coMtainer: Compilation-assisted HPC Container Images with Enhanced Adaptability

Yuhao Gu
Sun Yat-sen University
Guangzhou, China
guyh9@mail2.sysu.edu.cn

Haoquan Chen
Sun Yat-sen University
Guangzhou, China
chenhq79@mail2.sysu.edu.cn

Xianjie Chen
Sun Yat-sen University
Guangzhou, China
chenxj275@mail2.sysu.edu.cn

Jiangsu Du
Sun Yat-sen University
Guangzhou, China
dujiangsu@mail.sysu.edu.cn

Zhiguang Chen
Sun Yat-sen University
Guangzhou, China
chenzhg29@mail.sysu.edu.cn

Nong Xiao*
Sun Yat-sen University
Guangzhou, China
xiaon6@mail.sysu.edu.cn

Xianwei Zhang*
Sun Yat-sen University
Guangzhou, China
zhangxw79@mail.sysu.edu.cn

Yutong Lu
Sun Yat-sen University
Guangzhou, China
luyutong@mail.sysu.edu.cn

Abstract

The increasing interconnectivity of HPC systems has highlighted the need for efficient application migration across different environments. Containers, widely adopted for this purpose, simplify deployment but often fail to deliver optimal performance due to the separated build and execution container workflow. This leads to generic container images that miss out on system-specific software stack advantages, a challenge we define as the adaptability issue.

We propose coMtainer, a compilation-assisted image transformation framework that embeds build-time information into images. This enables remote HPC systems to specialize and rebuild the container using native toolchains and libraries. coMtainer preserves image neutrality while resolving the adaptability issue, allowing optimized execution without user involvement. Moreover, the embedded metadata unlocks advanced compiler optimizations such as LTO and PGO. We implement and evaluate coMtainer across a variety of real-world HPC applications, demonstrating coMtainer's practicability, applicability and effectiveness.

CCS Concepts

• **Software and its engineering** → **Compilers**; Software libraries and repositories; **Software performance**; *Cloud computing*.

Keywords

High Performance Computing, Container, Adaptability, Software Stack, Compiler Optimization

*Corresponding author.

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759790

ACM Reference Format:
Yuhao Gu, Haoquan Chen, Xianjie Chen, Jiangsu Du, Zhiguang Chen, Nong Xiao, Xianwei Zhang, and Yutong Lu. 2025. coMtainer: Compilation-assisted HPC Container Images with Enhanced Adaptability. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3712285.3759790

1 Introduction

With the continuous advancement of open science [37], the emergence of new application scenarios such as artificial intelligence [51], and democratization of computational resources, HPC systems have been increasingly interconnected [15, 32, 48, 50], further leading to more frequent cross-platform transfers of both application and data. Concurrently, the post-Moore's Law era [22] has witnessed an explosion of heterogeneous computing architectures [12], pushing HPC systems toward unprecedented diversity [35, 56]. While such architectural diversification enables specialized performance gains, it simultaneously introduces substantial challenges of application development and deployment [34], which must be deeply adapted to specific HPC systems to fully exploit the computational capabilities.

Featuring almost no runtime overhead [7, 25, 30, 54, 57], which is critical for HPC applications, containers have emerged as an effective solution and have been extensively validated in large-scale production deployments. The HPC community has further proposed several tailored solutions, including Apptainer/Singularity [31], Charliecloud [43], Sarus [9], and Shifter [19], to help exchange and run HPC applications. As shown in Figure 1, HPC container workflow generally mirrors cloud computing paradigms: (1) Users build application images on local machines, encapsulating all necessary components including programs, data, configurations and third-party dependencies; (2) These images are then distributed via repositories; (3) Remote HPC systems pull and execute the container images. This process is enabled by containers' ability to provide uniform and isolated OS-level execution environments, ensuring consistent application behavior across diverse HPC platforms.

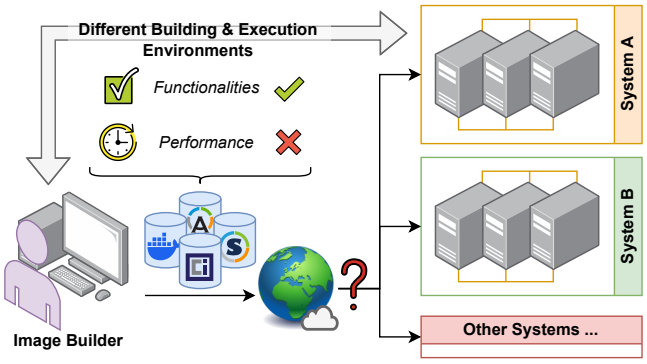


Figure 1: The current workflow of (HPC) container images, which is incapable to deliver high performance because of the ignorance of target HPC systems on the image builder side.

In contrast to conventional deployment scenarios of general computing, HPC containers must satisfy not only functional requirements but also stringent performance demands. To fully harness the computation capabilities of modern HPC systems, applications must be tightly coupled with the underlying hardware and software stacks. This often involves using dedicated libraries, compiler toolchains, and environment settings specifically tailored for the target HPC system [7, 8, 47, 52, 55]. However, existing HPC container technologies fall short of these requirements due to a fundamental limitation: containers only provide runtime environments, and the responsibility for making programs and images falls entirely on the users.

In the current container workflow, container images are built on the user side but executed on remote HPC systems. Users typically create generic system-agnostic images using the default software stack of the base image to ensure broad compatibility. However, when such generic images run on remote HPC systems, they often suffer from performance degradation or even execution failures due to the insufficient coupling between containerized applications and the underlying HPC systems [46, 47, 55]. We refer to this as the **adaptability issue** between HPC container images and HPC systems. An alternative approach is to build dedicated images specifically optimized for a target HPC system, but this inevitably leads to portability issues when using the same image on various systems. In essence, there is an inherent trade-off between the portability and performance for HPC container images.

To address this adaptability issue, we believe that the key lies in finding a better division of responsibilities between the user side and the HPC system side. An ideal solution would allow users to publish generic, standardized application images, while enabling the system side to specialize these images according to its specific configuration, thus avoiding performance loss caused by differences between the two sides. One feasible approach is thus to establish an image system which allows to distribute the applications at higher semantic levels, e.g. high-level language codes or compiler intermediate representations (IRs) to leverage the system irrelevance and offload system-specific build processes to the system side. This would keep the neutrality of the image and open up space for system-specific optimizations as well.

Driven by this insight, we propose `coMtainer`, a compilation enhanced container image building framework designed to alleviate the adaptability issue in HPC environments. Compared to existing containers, `coMtainer` extends the content of the image by incorporating information from its build process. Specifically, `coMtainer` analyzes the image construction procedure, collects additional intermediate data from the build environment and embeds them into the final image content. Such extra data enables system-side image rebuilding, thereby supporting the injection of system-specific optimizations. With `coMtainer`, users can create and distribute standardized application images in the most generic manner. Meanwhile, the HPC systems can specialize these standardized images according to their dedicated configurations, eliminating performance losses caused by the adaptability issue. Furthermore, the additional build-process information included in `coMtainer` images can facilitate other advanced optimizations on the system side, further improving application performance beyond the basic adaptation.

In summary, the contributions of this paper are:

(1) We identify that discrepancies between the user-side build environment and the system-side execution environment in current HPC container workflows can lead to incomplete performance fulfillment or even runtime failures. We define this challenge as the **adaptability issue** between HPC container images and HPC systems.
(2) We propose a novel approach that embeds additional build-time data into container images, thus enabling the system side to take over portions of the build process originally performed on the user side. This method ensures the portability of container images while allowing system-specific optimizations.
(3) We implement `coMtainer`, a prototype framework designed to validate our approach. We outline the overall workflow of `coMtainer`, introduce its internal components along with key implementation details, and evaluate its effectiveness on real HPC applications and systems.
(4) Evaluation results demonstrate that the `coMtainer` prototype can process a wide spectrum of real-world HPC applications, recovering performance losses caused by the adaptability issue. Additionally, it enables advanced optimizations such as Link-Time Optimization (LTO) and Profile-Guided Optimization (PGO), delivering extra performance improvements.

2 Background

2.1 HPC Environment Differences

HPC systems are clusters, i.e., supercomputers, with massive homogeneous nodes. The same hardware configuration is applied to all nodes to ensure that programs run seamlessly across the entire system. However, differences between HPC systems can be as significant as those between personal computers (PCs), encompassing factors such as CPU architectures, memory size and bandwidth, attached accelerators, and more. Additionally, HPC systems are characterized by high-speed interconnection networks that vary

in terms of latency, bandwidth, topology, and other performance metrics.

These hardware-level differences propagate to the software layer, resulting in system-specific drivers, optimized libraries, and customized environment settings. Beyond that, similar to PCs, different HPC clusters often host vastly different application software stacks and the file system structure. To develop an application that unleashes the full power of an HPC system, the programs must be deeply coupled with the system's software stack. As a typical example, many vendors of HPC systems provide their own MPI implementations to utilize the underlying high-speed network, like Cray MPI [20] for HPE systems, Fujitsu MPI [39] for the Fugaku supercomputer, the Intel MPI Library, etc.

Effectively utilizing an HPC system therefore requires users to be intimately familiar with its hardware and software environment — just as they would with their PCs. As a result, users tend to stick with a single familiar system for most of their work. However, HPC systems are typically shared among users, and regular users do not have *root* access to conduct low-level operations. This lack of administrative privileges introduces significant challenges for installing and developing software applications on these systems [18, 21], which has in turn driven the adoption of container technologies in the HPC community.

2.2 HPC Containers

As a lightweight OS-level virtualization technology, containers create isolated, standalone OS environment using kernel mechanisms such as namespaces [45]. Within these environments, applications can run with their own user-defined software stacks (UDSS) [44], ensuring that containerized applications behave consistently across different systems. Beyond UDSS, by encapsulating the entire runtime environment including data, dependencies, and configurations, containers also bring reproducibility to HPC applications, a long-standing challenge in scientific researches [36].

