYNC` to N . In this mode, LIBSYNC calls the synchronization function `fdatasync` every N flushed buffers during write operations, only flushing data and the metadata needed to reconstruct it, thus speeding up the synchronization operation. The specification of `BUFFER_NUM_SYNC` offers the possibility to set the synchronization event frequency during the steady-state write phase, which can be decoupled from the write calls/operations.

Unlike the NFS protocol, which by default implements *Close-to-Open* (CTO) [22] consistency semantics that require flushing data to the server when closing a file, modern local file systems, such as ext4 [23] and XFS [24], use the *Linux kernel Page Cache* [25] to delay writes. Similarly, parallel file systems like LUSTRE use an extensive distributed locking model to keep modified data in local client caches throughout the lock life-cycle, avoiding the mandatory synchronous synchronization when closing file descriptors [26]. In this context, the `SYNC_CLOSE` feature was introduced. When enabled, LIBSYNC synchronizes the mirrored file before closing it via `fdatasync`, even if the value of N defined in `BUFFER_NUM_SYNC` has not been reached. This allows users to force synchronization of a mirrored file on close, regardless of the file system.

Close phase under eviction. When a Spot VM receives an eviction notice, all pending write operations of the *Close phase* must complete within the eviction window. Simulation workloads often produce large checkpoint files, making latency minimization in this phase critical. Figure 5 depicts the LIBSYNC *Close phase* workflow, designed to flush all remaining buffered data to the on-premises storage. Upon intercepting a `close()` syscall, LIBSYNC marks the current partial buffer as full and signals the worker thread. However, the worker thread must wait for CTO completion, which may exceed the remaining time of the eviction notification window, especially over high-latency connections.

Streaming compression. To reduce the bottleneck caused by CTO, particularly when a large checkpoint file is trans-

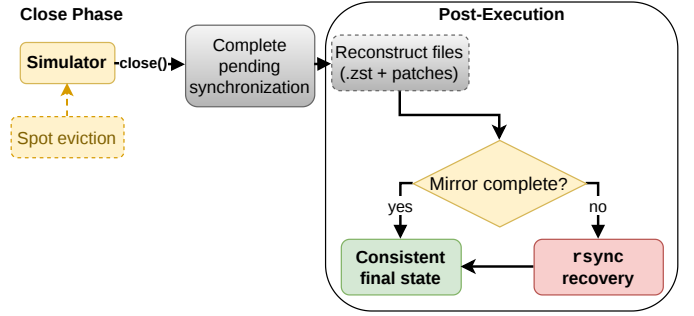


Fig. 5: Close and Post-execution phase workflow.

ferred over a high-latency connection during a Spot eviction, LIBSYNC provides a streaming compression feature that reduces the amount of data transmitted. LIBSYNC employs the Zstandard (ZSTD) compression algorithm [27], [28] that provides high compression ratios at near-memory-bandwidth speeds [29]. Its streaming compression API enables single-pass compression with bounded memory usage, making it suitable for integration into the worker flush path without buffering the entire file.

While implementing a compression feature offers a promising approach, significant concerns arise when it is applied to the checkpoint file generated by the reservoir simulator. Since this file uses a strictly non-sequential binary format, the metadata structure at the beginning of the file is updated after the data blocks are written to higher offsets. A naive streaming compressor cannot handle these backward writes without full-file buffering. LIBSYNC maintains a compression cursor tracking the highest byte offset written. At flush time, each segment is classified: forward segments ($offset \geq cursor$) are fed into the ZSTD streaming compressor and appended to the mirror on-premises file (`.zst`); backward segments ($offset < cursor$) are written as raw patch records to a lightweight sidecar file. In practice, backward writes account for fewer than 0.01% of total bytes.

Post-execution phase. Step-④ of Figure 3, represents the LIBSYNC *Post-execution phase*, where, after the simulation job terminates, two verifications are performed: (i) if compression was used, a standalone tool called `libsyntax_reconstruct` recovers the original file by decompressing the stream file on the on-premises storage and applies each patch recorded via `pwrite()` at its offset. The reconstructed file is byte-for-byte identical to the original; (ii) in the event of a Spot eviction, a *flag-file* mechanism is used to check whether the simulation’s output files were successfully closed on-premises. If the file exists, the simulation finished as expected and no further action is required; if not, like the I-RSYNC approach, an additional small VM is launched to run a `rsync` daemon to eliminate any residual differences in the files written on-premises, as shown in step-⑤ of Figure 3 and detailed in Figure 5.

SYNC ENGINE combines two complementary synchronization mechanisms in its design. I-RSYNC relies on a mature, file-level synchronization mechanism that is independent of application

I/O semantics, while LIBSYNC leverages LD_PRELOAD to overlap data persistence with computation without requiring modifications to applications or storage systems. This dual design avoids dependence on a single synchronization strategy and broadens the framework’s applicable deployment scenarios.

