计算机科学
COMPUTER SCIENCE

基于CRIU的高性能计算容器检查点技术研究

陈轶阳, 王小宁, 闫晓婷, 李冠龙, 赵一宁, 卢莎莎, 肖海力

引用本文

陈轶阳, 王小宁, 闫晓婷, 李冠龙, 赵一宁, 卢莎莎, 肖海力. 基于CRIU的高性能计算容器检查点技术研究[J]. 计算机科学, 2024, 51(9): 40-50.

CHEN Yiyang, WANG Xiaoning, YAN Xiaoting, LI Guanlong ZHAO Yining, LU Shasha, XIAO Haili. Study on High Performance Computing Container Checkpoint Technology Based on CRIU [J]. Computer Science, 2024, 51(9): 40-50.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.231000221

基于 CRIU 的高性能计算容器检查点技术研究

陈轶阳[1,2]　王小宁[1]　闫晓婷[1,2]　李冠龙[1,2]　赵一宁[1]　卢莎莎[1]　肖海力[1]
1 中国科学院计算机网络信息中心　北京 100190
2 中国科学院大学计算机科学与技术学院　北京 100049
(chenyiyang@cnic.cn)

摘　要　容错一直是高性能计算领域的热点和难点问题。检查点是解决容错问题的一种常用技术手段,它能够将运行进程的状态转储成文件并恢复。容器具有较强的资源隔离能力,可以为检查点技术提供更理想的运行环境与载体,避免迁移后任务在节点变更的情况下由于环境与资源变化而出现异常。因此,容器和检查点相结合能够更好地支撑任务迁移的研究与实现。文中围绕基于 CRIU(Checkpoint/Restore In Userspace)的 Singularity 容器检查点方案的设计和优化展开,根据检查点技术在高性能计算容器应用中的特点,在 CRIU 安全使用、迁移性能优化、保持网络状态方面给出了有效的解决方案,基于这些方案拓展了 Singularity 容器检查点功能,并且实现了原型工具 Migrator 来验证容器迁移性能。期望本工作能为后续实现高性能计算任务迁移提供有效的支撑。

关键词:容器;检查点;高性能计算;热迁移;容错
中图分类号　TP311.1

Study on High Performance Computing Container Checkpoint Technology Based on CRIU

CHEN Yiyang[1,2],WANG Xiaoning[1],YAN Xiaoting[1,2],LI Guanlong[1,2] ZHAO Yining[1],LU Shasha[1] and XIAO Haili[1]
1 Computer Network Information Center,Chinese Academy of Sciences,Beijing 100190,China
2 School of Computer Science and Technology,University of Chinese Academy of Sciences,Beijing 100049,China

Abstract　Fault tolerance has always been a hot and difficult problem in the field of high performance computing. Checkpointing is a common technical means to solve the fault tolerance problem,which can dump the state of running processes into files and recover them. Containers have strong resource isolation capability,which can provide a more ideal running environment and carrier for checkpointing technology and avoid the abnormality caused by the change of environment and resources in the case of node change after migration. Therefore,the combination of container and checkpointing can better support the research and implementation of task migration. This paper focuses on the design and optimization of Singularity checkpointing scheme based on CRIU (Checkpoint/Restore In Userspace). Based on the characteristics of checkpointing technology in HPC container applications,effective solutions are given in terms of safe use of CRIU,migration performance optimization,and maintaining network status. The paper extends the checkpointing function to Singularity and implements the prototype tool Migrator to verify the container migration performance. It can provide support for the subsequent implementation of HPC task migration.

