GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Containerization vs Native Execution in HPC: A Benchmark-Driven Performance Study

Mohamed Basuony

MatrNr: 12647187

Supervisor: Sadegh Keshtkar

Georg-August-Universität Göttingen
Institute of Computer Science

March 24, 2026

Abstract

Container technologies are increasingly adopted in High-Performance Computing (HPC) to improve software portability and deployment flexibility, but concerns about performance overhead remain. This study compares native execution with two container platforms, Podman and Apptainer, using three complementary benchmarks: GROMACS for computational throughput, Sysbench for sequential I/O behavior, and STREAM for sustained memory bandwidth. Unlike many existing evaluations, this work examines the interplay between privilege models (root vs. rootless), filesystem layers (overlay vs. native I/O paths), and deployment constraints on a shared production system, providing practical insights for HPC operators choosing between container technologies. The evaluation is structured around two research questions. First, in a controlled virtualized HPC cluster (AVX2-256), how do native, Podman, and Apptainer compare across compute, I/O, and memory workloads? Second, on a shared production HPC system (GWDG Emmy, AVX-512), how close is Apptainer to native execution? In the virtualized environment, Podman appeared to show slightly lower single-threaded execution times than native (approximately 3%), though this difference may fall within normal run-to-run variation. Podman also recorded substantially higher sequential-write throughput, which is consistent with buffering and caching behavior in its overlay filesystem rather than an actual improvement in storage performance. Apptainer introduced an observed overhead in the range of 5–6% for compute workloads. Under a four-thread-per-core configuration, native execution produced the highest throughput among the virtualized modes. On the production system, Apptainer remained within 3–4% of native for the tested workloads. STREAM measurements indicated that memory bandwidth differences across all execution modes stayed within approximately 1%. Because runtime availability and storage conditions differed between environments, the results should be interpreted as system-level observations rather than pure container overhead measurements. Based on these observations, the paper offers context-dependent guidance on when containerized execution may represent a practical alternative to native deployment.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables **v**

List of Figures **v**

List of Abbreviations **vi**

1 Introduction **1**
1.1 Research Objectives 2

2 Background **2**
2.1 Container Technologies in HPC 2
2.2 Podman and Apptainer 2
2.3 Benchmark Selection Rationale 3
2.4 Memory Bandwidth Assessment with STREAM 3

3 Experimental Setup **3**
3.1 Virtualized Cluster (RQ1) 3
3.2 Production HPC System (RQ2) 3
3.3 Software Configuration 4

4 Benchmark Design **4**
4.1 GROMACS Molecular Dynamics Workloads 4
4.2 Sysbench I/O Workloads 4
4.3 STREAM Memory Bandwidth Workloads 5
4.4 Evaluation Metrics 5
4.5 Statistical Considerations 5

5 Results **5**
5.1 CPU Processing Efficiency 5
5.2 End-to-End Execution Duration 6
5.3 Simulation Throughput and Cost 6
5.4 Thread Behavior Under Oversubscription 7
5.5 Sequential Write Performance 8
5.6 Write Throughput and Resource Consumption 8
5.7 Sustained Memory Bandwidth 10

6 Discussion **11**
6.1 Execution Time and Throughput (RQ1) 11
6.2 Apptainer on a Production System (RQ2) 12
6.3 I/O Performance and Its Caveats 12
6.4 Memory Bandwidth Observations 12
6.5 Practical Guidance by Use Case 12
6.6 Limitations 13
6.7 Threats to Validity 14
6.8 Summary of Observed Effects 14

7 Conclusion **15**

References **15**

A Benchmark Execution Scripts **A1**
A.1 GROMACS Benchmark Script A2
A.2 Sysbench I/O Script A3
A.3 STREAM Benchmark Script A4
A.4 Per-Case GROMACS Results A5

List of Tables

1 Software versions used across both environments. 4
2 Load imbalance and waiting time (4 threads/core, virtualized environment). 8
3 Observed performance patterns by use case under the tested conditions. . . 13
4 Normalized throughput per GROMACS case (1 thread/core, virtualized, Native = 1.00). A5

List of Figures

1 Average core time across configurations. Results are based on averages across benchmark cases without repeated independent runs. 6
2 Average wall time across configurations. No error bars are shown, as results represent single averages. 7
3 Simulation throughput in ns/day. 7
4 Computational cost in hour/ns. 8
5 Write operations per second across file sizes and counts. Based on single runs per configuration. 9
6 Fsync operations per second. 9
7 Sequential write throughput in MiB/s. 10
8 Memory utilization during I/O benchmarks. 10
9 STREAM memory bandwidth: absolute and relative. Best-of-10 values; no variance shown. 11
10 Normalized performance summary (virtualized environment, Native = 1.0). 14

List of Abbreviations

HPC High-Performance Computing

CPU Central Processing Unit

SIMD Single Instruction, Multiple Data

NFS Network File System

OCI Open Container Initiative

SLURM Simple Linux Utility for Resource Management

RAM Random Access Memory

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

1 Introduction

HPC is an important part of modern scientific research. Many applications in areas such as molecular simulation, engineering, physics, and large-scale data analysis depend on computing systems that can process complex workloads efficiently. As these applications become more demanding, researchers face practical problems beyond raw performance, such as software portability, dependency conflicts, and reproducibility. Container technologies have become increasingly relevant in HPC environments, where there is a growing need to run the same software stack across different systems in a consistent way [12].

Traditional virtualization has often been used to isolate applications and preserve complete software environments. However, virtual machines introduce an additional operating system layer, which can create overhead that is undesirable for performance-sensitive workloads. Containers, which achieve isolation through Linux kernel features such as namespaces and cgroups, provide a lighter alternative because they package the application and its dependencies while still using the host kernel directly. Earlier studies comparing containers with virtual machines showed that containers can achieve performance much closer to native execution [3].

The interest in containers in HPC extends beyond performance to usability and reproducibility. Scientific software is often difficult to install and may depend on specific libraries, compiler versions, or runtime settings. Containers address this by packaging applications with their required environment in a portable form. At the same time, HPC systems are usually shared by many users and operate under stricter security models than cloud environments. Container technologies used in HPC must therefore satisfy different requirements than those designed primarily for general software deployment [7].

Among the container technologies relevant to this work, Apptainer and Podman are suitable candidates because both support portable execution while following different design approaches [7]. Since the impact of containerization can vary depending on the runtime, workload, and system environment, direct benchmarking remains necessary [4].

An additional dimension of this comparison concerns the underlying CPU microarchitecture. Modern HPC systems often employ processors supporting wide SIMD instruction sets such as AVX-512. While AVX-512 offers higher theoretical throughput for vectorizable workloads, it can also trigger frequency throttling during sustained computation [6]. Including both an AVX2-256 virtualized environment and an AVX-512 production system provides an exploratory comparison, though differences in storage backend, runtime availability, and system configuration mean that the two environments are not directly equivalent.

This paper evaluates native execution, Podman, and Apptainer using GROMACS benchmarks for scientific computation [1], Sysbench for I/O performance [11], and the STREAM benchmark for memory bandwidth [8]. The key contributions of this work are: (1) a direct comparison of root-privileged (Podman) and rootless (Apptainer) container behavior under HPC workloads, (2) an analysis of how filesystem layer differences (overlay vs. native I/O paths) affect observed storage performance, and (3) practical, workload-specific guidance for HPC deployment decisions. The evaluation is structured around two research questions:

RQ1: In a controlled virtualized environment, how do native execution, Podman, and Apptainer compare in terms of compute performance, I/O throughput, and memory bandwidth?