IV. EXPERIMENTAL EVALUATION

In this section, the integration of SYNC ENGINE with the cloud-bursting workflow and the application of its two data-persistence implementations (I-RSYNC and LIBSYNC) are evaluated to assess the benefits and limitations of each strategy, in particular, in terms of temporal and financial impact. Furthermore, given that intensive HPC users find On-demand instances relatively expensive, Spot market instances offer a strong financial incentive, as they can be obtained at as little as 10% of the cost of their On-demand equivalents. However, their transient availability can significantly affect the viability of some data persistence implementations. We investigate solutions to circumvent this situation that can benefit from using Spot cloud resources. Finally, the implications for cloud-bursting large reservoir models on Spot instances with on-premises storage via NFS are discussed.

A. Experimental Setup

All simulation jobs executed in the experiments ran on an AWS `m7i.24xlarge` instance, allocated in the `us-east-1` region. This is a high-performance, general-purpose computing instance with 96 vCPUs, 384 GiB of RAM, and up to 50 Gbps of network bandwidth. Furthermore, Hyper-threading was disabled, and only 48 physical cores were available. This instance type and configuration were selected based on internal benchmarking and workload profiling for reservoir simulation models being studied. A `c5.2xlarge` instance was chosen for persistence approaches that require the use of a small VM during their *Post-execution phase*.

For these experiments, we simulated Spot instance eviction using On-demand instances to ensure that the eviction notifications were always sent at the exact same moment. The NFS client version 3, with `rsize` and `wsize` set to 256 MB, was used to mount the on-premises namespace due to the backend storage configuration. The network connection between the cloud provider and the on-premises facilities uses AWS Direct Connect [30].

Red Hat Enterprise Linux 8.10 was used, with kernel tunings applied to the VM’s image to improve data transfer over a high-latency NFS connection. Table II lists the parameters and the values that were changed to improve TCP Auto-Tuning, TCP send/receive, TCP Congestion Algorithm Control, VM Dirty Page Cache, and RPC Slot Table.

B. Comparing Persistence Approaches

Five approaches were evaluated under identical workload conditions and the same read path from on-premises storage:

- **Baseline:** The simulator writes only to cloud-local storage, specifically to the Amazon Elastic File System (EFS). No

TABLE II: Linux Kernel Tuning Parameters.

Parameters	Values
<code>net.core.rmem_max</code>	2147483647
<code>net.core.wmem_max</code>	2147483647
<code>net.ipv4.tcp_rmem</code>	4096 16777216 1073741824
<code>net.ipv4.tcp_rmem</code>	4096 16777216 1073741824
<code>net.ipv4.tcp_congestion_control</code>	bbr
<code>net.core.default_qdisc</code>	fq
<code>vm.dirty_bytes</code>	3221225472 (3 GB)
<code>vm.dirty_background_bytes</code>	536870912 (512 MB)
<code>sunrpc.tcp_slot_table_entries</code>	128

data is returned to on-premises storage during execution. This configuration is not a viable production solution; it serves solely to define a lower bound on write overhead.

- **Current:** The current production configuration, in which each simulation job writes output directly to the on-premises storage via NFS, without the use of cloud-local storage.
- **Naive:** A naive **post-hoc copy** strategy, transferring data to the remote server with the command-line utility `scp` [31]. In this scenario, after a **Baseline** job completes, the user creates another instance in the cloud to manually `scp` the output data from the cloud storage back on-premises.
- **I-RSYNC:** In the first SYNC ENGINE approach, the simulation job writes directly to EFS (cloud-local storage); independently, `rsync` periodically synchronizes the output files in their current state to on-premises storage.
- **LIBSYNC:** In the second SYNC ENGINE approach, the middleware writes the job’s output to EFS while asynchronously mirroring the same files to the designated on-premises target, overlapping I/O with computation.

C. Experimental Methodology and Metrics

All experiments were run under the same job submission configuration, typical of the production simulation jobs run daily using the commercially licensed CMG software. In addition to its widespread use in the Oil and Gas Industry, we specifically selected the CMG reservoir simulator because it exhibits the operational characteristics that motivated Sync Engine’s design: intensive output generation, large checkpoint files, strict on-premises persistence requirements, and time-sensitivity to Spot evictions. The test job was a curtailed version of a compositional reservoir simulation of a semi-synthetic Pre-Salt model at 100×100 meter resolution, a model representative of real-world models with their high I/O intensity and large output files. The simulation jobs were executed with 40 threads, reflecting production engineers’ ideal configuration choice when submitting this particular reservoir simulation model.

The primary metric assessed is the *Data Availability Time (DAT)*, defined as the total elapsed time since the actual start of the simulation until the whole set of output files is fully available on the on-premises storage (the exception being the **Baseline** scenario, in which case, the *DAT* is simply the simulation’s local execution time). To help interpret the *DAT* values, the duration of the *Close phase* has also been measured and corresponds to the elapsed time from the end

of the execution of the simulation job until the actual closing of the files (including a checkpoint due to Spot revocation).

The evaluation was carried out under two scenarios: (i) where simulation jobs configured for the different persistence approaches (defined in IV-B) were submitted to the on-premises job scheduler for execution on the chosen cloud instance without the occurrence of evictions, and; (ii) where a simulated Spot VM eviction notice is sent to the job to initiate the simulation’s graceful termination, which includes creating a checkpoint. In both cases, the information from the job’s log file and the LIBSYNC debug file (enabled with `DEBUG_SYNC=1`) is used to determine the total execution time. Resource usage in terms of CPU, RAM, and network consumption is also reported.