Keywords　Container,Checkpoint,High performance computing,Live migration,Fault tolerance

容错一直以来都是大规模高性能计算系统设计与服务面临的主要挑战之一[1]。在单个节点故障率一定的情况下,系统所含节点数量越多,系统平均故障间隔时间就越短。随着E级计算的到来,计算节点的增加使得并行程序出错的概率大幅上升。因此,大规模高性能计算系统除了需具备在硬件设计层面的容错能力,在系统软件方面也需要提供一定的容错支撑,才能高效正确地为用户提供服务。

容错能力指在高性能计算系统发生故障时仍能保证系统完成预定功能的能力。容错涉及多个层次,包括硬件层面和软件层面。现阶段,高性能计算系统中软件层面最常用的容错技术是检查点/重启技术(Checkpoint/Restart,CR)。检查点技术是一种广泛应用于大规模系统的回滚恢复容错技术,这种技术在程序执行期间将计算状态保存到稳定存储介质上[2]。一旦发生故障,当应用程序重新启动时,它不需要从头开始,而是从稳定存储中读取最新状态("检查点")并继续执行。

近年来,容器技术飞速发展,其不仅被大量应用于云服务平台,在高性能计算平台中也得到了普遍的认可和较为广泛的应用。容器的资源隔离能力可以为任务保障执行环境的

到稿日期:2023-10-31　返修日期 2024-03-28
基金项目:国家重点研发计划青年项目(2021YFB0300800)
This work was supported by the Young Scientist Project of National Key R&D Program of China(2021YFB0300800).
通信作者:王小宁(wxn@sccas.cn)

一致性，因此容器和检查点技术的结合可以更好地支撑任务迁移的研究与实现。

本文围绕基于 CRIU（Checkpoint/Restore In Userspace）的 Singularity 容器检查点方案的设计和优化展开，为 Singularity 增加了 CRIU 的拓展实现。本文的主要贡献点如下：1）根据检查点技术在高性能计算容器应用的特点，在 CRIU 安全使用、迁移性能优化、保持网络状态方面给出了有效的解决方案；2）基于这些方案为 Singularity 容器拓展了检查点功能，并且实现了原型迁移工具 Migrator 来验证容器迁移性能，为后续实现大规模高性能计算的局部任务迁移提供了支撑。

本文第 1 章介绍了高性能计算容器技术；第 2 章介绍了检查点技术及与容器的相关集成现状；第 3 章介绍了高性能计算容器集成检查点技术的关键问题和解决方案，包括 CRIU 安全使用、保持网络状态一致和迁移性能优化；第 4 章描述了 Singularity 检查点功能拓展和迁移原型工具 Migrator 的设计与实现；第 5 章对整体系统进行了测试；最后总结全文并提出了未来的工作方向。

1 高性能计算容器技术

容器本质上是一种虚拟化技术。在计算机领域中，虚拟化是一种将计算资源（CPU、内存、磁盘、网络等）抽象、转化、分割，以呈现一个或多个计算资源的技术。主流的虚拟化技术分为基于虚拟机的硬件虚拟化[3]和基于容器的操作系统虚拟化[4]。

从目前来看，操作系统虚拟化的容器技术已成为一种流行的开发、测试、部署应用的方式，它们允许应用程序拥有自己完整的软件堆栈，甚至是操作系统发行版。许多研究表明，如果操作正确，容器几乎不会引入性能损失[5-11]。这更加推进了容器在高性能计算环境的普及。

随着 Docker[12]在 2013 年的推出，操作系统虚拟化技术在工业界和学术界受到了广泛关注。从私有数据中心到公有云，容器因其轻量、可迁移等特性彻底改变了很多企业服务开发和部署的方式[13]。2008－2015 年，Linux 内核基本提供了内置的资源隔离和资源控制的机制，通用的容器软件开始出现。2008 年发布的 Linux containers（LXC）是第一个依赖 Linux 原生内核的完整容器管理实现，其底层使用了内核提供的 namespace[14]和 cgroup[15]机制。Namespace 对资源进行隔离，cgroup 对资源进行控制。2013 年，Docker 作为云计算公司 dotCloud 的开源项目被公开。Docker 基于 LXC 做资源隔离和限制，并且加入写时复制的文件系统作为镜像。这些工具的结合带来了封装应用的全新方式，DevOps[16]、系统管理员、开发人员都开始使用容器做环境的建设、封装和部署。自 Docker 发布后，围绕使用容器作为软件交付的标准单位的想法出现了一个大型社区。随着商业公司开始使用容器来包装和部署其软件，Docker 逐渐无法满足工程团队可能拥有的所有技术和业务需求。为此，社区开始开发具有不同实现和能力的新容器运行时。为了确保所有容器运行时相互兼容，2015 年，在 Docker 公司的牵头下，Linux 基金会启动了开放容器计划（Open Container Initiative，OCI）。OCI 旨在围绕容器成立开放的工业标准，解决容器的构建、分发和运行问题[17]，并逐步建立了标准化的规范。