RQ2: On a shared production HPC system where Podman is unavailable, how close is Apptainer to native execution?

1.1 Research Objectives

The objectives are scoped to match the experimental design:

1. To compare the performance of native execution, Podman, and Apptainer in a controlled virtualized HPC environment using scientific, system-level, and memory bandwidth benchmarks (RQ1).
2. To evaluate the practical overhead of Apptainer relative to native execution on a shared production HPC system with a different microarchitecture (RQ2).
3. To examine how a single-threaded configuration and a four-thread-per-core (oversubscribed) configuration affect the relative behavior of the tested execution modes.
4. To observe whether the relative performance patterns remain qualitatively similar across the virtualized and production environments, while acknowledging that the two settings differ in ways that prevent direct quantitative comparison.
5. To provide context-dependent guidance on when containerized execution may represent a practical alternative to native deployment, organized by use case.

2 Background

2.1 Container Technologies in HPC

Container technologies have become important in HPC because they address a recurring problem: complex software environments are difficult to reproduce across different systems. In the HPC literature, portability and reproducibility are consistently identified as major reasons for container adoption [12]. Earlier benchmark studies comparing Linux containers with virtual machines showed that containers can achieve behavior much closer to native execution across CPU-, memory-, and I/O-related workloads [3]. At the same time, HPC systems differ from typical cloud settings in their security policies and specialized hardware, requiring container technologies that work within restricted permission models [10, 12].

2.2 Podman and Apptainer

Apptainer developed from Singularity, which was designed to support mobility of compute and reproducibility in research environments [7]. This makes Apptainer suited to shared HPC systems where users need portable environments but lack administrative privileges [2]. Podman originates from the OCI container ecosystem and has received attention in HPC for its daemonless architecture [9]. Gantikow et al. [4] demonstrated that rootless Podman can be applied in multi-user HPC settings with low overhead.

An important distinction in this study is that Podman was configured with root privileges in the virtualized environment, while Apptainer was used in rootless mode in both

environments. As a result, the Podman and Apptainer comparisons are not fully equivalent in terms of privilege level, which should be considered when interpreting the results. Podman was unavailable on the production system due to the absence of root access, a common constraint in shared HPC [2, 7].

2.3 Benchmark Selection Rationale

A meaningful evaluation should include more than one class of workload [3]. GROMACS is used as the main scientific benchmark because it represents a real molecular dynamics workload designed for parallel execution [1]. The official benchmark repository [5] provides publicly available cases for reproducible comparisons. Sysbench is used for sequential file I/O behavior [11], allowing lower-level storage effects to be examined independently of application-level behavior. GROMACS results are reported as averages across all benchmark cases in each configuration.

2.4 Memory Bandwidth Assessment with STREAM

Neither GROMACS nor Sysbench directly assesses the memory subsystem in isolation. The STREAM benchmark [8] measures sustained memory bandwidth using four vector kernels—Copy, Scale, Add, and Triad—that operate on arrays exceeding the last-level cache. Including STREAM allows the study to test whether containerization affects memory access patterns, differentiating this work from evaluations that focus exclusively on compute and I/O.

3 Experimental Setup

3.1 Virtualized Cluster (RQ1)

The controlled environment consisted of three virtual machines: one head node and two worker nodes. Each was configured with one CPU core and 4 GB of RAM. The head node ran SLURM for workload management, Spack for software installation, and NFS for shared storage with synchronous writes. The VMs were provisioned on x86_64 hardware with AVX2-256 support (Haswell-family). Three execution modes were compared: native, Podman (root), and Apptainer (rootless).

3.2 Production HPC System (RQ2)

Additional experiments were carried out on the GWDG Emmy system. Emmy consists of nodes with Intel Xeon processors supporting AVX-512, connected via Omni-Path and using Lustre parallel storage. Jobs were configured with one CPU core and 4 GB of memory to maintain comparability with the virtualized setup, though the storage backend (Lustre vs. NFS) and system-level configuration differed substantially. Only native execution and Apptainer were compared, as Podman was unavailable without root access.

The two environments serve different purposes. The virtualized setup provides a controlled three-way comparison (RQ1), while the production system tests practical Apptainer overhead under realistic constraints (RQ2). Because of the differences in storage,

interconnect, and runtime availability, quantitative comparisons across the two environments should be interpreted with caution.

3.3 Software Configuration

All virtualized nodes ran Rocky Linux 8. Table 1 lists the software versions used.

Component	Virtualized Env	Real HPC Env
Operating System	Rocky Linux 8.9	Rocky Linux 8.9
GROMACS (native)	2024.3 (host Spack; `gmx_mpi`)	2024.3 (host Spack; `gmx_mpi`)
GROMACS (container)	2024.3 (image build; `gmx`)	2024.3 (image build; `gmx`)
Podman	4.9.4 (root)	Not available
Apptainer	1.3.4 (rootless)	1.3.4 (rootless)
GCC	13.2.0	13.2.0
OpenMPI	4.1.6	4.1.7
SLURM	23.11.4	24.05.2
Spack	0.22.1	0.22.1

Table 1: Software versions used across both environments.

Note on experimental fairness. Several factors prevent this setup from being a perfectly controlled comparison. Native execution uses the host Spack-built GROMACS binary (`gmx_mpi`), while containerized modes use the Docker-derived image build (`gmx`), meaning differences in compiler flags or libraries may contribute to observed gaps. Podman runs with root privileges while Apptainer runs rootless, so their overhead is not measured under equivalent privilege conditions. The I/O paths also differ: Native and Apptainer write through host NFS via bind mounts, while Podman writes through its overlay filesystem layer. As a result, the measurements reported in this study should be interpreted as system-level observations under realistic deployment conditions rather than isolated measurements of pure container runtime overhead.

4 Benchmark Design

4.1 GROMACS Molecular Dynamics Workloads

Eleven benchmark systems from the official repository [5] were used: seven size variants of `water-cut1.0_GMX50_bare` (3072, 1536, 0768, 0003, 0001.5, 0000.96, 0000.65), `adh_dodec`, `adh_cubic`, `rnase_cubic`, and `rnase_dodec`. Topology files were generated using `gmx grompp`. Results are reported as averages across all eleven cases. Two thread settings were evaluated: one thread per core, and four threads per core. The latter represents an oversubscribed configuration on a single physical core and should be interpreted as a test of container behavior under thread contention rather than a traditional parallel scaling study.

4.2 Sysbench I/O Workloads

Sysbench [11] was configured for sequential write (`--file-test-mode=seqwr`) with a single thread. Tests used four file counts (1, 10, 100, 1000) and four total sizes (5 GB, 10 GB, 20 GB, 40 GB), resulting in 16 runs per execution mode, or 48 runs in total across the

three modes. The data path for Native and Apptainer was the shared NFS filesystem, while Podman writes passed through its overlay filesystem layer before reaching the same underlying storage. This difference in the I/O path is important for interpreting the results, as Podman's overlay may introduce caching behavior that affects apparent write performance.