The times presented for each approach are averages over five executions. In the case of the I-RSYNC and LIBSYNC approaches, their respective parameters were varied in order to find the best configuration setup, as discussed below:

I-RSYNC: For each of the two `rsync` execution modes: `SSH` and `Daemon`, three values for the time interval (I_{rsync}) were chosen: 5, 10, and 15 minutes. Each combination was also executed with and without `rsync`’s built-in `ZSTD` compression option, available since `rsync` version 3.2.0, and chosen for its superior performance compared to the `zlib` algorithm [32].

LIBSYNC: The three buffer sizes (`BUFFER_SIZE` in MB) were used: 32, 256, and 1024 MiB to demonstrate the impact of small, medium, and large buffers. Regarding the number of buffers, which defines the ring buffer size, a 2% RAM limit was set, so each configuration used approximately the same amount of ring buffer space. Given the chosen VM has 384 GiB of memory, the corresponding `BUFFER_NUM` values for each buffer were set to 246, 31, and 8, respectively.

In order to evaluate the impact of synchronization frequency using `fdatsync` in relation to when a buffer is flushed, three scenarios were considered by setting `BUFFER_NUM_SYNC` to: 0 to keep this feature disabled and allow the operating system to handle write-back operations, when set to 1, `fdatsync` is run after each buffer flush, and when set to 10, `fdatsync` is only run after the buffer has been flushed 10 times. Because the path defined by `SYNC_TARGET_PATH` uses an NFS file system, it was not necessary to use `SYNC_CLOSE` during the tests due to NFS CTO. Finally, all of the LIBSYNC tests were executed with compression, enabled only for checkpoint files by setting `SYNC_COMPRESS_FILE=rst`, and without compression.

D. Results

This section first presents the results of a comparison of the average *Data Availability Times (DATs)* for different configurations of the two SYNC ENGINE approaches, I-RSYNC and LIBSYNC. This is followed by a comparison of different persistence approaches to assess their relative performance and the overhead of persisting data on-premises.

Table III presents the DATs for each of the two SYNC ENGINE approaches, without (NC) and with compression (C). A further nine alternative LIBSYNC configurations derived from

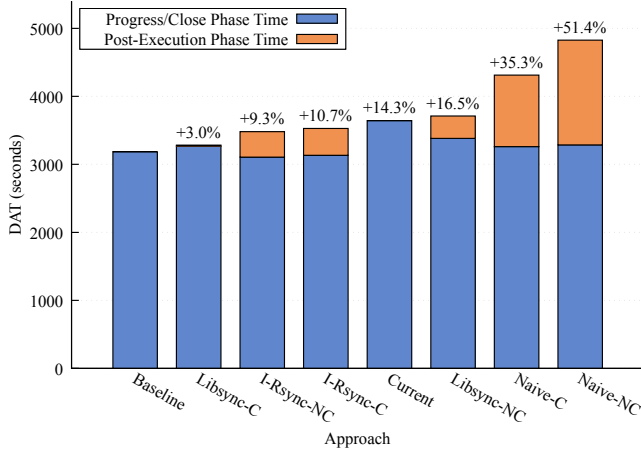
TABLE III: Data Availability Times (DATs) (in seconds) for different configurations of the two SYNC ENGINE approaches.

	No Compression (NC)			With Compression (C)		
LibSync	BUFFER_NUM_SYNC					
	0	1	10	0	1	10
32 MB	3731.2	3712.0	3822.7	3297.6	3318.0	3283.0
256 MB	3776.0	3788.2	3755.6	3377.0	3329.8	3345.8
1024 MB	3806.6	3793.8	3762.6	3370.0	3373.2	3396.8
I-Rsync	I_{rsync} (min)					
	5	10	15	5	10	15
SSH	3680.4	3632.2	3597.2	3542.4	3578.2	3600.0
Daemon	3535.6	3517.8	3481.8	3527.8	3550.2	3545.6

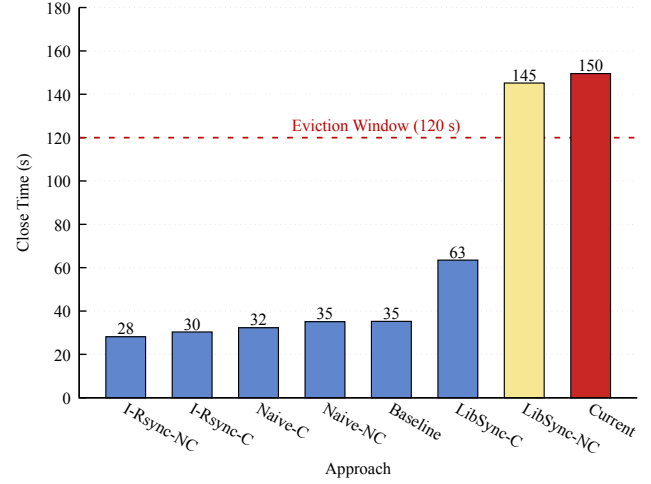
combinations of buffer sizes and buffer synchronization rates, and three I-RSYNC intervals were evaluated. The difference between the fastest and slowest configurations was relatively small (between ~ 70 and ~ 120 s), in three of the respective tests, but as much as ~ 200 s for I-RSYNC-NC. In general, smaller buffers are better, and with compression, LIBSYNC was significantly faster than without. However, for I-RSYNC-C, compression did not provide any considerable benefit in terms of DATs. In the *Post-execution phase*, most output data is already synchronized, making chunk comparison the dominant bottleneck. Furthermore, while there is a highly compressed 12 GB HDF5 [33] file among the simulation’s outputs, both `rsync` and LIBSYNC can selectively choose which file types in the `workdir` directory being synchronized should be compressed, and avoid spending time for little gain. The best configurations, highlighted in bold, were compared with other persistence approaches in the following experiment.