很多超算中心受到启发，希望应用容器技术来解决高性能计算程序可移植性的问题。国外的超算中心开发了 Singularity[18]，Shifter[19]，Charliecloud[20]，Sarus[21]等适应超算环境的容器软件，允许用户自定义软件栈，将程序的依赖封装成单独的镜像。美国国家能源研究科学计算中心（NERSC）支持在超算上使用 Shifter。根据其 2014 年[22]和 2018 年[23]的系统运行统计报告，容器的使用量从 2014 年的 1%上升到了 2018 年的 8%。美国国家航空航天局（NASA）的超级计算中心在机器上加入了 Singularity 模块[24]。上海交通大学的校级计算公共服务平台也部署了 Singularity[25]。亚马逊公司的云服务 AWS 研发了超算云，利用脚本 AWS ParallelCluster[26]快速创建高性能计算集群，并提供了不同的容器运行时[27]，支持 runc，Singularity 和 Sarus。阿里云的弹性高性能计算 E-HPC[28]平台也支持 Singularity 来完成高性能计算作业[29]。

在众多的高性能计算容器实现中，Singularity 是目前最流行的面向高性能计算系统的容器软件。2015 年，美国劳伦斯伯克利国家实验室（LBNL）在 Gregory Kurtzer 的领导下开发了 Singularity 的初始版本。Singularity 很快引起了全世界计算密集型科学机构的关注。截至 2018 年，根据社区公开统计的调研数据，Singularity 用户群估计超过 25000 人，其中用户不乏来自于俄亥俄州立大学和密歇根州立大学等学术机构以及德州高级计算中心、圣地亚哥超级计算机中心和橡树岭国家实验室等顶级高性能计算中心[30]。Singularity 放弃了许多 Docker 支持的容器特性，专注于实现软件的可移植性，用户可以自己构建镜像，并允许交互式修改。之后将镜像传输到运行环境上，用户能够以无特权的方式运行镜像中的程序。

2 检查点技术及其与容器集成

检查点是一种为计算系统提供容错的技术，它能够保存应用程序状态的快照，以便应用程序能够在出现故障时从该点重新启动。根据实现的层次，检查点技术可以分为系统级和应用级两种。系统级检查点不需要对应用的代码做更改，就能够实现保存完整的程序状态，并在故障后从保存的检查点重新启动程序。常用的系统级检查点技术有 DMTCP[31]和 CRIU[32]。

DMTCP（Distributed MultiThreaded CheckPointing）是由美国东北大学 Coopman 教授主导的一个活跃的项目，其通过动态库代理的形式实现对进程状态的获取。CRIU（Checkpoint/Restore In Userspace）是一个面向 Linux 系统的检查点工具，它由 OpenVZ 开发团队于 2011 年推出，其显著特点是通过使用现有的系统调用来实现，并且主要工作在用户空间中。

CRIU 支持对进程 namespace 和 cgroup 信息的保存和恢复，因此能对容器提供很好的支持。Docker1.13 版本开始支持集成 CRIU 来提供检查点功能。Libcontainer 是 Docker 的原生执行驱动程序，它提供了一组 API 来创建和管理容器。为了支持 CRIU 检查点，Docker 向 libcontainer 引入两个方法，checkpoint()和 restore()，并且向 Docker 本身添加检查