Container technologies can fundamentally transform HPC application development by decoupling the development and deployment phases through standardized runtime environments. This paradigm enables developers to construct and validate applications locally before packaging them into portable container images for deployment across heterogeneous HPC systems. This approach dramatically simplifies development, accelerates iteration, and facilitates CI/CD practices in scientific computing [21, 27]. In response to these benefits, the HPC community has developed specialized container solutions including Apptainer/Singularity [31], Charliecloud [43], and Shifter [19], which have gained substantial adoption in production environments [26, 27, 38], and are expected to see expanded usages [60].

2.3 Building HPC Application Image

HPC applications are usually developed with compiled languages like C/C++/Fortran for the sake of performance. To containerize these applications, it is recommended to use a *multi-stage build* [4] approach to generate application images, as this helps reduce the final image size. Figure 2 illustrates an example of a two-stage build process, in which two containers are created. The first container (`build`), installs build-time dependencies such as compiler toolchains and libraries, then compiles the application source code and generates program binaries. The second container (`dist`) is usually based on the same base image but only includes the runtime dependencies and compiled binaries from the `build` container. This eliminates unnecessary build-time files from the final image and significantly reduces its size.

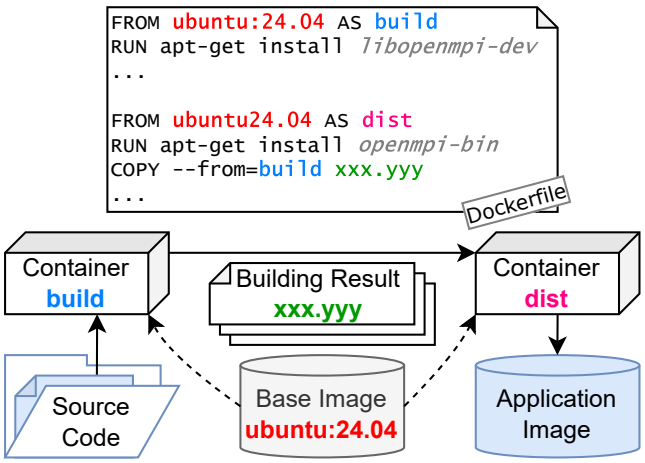


Figure 2: A two-stage container image build process and its corresponding `Dockerfile`.

3 Motivation

In this section, we introduce the origin of the adaptability issue and our motivation from the perspective of an image maker. Suppose we are developers of a certain HPC application and intend to publish our work in the form of container images. As illustrated in Figure 2, we need to compile the source codes and generate the program binaries on our user-side machine, then package them along with third-party libraries, environment configurations, and data files—into an eventual container image.

However, the build process is often far from straightforward. To achieve optimal runtime performance, we need to select the best build configuration (compiler options, 3rd-party libraries, etc.) for the target systems. But this leads to a critical challenge: we have no prior knowledge with the HPC systems on which the image will eventually be run. Hence in practice, most application developers tend to create generic images using default toolchains and software stacks from mainstream base images like *Ubuntu*/*Debian*/*Alphine*, targeting the core subsets of a few mainstream ISAs and HPC clusters. It is important to note that even within the same ISA, there can be significant performance differences across vendors and between different product lines from the same vendor. These system-specific factors are inaccessible at the time of building images, leading to a mismatch between the general-purpose image and the specific HPC system, which is the essence of the adaptability issue.

The conventional container workflow faces several fundamental challenges in HPC environments. First, even if we could identify several possible target HPC systems in advance, it would still be infeasible to account for all of them. Moreover, building highly optimized, system-specific container images for each target demands

significant time and efforts, substantially increasing the burden of both releasing and maintaining the application. Let alone the potential compatibility issue about running the remote system's software stack on the user side during image building. On the other hand, if the target HPC systems are known and fixed, separating the container workflow into building and execution phases may seem unnecessary, since applications can be developed, built, and deployed directly on the target systems. Collectively, these factors explain why the conventional container workflow often leads to performance degradation in HPC scenarios caused by the adaptability issue.

To further illustrate the severity of this performance loss, we examine *LULESH*–a representative HPC application–on a single node from our x86-64 and AArch64 systems respectively. Specifically, we compare its performance under two scenarios. In the first scenario, we create a generic containerized version of *LULESH* using a mainstream base image (`ubuntu:24.04`) with the default toolchain and software stack it provides. In the second scenario, we build *LULESH* natively on our HPC systems, incrementally enabling a series of optimizations without modifying its source code. Figure 3 presents the experimental results, where `COST` shows the time cost and other legends show the effect of optimizations.

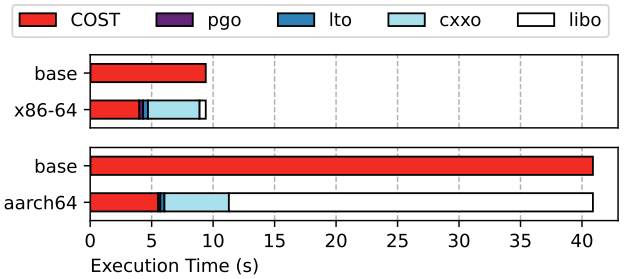


Figure 3: The performance of *LULESH* in the generic image compared to that of the native version optimized for x86 and ARM target HPC systems.

As shown in Figure 3, system-specific optimizations yield substantial performance improvements. We enumerate the applied optimizations as below:

- `libo`: Replacing default libraries in the image with optimized alternatives from the system's native software stack, such as vendor-provided C/C++ standard libraries and third-party libraries that have already been optimized for the target system.
- `cxxo`: Using the target system's native compiler toolchain (in this case, the C++ compiler).
- `lto`: Enabling Link Time Optimization (LTO), which can significantly increase compilation time and hence make it prohibitive on the user side, yet feasible on the system side where ample compute resources are available.
- `pgo`: Enabling Profile-Guided Optimization (PGO), which collects runtime profiling data to guide the compiler optimizations, but also makes it highly sensitive to the target system's characteristics.

Overall, `libo` and `cxxo` bring in up to 50% and 72% time reductions on our x86-64 and AArch64 systems respectively, which effectively recovers the performance lost due to the adaptability issue and achieves performance comparable to a native build. Furthermore, `lto` and `pgo` yield an additional 17.5% and 9.6% performance boost respectively on top of the already optimized versions. It is also worth noting that many other advanced optimizations (like binary-level layout optimization [40, 41, 59]) are not included here, suggesting greater space for potential performance gains.

4 Design

To address the adaptability issue and enable the optimizations listed in §3 for HPC containers, we envision a compilation-assisted workflow (i.e. the `coMtainer` procedure) as illustrated in Figure 4. The workflow introduces an analysis procedure on the user side, where additional build-time data is collected from building processes of the image and inside applications. The data is layered upon the original image, forming an *extended image*, and used by remote HPC systems later to rebuild and redirect the image, creating optimized versions tailored for themselves.

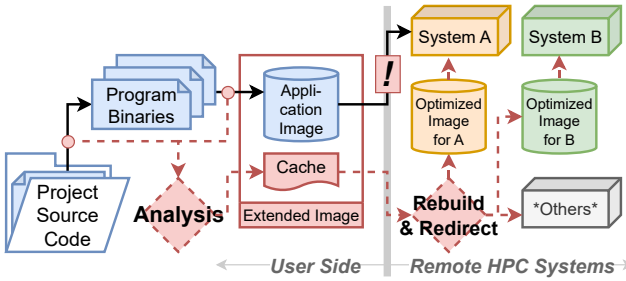


Figure 4: A brief illustration of the `coMtainer` workflow.

We put this idea into practice and develop the image building framework–`coMtainer`, which we will introduce in a top-down fashion in this section. First, we detail the `coMtainer` workflow. Next, we introduce its toolset architecture that supports the entire workflow. We then delve into the core components of the toolset , i.e., the process representations. Following that, we list the optimizations implemented in our current prototype, which is used in the experimental section later. Finally, we provide some key implementation details and discussions about design choices.

4.1 The `coMtainer` Workflow

The left part of Figure 5 depicts the overall workflow of `coMtainer`, which is based on the conventional two-stage container image building process, with two key modifications:

1) The two build-stage containers now use `coMtainer` 's `Env` and `Base` images respectively, rather than using an identical mainstream base image. This change is reflected in the Dockerfile as shown in Figure 6. Notably, `coMtainer`'s `Env` and `Base` images remain compatible with standard base images, ensuring that the modified Dockerfile works as before.

2) After the two-stage build process is completed, `coMtainer` reintroduces the generated image back into the `build` container. Within this container, `coMtainer` 's toolset performs an in-depth

analysis of the image, collects necessary files from the environment, and adds a new cache layer to the image, thus forming an coMtainer *extended image*. The extended image remains compliant with the OCI standard, thereby allowing it to be pushed to OCI-compliant image registries and seamlessly integrated into existing OCI ecosystems.

```
FROM ubuntu:24.04 as build
...
FROM ubuntu:24.04 as dist
COPY --from=build ...
...
```

```
FROM comt:ubuntu24.env as build
...
FROM comt:ubuntu24.base as dist
COPY --from=build ...
...
```

Figure 6: Modification (right) to the user-side `Dockerfile`

Assuming we are building a container image named `xxx`, the following commands illustrate the semantics performed in `coMtainer-build` on the user side. We use `buildah` for demonstration here, as it adheres most closely to the OCI standard. The first two commands build the two stages of the `Dockerfile` as separate images. The third command exports the `xxx.dist` image as an OCI layout directory (`./xxx.dist.oci`). This directory is then mounted into the `xxx.build` container, where the `coMtainer-build` command is executed. When the command completes, the cache layer will be generated in `./xxx.dist.oci`, making the directory a coMtainer extended image.

```
$ buildah build --target build -t xxx.build .
$ buildah build --target dist -t xxx.dist .
$ buildah push xxx.dist oci:./xxx.dist.oci
$ buildah from --name xxx.build
  -v .../xxx.dist.oci:/.coMtainer/io xxx.build
$ buildah run xxx.build -- coMtainer-build
```

The right part of Figure 5 shows the rest of workflow on remote HPC systems. After the extended image is pulled from the repository, the system then creates a `rebuild` container based on coMtainer 's Sysenv image. This container utilizes the additional data carried in the cache layer of the extended image to rebuild optimized programs, configurations, and other system-specific settings. The results of this process are collected into a `rebuild` layer, which is added to the extended image, forming the coMtainer *rebuilt image*. Next, the system creates an empty `redirect` container from the Rebase image. Inside this container, coMtainer 's toolset extracts the necessary data from the rebuilt image and configures the environment accordingly. Once the setup is complete, the system commits the `redirect` container, producing the final optimized image, which is now fully adapted to the target HPC system. Note that the rebuilding and redirecting can be performed many times during the image's lifetime, which thus enables optimizations requiring runtime information like PGO.