Figure 6(a) illustrates the DATs broken down into the respective durations of the simulation’s execution, as the sum of the *Progress phase* and *Close phase*, in blue. The orange portion of the bars is the duration of the *Post-execution phase*, if necessary. Above each bar is how much slower each approach was to execute the simulation and persist all output data on-premises, as a percentage, in relation to the **Baseline**’s cloud-only execution. Figure 6(b) presents the duration of just the *Close phase* for each respective approach in relation to AWS’s two-minute eviction notification window.

1) *On-premises Data Availability Latency:* The primary metric, *DAT*, measures the total elapsed time required to fully replicate the job’s output files to the on-premises NFS namespace. As shown in Figure 6(a), *LIBSYNC with compression* (denoted as LIBSYNC-C) incurs the lowest overhead across all approaches, being only **+3.0%** (96 s) above the **Baseline**’s lower bound. This overhead is due to the memory copies carried out by LIBSYNC when buffering mirrored writes. The next best results are produced by the I-RSYNC approaches, first with no compression (denoted as I-RSYNC-NC) then with compression (I-RSYNC-C). While both I-RSYNC solutions can match the simulation execution times of the **Baseline** approach, they both have *Post-execution phases* that incur overheads of **+9.3%** (for NC) and **+10.7%** (with compression). The **Current** approach writes directly to the on-premises NFS storage during the simulation’s execution, incurring an



(a) Data Availability Time.



(b) Close phase duration.

Fig. 6: (a) Shows the average *Data Availability Time* for each approach. Each bar portrays the *Progress phase* and *Close phase* time (in blue) with the *Post-execution phase* (in orange). On top of each bar, the percentage increase relative to the **Baseline** is shown. (b) Presents the average *Close phase* duration, where the dashed line marks the Spot eviction period.

additional **+14.3%** delay. The worst results were obtained by the **Naive** approaches, reaching up to overheads of **+35.3%** (Naive-C) and even **+51.4%** (Naive-NC) to transfer the entire volume of output post-execution. Although compression helps reduce data transfer times, these results indicate that overlapping data mirroring with computation is essential to avoid incurring additional cloud VM costs while synchronizing outputs over a high-latency path.

2) *Close phase duration and Spot-eviction resilience*: In addition to the write overhead, another crucial aspect is that executing a job on a Spot instance imposes a hard deadline upon receiving an eviction notice. To avoid losing the previous computation, the application must write a consistent checkpoint to persistent storage during its *Close phase*. It is therefore beneficial that the duration of this phase be shorter than the eviction notification period.

A comparison of the durations of the *Close phases* of the different approaches is shown in Figure 6(b). All approaches, except **CURRENT** and **LIBSYNC-NC**, completed their *Close phases* within this eviction notification period, with **I-RSYNC-C** achieving the best performance at 28 s. The approach with the slowest time, yet still safely within the **120 s** window, was **LIBSYNC-C**, with a time of **63 s**. Conversely, both **LIBSYNC** with no-compression (**LIBSYNC-NC**), in yellow, with a time of **145 s**, and the **CURRENT** approach, in red, with **150 s**, exceed the eviction notification period. While this prevents the latter from reliably persisting large final checkpoints under an eviction scenario, the **SYNC ENGINE** is able to recuperate incomplete on-premises output files, including the checkpoint, during its *Post-execution phase* using the *flag-file* and *rsync*, as shown in Figure 5. Although the **I-RSYNC** variants exhibit short *Close phase* times (28–30 s), it is important to note that the resulting *DATs* are higher than that of **LIBSYNC-C** (Figure 6(a)) since

they always create a new VM to run *rsync* in their *Post-execution phase*.

Overall, the joint interpretation of Figures 6(a)–(b) is that **LIBSYNC-C** is the only approach in the comparison that simultaneously keeps *DAT* near the **Baseline** one and meets the eviction notification period for the use case considered in this work. Nevertheless, **I-RSYNC-NC** should not be discarded.

3) *Compression benefits*: Compression is the key enabler for Spot-safe final checkpoint persistence under high latency with **LIBSYNC**. With compression enabled only for checkpoint files (*.rst*), the amount of data transferred during checkpoint mirroring drops from 3 GB to ~600 MB (an **80%** reduction). As a result, all the executions under data compression (**LIBSYNC-C**) complete the *Close phase* within the desirable period. This shift is operationally significant because it avoids relying on frequent periodic checkpoints by enabling a **single final checkpoint** to be created and persisted upon receipt of an eviction signal. Although compression is an important option for optimizing data transfer, its effectiveness naturally depends on the characteristics of the output data. Datasets with high entropy may benefit less from compression.