点和恢复子命令。此外，LXC，Podman 等云计算容器也都采用了修改容器引擎源码的方式集成 CRIU 来实现 C/R 功能。

为了支持检查点的功能，Singualrity 选择 DMTCP 项目作为集成的基础。这是一个实验性质的特性，并不推荐在生产环境下使用。Singularity 在容器创建时将 DMTCP 相关的二进制文件从主机注入容器中，并使用它来包装容器进程的启动。启动进程时，DMTCP 将监视衍生的应用程序进程及其子进程，以便在触发时完全检查应用程序的状态。

尽管 Singualrity 实验性地支持了 DMTCP，借助 DMTCP 协调检查点协议，可实现并行 MPI 程序在故障发生时转储并在正常的计算节点环境恢复；但随着 E 级计算的到来，高性能计算程序的并行规模也随之剧增，全部计算任务进程需要转储和恢复的存储代价和时间代价都将是不可接受的。因此，本文尝试通过 Singualrity 集成 CRIU 的方式来支持大规模并行计算中局部任务的转储与恢复；基于 CRIU 实现不协调检查点机制，通过让其他进程等待，单独恢复某个进程，来实现整个程序的容错。

另外，DMTCP 需要给运行库提供代理并储存数据才能够恢复，但后者并没有为所有的内核 API 提供代理（例如，inotify()是已知不支持的），并且这种代理可能会产生一些问题，比如 DMTCP 通过拦截 getpid()库调用并向应用程序提供假的 PID 值来欺骗一个进程，从而实现进程号的虚拟化。这种行为是非常危险的，因为如果应用程序试图通过其 PID 访问 procfs 文件系统，它可能会看到错误的文件。同样地，由于采用动态库代理的方式，它无法用于静态链接的应用，且 DMTCP 由于请求的代理而可能出现的潜在的性能问题。

基于以上考虑，本文选择检查点工具时更加倾向于 CRIU。

3 高性能计算容器检查点技术

3.1 问题分析

本文工作的目标是实现高性能计算容器集成检查点技术，以支持未来大规模并行计算局部任务因运行故障可迁移的应用场景，避免全部任务进程转储和恢复引入的不必要的存储和时间开销。

由于高性能计算系统的特殊软件架构和使用场景，本文需要对检查点功能集成方案的设计作出特殊要求，以确保其适用性和有效性。

Singularity 为了满足高性能计算系统多租户安全使用的需求，抛弃了 Docker 采用的 C/S 架构，并没有一个 root 权限的守护进程来完成一系列特权操作。可以看到，高性能计算环境多租户的场景给 Singularity 设计和实现带来了许多影响。而 CRIU 恰好是一个需要特权才能使用的工具，高性能计算环境中普通用户不具有 root 权限。后续基于 CRIU 拓展检查点功能时，安全性需要得到保障。

在高性能计算集群中，许多应用都需要通过网络进行协同工作，因此在进行容器迁移时，网络的一致性显得尤为重要，需要采取措施来确保迁移后容器的网络设置与之前相同，保持网络连接不中断，从而保证应用的正常运行。

当进行容器迁移时，需要将所有的工作负载从一个节点迁移到另一个节点，这一过程需要大量的时间和资源。随着集群规模的扩大，同时进行多个容器迁移的情况也会变得更加普遍，这将进一步增加集群的负载，可能导致资源的争用和拥堵，进而影响容器迁移的效率和成功率。因此，为了实现容器迁移后应用的快速恢复，整体迁移的时间消耗应尽可能小。

基于上述分析，并综合考虑高性能计算集群环境的使用特点和 Singularity 的实现方式，本文将从 CRIU 的安全使用、一致的网络状态、迁移性能优化这 3 个方面开展 CRIU 和 Singularity 集成的研究。

3.2 CRIU 安全使用

高性能计算系统通常面向多个用户，每个用户可能需要运行自己的程序或任务。对于一些需要进行特权操作的程序，普通用户可能没有足够的权限来执行这些操作。CRIU 运行时需要进行特权操作，Singularity 必须在保证系统和应用安全的情况下，实现 CRIU 的正常运行。

