exSIF: A Portable, Self-Contained Extension to Apptainer for Big Data Containers

Puttaranun Boonchit Shivani Hukkeri Jack Yu Anmol Gupta

I. Abstract

Big data systems require direct access to low-level hardware, yet traditional container frameworks impose resource isolation that hampers their performance. Apptainer addresses these challenges but depends heavily on pre-installed runtimes and environment configurations, which may not be universally available. We present exSIF, a self-contained extension of the Apptainer SIF format, which includes an embedded Apptainer runtime and bootstrap mechanism. exSIF enables users to execute containerized big data applications seamlessly on any Linux-based host that supports Python and GNU Coreutils. We evaluated exSIF using popular big data frameworks (Hadoop, Spark, Flink, and Kafka), demonstrating that our approach incurs minimal startup and memory overhead, particularly after initial caching. Our evaluation shows that exSIF effectively simplifies dependency management and administration, significantly reduces compatibility issues, and offers performance close to standard Apptainer containers after the initial extraction. This makes exSIF ideal for containerized big data deployments in shared cloud and HPC environments.

II. Introduction & Motivation

Big data systems struggle in containerized environments due to their need for low-level hardware access like GPUs, RNICs, and HBAs for optimal computing performance. Existing container frameworks such as Kubernetes and Docker enforce strong resource isolation, which impacts performance [1][2]. Disabling this isolation demands expertise in both the container runtime and the big data system, posing scalability challenges for organizations managing numerous in-house applications across various frameworks.

Big data systems often face compatibility issues, relying on hundreds of libraries with strict version requirements. For example, CUDA and NVIDIA drivers, allow only one installed version at a time. Managing these dependencies demands significant administrator effort, especially when multiple systems must coexist and interoperate.

To address these challenges, tools like Apptainer, formerly Singularity, have been designed for running containerized Big Data systems in HPC environments [3]. Users can start a rootless container by running a binary file, enabling native driver access. This approach isolates systems and their dependencies while maintaining relatively good performance.

However, Apptainer's niche use limits its awareness among system administrators, risking misconfiguration or unavailable on all clusters. To address this, our goal is to extend the Apptainer SIF format with a self-extracting runtime. This new format, exSIF, enables SIF applications to run on any system without administrator intervention.

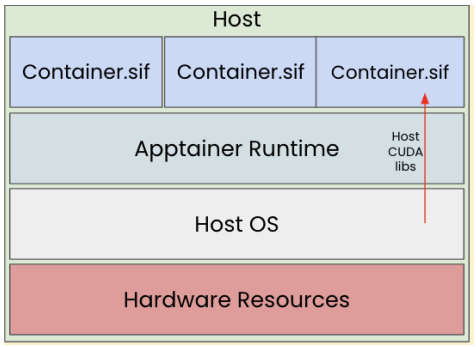

Fig. 1. High-level view of Apptainer running on host system

III. Background

Apptainer is a container runtime that simplifies running containers requiring low-level system access in HPC environments. Containers are binary files containing a system image with one or more applications. With Apptainer installed, users can start a container by executing the binary container file, opening a shell with their home directory mounted inside. This allows users to launch containerized applications without managing container sharing permissions or configuration commands.

Figure 1 depicts how Apptainer containers integrate with the host system. From the lowest level, hardware resources and the host OS provides the foundational environment. On top of that, the Apptainer runtime manages container execution and mapping resources like host CUDA libraries. Each container is a single `.sif` file that bundles applications and libraries. By relying on the host's kernel and drivers, Apptainer achieves native hardware performance, important for big data and HPC workloads. This architecture enables secure, rootless containers with efficient hardware access for users.

IV. System Design

The system design for the project, called exSIF (extended SIF), is based on the existing Apptainer SIF binary format.

A traditional Apptainer SIF file contains a filesystem image and metadata for running with low-level hardware access. However, this depends on a properly configured Apptainer installation, which may be missing or misconfigured on some hosts. To solve this, exSIF embeds its own Apptainer runtime copy as well as a small "bootstrap" mechanism so that it can run anywhere Python and GNU Coreutils are available.