Following the previous example, the remote side performs the semantics as the following commands. The first two commands perform the `rebuild` process, while the last three commands handle the `redirect` process:

```
$ buildah from -v .../xxx.dist.oci:/.coMtainer/io
  --name xxx.rebuild Env-ubuntu24
$ buildah run xxx.rebuild -- coMtainer-rebuild
$ buildah from -v .../xxx.dist.oci:/.coMtainer/io
  --name xxx.redirect Rebase-ubuntu24
$ buildah run xxx.redirect -- coMtainer-redirect
$ buildah commit xxx.redirect oci:./xxx.redirect.oci
```

4.2 Toolset Architecture

The coMtainer toolset is implemented as a set of Python scripts embedded within the Env, Sysenv, and Rebase images (and their derived containers) to support the overall workflow. Inspired by the classic three-phase compiler architecture, it consists of a front-end, back-end, and process models functioning like the intermediate representation (IR), with the addition of system adapters and the cache storage. Just like usual compilers, the core of the toolset is the process models, which are composed of three parts: the compilation model, which captures individual compilation processes; the build

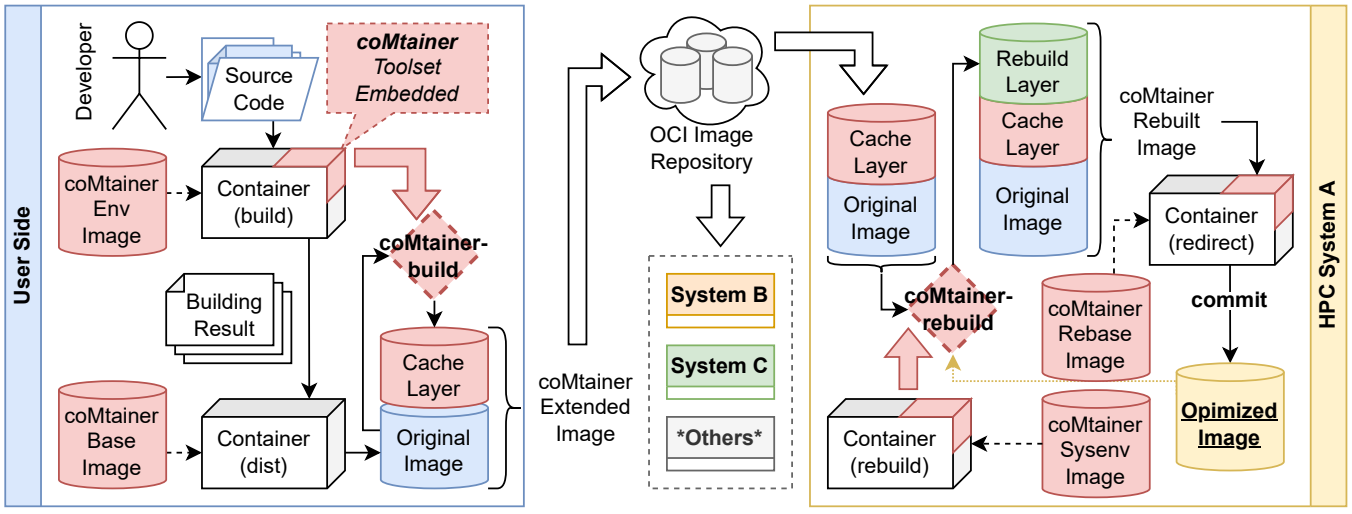


Figure 5: Detailed user and system side workflows of coMtainer.

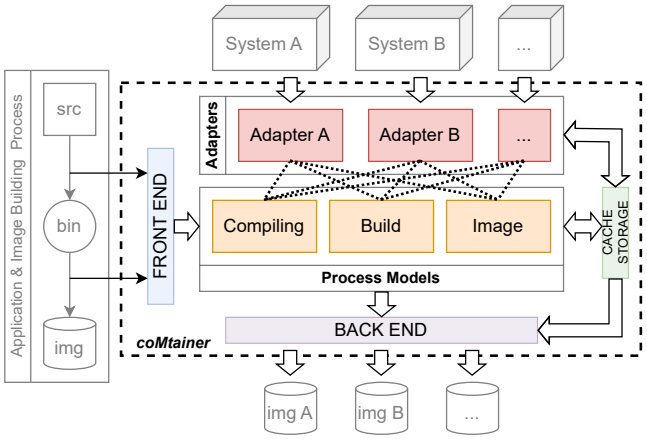


Figure 7: Architecture of `coMtainer` toolset.

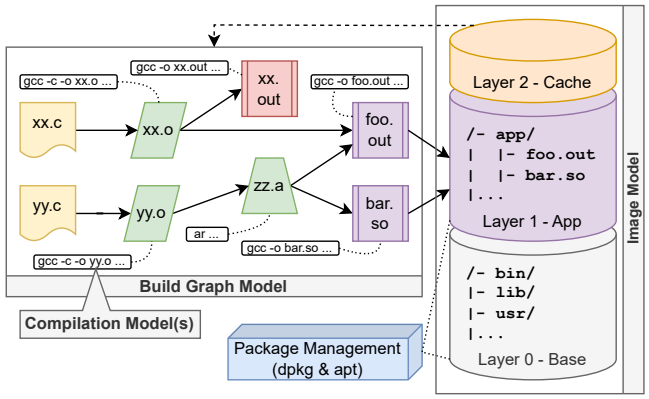


Figure 8: The process models of `coMtainer`.

model, which represents the data conversions in the entire build process with directed acyclic graphs; and the image model, which represents the structure and composition of the final application image.

The front-end works on the user side, records and parses the complete build workflow to generate the three models. The back-end works on the system side, retrieves data from cache storage and produces the optimized system-specific image. Cache storage enables data transfer between the user and system sides while ensuring compatibility with OCI formats. System adapters, akin to compiler optimization passes, operate on independent copies of the process models, tailoring transformations to specific HPC systems. These adapters analyze and modify process models, collect additional data from the build environment, and perform the image rebuilding and redirection on the target system.

All components in the architecture except the system adapters are universal across different systems. Since adapters are tied to specific HPC systems, they are designed as extensible plugins. However, the toolset includes built-in adapters for common HPC setups, which have broad applicability. Therefore, `coMtainer` itself is an infrastructure framework, and adapter plugins should be developed by the system side and integrated into the framework.

4.3 Process Models

The process models form the core of the `coMtainer` toolset. They model the entire build process of an application image and represent the process as structured data to enable analysis and transformation. They consist of three components, which are as shown in Figure 8:

Image Model represents the structure and content of the application image. The files are categorized into five types, including those from the base image, those from the package manager, those from the build process, platform-independent data, and unknown origins. This classification guides system-side transformations, particularly file replacements. Those files generated by the build process are associated with the nodes in the build graph model.

Build Graph Model is a DAG that represents all data transformations within the build process. Its structured nodes resemble syntax tree nodes in compilers rather than homogeneous nodes in graph databases. Each node tracks its dependencies , namely incoming edges, and stores metadata for analysis and transformation, such as the command lines that generate the node.

Compilation Models are specialized sub-models that capture the generation process of individual nodes. Different node types correspond to different compilation models, serving distinct analysis and transformation purposes.

The package management system plays a key role in enabling system-side optimizations at the image level, such as package replacement. The build graph is extensible and should be designed accordingly along with the compilations models, allowing support for new language ecosystems and application domains by adding new node types. For the HPC applications written in C/C++/Fortran focused by this paper, our build graph currently models source files, `.a`/`.o`/`.so` files during compilation, and together with other node types. The compilation model of `.a` nodes represents the archive contents, while those of `.a`/`.o`/`.so` nodes are structural data representing GCC command lines - Deriving this compilation model was a non-trivial task, requiring us to manually extract it by systematically reviewing the entire GCC user manual. Due to the complexity of GCC's command-line options, we continue refining our model to ensure accuracy and compatibility.

4.4 Performance Retention and Optimizations

`coMtainer` aims to retain the lost performance that occurs when running generic container images on specific target HPC systems. This performance loss primarily stems from the lack of system-specific optimized software stack during user-side image building. By leveraging `coMtainer` 's process model, the system side gains the ability to perform system-specific optimizations in two ways. First, it can replace default libraries from the package manager with counterparts optimized for target systems. Second, during rebuilding, it can recompile the application using the system's dedicated toolchains. These optimizations often restore performance to the same level as natively building applications on the HPC system.

In addition to package replacement, `coMtainer` can also enable advanced compiler optimizations on the system side that are often overlooked on the user side. In this work, we focus primarily on

two of them, including Link-Time Optimization (LTO) and Profile-Guided Optimization (PGO).