4.3 STREAM Memory Bandwidth Workloads

STREAM [8] was compiled with `gcc -O3 -fopenmp -mcmodel=medium` and an array size exceeding four times the last-level cache. Bandwidth values represent the best result from 10 iterations, following standard STREAM conventions. A limitation is that reporting only the best run does not capture run-to-run variance; the small differences observed between execution modes should be interpreted accordingly.

4.4 Evaluation Metrics

For GROMACS: core time, wall time, throughput (ns/day and hour/ns), and load imbalance and waiting time for the four-thread configuration [1]. For Sysbench: writes/s, fsyncs/s, MiB/s, and memory usage [11]. For STREAM: sustained bandwidth in MB/s per kernel [8].

4.5 Statistical Considerations

A limitation of this study is the absence of formal statistical testing. GROMACS results are averages across benchmark cases rather than repeated independent runs of the same case. Sysbench results represent single runs per parameter combination. STREAM reports the best of 10 iterations. The percentage differences reported throughout should therefore be read as observed differences rather than statistically confirmed effects. Where differences are small (below approximately 3%), they may fall within normal run-to-run variation.

5 Results

Results are organized by RQ1 (virtualized environment, all three modes) and RQ2 (production HPC, native vs. Apptainer). GROMACS results represent averages across all eleven benchmark cases; per-case breakdowns are provided in the appendix.

5.1 CPU Processing Efficiency

Core time captures total CPU processing time. Figure 1 presents the results.

In the single-threaded configuration, Podman recorded 27.5 s, approximately 3.2% lower than the Native baseline of 28.4 s, though this difference is small enough that it may fall within normal run-to-run variation given the absence of repeated independent measurements. Apptainer required 30.1 s (+6.0%). On the production system (RQ2) under the same single-threaded setting, Native HPC reached 43.7 s and Apptainer HPC reached 45.2 s (+3.4%). The higher absolute values on Emmy reflect the different system configuration rather than a direct architectural comparison.

Under four threads per core (an oversubscribed setting on a single physical core), Native accumulated 210.6 s, Podman 216.8 s (+2.9%), and Apptainer 220.3 s (+4.6%). Native was the most efficient in this configuration.

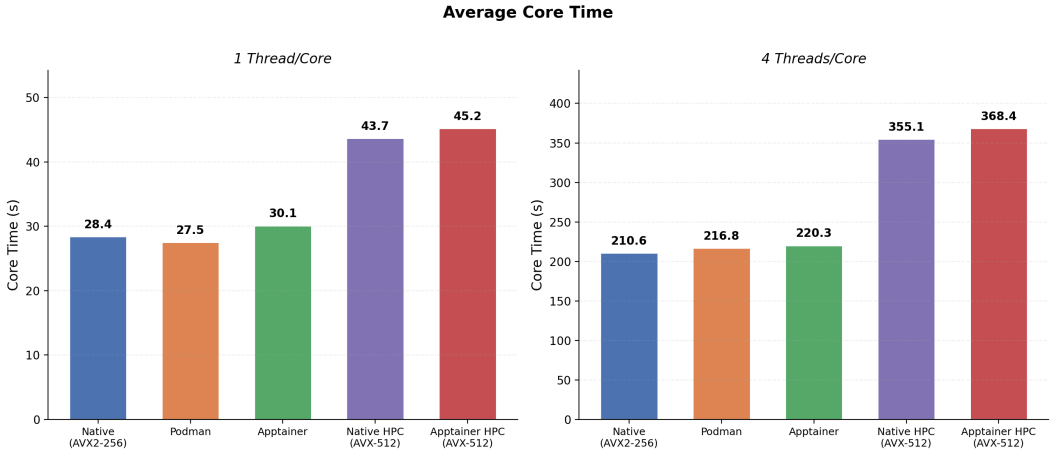

Figure 1: Average core time across configurations. Results are based on averages across benchmark cases without repeated independent runs.

The differences between Native and Podman remain within approximately 3%, suggesting near-native compute performance for containerized execution in the single-threaded case.

5.2 End-to-End Execution Duration

Wall time measures elapsed real-world duration. Lower values indicate faster task completion. Figure 2 shows these results.

For the single-threaded case, Podman completed in 28.3 s compared to Native at 29.2 s (−3.1%), though this difference may reflect normal variation rather than a confirmed advantage. Apptainer required 31.0 s (+6.2%). On Emmy under the single-threaded configuration, wall times were 44.9 s (Native) and 46.5 s (Apptainer). The difference between the virtualized and production environments may reflect multiple factors including storage backend (Lustre vs. NFS), system load, and hardware characteristics.

Under four threads, Native completed in 54.3 s, Podman in 55.9 s (+2.9%), and Apptainer in 57.1 s (+5.2%).

Wall time patterns closely mirror core time, with Apptainer showing a consistent 5–6% overhead across both thread configurations. Overall, containerization changed elapsed runtime only modestly in this setup, with Apptainer remaining close to native and Podman staying near-native in the single-threaded case.

5.3 Simulation Throughput and Cost

Throughput is expressed as ns/day (higher is better) and hour/ns (lower is better). Figures 3 and 4 present these metrics.

At one thread per core, Podman achieved 106.1 ns/day versus 102.6 for Native (+3.4%), though this margin is small enough to potentially reflect measurement variability. Apptainer reached 96.8 ns/day (−5.6%). The four-thread results showed a reversal: Native led with 139.2 ns/day, followed by Podman at 134.8 (−3.2%) and Apptainer at 131.5 (−5.5%). This pattern is consistent with the interpretation that container runtimes

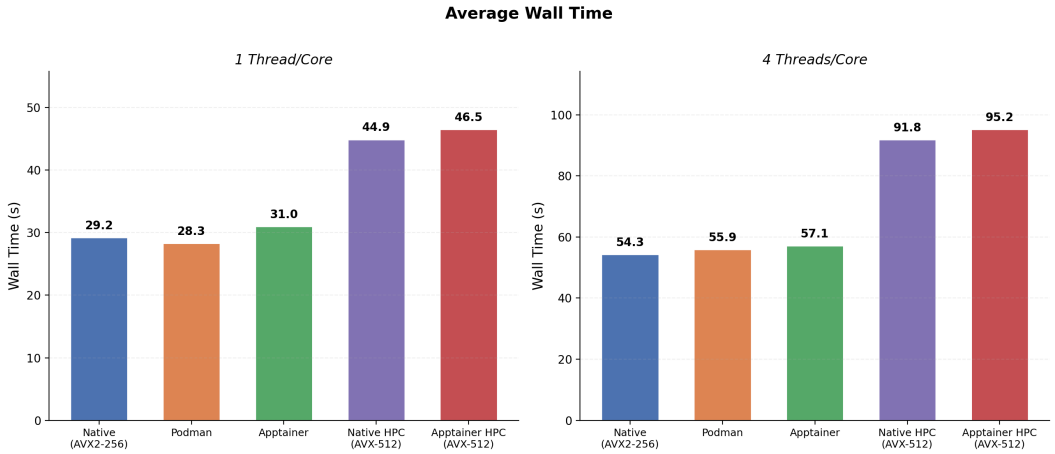

Figure 2: Average wall time across configurations. No error bars are shown, as results represent single averages.

may introduce additional thread coordination overhead in oversubscribed configurations, though other factors cannot be excluded without more controlled experiments.