Unlike regular SIF files that rely on the Apptainer runtime, exSIF launches a runtime management daemon via an interpreter. The daemon first checks for a compatible Apptainer runtime on the host. If so, it uses the existing runtime;

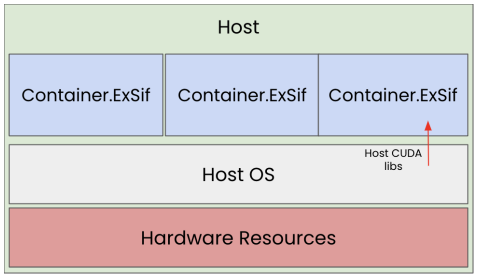

Fig. 2. High-level view of Apptainer with exSIF running on host system

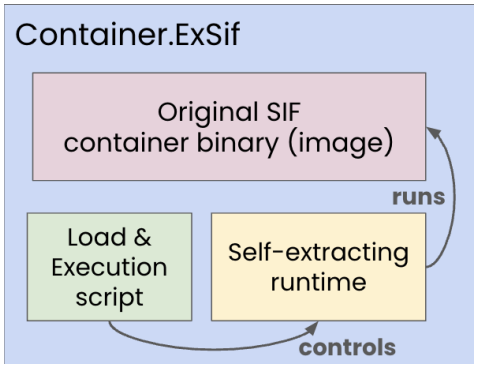

Fig. 3. Internal structure of exSIF container

otherwise, it extracts its own embedded Apptainer runtime to a ramfs. Once a runtime is available, exSIF runs the main container image, providing the same interface as a standard Apptainer container. By embedding the runtime and core dependencies, exSIF ensures that container can run without special privileges or complex setup, even on systems without Apptainer installed.

Figure 2 illustrates how exSIF containers interact with the host system in high-level. Unlike a standard setup as Figure 1, which relies on an external Apptainer runtime, exSIF removes this dependency. Each `Container.ExSIF` runs directly on the host OS, accessing hardware resources like GPUs via host CUDA libraries while maintaining portability across systems.

V. Implementation Details

Our exSIF implementation divides the container design into three components, as shown in Figure 3. Additionally, we provide a build process that automates the creation of exSIF container images from standard SIF container images.

1) Loading & Execution Script: These components include the runtime management daemon and a bootstrap script to start it. Both are stored at the beginning of the exSIF file, with the daemon as a python script and the bootstrapper as a simple shell script that primes the runtime management daemon.

The separate bootstrapper is necessary because the Python interpreter parses syntax before executing commands. If binary data is embedded in the same file, CPython interpreter raises a syntax error, even if the data isn't executable. The shell script one use pure POSIX sh to filter out the binary data before anything is handed off to the python interpreter.

The runtime daemon verifies if a functioning Apptainer installation exists and extracts the embedded runtime to RAM if needed. The runtime remains active until all containers have stopped, after which RAM is reclaimed. Tracking containers is challenging since buggy applications or scripts may terminate containers uncleanly. Our solution is to rely on the OS's file descriptor management. Each container spawns a child process with a Unix domain socket connection to the daemon. When the container exits, the child process automatically terminates, releasing its resources. Since the socket's other end is in the daemon, this triggers a notification, allowing the daemon to track ongoing containers runtime usage.

For runtime daemon management, we assign one runtime daemon per user running containers. So that this approach allows ramfs usage to be tracked via Cgroups, which are commonly used to manage resources in HPC and shared cloud environments.

2) Embedded Apptainer Runtime: If Apptainer is already available, exSIF bypasses runtime extraction. Otherwise, it extracts the embedded runtime into RAM during daemon initialization. This statically compiled Apptainer runtime runs in rootless mode, eliminating the need for a suid bit, like in Docker.

3) Container Filesystem Image (SIF Data): The main filesystem image, identical to a regular SIF file, is bundled at the end of exSIF. It includes the big data applications, libraries, and dependencies needed by the user's workload. This image is also extracted to RAM before execution.

Since extracting a large binary blob to RAM is still inefficient, we utilize the container runtime daemon to deduplicate these images. Each image is indexed in a ramfs by its sha256 checksum, allowing multiple container files with different names to reuse identical binary data. On the first launch, the container is extracted to RAM and cached, so that the subsequent launches under the same daemon reuse the same image.

To prevent corrupted extracted image data in RAM from early exSIF container termination, we add an extra checksum step to the image cache. If the exSIF file itself is valid, any partially extracted or modified images are detected and overwritten by the newly launched container.

A. Build Process

Generating exSIF images from SIF images is automatically handled by a shellscript, by detecting the component length, concatenating them into a single image file, and then configuring baked length offsets for the bootstrapper script to extract the correct sections. The detailed steps are as follows:

1) First, build or obtain a normal Apptainer SIF image (using standard Apptainer commands).
2) `make_exsif` makes the SIF file, injects the compiled Apptainer runtime, and prepends the bootstrap script and Python code to form a single self-extracting binary.
3) At that point, it can be distributed and run on any Linux host with Python 3.6+ and basic core utilities.