LTO delays the final compilation of IRs until the linking stage, allowing whole-program optimization across all translation units (`.o` files). While this can significantly improve performance, it is rarely used due to the significantly lengthened compilation time, which is intolerable for normal users. However, on HPC clusters, computation resources are often abundant, making LTO a viable optimization strategy. `coMtainer` seamlessly enables LTO and can flexibly control its scope since the whole build process is represented as an explicit graph data.

PGO optimizes compilation using runtime profiling data, either from instrumented trial runs or from hardware performance counters. Despite its potential, PGO is rarely used in pre-built HPC applications due to two major challenges: (1) the difficulty of defining "typical" input data for profiling and (2) the inconvenience of collecting profiling data on remote HPC systems for recompilation. `coMtainer` overcomes these limitations by integrating the entire build and execution process, and enables a fully automated PGO feedback loop.

4.5 Implementation

`coMtainer`'s front end generates build process models by parsing the raw build process, which is the recorded history of executed command lines during the building process. The recording is performed by a simple command line hijacker program that logs the arguments, environment variables, etc., and transparently forwards the execution to the real program via `execvp`. The hijacking is achieved by replacing the default programs in the `Env` image with symbolic links to the hijacker program.

To construct the models, `coMtainer` needs to parse command lines and OCI images. Parsing GCC command lines is particularly challenging due to their complexity (2314 options in total), and parsing OCI images requires a POSIX file system simulator to compute the final file system state after applying all image layers. `dpkg/apt` data inside the image are parsed further to get the dependency list needed by the image model.

System adapters are implemented as Python modules integrated into the `coMtainer` toolset. Each adapter consists of callback functions executed within the `Env`, `Sysenv`, and `Rebase` containers. The backend sets up the `redirect` container by installing the runtime dependencies and extracting files from the rebuild cache. The cached files are placed at the same path as the original image, and the container's final state is committed as the optimized image. The cache storage provides directory services to system adapters, encodes their data into new layer tarballs, generates new `config.json` and `manifest.json` files to mark the tarballs as new images so that the system side can pull them as needed. Thanks to the layered nature of OCI images, the injection of additional data introduces no changes to the original image.

4.6 Discussion

We begin by addressing a common misconception: given that the container image build process is usually system-agnostic, why not simply package and transfer the `Dockerfile` along with its build context to the target HPC system for rebuilding, rather than developing `coMtainer`? The answer is straightforward that with current container build tools, the same `Dockerfile` and build context will always produce an identical image, regardless of where the building is performed. Thus the adaptation is impossible even when the image is built on the target system. As a result, to truly address the adaptability issue, we must model the internal structure of the software stack within the image. This level of introspection and adaptability is fundamentally lacking in today's containers, and `coMtainer` is designed precisely to bridge that gap.

`coMtainer` currently relies on the package manager of the base image to analyze the application software stack. Users must use packages from the designated package repositories chosen by the `Base` image, and ensure that their application's package dependencies are not tied to a specific version–for example, by relying on private APIs or bug features. Only in this way can the dependencies be substituted by optimized equivalent versions provided by the target system. We believe that the package ecosystems of major Linux distributions are sufficiently comprehensive to meet the needs of most HPC applications. Additionally, third-party programs compiled and built in the container's `Env` environment can also be treated as part of the application by `coMtainer`, ensuring good applicability. Currently, our prototype only implements parsing for dpkg/apt and supports Debian-based distributions only. However, our approach is equally applicable to other package managers, such as *RPM*.

Finally, is embedding high-level build-time data like the source code into the container image too demanding? We believe it is not, as the included sources don't have to be in their original form — they can be obfuscated to protect intellectual property while still enabling all the system-side adaptation and optimizations. Besides, we can use other higher-level IRs, such as LLVM IR as alternatives to source code. But this approach limits package replacement flexibility since many packages only guarantee API compatibility. Once compiled, the application becomes tightly coupled with specific package versions. Finally, we argue that if one attempts to overcome system differences by raising the abstraction level of distribution, this problem is unavoidable, as source code is at the highest abstraction level. If an application's source code is not portable across systems, then no other approach can enable efficient cross-platform execution.

5 Evaluation

5.1 Experimental Methodology

5.1.1 Testbed. The detailed configuration of our HPC systems is given in Table 1, including both an x86-64 cluster and an AArch64 cluster. Images used in the experiments are built with *Buildah* on local workstations and executed with *Charliecloud* on the remote HPC system.

5.1.2 Workloads. The evaluated workloads are given in Table 2, including nine popular HPC benchmarks of HPL [33], HPCG [16], LULESH [28], those from Mantevo benchmark set [14]) and 2 large-scale real-world HPC applications (LAMMPS [53] and OpenMX [10]. Also, it should be noted that HPCG is designed to be run for a fixed

Table 1: Our x86-64 and AArch64 HPC systems

	x86_64	aarch64
CPU	2 x Intel Xeon Platinum 8358P @ 2.60GHz	1 x Phytium FT-2000+/64 @ 2.2GHz
RAM	512GB	128GB
OS	Ubuntu 22.04	Kylin Linux Advanced Server V10
Nodes	16	16

time and output the GFLOPS value. For consistency, we convert the GFLOPS value to the theoretical time cost.

Table 2: Workloads (Wkld) used in evaluation.

App	Wkld	LoC	App	Wkld	LoC
/	hpl	37556	/	hpccg	1563
/	hpcg	5529	/	miniaero	42056
/	lulesh	5546	/	miniamr	9957
/	comd	4668	/	minife	28010
lammps	chain chute eam lj rhodo	2273423	/	minimd	4404
			openmx	awf5e awf7e nitro pt13	287381

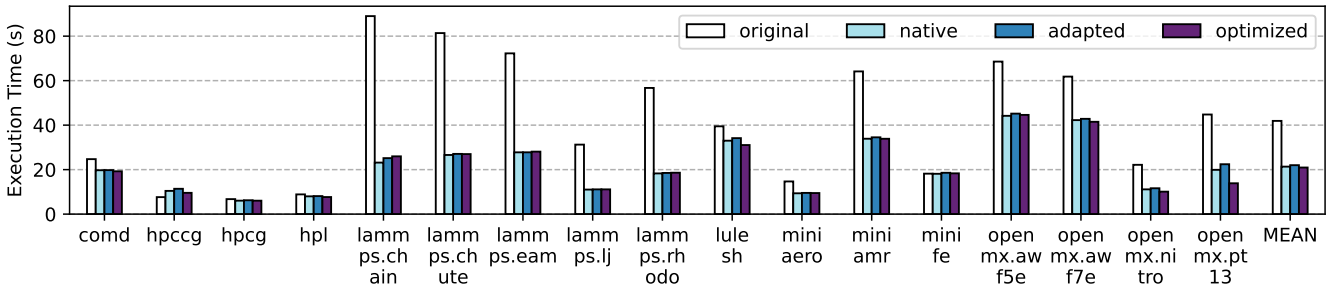
5.1.3 Metrics and Schemes. We evaluate `coMtainer` with the following metrics: 1) performance, and 2) image size, for both our x86-64 and AArch64 HPC systems. For performance, we compare the following schemes:

- `original`: the general image built with the default toolchain and software stack from the standard base image.
- `native`: native programs built and run on the target HPC systems.
- `adapted`: `coMtainer` images adapted with target system's toolchain and software stack.
- `optimized`: adapted `coMtainer` images further optimized with LTO and PGO.

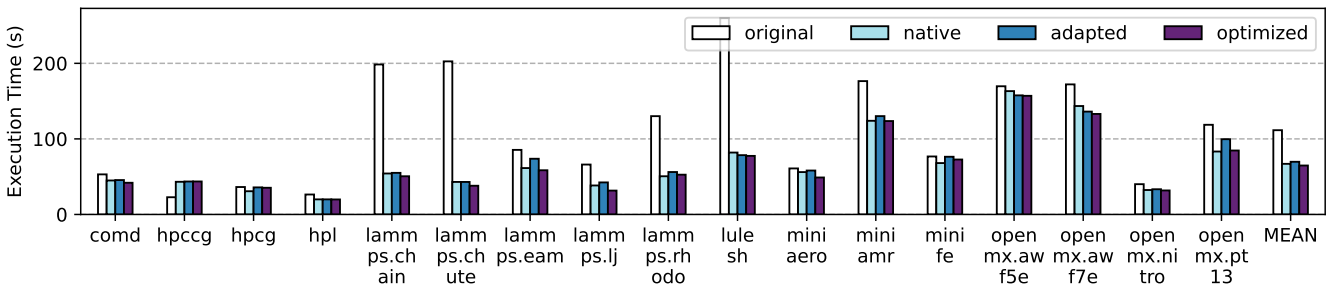
5.2 Performance Retention

Figure 9 presents the execution time of all workloads under the four schemes of `original`, `native`, `adapted`, and `optimized`. It can be observed that, for the majority of workloads, building and running the program natively on the system (`native`) significantly outperforms `original` images, with an average performance improvement of 96.3% on the x86-64 system and 66.5% on the AArch64 system. But when `adapted` by `coMtainer`, the average execution time (22.0s and 69.7s) becomes comparable to that of `native` (21.35s and 67.0s), indicating that `coMtainer` effectively retains performance.

`coMtainer`'s adaptation yields immediate and significant improvements compared to `original` images. We owe this to the system-specific software stack, including dedicated toolchains and optimized library dependencies. Notably, the two large applications

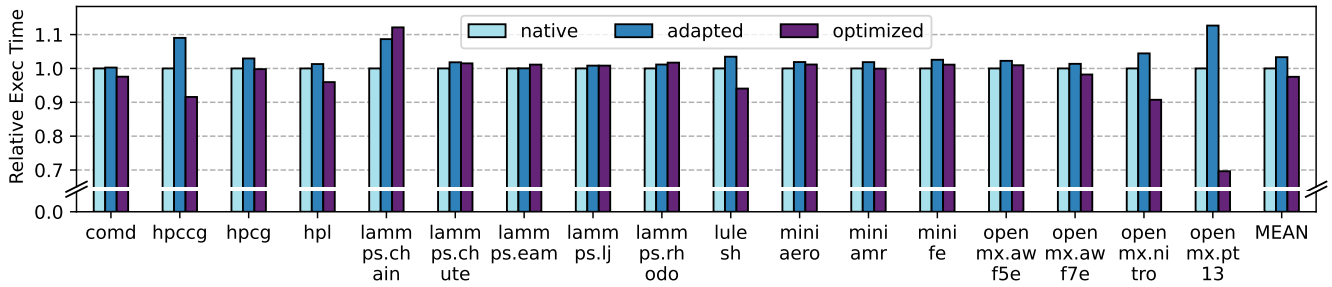


(a) Execution time on the x86-64 system.