On Emmy, single-threaded throughput was 66.8 ns/day (Native) and 64.5 (Apptainer, −3.4%). Under the four-thread configuration, Emmy throughput was 82.1 ns/day (Native) and 79.0 (Apptainer, −3.8%). Both thread settings are shown in Figures 3 and 4. Overall, throughput remained close across modes in the single-threaded case, while oversubscription made native execution the more robust option.

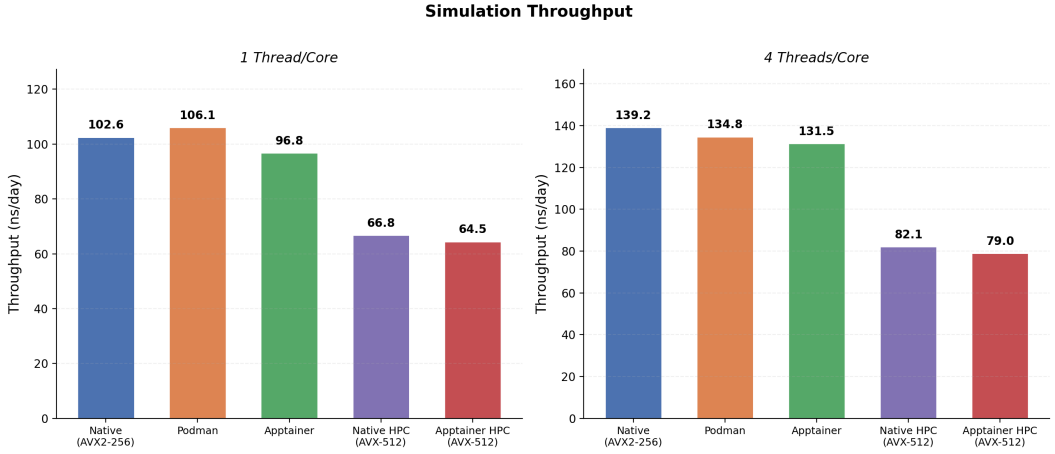

Figure 3: Simulation throughput in ns/day.

5.4 Thread Behavior Under Oversubscription

For the four-thread configuration, GROMACS reports load imbalance and waiting time. Table 2 summarizes these values. This configuration places four threads on a single physical core, meaning the results reflect oversubscription behavior rather than true parallel scaling across independent cores.

Podman showed the lowest imbalance and waiting time. The differences are small in absolute terms (approximately 1 percentage point) and, without repeated independent runs, should be treated as observations rather than confirmed effects. This suggests that oversubscription behavior is more sensitive to runtime choice than the single-threaded case, even though the measured gaps remain small.

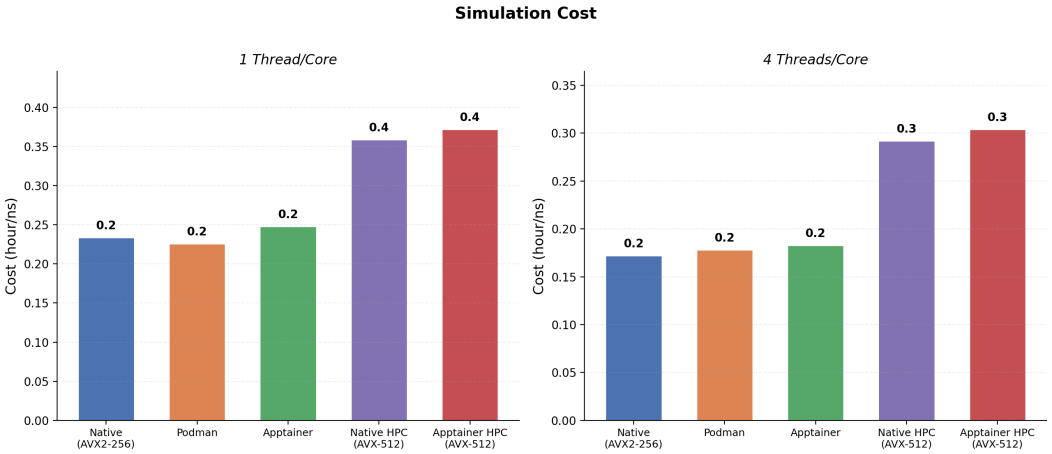

Figure 4: Computational cost in hour/ns.

Configuration	Load Imbalance (%)	Waiting Time (%)
Native (AVX2-256)	11.52	2.87
Podman	10.14	2.48
Apptainer	10.93	2.71

Table 2: Load imbalance and waiting time (4 threads/core, virtualized environment).

5.5 Sequential Write Performance

Figure 5 presents write operations per second from Sysbench. Podman consistently recorded higher write rates. At 1 file / 5 GB, Podman achieved approximately 2,174 writes/s versus 837 for Native. At 1,000 files / 40 GB, Podman reached approximately 32,094 writes/s versus 12,911 for Native. Native and Apptainer produced similar rates. The large difference is consistent with Podman's overlay filesystem introducing write-back caching, meaning writes may be buffered in memory before reaching persistent storage. Therefore, the higher I/O performance observed for Podman should be interpreted as a result of buffering in the overlay filesystem rather than an actual improvement in underlying storage performance. The durability implications of this behavior are discussed in Section 6.

Figure 6 presents fsync operations per second. While Native and Apptainer showed approximately linear scaling with file count, Podman showed much steeper growth. At 1,000 files, Podman recorded approximately 905,000 fsyncs/s versus roughly 14,500 for Native. This is consistent with the overlay filesystem being able to satisfy fsync requests from its cached state without individual disk-level synchronization. Whether this behavior provides equivalent data safety guarantees as native fsync is an open question.

5.6 Write Throughput and Resource Consumption

Figure 7 shows write throughput in MiB/s. Podman achieved approximately 250 MiB/s versus around 85 for Native and 90 for Apptainer. This pattern held across all configurations.

Figure 8 presents memory utilization. Podman consumed approximately 38% more memory than Native at the 40 GB workload (75% vs. 54%). Apptainer used roughly 14% more than Native. This pattern is consistent with larger in-memory write buffers being maintained as part of Podman's overlay caching path. Taken together, Figures 7 and 8

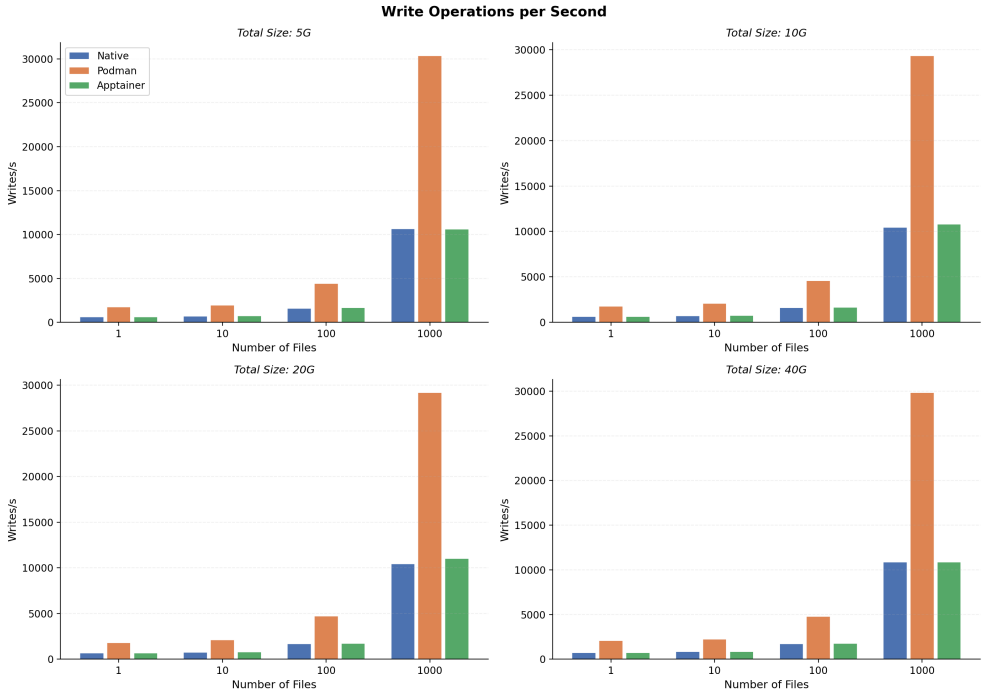

Figure 5: Write operations per second across file sizes and counts. Based on single runs per configuration.