4) *System resource usage*: The performance evaluation of each approach requires not only measuring the execution time, but also understanding how system resources are used. CPU, memory, and network usage can influence the choice of approach, especially in shared or resource-constrained environments. To capture these aspects, a *psutil*-based [34] resource usage collection script was employed to collect the CPU, memory, and network usage of the VM during the execution of the best performing approaches from the experiments shown in Figure 6a.

Regarding CPU usage, each *rsync* call relies on a single core during its execution, while **LIBSYNC** consumes additional CPUs to compress and flush buffers. Despite these aspects, all

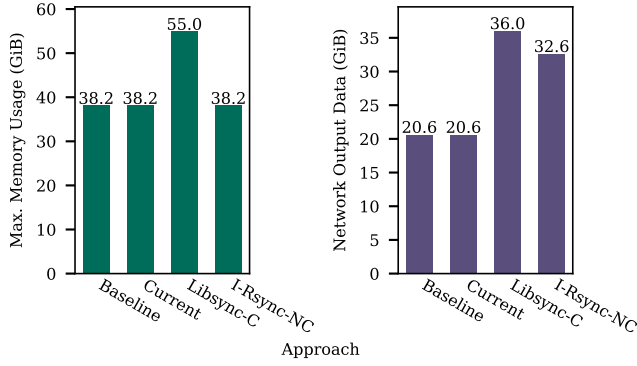


Fig. 7: Memory consumption (left) and output data transferred over the network (right) of the best two SYNC ENGINE approaches in Fig. 6, compared with **Baseline** and **Current**.

approaches used the same number of cores on average during execution. This behavior is expected, as the LIBSYNC worker threads remain idle for most of their life cycle while waiting for writes to complete, and `rsync` is only invoked at the configurable I_{rsync} intervals.

Figure 7 highlights differences in memory and network usage. In the green bars, which represent maximum memory usage, LIBSYNC-C shows the highest value at 55 GiB, compared to 38.2 GiB for the other approaches. This 16.8 GiB increase, incurred by LIBSYNC’s per-file ring buffers that are used to absorb output bursts and overlap synchronization with computation, did not pose a practical limitation for the evaluated workload. This is not a concern, since LIBSYNC’s memory footprint is configurable, allowing memory usage and buffering capacity to be adjusted to the workload characteristics, while the instance required to run the simulation provides 384 GiB of RAM. However, this may become a problem in resource-constrained environments, as LIBSYNC requires additional memory to avoid performance losses under I/O pressure.

The purple bars represent the total amount of output data transferred over the network during the execution of each approach, including data written to both cloud and on-premises storage. LIBSYNC-C and I-RSYNC-NC transfer substantially more data because both write data to both types of storage. LIBSYNC-C transfers the largest volume because it effectively mirrors all the writes throughout the job. However, it does not reach twice the **Baseline** volume due to checkpoint compression. I-RSYNC-NC, on the other hand, transfers less data because it still relies on a final `rsync` step on a separate VM to send the remaining data to the on-premises NFS storage. Note that excessive network usage may lead to performance losses under network contention.

5) *Monetary costs*: Since cloud billing is proportional to the time the VM is in use, reductions in *DAT* directly translate into lower monetary costs for the workload. Furthermore, during the **Post-execution phase** executed by SYNC ENGINE, an additional single CPU instance might be necessary. Nonethe-

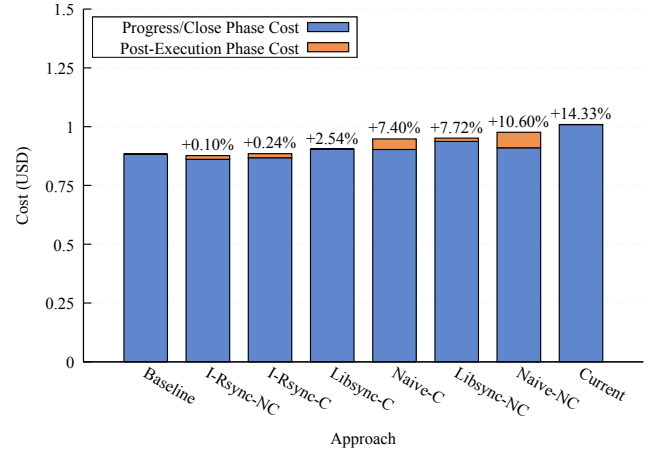


Fig. 8: Average execution cost for each approach, in USD, based on the AWS US-east-1 Spot prices at 13.40 GMT on June 2nd, 2026. The percentage above each bar indicates the cost difference relative to the **Baseline** cost of US\$0.883.

less, despite the additional cost, this is usually very cheap relative to VM instances required by the HPC application.

The cost analysis considers the cloud compute resources used during execution and persistence activities. Data egress charges were not included, as they vary substantially across cloud providers, services, contracts, and enterprise agreements. In our production environment, compute costs represent the dominant component of the total execution cost.