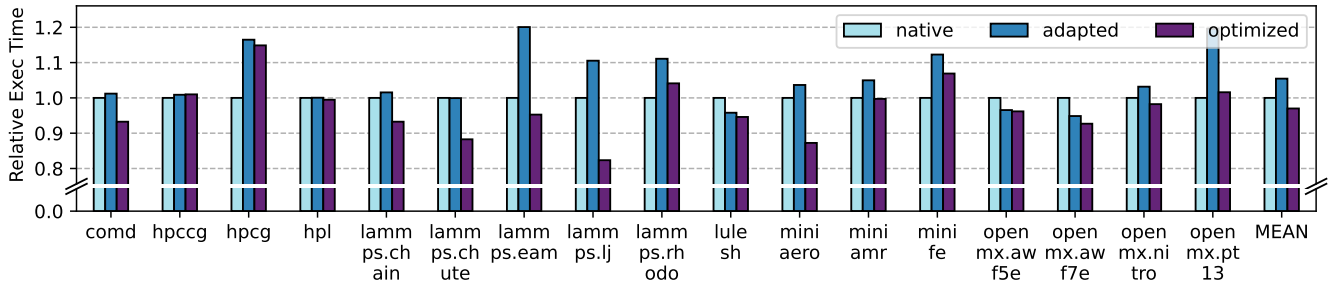


(b) Execution time on the AArch64 system.

Figure 9: Performance results. Execution time of each workload for varying schemes (the lower the better). The `adapted` images achieve the same performance as `native` builds, and `optimized` images is even better.



(a) Relative execution time on the x86-64 system.



(b) Relative execution time on the AArch64 system.

Figure 10: Relative execution time to the `native` builds (the lower the better).

`lammps` and `openmx` show a remarkable maximum improvement of 253% and 99.7% on the x86-64 system respectively, indicating that `coMtainer` can be more profitable for large applications. `lulesh` shows a huge improvement of 231% on the AArch64 system in contrast to 15.6% on the x86-64 system. This is because `lulesh` becomes communication-intensive on large scales. However, the MPI library in `original` fails to utilize the system's specialized high-speed network due to the lack of dedicated plugins, resulting in significantly higher communication overhead. On the other hand, since the communication overhead dominates when `lulesh` scales to 16 nodes, the improvement here becomes unobvious compared with the result in Figure 3. Finally, `hpccg` is the only workload that shows performance degradation in `native` and `adapted`. We attribute this to the over-aggressive optimizations of system-specific compiler toolchains.

We conclude that the results show the benefits of `coMtainer`, which can recover the lost performance caused by the adaptability issue, especially for large applications.

5.3 Performance Optimization

Beyound the adaptation, we can further apply advanced optimizations using the additional build-time data embedded in `coMtainer` images. In our evaluation, we experiment with two advanced compiler optimizations of LTO and PGO. The reduced execution time relative to `native` is shown in Figure 10a and Figure 10b for the x86-64 system and AArch64 system.

Overall, the two optimizations bring an extra performance improvement of 5.6%(AArch64) / 8%(x86-64) compared to `adapted`, and 3.4%(x86-64) / 3%(AArch64) compared to `native`. The optimization effects vary greatly across different workloads. On the x86-64 system, `openmx.pt13` achieves the best improvement of 30.4%, but `lammps.chain` gets a degradation of -12.1%. Similar trends are observed on the AArch64 system, with a maximum improvement of 17.7% for `lammps.lj` and a minimum of -14.9% for `hpcg`. This variation is less pronounced on x86-64, possibly due to its more mature compiler toolchain, whose default optimizations already resemble those achieved by LTO and PGO.

The performance gains from these advanced optimizations seem negligible compared to those from adaptation. But the results align with the general understanding of advanced compiler optimizations, whose effectiveness is highly application-dependent. The pronounced imbalance in the results of `LTO` and `PGO` aligns with expectations that program execution time often concentrates in a few hot-spot code segments based on principle of locality; significant impact on overall runtime only occurs when optimizations affect these regions. Given the randomness and unpredictability of optimization effects, developers usually perceive them as unprofitable, given that the additional compilation time cost along and manual efforts. Thus they tend to use a conservative set of optimizations. This reinforces the motivation behind `coMtainer`, which automates the optimization process, leverages idle compute resources to absorb the compilation cost, and amplifies runtime benefits through large-scale deployment in HPC clusters, which makes aggressive optimizations practically worthwhile.

The above results reveal that `coMtainer` successfully opens up optimization space and serves as a framework capable of accommodating many advanced compiler optimizations. Thus far, we

have only tried `LTO` and `PGO`. It is highly likely that `coMtainer` image will become even better after the inclusion of more advanced optimizations, which are left as future work.

5.4 Image Size

Finally, we evaluate the design overhead from the user's perspective. Since `coMtainer` operates exclusively on container images, it introduces no overhead for container execution. The primary cost of improving system adaptability is to include the additional cache data into the image. Therefore, we measure the image size and cache layer size for each application.

Table 3 lists the size of the `original` images and their corresponding cache layers. The size of the `coMtainer` extended image can be considered as the sum of the two. In our evaluation, the cache layer mainly contains the source files involved in the building process. Since the build processes are identical for the x86-64 and AArch64 systems, the cache layer sizes vary minimally.

We can see from the table that the x86-64 `original` images are significantly larger than the AArch64 images, indicating that x86-64 has a more bloated software stack. Across both architectures, the cache layer is significantly smaller than the `original` images—with a maximum of 7.1% on x86-64 and 11.3% on AArch64 of the `original` image size. Hence `coMtainer` incurs little overhead for image distribution.

Table 3: Size (in MiB) of `original` images and cache layers.

App	Image (x86-64)	Image (AArch64)	Cache
comd	170.36	94.87	0.75
hpccg	170.40	94.77	0.59
hpcg	170.04	95.37	0.80
hpl	170.76	94.86	1.32
lulesh	170.29	96.12	0.66
miniaero	170.12	94.63	0.62
miniamr	170.10	94.62	0.80
lammps	203.30	127.23	14.42
openmx	440.97	359.14	23.99

5.5 Attempts to Cross ISA

In principle, if all the sources involved in building a container image are ISA-agnostic, and the application's direct dependencies have implementations across different ISAs, then `coMtainer` should, in theory, be able to leverage the data in the cache layer to rebuild and redirect a container image from one ISA to another – enabling cross-ISA workflow of container images. While cross-ISA support was not the initial goal of `coMtainer`, this potential is both intriguing and worth exploring.

To investigate this, we pull the x86-64 `coMtainer` extended images for all applications listed in Table 2 and attempt to process them on the AArch64 system. As expected, most of the images fail due to ISA-specific contents in their build scripts or source code, e.g. inline assembly, preprocessor macros, ISA-specific compiler flags, etc. However, once we relax our constraints slightly and allowed minor modifications to their build scripts, we find that many of the applications can successfully be rebuilt and redirected across ISAs using `coMtainer`.

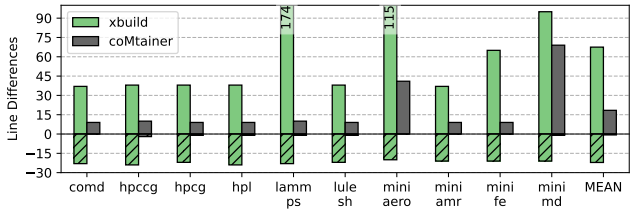


Figure 11: Line differences with `original` of image build scripts using 1) cross compilation and 2) `coMtainer`. The bar above zero show added lines, and the bar below zero with hatches show deleted lines.

Figure 11 lists the applications that were able to cross ISA with `coMtainer`, along with the number of line changes made to their image build scripts compared to `original`. For comparison, the figure also shows the modifications required when using a traditional cross-compilation approach (`xbuild`). The results show that `coMtainer` greatly reduces the amount of manual work compared to cross-compilation. On average, users only need to change 5 lines in the build scripts with `coMtainer` — just 10% of the effort required for cross-building, which are averaged to 47 lines.

This demonstrates that `coMtainer` holds real potential for enabling ISA-agnostic container images, laying a promising foundation for future efforts in building a cross-ISA container ecosystem.

6 Related Works

6.1 Distribute HPC Applications

Performance is a major challenge for distributing HPC applications. One of the most straightforward solutions is to distribute source code directly with auto-configuration systems like GNU Automake, CMake, Meson, etc. These tools help developers organize projects and configure the build process to match the host environment. However, fully unlocking application performance still requires developers to manually write build scripts. To address this challenge, Magdalena Slawinska, Jaroslaw Slawinski, et al. proposed Harness Workbench [49] and ADAPT [11] to enable automatic adaptation of HPC applications.

Building applications from source is challenging due to the "dependency hell" problem [17]. Casual users therefore prefer package managers like Debian's dpkg/apt, which distribute pre-built binaries but restrict users to fixed versions – with no consideration for performance adaptability. In contrast, the HPC community has developed source-based package managers, such as Spack [18] and EasyBuild [24]—tools that build and install applications from source using toolchains in the user's environment. When configured appropriately, these tools can ensure that applications achieve optimal performance tailored to the system.