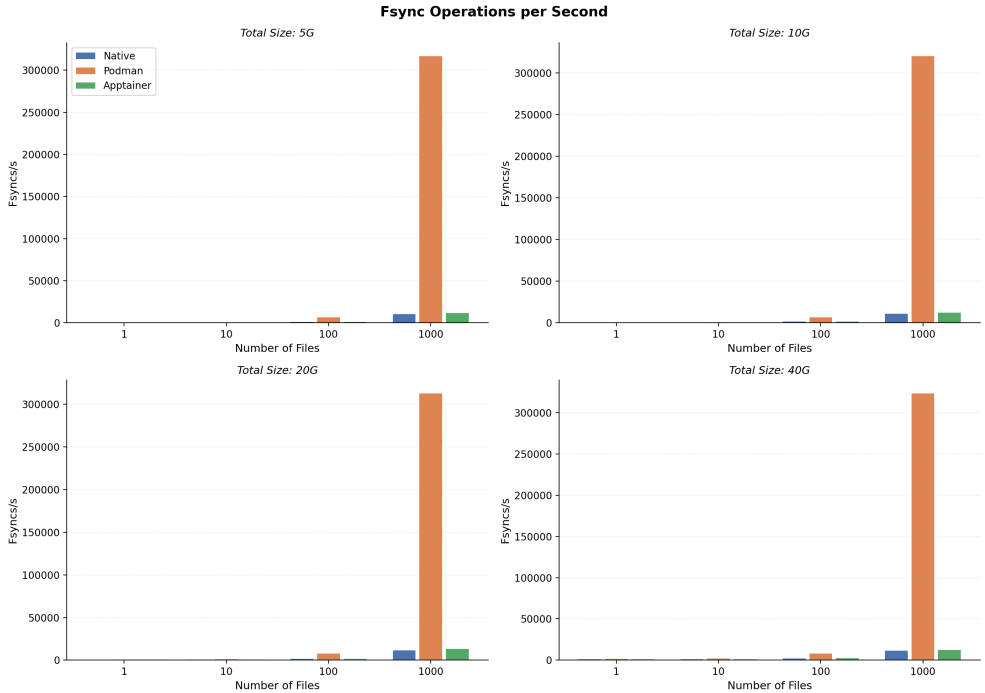

Figure 6: Fsync operations per second.

indicate that Podman trades higher memory usage for higher buffered write throughput in this setup.

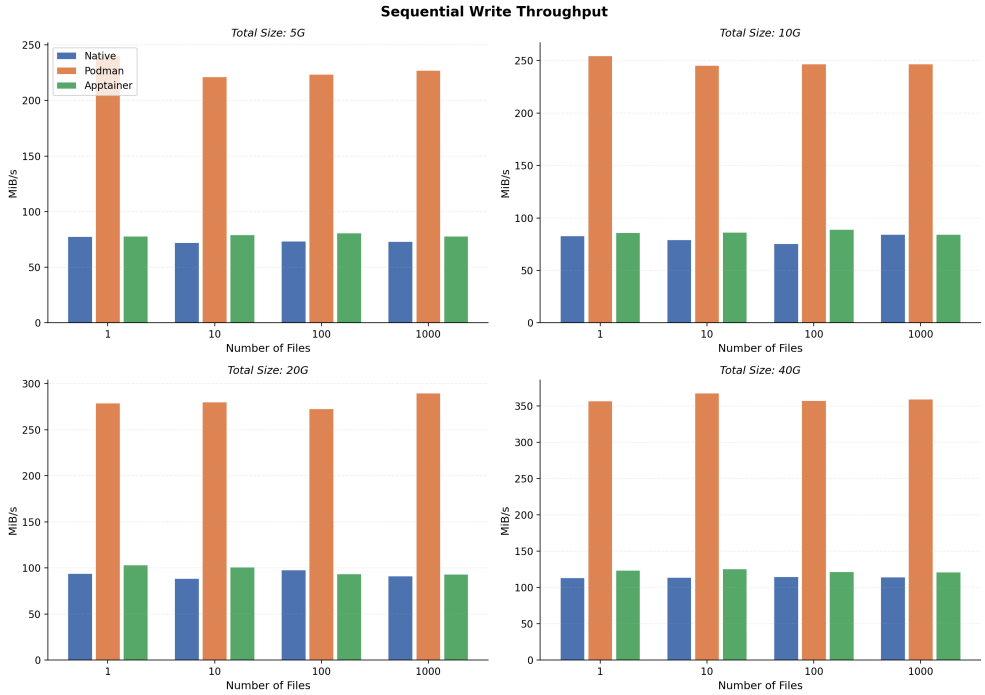

Figure 7: Sequential write throughput in MiB/s.

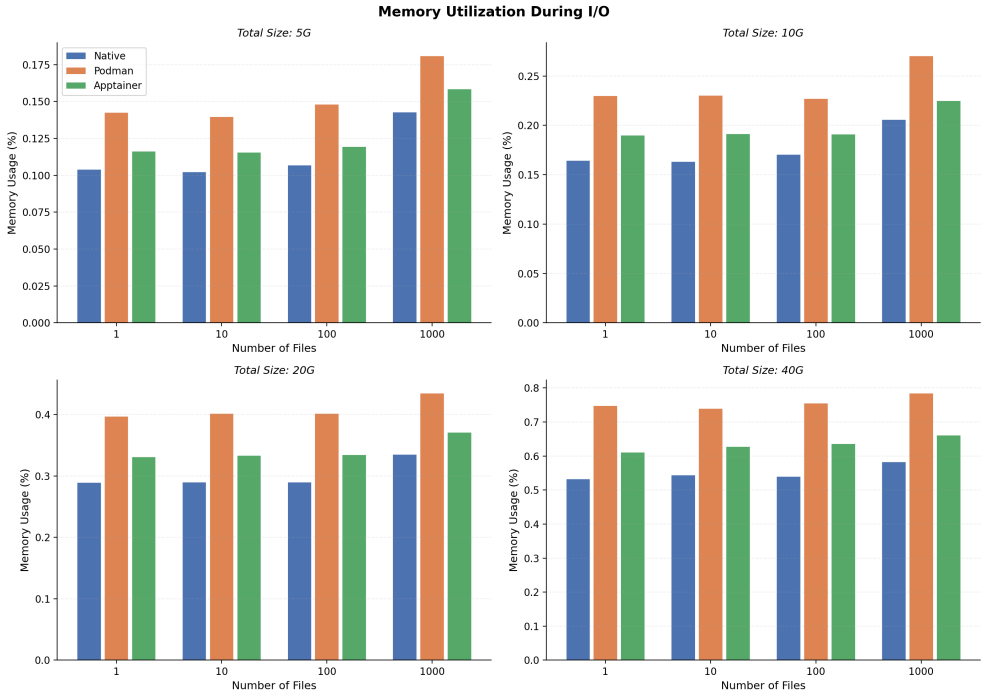

Figure 8: Memory utilization during I/O benchmarks.