Figure 8 compares the total cost of executing the simulation job across the approaches under comparison. The percentage increase relative to **Baseline** is shown on top of the bars. It reveals that I-RSYNC-NC and I-RSYNC-C incur lower costs than LIBSYNC-C, even though the latter obtained the smallest overall *DAT*. This is because faster simulation execution under the I-RSYNC approaches compensates for the cost of allocating an additional VM to execute the **Post-execution phase**.

LIBSYNC-C reduces the **Current** per-simulation cost from \$1.009 to \$0.905. While this seems like a small reduction at first glance, notice that, as stated in Section IV-C, the experiments used a 1-hour curtailed version of a 12-hour simulation. To demonstrate the significance of the cost savings, we ran a single full reservoir simulation for the approaches **Baseline**, **Current**, **I-Rsync-NC**, and **Libsync-C** to estimate costs for a scenario of 1000 full reservoir simulations, a scale typical of iterative ensemble workflows using the Ensemble Kalman Filter (EnKF) [35] for history matching. Based on the results, our solution would reduce the total expense from US\$ 15016 to US\$ 14193 with I-RSYNC-NC and to US\$ 14067 using LIBSYNC-C, yielding savings of **6.3%** (US\$ 949) for standard engineering workloads, compared to **Current**. Note, however, that in practical terms, these savings represent a lower bound. SYNC ENGINE saves more time and money for long-duration simulations by eliminating the need for regular checkpoints and any lost computation incurred each time

revocations occur.

V. RELATED WORK

This section briefly reviews related research and situates our contribution within the broader context.

Cloud-to-On-Premises Data Synchronization. Managed network utilities such as AWS DataSync [36] can automate data movement between cloud and on-premises endpoints. However, AWS DataSync operates independently of the application runtime, functioning as a periodic, batch-oriented synchronization tool. Thus, it cannot mitigate the synchronous I/O blocking penalties that arise when an active application writes large payloads over high-latency connections. In contrast, the SYNC ENGINE architecture provides a runtime-level approach and transparently decouples the simulation’s compute loop from network constraints. This dual-layer approach bridges the gap between application-level runtime performance and hybrid storage synchronization, without requiring any application modifications or a dedicated virtual machine appliance, and without vendor lock-in to a single cloud provider.

Multi-tier Buffering, Data Staging and Tiering. Prior work has explored buffering, staging, and tiered placement to reduce data movement costs across heterogeneous storage layers. Hermes [37] uses multi-tier distributed buffering to absorb I/O bursts and move data asynchronously through deep storage hierarchies. In cloud settings, Mutant [38] shows that policy-driven placement across storage tiers can balance cost, latency, and durability under changing workloads. Compared with these systems, SYNC ENGINE targets a narrower, but more deployable integration point: transparent mirroring at the POSIX layer for a fixed application stack and cloud-bursting execution workflows.

Caching and Asynchronous I/O. Another approach reduces I/O overhead by using asynchronous persistence and caching close to the application. HDF5 Cache VOL [39] integrates caching and staging into the HDF5 stack and asynchronously moves data from node-local storage to the parallel file system. An HDF5 VOL connector was used as a prototype in [40] to implement asynchronous I/O using background threads, overlapping I/O operations with computation. However, these solutions are unsuitable because CMG Simulators are built with an HDF5 library version that does not support HDF5 VOL connectors. SYNC ENGINE is an asynchronous, agnostic tool that can overlap I/O operations with computation without requiring any changes to the application.

Compression in Scientific Data Paths. Compression is another effective way to reduce storage traffic in scientific workflows. Zstandard (ZSTD) [27] is a standardized loss-less compression format designed for high-throughput, data-stream-oriented use [29]. In scientific data management, compression is commonly integrated into netCDF-4/HDF5 filter pipelines [41]. SYNC ENGINE differs by applying optional streaming compression at the POSIX mirroring layer rather than within the scientific I/O library, thereby preserving transparency for proprietary simulators while reducing the volume written back to on-premises storage.

VI. CONCLUSIONS AND FUTURE WORK

This paper addresses a critical challenge in HPC cloud bursting: how to reliably repatriate large checkpoint files (GB-scale) to on-premises storage within the Spot instance eviction period specified by cloud providers, without modifying applications. Our evaluation reveals that current typical approaches fail in two ways: (1) high I/O latency reduces VM efficiency, and (2) the Spot eviction period can be insufficient to persist multi-gigabyte checkpoints. SYNC ENGINE, proposed in this work, provides a means to address both challenges simultaneously.

By comparing the current production workflow, SYNC ENGINE can actually offer a more effective balance among transparency, performance, and resilience through two mechanisms: periodic I-RSYNC and the proposed LIBSYNC mirroring middleware. In the evaluated configurations, LIBSYNC with compression (LibSync-C) delivered the best overall performance, incurring only **3%** additional *Data Availability Time* relative to a cloud-local **Baseline** time and also completes the *close phase*, including writing locally and transferring on-premises a 3 GB checkpoint file, in **63s**, which is within the AWS **120s** Spot eviction period. In addition to these performance benefits, this configuration also provided a 6 to 10% monetary saving.