为了允许普通用户执行特权操作，一种常见的方法是给程序设置 setuid 位。设置 setuid 位可以将程序的权限设置为其拥有者（通常是 root 用户）的权限，从而允许普通用户以 root 权限运行该程序。这种方法需要谨慎使用，因为设置 setuid 位可能会带来安全风险，即被攻击者利用来获取系统权限。因此，在设置 setuid 位之前，必须对程序进行充分的安全性评估和测试，并采取适当的安全措施来最大程度地降低安全风险。Singularity 就是通过给它的运行时程序设置 setuid 位来解决普通用户需要特权操作的问题。CRIU 程序是一个非常特殊的工具，它可以通过命令行提供的进程号对任意进程树进行检查点生成操作；并且 CRIU 在生成检查点后会默认杀死进程，对于用户来说，这意味着使用 CRIU 需要非常谨慎。

Singularity 的运行时组件 starter-suid 是设置了 setuid 位的，如图 1 所示，它是 Singularity 启动容器的运行时。它经过小心谨慎地设计，会在需要特权操作的阶段，比如创建命名空间或者挂载 rootfs 等，短暂地切换进程的 euid 为 root 用户，从而拥有权限完成容器启动。在 Singularity 调用 CRIU 时，会通过运行时 starter-suid 在容器内启动进程，可以针对 CRIU 的调用做特殊处理，保留特权。将 CRIU 的工作范围限制在容器内，还可以避免 CRIU 对容器外部的进程造成影响。这是因为容器创建了 pid 命名空间，进程号被隔离在容器内，容器外部的进程在容器内甚至不可见。

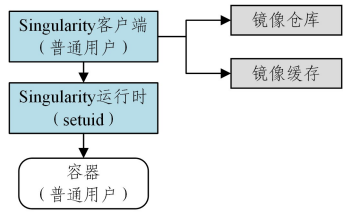

图 1 Singularity 软件架构

Fig. 1 Singularity architecture

针对 starter-suid 对 CRIU 的调用，如果只是简单地保留 root 用户身份，这种做法违反了最小特权原则。Linux 的 Capabilities 机制就是一种在 Linux 系统上实现细粒度权限控制的机制，旨在降低特权过度的程序带来的风险。因此可以给 CRIU 进程保留必须的 capability。整体的方案思路如图 2

所示。Singularity 通过特定的命令由运行时 starter-suid 调用 CRIU，调用时 starter-suid 会抛弃自己的 root 身份，并设置必须的 capability，从而 CRIU 进程能够以普通用户的凭证运行。

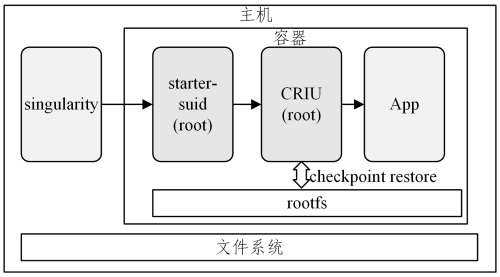

图 2 CRIU 安全使用

Fig. 2 CRIU usage safety

3.3 一致的网络状态

在进行容器迁移时，除了要考虑进程的状态和数据的迁移之外，还需要解决迁移前后网络一致性的问题。Singularity 默认启用的是主机模式。主机模式意味着容器的网络协议栈没有与主机隔离，容器与主机共享同一个网络命名空间。因此，在容器迁移后，工作进程的网络环境可能会发生变化，包括 IP 地址、MAC 地址等网络标识符。为了避免变化后的网络环境导致建立好的网络连接中断，可以利用基于容器的网络虚拟化功能来实现网络保持一致。