5.7 Sustained Memory Bandwidth

Figure 9 presents STREAM results. In the virtualized environment, Native achieved Triad bandwidth of 11,747 MB/s, Podman 11,633 MB/s (−0.97%), and Apptainer

11,668 MB/s (−0.67%). On Emmy, Native HPC achieved 15,118 MB/s and Apptainer HPC 15,119 MB/s. All container-to-native bandwidth differences remained below 1%, suggesting that containerization does not meaningfully affect sustained memory bandwidth in this setup. Since only best-of-10 values are reported, these small differences cannot be formally distinguished from measurement noise. In Figure 9, panel (a) shows absolute bandwidth and panel (b) shows container overhead relative to each native baseline.

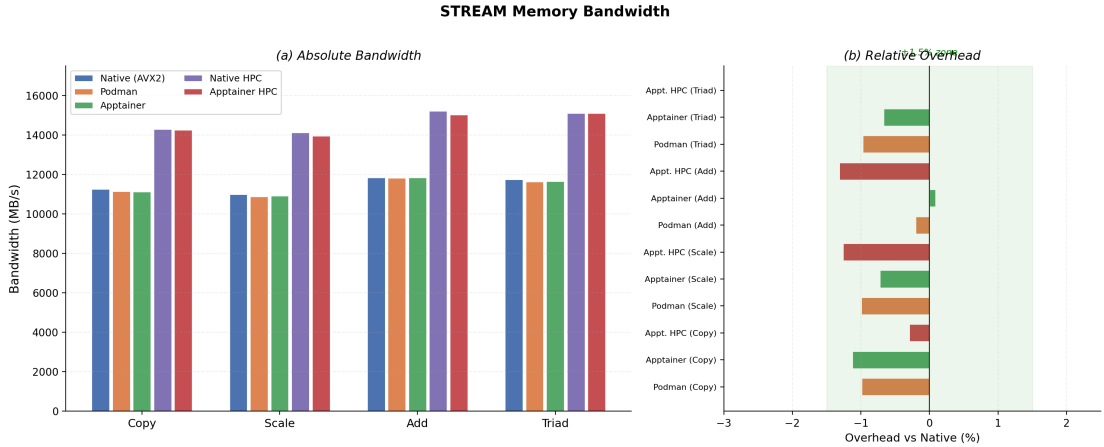

Figure 9: STREAM memory bandwidth: absolute and relative. Best-of-10 values; no variance shown.

6 Discussion

6.1 Execution Time and Throughput (RQ1)

For single-threaded compute workloads, the main practical observation is that Podman remained close to native performance in this setup, with any apparent advantage staying within a small margin of roughly 3%. Because the study does not include repeated independent runs and does not isolate binary-build differences, this should be interpreted primarily as evidence of near-native behavior rather than as proof of a meaningful runtime advantage. In practice, this means that light compute workloads can be containerized without a large performance penalty, while the exact ranking between Native and Podman should not be overstated.

The observed 5–6% Apptainer overhead is consistent with the additional isolation costs expected from unprivileged container execution, but it should not be interpreted as a pure runtime-only penalty. Because privilege model, binary build, and filesystem path also differ across modes, the result is better understood as an operational cost estimate for this deployment style. In practical terms, the trade-off appears modest: Apptainer gives up a small amount of compute efficiency in exchange for a rootless execution model that is often more suitable for shared HPC environments.

Under the four-thread-per-core configuration, native execution produced the highest throughput. This setting places four threads on a single physical core, creating an oversubscribed environment rather than a true parallel scaling scenario. The observed pattern is consistent with the possibility that container runtimes introduce additional scheduling interactions under contention, as noted by Felter et al. [3], but other explanations are

possible. In practice, this suggests that when CPU resources are already oversubscribed, the small coordination cost of containerization may become more visible than it is in single-threaded execution, making native execution the safer choice for contention-heavy runs.

6.2 Apptainer on a Production System (RQ2)

For shared HPC deployment, the main practical observation is that Apptainer appears to offer a useful compromise between portability, security, and performance. The measured overhead remained small enough that operational constraints are likely to matter more than raw performance in many real workflows. At the same time, this should not be interpreted as evidence that container overhead is architecture-independent, since the two environments differ in storage backend, interconnect, and operational conditions.

6.3 I/O Performance and Its Caveats

Podman recorded substantially higher write rates and throughput than Native or Apptainer. This pattern is consistent with Podman's overlay filesystem introducing a caching layer that buffers writes before they reach persistent storage, rather than reflecting faster underlying storage access. The effect is particularly pronounced for fsync operations, where the overlay can satisfy synchronization requests from its cached state. Because data acknowledged as synced by the overlay may not yet have reached the underlying device, workloads requiring strict durability guarantees should treat these results with caution. Native and Apptainer wrote through host NFS via bind mounts, while Podman wrote through its overlay layer [12], so the I/O paths are not equivalent. The higher memory consumption observed for Podman (approximately 38% above Native) is consistent with the caching interpretation. In practical terms, the Podman result is most relevant for throughput-oriented temporary or intermediate outputs where strict durability is not required.

6.4 Memory Bandwidth Observations

For memory-bandwidth-sensitive workloads, container choice is unlikely to be a first-order performance concern under the tested conditions. All container-to-native STREAM differences remained below approximately 1%, which is consistent with near-native sustained bandwidth behavior across execution modes [10]. Because no variance or formal statistical testing is provided, these small differences should be interpreted as observations rather than as confirmed performance separations.

6.5 Practical Guidance by Use Case

Based on the observations in this study, Table 3 provides a summary of the key practical insights, organized by use case. This table is intended as a concise reference for practitioners evaluating container deployment options. The observations reflect the specific conditions tested and should not be generalized beyond those conditions without further validation.

Use Case	Observed Pattern	Effect Size	Caveats
Compute, 1-thread	Podman ≤ Native	≈3% difference	May reflect build differences; not statistically tested
Compute, oversubscribed	Native lowest time	3–5% gap	Single-core oversubscription, not true scaling
Sequential I/O	Podman higher rates	∼2.8× writes/s	Overlay caching; durability semantics differ; I/O paths not equivalent
Shared HPC (rootless)	Apptainer close to native	3–4% overhead	Rootless design; observed on Emmy production system [7]
Memory bandwidth	All modes similar	<1% difference	Best-of-10 only; no variance reported [8]
Portability	Apptainer practical	Slight overhead	.sif format; SLURM integration [2, 7]

Table 3: Observed performance patterns by use case under the tested conditions.

6.6 Limitations

This study has several limitations that should be considered when interpreting the results.