Overall, the paper demonstrates that large-scale CMG reservoir models can be executed on transient cloud resources while still satisfying on-premises persistence requirements and data sovereignty constraints. In this sense, SYNC ENGINE advances the practical viability of cloud bursting for industrial reservoir simulation workloads by decoupling the simulator’s hot write path from the latency cost of on-premises storage, without requiring changes to the simulator itself.

As future work, two directions appear particularly relevant. First, integrating SYNC ENGINE with a multi-region, multi-market VM selection framework that is checkpoint-driven and supports restarts is a straightforward direction. Second, LIBSYNC can be extended with *on-the-fly client-based deduplication* to further reduce mirrored traffic and storage footprint when output or checkpoint streams contain repeated regions. Also, the compression ratio depends on the checkpoint file structure (binary, non-sequential), so other workloads may benefit differently, and further exploration is required.

As a final remark, our contribution is not merely a faster synchronization mechanism but a production-ready solution that brings us one step closer to enabling transparent, economically viable hybrid cloud-bursting HPC operations for reservoir engineering workloads under strict persistence constraints.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions and Petróleo Brasileiro S.A. (PETROBRAS) for providing access to computational resources and funding this work. This work has been carried out within the scope of the project BCD.S+M, reference 14436 (NORTE2030-FEDER-00584600), and the WISE Extra Exploratory Research Project from the UT Austin Portugal Program.