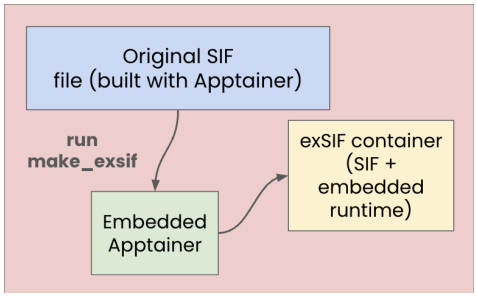

Fig. 4. Build and Run Flow

B. Execution Flow

When a user runs `./my_container.exsif`, the following happens:

1) Bootstrap Script: The system invokes the bootstrapper script (`bootstrapper.sh`) from the executable file `./my_container.exsif`. The script extracts values like `SCRIPT_LEN`, `DAEMON_LEN`, and `RUNTIME_LEN` from its contents, then hands off execution to the embedded Python daemon (`daemon.py`) with the required parameters.

2) Check for Existing Runtime: `daemon.py` first attempts to connect to an existing runtime daemon via a Unix socket. If another exSIF container is already running, it tries to reuse its runtime; otherwise, it will start a new one.

3) Handling Runtime Availability: If an exSIF runtime is already running, the client process (`daemon.py`) gets the runtime path, unpacks the container image, and executes it. Otherwise, it forks a new runtime daemon, which extracts the embedded Apptainer runtime from the exSIF file to a temporary directory, and then binds to the socket for future launches.

4) Unpacking the Container Image: If the container image has not already been unpacked, `daemon.py` extracts it, and uses the unpacked runtime to execute the container.

5) Container Execution: The runtime binary is executed with the extracted container image, launching in a rootless mode while retaining direct access to host hardware.

6) Handling Multiple Containers: The runtime can be shared across multiple exSIF instances. When all containers exit, the runtime daemon automatically shuts down, freeing up resources.

C. Code

The source code for this project is available on GitHub: https://github.com/xythrez/exSIF/ [4]. Detailed comments have been included for clarity.

VI. Evaluation

A. Overview & Setup

To assess the viability and performance impact of embedding a container runtime within each exSIF file, we evaluated our approach using four popular Big Data frameworks: Hadoop [5], Spark [6], Flink [7], and Kafka [8]. We measured the container image sizes and startup times under three distinct scenarios with container checksums disabled. We define startup time as the amount of time that is needed to setup the container before invoking the launch command of the big data system. **This does not include the startup time of the system that is being run.**

1) Standard SIF launch: Apptainer installed on the system.

2) exSIF first launch: When the embedded runtime was not yet extracted/cached.

3) exSIF repeated launch: Subsequent launches after the embedded runtime has already been extracted.

Our experimental testbed is as follows:

- Processor: AMD 7800X3D (16 HW Threads@5GHz, 96MB LLC)
- Memory: 64GB DDR5 6000MT (2 memory channels)
- Storage: 2TB Samsung 990 Pro PCIe 4.0 m.2 SSD

While apptainer is able to handle containers that require native CUDA driver access, we did not include this in our evaluation as we did not have any NVIDIA GPUs available. However, since we did not modify how apptainer operates we believe our results are generally applicable to containers that require native driver access as well.

B. Results

Table I and Table II summarize the container sizes and launch times for each framework using standard SIF and exSIF containers.

System	SIF Image Size	ExSIF Image Size
Hadoop	1.1GB	1.1GB
Spark	506MB	546MB
Flink	551MB	591MB
Kafka	205MB	245MB

TABLE I
COMPARISON OF SIF AND EXSIF IMAGE SIZES ACROSS DIFFERENT SYSTEMS.

System	SIF Launch Time	ExSIF First Launch Time	ExSIF Repeated Launch Time
Hadoop	0.87s	0.802s	0.104s
Spark	0.082s	0.470s	0.102s
Flink	0.569s	0.982s	0.585s
Kafka	0.087s	0.278s	0.105s

TABLE II
COMPARISON OF SIF AND EXSIF IMAGE LAUNCH TIMES ACROSS DIFFERENT SYSTEMS.

C. Analysis

1) Image Size: For Spark, Flink, and Kafka, exSIF is about 40 MB larger on average than the original SIF. This overhead is due to Apptainer's runtime and associated libraries within the container file. For Hadoop, there was effectively no measured increase in size (both are 1.1 GB). The exSIF overhead in this case was negligible compared to Hadoop's base image size (Hadoop's base image + large libraries + components sometimes size up to more than 1 GB).