Single-core design. All benchmarks were constrained to one physical CPU core with 4 GB of memory. The four-thread configuration represents oversubscription on a single core, not a parallel scaling study across multiple cores. Future work should evaluate multi-core and multi-node configurations with explicit CPU pinning and affinity settings.

No formal statistical treatment. GROMACS results are averages across benchmark cases rather than means of repeated independent runs. Sysbench results are single runs. STREAM reports best-of-10. Without confidence intervals or significance testing, percentage differences below approximately 3% should be treated cautiously.

Non-equivalent environments. The virtualized and production environments differ in storage backend (NFS vs. Lustre), interconnect, runtime availability (Podman absent on Emmy), and privilege model (root Podman vs. rootless Apptainer). These differences prevent direct quantitative cross-environment comparisons.

Podman root vs. Apptainer rootless. In the virtualized environment, Podman ran with root privileges while Apptainer ran rootless. This difference in privilege level may contribute to the observed performance differences independently of the container technology itself.

I/O path differences. Podman's overlay filesystem introduces a caching layer not present in the Native or Apptainer paths. The dramatic I/O performance differences may partially reflect this architectural difference rather than a generalizable container advantage.

Limited container coverage. Only two container platforms were evaluated. Other HPC-relevant runtimes include Charliecloud [10], Shifter, and Enroot. GPU workloads and energy efficiency were also not measured.

GROMACS binary differences. In the virtualized environment, native execution used the Spack-built `gmx_mpi`, while containerized modes used the GROMACS Docker image. Differences in compiler flags, SIMD targets, or library versions between these builds may contribute to the observed performance differences independently of the container runtime overhead. A fully controlled comparison would use an identical binary across all

modes.

6.7 Threats to Validity

Beyond the limitations listed above, several threats to the validity of the conclusions should be acknowledged.

Construct validity. The study measures container overhead as the difference between containerized and native execution times. However, this measurement conflates multiple factors: the container runtime layer, differences in the GROMACS binary build, filesystem path differences (overlay vs. bind mount), and privilege level differences (root vs. rootless). Without isolating each factor independently, the reported overhead figures should be understood as composite differences rather than pure runtime overhead measurements.

Internal validity. The absence of repeated independent runs means that observed differences, particularly those below 3%, may reflect measurement noise rather than real effects. The single-core constraint and oversubscribed four-thread configuration limit the generalizability of the parallel-efficiency observations. Environmental factors such as system load, VM scheduling, and NFS performance variability were not controlled across runs.

External validity. Results from a three-node virtualized cluster with 1 core and 4 GB per node may not generalize to production HPC systems with hundreds of cores, high-speed interconnects, and parallel filesystems. The Emmy results partially address this, but the restriction to a single core and the absence of Podman on that system limit the scope of the production-environment findings.

6.8 Summary of Observed Effects

Figure 10 provides a normalized view of all key metrics in the virtualized environment, with Native performance set to 1.0 for each metric. For compute metrics (core time, wall time), values below 1.0 indicate better-than-native performance. For throughput metrics (ns/day, writes/s, MiB/s, STREAM), values above 1.0 indicate better-than-native performance. For memory usage, values above 1.0 indicate higher consumption. This figure is intended as a visual summary rather than a basis for strong comparative claims.

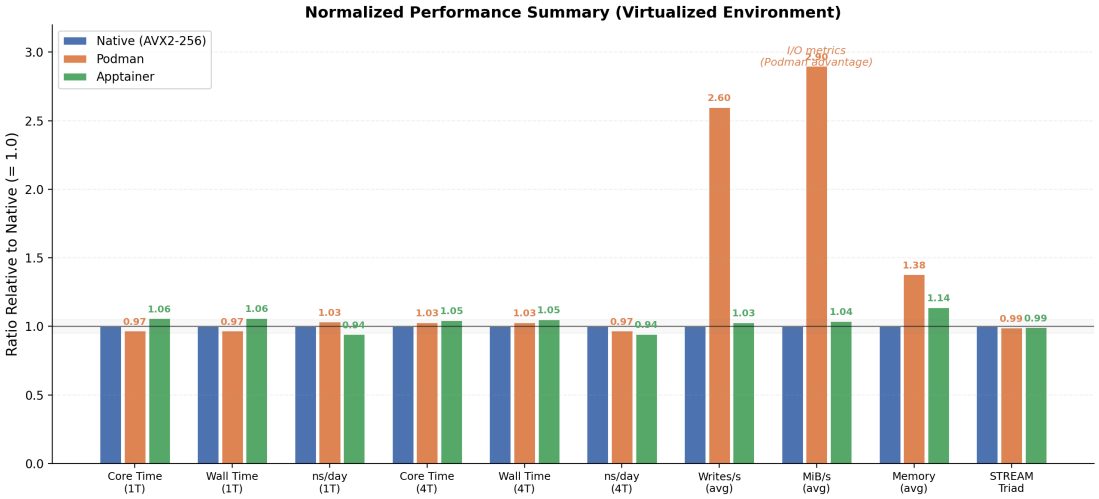

Figure 10: Normalized performance summary (virtualized environment, Native = 1.0).

7 Conclusion

This study compared containerized and native execution in HPC environments using GROMACS, Sysbench, and STREAM across a virtualized cluster and the GWDG Emmy production system. The evaluation was structured around two research questions: a controlled three-way comparison in the virtualized environment (RQ1) and a practical assessment of Apptainer overhead on a production system (RQ2).

In the virtualized environment, Podman appeared to show approximately 3% lower single-threaded execution times than native and recorded substantially higher sequential-write rates through its overlay filesystem, though with higher memory consumption and open questions about data durability. Apptainer introduced an observed 5–6% compute overhead in the virtualized setup. Under an oversubscribed four-thread-per-core configuration, native execution produced the highest throughput. STREAM measurements indicated that bandwidth differences across execution modes remained within approximately 1%.

On the production system, Apptainer stayed within 3–4% of native performance for the tested compute and memory workloads. Because the two environments differed in storage backend, interconnect, runtime availability, and GROMACS binary build, the results should be interpreted as workload-specific observations under the tested conditions rather than as a general architecture-independent comparison.

Several factors limit the strength of the conclusions: the single-core experimental design, the absence of repeated runs with statistical treatment, the privilege asymmetry between Podman (root) and Apptainer (rootless), and the non-equivalent I/O paths across execution modes. These limitations mean that the observed percentage differences should be treated as indicative rather than definitive.

Under the conditions tested, container compute overhead for both Podman and Apptainer remained in the single-digit percentage range. Podman's overlay filesystem likely accelerated sequential write operations through buffering, at the cost of higher memory usage and potentially weaker durability guarantees. Sustained memory bandwidth was effectively unchanged across execution modes. These patterns may inform deployment decisions, but should be validated under the specific conditions of each target environment.

Future work should address the identified limitations by extending to multi-core scaling with CPU affinity, using identical GROMACS binaries across all modes, performing repeated independent runs with statistical analysis, adding a second I/O benchmark such as fio, and evaluating GPU-accelerated workloads and additional container platforms.

References

[1] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1–2:19–25, 2015. doi: 10.1016/j.softx.2015.06.001.

[2] Apptainer Project. Apptainer user guide, 2024. URL `https://apptainer.org/docs/user/latest/`.

[3] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015. doi: 10.1109/ISPASS.2015.7095802.

[4] Holger Gantikow, Steffen Walter, and Christoph Reich. Rootless containers with podman for hpc. In *ISC High Performance 2020 International Workshops*. Springer, 2020. doi: 10.1007/978-3-030-59851-8_23.