However, containers differ from package managers in that the images can also carry data and customized configurations, encompassing arbitrary combinations of software. This makes containers not just a tool for deployment, but also a means of encapsulating

entire workflows. Thus, package managers do not overlap with the issues this paper aims to address.

6.2 Build Container Image from Source

coMtainer is not the first attempt to take over the building stage of container images. A number of existing tools aim to create container images directly from source code, with the goal of simplifying containerization and enhancing cross-platform portability.

Buildpack [1] automatically detect the type of application, set up the appropriate build environment, and assemble container images directly from source code—without the need for explicit Dockerfiles or low-level configuration. Buildpack supports a wide range of programming languages and frameworks, and is widely adopted in cloud-native development workflows. Similarly, Source-to-Image (S2i) [5] provides a flexible framework that combines an application's source code with a pre-configured builder image to produce a runnable container. It emphasizes reproducibility and automation, making it particularly well-suited for web applications, scripting environments, and CI/CD pipelines. Other tools such as Jib [2] and ko [3] serve similar purposes but are tailored for Java and Go applications, respectively, simplifying containerization in those ecosystems.

While these tools successfully decouple image building from low-level configuration, they are primarily designed for cloud-native workloads and CI/CD scenarios. They fall short in high-performance computing (HPC) contexts, where fine-grained control over compilers, system-tuned libraries and hardware-specific optimizations is essential. In contrast, coMtainer targets HPC-specific needs by enabling deferred system-aware specialization of container images, bridging the gap between source-level portability and runtime performance optimization on diverse HPC systems.

6.3 The Performance of HPC Containers

After becoming popular, Docker garnered attention from the HPC community and started to be deployed on HPC systems. Quickly, multiple studies [13, 29, 58] emerged to evaluate Docker's performance on these systems, as well as efforts to optimize network communication [57], resource management [23], etc. Given that Docker is a heavyweight container product designed for microservices, people soon realized that its deep virtualization capabilities are unnecessary for HPC systems and introduce performance overhead. As a result, HPC-dedicated container products such as Singularity [31] and Charliecloud [43] have been proposed, which are proven to have the near-zero overhead by many evaluation studies [6, 25, 30, 54, 57], provided they are used properly.

Existing performance optimization efforts for HPC containers primarily focus on the container themselves at runtime, with scant attention given to the container images. However, the adaptability issue of HPC application images has been recognized early on, particularly the compatibility between applications within the image and the MPI libraries in the system. Current HPC containers' mitigation measures are to hijack the library binaries by LD_PRELOAD_LIBRARY and mounting directories from the host system. Nevertheless, this approach requires that the applications within the image be dynamically linked, and have ABI compatibility issues. In a recent work [42], Charliecloud attempts to support making images directly on HPC systems to leverage the host's software stack. However, their methodology primarily targets CI/CD scenarios and does not help the general image making workflows. To the best of our knowledge, we are the first to combine compilation techniques and container images. By addressing the adaptability issue at the compilation phase, we can fully exploit the portability of the project source code, and bypass problems such as ABI compatibility.

7 Conclusion

In this paper, we propose coMtainer, a container image transformation and optimization framework that integrates compilation techniques. We then introduce the coMtainer workflow and its key designs, demonstrating how embedding high-level build-time data into container images enables system-side rebuilding and redirection. This approach ensures the neutrality of user-side distributed images while simultaneously allowing target-specific optimizations to be applied transparently.

Our experimental evaluation shows that the current coMtainer prototype can already be applied to real-world large-scale HPC applications and successfully recovers the performance loss caused by the adaptability issue. Additionally, by embedding the build process into the image, opportunities for advanced compiler-level optimizations—such as LTO and PGO—are unlocked, achieving non-trivial performance gains across a range of workloads.

We believe our work pioneers the transformation of container images for performance optimization. Furthermore, our extensible design suggests that this methodology can be generalized well beyond traditional C/C++/Fortran-based HPC applications to high-level language ecosystems with similar needs. We are actively exploring further possibilities for container image transformation and optimization based on coMtainer, with the vision of integrating it seamlessly into future HPC software workflows.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive comments and suggestions. This work is supported by National Key R&D Program of China (Grant No. 2025YFB3003501), NSFC Grant #62461146204, and is sponsored by CCF-Huawei Populus Grove Fund (CCF-HuaweiSY202409). Xianwei Zhang and Nong Xiao are the corresponding authors.

References

[1] [n. d.]. Cloud Native Buildpacks. https://buildpacks.io/.
[2] [n. d.]. GoogleContainerTools/Jib: Build Container Images for Your Java Applications. https://github.com/GoogleContainerTools/jib.
[3] [n. d.]. Ko: Easy Go Containers. https://ko.build/.
[4] [n. d.]. Multi-Stage | Docker Docs. https://docs.docker.com/build/building/multi-stage/.
[5] [n. d.]. Openshift/Source-to-Image: A Tool for Building Artifacts from Source and Injecting into Container Images. https://github.com/openshift/source-to-image.
[6] Subil Abraham, Arnab K. Paul, Redwan Ibne Seraj Khan, and Ali R. Butt. 2020. On the Use of Containers in High Performance Computing Environments. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 284–293. doi:10.1109/CLOUD49709.2020.00048
[7] Abdulrahman Azab. 2017. Enabling Docker Containers for High-Performance and Many-Task Computing. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*. 279–285. doi:10.1109/IC2E.2017.52
[8] Maxim Belkin, Roland Haas, Galen Wesley Arnold, Hon Wai Leong, Eliu A. Huerta, David Lesny, and Mark Neubauer. 2018. Container Solutions for HPC Systems: A Case Study of Using Shifter on Blue Waters. In *Proceedings of the Practice and Experience on Advanced Research Computing: Seamless Creativity*

(PEARC '18). Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3219104.3219145

[9] Lucas Benedicic, Felipe A. Cruz, Alberto Madonna, and Kean Mariotti. 2019. Sarus: Highly Scalable Docker Containers for HPC Systems. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode (Eds.). Springer International Publishing, Cham, 46–60. doi:10.1007/978-3-030-34356-9_5

[10] Steven Boker, Michael Neale, Hermine Maes, Michael Wilde, Michael Spiegel, Timothy Brick, Jeffrey Spies, Ryne Estabrook, Sarah Kenny, Timothy Bates, Paras Mehta, and John Fox. 2011. OpenMx: An Open Source Extended Structural Equation Modeling Framework. *Psychometrika* 76, 2 (April 2011), 306–317. doi:10.1007/s11336-010-9200-6

[11] Julien Bourgeois, Vaidy Sunderam, Jaroslaw Slawinski, and Bogdan Cornea. 2011. Extending Executability of Applications on Varied Target Platforms. In *2011 IEEE International Conference on High Performance Computing and Communications*. 253–260. doi:10.1109/HPCC.2011.41

[12] Lattner Chris. 2021. The Golden Age of Compiler Design in an Era of HW/SW Co-Design.

[13] Minh Thanh Chung, An Le, Nguyen Quang-Hung, Duc-Dung Nguyen, and Nam Thoai. 2016. Provision of Docker and InfiniBand in High Performance Computing. In *2016 International Conference on Advanced Computing and Applications (ACOMP)*. 127–134. doi:10.1109/ACOMP.2016.027

[14] Paul Crozier, Heidi Thornquist, Robert Numrich, Alan Williams, Harold Edwards, Eric Keiter, Mahesh Rajan, James Willenbring, Douglas Doerfler, and Michael Heroux. 2009. *Improving Performance via Mini-Applications*. Technical Report SAND2009-5574, 993908. SAND2009–5574, 993908 pages. doi:10.2172/993908

[15] Eli Dart, Jason Zurawski, Carol Hawk, Benjamin L. Brown, and Inder Monga. 2023. *ESnet Requirements Review Program Through the IRI Lens: A Meta-Analysis of Workflow Patterns Across DOE Office of Science Programs (Final Report)*. Technical Report LBNL-2001552; 24-SN-35053. Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States). doi:10.2172/2008205

[16] Jack Dongarra, Piotr Luszczek, and M Heroux. 2013. HPCG Technical Specification. *Sandia National Laboratories, Sandia Report SAND2013-8752* (2013).

[17] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 463–474. doi:10.1145/3395363.3397388

[18] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. 2015. The Spack Package Manager: Bringing Order to HPC Software Chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2807591.2807623

[19] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. 2017. Shifter: Containers for HPC. *Journal of Physics: Conference Series* 898, 8 (Oct. 2017), 082021. doi:10.1088/1742-6596/898/8/082021

[20] Richard L Graham, George Bosilca, and Jelena Pje˘sivac-Grbovic. [n. d.]. A Comparison of Application Performance Using Open MPI and Cray MPI. ([n. d.]).