2) First Launch Time: For Spark, Flink, and Kafka, the first exSIF launch is slower than the standard SIF launch. This is expected, since exSIF must extract its embedded runtime to a temporary location.

3) Repeated Launch Time: Once the runtime is extracted, subsequent exSIF launch times are considerably faster or about the same as the standard SIF launch time. For Hadoop, repeated launches were much faster than SIF. For Spark, Flink, and Kafka, repeated launches were close to or slightly above the baseline SIF times. This shows how caching the extracted runtime is beneficial for multiple launches.

D. Scalability

Evaluation results show that caching the extracted runtime greatly improves performance in multi-container scenarios. While the first container launch is slower due to runtime extraction, but it does most of the heavy lifting, allowing subsequent launches to be nearly instant. For instance, in a Spark job launching multiple containers on the same host, only the first container will face a delay while others will be fast.

More importantly, RAM savings can be substantial for workloads with numerous container processes, like MPI and Hadoop. In the case where each core is used to run a separate Hadoop node, there is a fixed 1.1GB cost in memory usage instead of 1.1GB for each container. In multi-terrabyte RAM systems, this is negligible, however, linear scaling of memory requirements is not.

VII. Impact & Limitations

A. Impact/Benefits

The biggest impact exSIF can have in the real world is its ease of use and compatibility. By packaging the container runtime alongside the application, exSIF eliminates the need for users to install Apptainer or Docker. They can just run a single executable container file which makes using containers and deployment much more accessible. Its supports for more advanced hardware features makes it ideal for specialized big data workloads.

For systems that already have Apptainer installed, exSIF can minimize oveerhead by skipping runtime extraction. Caching the runtime further accelerates repeated container launches on the same host, benefiting short-lived processes that require frequent launches, but perform fast once there is a runtime in place. As an extension of Apptainer, exSIF maintains security and containerization guarantees as Apptainer. Thus, it ensures privacy for HPC and big data features.

B. Limitations/Tradeoffs

Despite these strengths, exSIF has some limitations. The biggest one being that there is a noticable time overhead for hosts that do not have Apptainer installed since exSIF needs to first extract its runtime before launching the container. Depending on the use pattern, this may or may not be a tradeoff worth taking. An application that launches single microsecond-level containers before terminating may find our approach inefficient. Since exSIF relies on a rootless container model, some HPC or big data systems might need higher privileges which exSIF would not be able to provide without extra steps taken by the administrator. One common case is various network simulators that rely on Linux's qdisc mechanism. This is also a limitation that users would need to consider. Users will also need to ensure the host systems satisfy exSIF's basic requirements of having the required Linux commands and Python 3.6+. For highly specialized environments, this might require extra steps and is a tradeoff users would need to make to be compatible with exSIF.

VIII. Conclusion

The exSIF container format successfully addresses important usability and compatibility challenges faced by traditional container runtimes and simplifies the deployment of big data systems. By embedding the container runtime into the executable file, exSIF makes managing dependencies simpler and reduces administrative overhead. This makes it more accessible and easy to use. Although initial startup overhead can be a limitation on hosts without Apptainer already installed, the caching logic reduces this overhead in repeated launches. Overall, exSIF is a practical solution for containerizing complex big data applications since it balances easiness to use, security, and performance effectively.

IX. Member Contribution

All team members participated in initial project architecture discussion and intermediate project meetings, providing feedback and identifying issues. In addition to design, testing, and presentation, each of the members also contributed to the follows components in distinct ways:

- **Jack Yu:** Runtime daemon and build script automation
- **Puttaranun Boonchit:** Container checksum, corruption detection, and diagrams
- **Shivani Hukkeri:** System runtime detection and extraction bypass
- **Anmol Gupta:** Full component integration, documentation, report, and diagrams

References

[1] [Online]. Available: https://docs.docker.com/get-started/docker-overview/
[2] [Online]. Available: https://kubernetes.io/docs/concepts/overview/
[3] [Online]. Available: https://apptainer.org/docs/user/latest/index.html
[4] [Online]. Available: https://github.com/xythrez/exSIF/
[5] [Online]. Available: https://hadoop.apache.org/docs/r3.4.1/hadoop-project-dist/hadoop-common/ClusterSetup.html
[6] [Online]. Available: https://spark.apache.org/docs/latest/quick-start.html
[7] [Online]. Available: https://nightlies.apache.org/flink/flink-docs-stable/docs/try-flink/local_installation/
[8] [Online]. Available: https://kafka.apache.org/quickstart