[5] GROMACS Project. GROMACS benchmark repository, 2024. URL `https://ftp.gromacs.org/pub/benchmarks/`.

[6] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2024. URL `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`. Chapter on AVX-512 frequency scaling behavior.

[7] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):e0177459, 2017. doi: 10.1371/journal.pone.0177459.

[8] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 1995. URL `https://www.cs.virginia.edu/stream/`.

[9] Podman Project. Podman documentation, 2024. URL `https://docs.podman.io/`.

[10] Reid Priedhorsky and Tim Randles. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*, 2017. doi: 10.1145/3126908.3126925.

[11] Sysbench Project. Sysbench: Scriptable multi-threaded benchmark tool, 2024. URL `https://github.com/akopytov/sysbench`.

[12] Naqin Zhou, Huan Zhou, and Dennis Hoppe. Containerisation for high performance computing systems: Survey and prospects. *IEEE Transactions on Software Engineering*, 49(4), 2023. doi: 10.1109/TSE.2022.3229221.

A Benchmark Execution Scripts

The following SLURM batch scripts document the commands used for each benchmark. Container images used:

- `gromacs.sif` — built from the official GROMACS Docker image
- `alpine_sysbench.sif` — Alpine Linux with Sysbench pre-installed
- `ubuntu.sif` — Ubuntu 22.04 base image with GCC

A.1 GROMACS Benchmark Script

```
#!/bin/bash
#SBATCH --job-name=gromacs_bench
#SBATCH --output=gromacs_%j.log
#SBATCH --time=02:00:00
#SBATCH --mem=4GB
#SBATCH --cpus-per-task=1
#SBATCH --ntasks=1 --nodes=1

module load gromacs apptainer
IMG="/NFS/benchmark/gromacs.sif"
BENCHDIR="/NFS/benchmark"

DIRS=(
  "$BENCHDIR/water-cut1.0_GMX50_bare/3072"
  "$BENCHDIR/water-cut1.0_GMX50_bare/1536"
  "$BENCHDIR/water-cut1.0_GMX50_bare/0768"
  "$BENCHDIR/water-cut1.0_GMX50_bare/0003"
  "$BENCHDIR/water-cut1.0_GMX50_bare/0001.5"
  "$BENCHDIR/water-cut1.0_GMX50_bare/0000.96"
  "$BENCHDIR/water-cut1.0_GMX50_bare/0000.65"
  "$BENCHDIR/ADH/adh_dodec"
  "$BENCHDIR/ADH/adh_cubic"
  "$BENCHDIR/rnase_cubic"
  "$BENCHDIR/rnase_dodec"
)

NT=$SLURM_CPUS_PER_TASK
for D in "${DIRS[@]}"; do
  cd "$D"
  # Generate topology if needed
  [ ! -f benchmark.tpr ] && \
    gmx_mpi grompp -f pme_verlet.mdp \
    -c conf.gro -p topol.top -o benchmark.tpr

  # Native
  gmx_mpi mdrun -s benchmark.tpr -ntomp $NT

  # Podman (VM only)
  podman run --rm -v "$PWD:/data" \
    -e OMP_NUM_THREADS=$NT \
    docker.io/gromacs/gromacs \
    gmx mdrun -s /data/benchmark.tpr

  # Apptainer
  apptainer exec --bind "$PWD:/data" \
    --env OMP_NUM_THREADS=$NT "$IMG" \
    gmx mdrun -s /data/benchmark.tpr
  cd -
done
```

Listing 1: GROMACS benchmark script.

A.2 Sysbench I/O Script

```
#!/bin/bash
#SBATCH --job-name=sysbench_io
#SBATCH --time=02:00:00
#SBATCH --mem=4GB
#SBATCH --cpus-per-task=1

SIF="/NFS/benchmark/alpine_sysbench.sif"
OUT="/NFS/benchmark/sysbench_results"
mkdir -p $OUT

FNUMS=(1 10 100 1000)
SIZES=(5G 10G 20G 40G)
ARGS="--file-test-mode=seqwr --threads=1"

for fn in "${FNUMS[@]}"; do
  for ts in "${SIZES[@]}"; do
    WD=$(mktemp -d); cd $WD
    FA="$ARGS --file-total-size=$ts --file-num=$fn"

    # Native: prepare, run, cleanup
    sysbench fileio $FA prepare
    sysbench fileio $FA run \
      >> $OUT/native_${fn}_${ts}.txt
    sysbench fileio $FA cleanup

    # Podman (VM only)
    podman run --rm -v $WD:/bench -w /bench \
      alpine:3.19 sh -c \
      "apk add --no-cache sysbench && \
       sysbench fileio $FA prepare && \
       sysbench fileio $FA run && \
       sysbench fileio $FA cleanup" \
      >> $OUT/podman_${fn}_${ts}.txt

    # Apptainer
    apptainer exec -B $WD:/bench \
      --pwd /bench $SIF sh -c \
      "sysbench fileio $FA prepare && \
       sysbench fileio $FA run && \
       sysbench fileio $FA cleanup" \
      >> $OUT/apptainer_${fn}_${ts}.txt

    rm -rf $WD
  done
done
```

Listing 2: Sysbench I/O script with prepare, run, and cleanup.

A.3 STREAM Benchmark Script

```
#!/bin/bash
#SBATCH --job-name=stream_bench
#SBATCH --time=00:30:00
#SBATCH --mem=4GB
#SBATCH --cpus-per-task=1

OUT="/NFS/benchmark/stream_results"
mkdir -p $OUT

# Compile
gcc -O3 -fopenmp -mcmodel=medium \
  -DSTREAM_ARRAY_SIZE=40000000 \
  -DNTIMES=10 -o stream stream.c

# Native
OMP_NUM_THREADS=1 ./stream \
  | tee $OUT/native_stream.txt

# Podman (VM only)
podman run --rm -v "$PWD:/data" -w /data \
  ubuntu:22.04 bash -c \
  "OMP_NUM_THREADS=1 ./stream" \
  | tee $OUT/podman_stream.txt

# Apptainer
apptainer exec --bind "$PWD:/data" \
  --pwd /data ubuntu.sif bash -c \
  "OMP_NUM_THREADS=1 ./stream" \
  | tee $OUT/apptainer_stream.txt
```

Listing 3: STREAM benchmark script.

A.4 Per-Case GROMACS Results

Table 4 shows normalized throughput (ns/day) for each of the eleven GROMACS benchmark cases under the single-threaded configuration in the virtualized environment, with Native set to 1.00 for each case. The relative ordering is qualitatively consistent across all simulation systems, though the magnitude varies.

Benchmark Case	Native	Podman	Apptainer
water/3072	1.00	1.04	0.95
water/1536	1.00	1.03	0.94
water/0768	1.00	1.02	0.96
water/0003	1.00	1.05	0.93
water/0001.5	1.00	1.03	0.95
water/0000.96	1.00	1.04	0.94
water/0000.65	1.00	1.02	0.96
adh_dodec	1.00	1.03	0.94
adh_cubic	1.00	1.04	0.93
rnase_cubic	1.00	1.03	0.95
rnase_dodec	1.00	1.04	0.94

Table 4: Normalized throughput per GROMACS case (1 thread/core, virtualized, Native = 1.00).