[21] Jack S. Hale, Lizao Li, Christopher N. Richardson, and Garth N. Wells. 2017. Containers for Portable, Productive, and Performant Scientific Computing. *Computing in Science & Engineering* 19, 6 (Nov. 2017), 40–50. doi:10.1109/MCSE.2017.2421459

[22] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. doi:10.1145/3282307

[23] Stephen Herbein, Ayush Dusia, Aaron Landwehr, Sean McDaniel, Jose Monsalve, Yang Yang, Seetharami R. Seelam, and Michela Taufer. 2016. Resource Management for Running HPC Applications in Container Clouds. In *High Performance Computing*, Julian M. Kunkel, Pavan Balaji, and Jack Dongarra (Eds.). Springer International Publishing, Cham, 261–278. doi:10.1007/978-3-319-41321-1_14

[24] Kenneth Hoste, Jens Timmerman, Andy Georges, and Stijn De Weirdt. 2012. EasyBuild: Building Software with Ease. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 572–582. doi:10.1109/SC.Companion.2012.81

[25] Guangchao Hu, Yang Zhang, and Wenbo Chen. 2019. Exploring the Performance of Singularity for High Performance Computing Scenarios. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2587–2593. doi:10.1109/HPCC/SmartCity/DSS.2019.00362

[26] Kwangwoog Jung, Yang-Ki Cho, and Yong-Jin Tak. 2021. Containers and Orchestration of Numerical Ocean Model for Computational Reproducibility and Portability in Public and Private Clouds: Application of ROMS 3.6. *Simulation Modelling Practice and Theory* 109 (May 2021), 102305. doi:10.1016/j.simpat.2021.102305

[27] Sabah Kadri, Andrea Sboner, Alexandros Sigaras, and Somak Roy. 2022. Containers in Bioinformatics: Applications, Practical Considerations, and Best Practices in Molecular Pathology. *The Journal of Molecular Diagnostics* 24, 5 (May 2022), 442–454. doi:10.1016/j.jmoldx.2022.01.006

[28] I Karlin, J Keasler, and J Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973, 1090032. LLNL–TR–641973, 1090032 pages. doi:10.2172/1090032

[29] Animesh Kuity and Sateesh Kumar Peddoju. 2017. Performance Evaluation of Container-Based High Performance Computing Ecosystem Using OpenPOWER. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf (Eds.). Springer International Publishing, Cham, 290–308. doi:10.1007/978-3-319-67630-2_22

[30] Mandeep Kumar and Gagandeep Kaur. 2022. Containerized MPI Application on InfiniBand Based HPC: An Empirical Study. In *2022 3rd International Conference for Emerging Technology (INCET)*. 1–6. doi:10.1109/INCET54531.2022.9824366

[31] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017-5-11. Singularity: Scientific Containers for Mobility of Compute. *PLOS ONE* 12, 5 (2017-5-11), e0177459. doi:10.1371/journal.pone.0177459

[32] Zhuozhao Li, Ryan Chard, Yadu Babuji, Ben Galewsky, Tyler J. Skluzacek, Kirill Nagaitsev, Anna Woodard, Ben Blaiszik, Josh Bryan, Daniel S. Katz, Ian Foster, and Kyle Chard. 2022. funcX: Federated Function as a Service for Science. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (Dec. 2022), 4948–4963. doi:10.1109/TPDS.2022.3208767

[33] D. Maître. 2006. HPL, a Mathematica Implementation of the Harmonic Polylogarithms. *Computer Physics Communications* 174, 3 (Feb. 2006), 222–240. doi:10.1016/j.cpc.2005.10.008

[34] Paul Messina. 2017. The Exascale Computing Project. *Computing in Science & Engineering* 19, 3 (May 2017), 63–67. doi:10.1109/MCSE.2017.57

[35] Dejan Milojicic, Paolo Faraboschi, Nicolas Dube, and Duncan Roweth. 2021. Future of HPC: Diversifying Heterogeneity. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 276–281. doi:10.23919/DATE51398.2021.9474063

[36] David Moreau, Kristina Wiebels, and Carl Boettiger. 2023. Containers for Computational Reproducibility. *Nature Reviews Methods Primers* 3, 1 (July 2023), 1–16. doi:10.1038/s43586-023-00236-9

[37] National Academies of Sciences, Engineering, and Medicine, Policy and Global Affairs, Board on Research Data and Information, and Committee on Toward an Open Science Enterprise. 2018. *Open Science by Design: Realizing a Vision for 21st Century Research*. National Academies Press (US), Washington (DC).

[38] C. Nigro, C. Deil, R. Zanin, T. Hassan, J. King, J. E. Ruiz, L. Saha, R. Terrier, K. Brügge, M. Nöthe, R. Bird, T. T. Y. Lin, J. Aleksić, C. Boisson, J. L. Contreras, A. Donath, L. Jouvin, N. Kelley-Hoskins, B. Khelifi, K. Kosack, J. Rico, and A. Sinha. 2019. Towards Open and Reproducible Multi-Instrument Analysis in Gamma-Ray Astronomy. *Astronomy & Astrophysics* 625 (May 2019), A10. doi:10.1051/0004-6361/201834938

[39] FSTJ Editorial Office. 2012. MPI Library and Low-Level Communication on the K Computer. *FUJITSU Sci. Tech. J.* 48, 3 (2012).

[40] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. 2–14.

[41] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: Powerful, Fast, and Scalable Binary Optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC 2021)*. Association for Computing Machinery, New York, NY, USA, 119–130. doi:10.1145/3446804.3446843

[42] Reid Priedhorsky, R. Shane Canon, Timothy Randles, and Andrew J. Younge. 2021. Minimizing Privilege for Building HPC Containers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3458817.3476187

[43] Reid Priedhorsky and Tim Randles. 2017. 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)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3126908.3126925

[44] Reid Priedhorsky and Tim Randles. 2017. Linux Containers for Fun and Profit in HPC. 42, 3 (2017).

[45] Rami Rosen. 2016. Namespaces and Cgroups, the Basis of Linux Containers. *Seville, Spain, Feb* (2016).

[46] Oleksandr Rudyy, Marta Garcia-Gasulla, Filippo Mantovani, Alfonso Santiago, Raül Sirvent, and Mariano Vázquez. 2019. Containers in HPC: A Scalability and Portability Study in Production Biological Simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 567–577. doi:10.1109/IPDPS.2019.00066

[47] Amit Ruhela, Stephen Lien Harrell, Richard Todd Evans, Gregory J. Zynda, John Fonner, Matt Vaughn, Tommy Minyard, and John Cazes. 2021. Characterizing Containerized HPC Applications Performance at Petascale on CPU and GPU Architectures. In *High Performance Computing*, Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek (Eds.). Springer International Publishing, Cham, 411–430. doi:10.1007/978-3-030-78713-4_22

[48] Nickolaus Saint, Ryan Chard, Rafael Vescovi, Jim Pruyne, Ben Blaiszik, Rachana Ananthakrishnan, Mike Papka, Rick Wagner, Kyle Chard, and Ian Foster. 2023. Active Research Data Management with the Django Globus Portal Framework. In *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good (PEARC '23)*. Association for Computing Machinery, New York, NY, USA, 43–51. doi:10.1145/3569951.3593597

[49] Magdalena Slawinska, Jaroslaw Slawinski, and Vaidy Sunderam. 2009. Portable Builds of HPC Applications on Diverse Target Platforms. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. 1–8. doi:10.1109/IPDPS.2009.5160915

[50] Larry Smarr, Camille Crittenden, Thomas DeFanti, John Graham, Dmitry Mishin, Richard Moore, Philip Papadopoulos, and Frank Würthwein. 2018. The Pacific Research Platform: Making High-Speed Networking a Reality for the Scientist. In *Proceedings of the Practice and Experience on Advanced Research Computing: Seamless Creativity (PEARC '18)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3219104.3219108

[51] Rick Stevens, Valerie Taylor, Jeff Nichols, Arthur Barney Maccabe, Katherine Yelick, and David Brown. 2020. *AI for Science: Report on the Department of Energy (DOE) Town Halls on Artificial Intelligence (AI) for Science*. Technical Report ANL-20/17. Argonne National Lab. (ANL), Argonne, IL (United States). doi:10.2172/1604756

[52] Shinji Sumimoto. 2012. The MPI Communication Library for the K Computer: Its Design and Implementation. In *Recent Advances in the Message Passing Interface*, Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra (Eds.). Springer, Berlin, Heidelberg, 11–11. doi:10.1007/978-3-642-33518-1_3

[53] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. 2022. LAMMPS - a Flexible Simulation Tool for Particle-Based Materials Modeling at the Atomic, Meso, and Continuum Scales. *Computer Physics Communications* 271 (Feb. 2022), 108171. doi:10.1016/j.cpc.2021.108171

[54] Alfred Torrez, Timothy Randles, and Reid Priedhorsky. 2019. HPC Container Runtimes Have Minimal or No Performance Impact. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 37–42. doi:10.1109/CANOPIE-HPC49598.2019.00010

[55] Andrew J. Younge, Kevin Pedretti, Ryan E. Grant, and Ron Brightwell. 2017. A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 74–81. doi:10.1109/CloudCom.2017.40

[56] Andrew J. Younge, Kevin Pedretti, Ryan E. Grant, Brian L. Gaines, and Ron Brightwell. 2017. Enabling Diverse Software Stacks on Supercomputers Using High Performance Virtual Clusters. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 310–321. doi:10.1109/CLUSTER.2017.92

[57] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. High Performance MPI Library for Container-Based HPC Cloud on InfiniBand Clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*. 268–277. doi:10.1109/ICPP.2016.38

[58] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. Performance Characterization of Hypervisor-and Container-Based Virtualization for HPC on SR-IOV Enabled InfiniBand Clusters. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1777–1784. doi:10.1109/IPDPSW.2016.178

[59] Y. Zhang, T. A. Khan, G. Pokam, B. Kasikci, H. Litz, and J. Devietti. 1-5 Oct. 2022. OCOLOS: Online COde Layout OptimizationS. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 530–545. doi:10.1109/MICRO56248.2022.00045

[60] Naweiluo Zhou, Huan Zhou, and Dennis Hoppe. 2023. Containerization for High Performance Computing Systems: Survey and Prospects. *IEEE Transactions on Software Engineering* 49, 4 (April 2023), 2722–2740. doi:10.1109/TSE.2022.3229221

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 We identify that differences between the user-side build environment and the system-side execution environment in current HPC container workflows can lead to incomplete performance fulfillment or even runtime failures. We define this as **the adaptability issue** between HPC container images and HPC systems.

C_2 We propose the approach that embeds additional build-time data into container images, allowing system side to take over portions of the build process originally performed on the user side. This method ensures the portability of container images while enabling system-specific optimizations.

C_3 We implement `coMtainer`, a prototype framework designed to validate our approach. We outline the overall workflow of `coMtainer`, introduce its internal components and key implementation details, and evaluate its effectiveness on real HPC applications and systems.