REFERENCES

- [1] Gartner, “Forecast: Public cloud services, worldwide and regions, 2022-2028, 2Q24 update,” 2024, accessed: May, 2026. [Online]. Available: <https://www.gartner.com/en/documents/5541595>
- [2] Amazon Web Services, “Amazon web services,” 2026, accessed: April, 2026. [Online]. Available: <https://aws.amazon.com/>
- [3] Microsoft Azure, “Microsoft azure cloud computing platform,” 2010, accessed: April, 2026. [Online]. Available: <https://microsoft.com>
- [4] Google Cloud, “Google cloud platform,” 2008, accessed: April, 2026. [Online]. Available: <https://cloud.google.com/>
- [5] E. Borin, L. M. A. Drummond, J. Gaudiot, A. Melo, M. M. Alves, and P. Navaux, *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*. Springer, 2023.
- [6] L. S. Serrano, M. de Lima, F. A. Portella, and L. M. A. Drummond, “A multi-cloud approach to cost optimization with AWS and Azure Fleet Services,” in *2025 IEEE/SBC 37th International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, 2025, pp. 61–68.
- [7] K. Kim and K. Lee, “Making cloud spot instance interruption events visible,” in *Proceedings of the ACM Web Conference 2024*, ser. WWW ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2998–3009. [Online]. Available: <https://doi.org/10.1145/3589334.3645548>
- [8] D. M. Sodr , L. Serrano, M. de Lima, C. Boeres, L. M. Drummond, and V. E. Rebello, “Spotting the right cloud instances with multiple AWS EC2 Fleets,” in *2025 IEEE/SBC 37th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2025, pp. 69–79.
- [9] J. P. de Araujo Neto, D. M. Pianto, and C. G. Ralha, “A fault-tolerant agent-based architecture for transient servers in fog computing,” in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 282–289.
- [10] A. Bhosale, A. Tahilyani, L. Kale, and S. Kokkila-Schumacher, “Towards an adaptive runtime system for cloud-native HPC,” 2026. [Online]. Available: <https://arxiv.org/abs/2603.14630>
- [11] S. Dasgupta, M. Gerndt, and B. Gholami, “Elastic workflows in hybrid cloud for CAE simulations,” in *2024 IEEE International Conference on Cloud Engineering (IC2E)*, 2024, pp. 251–252.
- [12] H. Gupta, “Hybrid cloud architectures and big data,” *International Journal of Advanced Research in Science, Communication and Technology*, vol. 5, no. 4, pp. 199–207, Feb 2025. [Online]. Available: <https://ijarset.co.in>
- [13] F. J. Clemente-Castello, B. Nicolae, M. M. Rafique, R. Mayo, and J. C. Fernandez, “Evaluation of data locality strategies for hybrid cloud bursting of iterative MapReduce,” in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 181–185.
- [14] Computer Modelling Group Ltd., “CMG,” accessed: May, 2026. [Online]. Available: <https://cmgl.ca>
- [15] F. A. Portella and F. M. de Souza, *Reservoir Simulation in the Cloud*. Cham: Springer International Publishing, 2023, pp. 265–282. [Online]. Available: https://doi.org/10.1007/978-3-031-29769-4_13
- [16] Computer Modelling Group Ltd., “CMG Cloud,” accessed: May, 2026. [Online]. Available: <https://www.cmgl.ca/solutions/cloud/>
- [17] Amazon Web Services, “Spot instance interruptions,” Amazon Web Services, Inc., 2026, accessed: Jun, 2026. [Online]. Available: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html>
- [18] A. B. Yoo, M. A. Jette, and M. Grondona, “Slurm: Simple Linux utility for resource management,” in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.
- [19] The Samba Team, *rsync(1) — Remote Sync*. Samba.org. [Online]. Available: <https://linux.die.net/man/1/rsync>
- [20] R. Johnson, *Rsync Solutions: Definitive Reference for Developers and Engineers*. HiTeX Press, 2025.
- [21] Z. Zhang, H. Li, L. Tang, G. Xue, J. Shu, and Y. Zhang, “[SkySync]: Accelerating file synchronization with collaborative delta generation,” in *24th USENIX Conference on File and Storage Technologies (FAST 26)*, 2026, pp. 461–476.
- [22] S. Shepler, “NFS Version 4 Design Considerations,” RFC Editor, RFC 2624, jun 1999. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc2624.txt>
- [23] M. Cao, S. Bhattacharya, and T. Ts’o, “Ext4: The next generation of Ext2/3 filesystem,” in *2007 Linux Storage & Filesystem Workshop (LSF 07)*. San Jose, CA: USENIX Association, Feb. 2007. [Online]. Available: <https://www.usenix.org/conference/2007-linux-storage-filesystem-workshop/ext4-next-generation-ext23-filesystem>
- [24] A. Sweeney, “Scalability in the XFS file system,” in *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*. San Diego, CA: USENIX Association, Jan. 1996. [Online]. Available: <https://www.usenix.org/conference/usenix-1996-annual-technical-conference/scalability-xfs-file-system>
- [25] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. Sebastopol, CA, USA: O’Reilly Media, 2005.
- [26] M. Moore, P. Farrell, and B. Cernohous, “Lustre lockahead: Early experience and performance using optimized locking,” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, p. e4332, 2018.
- [27] Y. Collet, “Zstandard: Fast real-time compression algorithm,” 2016. [Online]. Available: <https://facebook.github.io/zstd/>
- [28] G. Senniappan and R. Lourdasamy, “Bridging the gap between traditional and contemporary compression algorithms: Analytical insights from LZ77 to Zstandard,” in *7th International Conference on Innovative Data Communication Technologies and Application (ICIDCA)*, 2025, pp. 1374–1379.
- [29] Y. Collet and M. Kucherauw, “Zstandard compression and the ‘application/zstd’ media type,” RFC Editor / IETF, Tech. Rep. RFC 8878, Feb. 2021. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc8878.pdf>
- [30] Amazon Web Services, “What is AWS Direct Connect?” Amazon Web Services, Inc., 2026, accessed: Jun, 2026. [Online]. Available: <https://docs.aws.amazon.com/directconnect/latest/UserGuide/Welcome.html>
- [31] The OpenBSD Project, *OpenSSH scp (Version 9.8)*. [Online]. Available: <https://openssh.com>
- [32] J.-I. Gailly and A. Mark, “Zlib: A massively spiffy yet delicately unobtrusive compression library,” 1995. [Online]. Available: <https://zlib.net/>
- [33] M. Folk, A. Cheng, and K. Yates, “HDF5: A file format and I/O library for high performance computing applications,” in *Proceedings of supercomputing*, vol. 99, 1999, pp. 5–33.
- [34] G. Rodola, “Psutil - process and system utilities for python.” [Online]. Available: <https://psutil.readthedocs.io/stable/>
- [35] A. A. Emerick and A. C. Reynolds, “Ensemble smoother with multiple data assimilation,” *Computers & Geosciences*, vol. 55, pp. 3–15, 2013. [Online]. Available: <https://doi.org>
- [36] Amazon Web Services, “AWS DataSync: Automated, Accelerated Online Data Transfer,” Amazon Web Services, Inc., 2024, accessed: May, 2026. [Online]. Available: <https://aws.amazon.com/datasync/>
- [37] A. Kougkas, H. Devarajan, and X.-H. Sun, “Hermes: A heterogeneous-aware multi-tiered distributed I/O buffering system,” in *Proceedings of HPDC ’18*, 2018. [Online]. Available: <https://dl.acm.org/doi/epdf/10.1145/3208040.3208059>
- [38] H. Yoon, J. Yang, S. F. Kristjansson, S. E. Sigurdarson, Y. Vigfusson, and A. Gavrilovska, “Mutant: Balancing storage cost and latency in lsm-tree data stores,” in *Proceedings of SoCC ’18*, 2018. [Online]. Available: <https://doi.org/10.1145/3267809.3267846>
- [39] H. Zheng, V. Vishwanath, Q. Koziol, H. Tang, J. Ravi, J. Mainzer, and S. Byna, “HDF5 Cache VOL: Efficient and scalable parallel i/o through caching data on node-local storage,” in *Proceedings of IEEE/ACM CCGrid*, 2022. [Online]. Available: <https://doi.org/10.1109/CCGrid54584.2022.00015>
- [40] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, “Enabling transparent asynchronous I/O using background threads,” in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2019, pp. 11–19.
- [41] X. Delaunay, A. Courtois, and F. Gouillon, “Evaluation of lossless and lossy algorithms for the compression of scientific datasets in netCDF-4 or HDF5 files,” *Geoscientific Model Development*, vol. 12, no. 9, pp. 4099–4113, 2019. [Online]. Available: <https://gmd.copernicus.org/articles/12/4099/2019/gmd-12-4099-2019.pdf>