C_4 Evaluation results demonstrate that the `coMtainer` prototype can process a wide spectrum of real-world HPC applications, recovering performance losses caused by the adaptability issue. Additionally, it enables advanced optimizations such as LTO and PGO, further improving performance.

A.2 Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.16920424
A_2 https://doi.org/10.5281/zenodo.16920424
A_3 https://doi.org/10.5281/zenodo.16920424

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figure 3
A_2	C_2 C_3	Figures 4-9
A_3	C_4	Table 1-3 Figures 10-12

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

A_1 contains the four application images for LULESH to demonstrate the adaptability issue, with the image build scripts placed at `/src` inside the containers. There are 4 images inside: the base image and target-specifically optimized image for x86-64 and AArch64 respectively, tagged with `x86_64.base`, `x86_64.spec`, `aarch64.base`, `aarch64.spec`.

Expected Results

The target-specific images (`.spec`) should perform much better than the base images (`.base`).

Expected Reproduction Time (in Minutes)

Artifact Setup (10~60 min),
Artifact Execution (3~5 min),
Artifact Analysis (1 min).

Artifact Setup (incl. Inputs)

Hardware. The x86-64 experiment uses Intel Xeon Platinum 8358P @ 2.60GHz, and the AArch64 experiment uses Phytium FT-2000+/64 @ 2.2GHz, both on single cluster node.

Software. The host OS in the x86-64 experiment is Ubuntu 22.04 and that in the AArch64 experiment is Kylin Linux Advanced Server V10. However the results are expected not to be affected by the host environment due to the isolation of containers.

To avoid runtime overhead, it is expected that the images are executed by HPC containers like Singularity or Charliecloud, which may necessitate the conversion from OCI format to other formats.

Datasets / Inputs. No datasets or inputs needed.

Installation and Deployment. Use the images in the repository as normal. To build the images, refer to the `/src/README.md` file inside the containers.

Artifact Execution

Run the four images and measure the execution time. Some HPC container engines may be unable to use the entry commands specified in the OCI config file. In such cases, manually run the LULESH program at `/app/lulesh`. For example, using Charliecloud:

```
$ time ch-run -w ./imgdir -- \
    /bin/bash -c 'time /app/lulesh'
```

Artifact Analysis (incl. Outputs)

Simply comparing the measured time will do.

B.2 Computational Artifact A_2

Relation To Contributions

A_2 contains the `Env`, `Base`, `Sysenv`, and `Rebase` images of `coMtainer`. They together realize the `coMtainer` workflow described in the paper. The source code of `coMtainer` is placed at `/.coMtainer` inside the `Env`, `Sysenv`, and `Rebase` images.

The repository provides the user-side `Env` and `Base` images for both x86-64 and AArch64. However we can't share our system-side `Sysenv` and `Rebase` images as they contain proprietary system-specific compiler toolchains which prohibits redistribution. So we choose to provide alternative `Sysenv` and `Rebase` images based on the free LLVM toolchains instead.

Expected Results

These images should be able to reproduce the `coMtainer` workflows described in the paper, and support the image building of applications given in the A_3 test suite.

Expected Reproduction Time (in Minutes)

Artifact Setup (5~10 min),
Artifact Execution (20~60 min),
Artifact Analysis (1 min).

Artifact Setup (incl. Inputs)

Hardware. The images should be able to work on any main-stream x86-64 or AArch64 PCs.

Software. It is advised to use `buildah` to reproduce the workflow. But in theory any other OCI-compliant container engine will do.

Datasets / Inputs. No datasets or inputs needed.

Installation and Deployment. Use the images in the repository as normal.

Artifact Execution

To use the four images, one first needs to prepare a two-stage Dockerfile like this:

```
FROM comtainer:x86-64.env as build
...
FROM comtainer:x86-64.base as dist
COPY -- from = build ...
...
```

Next, we take the LULESH test case in A_3 as an example. In its directory, use the following commands to build the LULESH `build` image and LULESH `dist` image.

```
$ buildah build --target build -t lulesh.build .
$ buildah build --target dist -t lulesh.dist .
```

Use the following command to export the `dist` image into an OCI layout directory for analysis and transformation:

```
$ buildah push lulesh.dist oci:./lulesh.dist.oci
```

Use the following commands to perform the `coMtainer-build` stage in the workflow, creating the `coMtainer` extended image:

```
$ buildah from --name lulesh.build
  -v $(pwd)/lulesh.dist.oci:/.coMtainer/io lulesh.build
$ buildah run lulesh.build -- coMtainer-build
```

Then we perform the `coMtainer-rebuild` stage in the workflow:

```
$ buildah from -v $(pwd)/lulesh.dist.oci:/.coMtainer/io
  --name lulesh.rebuild comtainer:x86-64.sysenv
$ buildah run lulesh.rebuild -- coMtainer-rebuild
```

Perform the `coMtainer-redirect` stage in the workflow:

```
$ buildah from -v $(pwd)/lulesh.dist.oci:/.coMtainer/io
  --name lulesh.redirect comtainer:x86-64.rebase
$ buildah run lulesh.redirect -- coMtainer-redirect
```

Finally, to get the optimized image, commit the redirected container:

```
$ buildah commit lulesh.redirect oci:./lulesh.redirect.oci
```

Artifact Analysis (incl. Outputs)

After the `coMtainer-build` stage is performed, a new manifest tagged with suffix `+coM` should be generated in `./lulesh.dist.oci/index.json`.

After the `coMtainer-rebuild` stage is performed, a new manifest tagged with suffix `+coMre` should be generated in `./lulesh.dist.oci/index.json`.

The final redirected image `lulesh.redirect.oci` should have a file system layout compatible with the original `lulesh.dist` image, and the LULESH application inside can be used likewise.

B.3 Computational Artifact A_3

Relation To Contributions

A_3 contains 4 workloads used in the evaluation. It can build the `original`, `adapted` and `optimized` versions of application images. The scripts for building the `native` version is system-related and removed for proprietary reasons.

Expected Results

It is expected that the `adapted` and `optimized` images run faster than the `original` images. However, as the provided system-side images are based on the free LLVM toolchains, which is still for general use. The improvements can be greatly diminished compared to vendor-specific toolchain like Intel OneAPI.

Expected Reproduction Time (in Minutes)

Artifact Setup (10~60 min),
Artifact Execution (120~180 min),
Artifact Analysis (10 min).

Artifact Setup (incl. Inputs)

Hardware. Same as A_1.

Software. No additional requirements apart from A_1 and A_2.

Datasets / Inputs. No datasets or inputs needed.

Installation and Deployment. Pull the Git repository.

Artifact Execution

Each test case corresponds to a sub-folder. Use the `build-original.sh`, `build-adapted.sh` and `build-optimized.sh` scripts to build the corresponding images into the `dist.oci` directory inside. Then the images can be run with the `run.sh` aside.

Artifact Analysis (incl. Outputs)

The `run.sh` will print the time cost, showing the performance of each image version.

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

A container engine is required for evaluation. We recommend *Podman*, but *Docker* will also do. Using HPC-dedicated container engines like *Singularity* or *Charliecloud* also requires the format conversion of tested images, as described in their documentations.

Artifact Execution

The use of containers greatly simplifies the evaluation process, as described below.

Execute the following commands to try the basic version:

```
$ podman load -i lulesh_x86-64.base.tar
$ time podman run --rm -it
  docker.io/yhgu2000/lulesh:x86-64.base /app/lulesh
```

Execute the following commands to try the target-specific version:

```
$ podman load -i lulesh:x86-64.spec.tar
$ time podman run --rm -it
  docker.io/yhgu2000/lulesh:x86-64.spec /app/lulesh
```

For AArch64 versions, replace the `x86-64` image tags with `aarch64`.

Artifact Analysis (incl. Outputs)

Compare the timing result of the two versions. It is expected that the target-specific version is significantly faster than the general basic version.

C.2 Computational Artifact A_2

Artifact Setup (incl. Inputs)

This evaluation demonstrates the overall workflow of *coMtainer*. Container image build tool (like *Buildah*) is required for this evaluation. Here we take the LULESH test case from A_3 as an example.

Artifact Execution

First, pull and re-tag the images:

```
$ buildah pull oci-archive:./comtainer_x86-64.env.tar
$ buildah tag
  docker.io/yhgu2000/comtainer:x86-64.env
  comtainer:x86-64.env
$ buildah pull oci-archive:./comtainer_x86-64.base.tar
$ buildah tag
  docker.io/yhgu2000/comtainer:x86-64.base
  comtainer:x86-64.base
$ buildah pull oci-archive:./comtainer_x86-64.sysenv.tar
$ buildah tag
  docker.io/yhgu2000/comtainer:x86-64.sysenv
  comtainer:x86-64.sysenv
$ buildah pull oci-archive:./comtainer_x86-64.rebase.tar
$ buildah tag
  docker.io/yhgu2000/comtainer:x86-64.rebase
  comtainer:x86-64.rebase
```

Then run the commands as described in the AD with the `Containerfile` in the `lulesh/build` directory.

Artifact Analysis (incl. Outputs)

After running the commands described in the AD successfully, two directory `./lulesh.dist.oci` and `./lulesh.redirect.oci` will be generated, and the later one works the same as the previous one, with the only difference that it was built with LLVM toolchains instead of GCC.

C.3 Computational Artifact A_3

Artifact Setup (incl. Inputs)

This evaluation is based on the two before and requires all the source files in the given Git repository.

Artifact Execution

`cd` into each case's directory and run the following commands:

```
$ ./build-original.sh
$ ./build-adapted.sh
$ ./build-optimized.sh
$ ./run.sh
```

For PGO, run `./run-pgo.sh` instead of `./run.sh`.

Please refer to the content of all the scripts to confirm the workability of artifacts.

Artifact Analysis (incl. Outputs)

The `./run.sh` script will print timing results. It is expected that the redirected images give better performance results than the original images.