容器可以利用网络命名空间提供独立的网络环境，使得容器内的应用程序可以独立于主机网络环境运行。如图 3 所示，Linux 内核提供 macvlan 机制来实现容器内的独立网络设备配置，包含 IP 地址、路由、网关等。macvlan 是 Linux 内核网络的模块，它基于 Linux 内核的虚拟网络设备机制实现[33]。在 Linux 内核中，每个网络接口都有一个唯一的 MAC 地址，这个 MAC 地址在网络中用于标识该接口。macvlan 模块可以从物理网络接口创建一个虚拟网络接口，虚拟网络接口被称为子接口，物理网络接口被称为父接口，并为子接口生成一个新的 MAC 地址。该地址基于父接口的 MAC 地址生成，并且是唯一的。这个新的虚拟网络接口就是一个新的网络设备，它与主机上的其他网络接口相互独立，可以单独配置 IP 地址等网络配置。

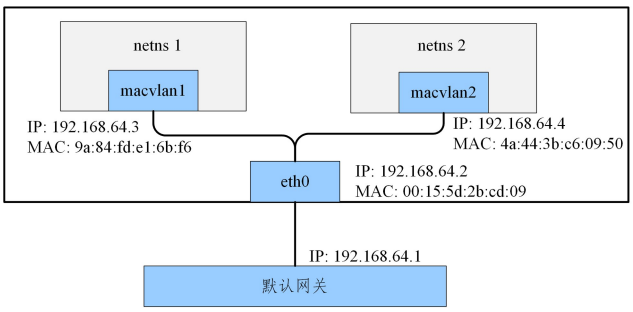

图 3 macvlan 虚拟设备

Fig. 3 Macvlan virtual device

总体来说，整个方案可以在容器启动时创建网络命名空间，并在这个网络命名空间内创建 macvlan 虚拟设备，并且配置 IP 地址、默认网关等信息。迁移后，容器重新建立新的网络命名空间和 macvlan 虚拟设备，并且配置相同的网络信息，从而恢复网络。

基于 macvlan 的容器网络方案可以提供迁移前后的网络状态一致性，结合 CRIU 保持网络连接的特性，可以实现分布式计算任务中对单任务的迁移。目前，CRIU 已经加入了对 TCP 连接恢复的支持。CRIU 团队给 Linux 内核提交了一些 TCP 修复连接的 patch，并在 3.5 版本加入内核主线。他们给 socket 提供了一个特殊的 TCP_REPAIR 选项[34]，用此选项时，socket 将切换到一种特殊模式。在这种模式下，其 socket 执行的任何操作都不会执行实际的网络处理，只改变了协议栈里的网络状态。未来，CRIU 会增加对 IB 网络的支持，从而更好地实现对高性能计算任务迁移的支撑。

3.4 迁移性能优化

在实际的容错场景中，一旦发生故障，作业需要能够快速地从故障节点迁移到另一个正常运行的节点，以确保计算任务的连续性和数据的一致性。在容器迁移的过程中，需要保证所有的数据都能够完整地转移到新节点上，避免数据的丢失和损坏，这些数据主要是指 CRIU 生成的检查点文件。容器的热迁移需要保证系统的性能。在容器迁移的过程中，需要尽量减少整体迁移对业务的影响。因此，在容器热迁移的过程中，需要对系统进行优化和调整，以确保系统的高效性和性能稳定性。

在高性能计算环境中，使用 Singularity 进行迁移的流程如图 4 所示，可以分为以下两个步骤：

1）在源节点上，需要在已运行的容器内启动 CRIU 创建检查点。这个过程分为 3 个阶段。

阶段 1 Singularity 客户端完成解析和验证配置。

阶段 2 Singularity 运行时配置容器环境，并启动 CRIU。

阶段 3 CRIU 完成创建检查点。这里包括一个隐含的检查点同步阶段，检查点文件会被存储到共享文件系统上，目标节点就可以直接在文件系统内访问检查点文件和容器的镜像文件。

2）检查点创建完毕后，可以在目标节点上创建容器，并通过 CRIU 来完成检查点的恢复。这个过程也分为 3 个阶段。

阶段 1 Singularity 客户端完成解析和验证配置。

阶段 2 Singularity 运行时初始化容器环境并创建容器。

阶段 3 启动 CRIU 来恢复工作进程。

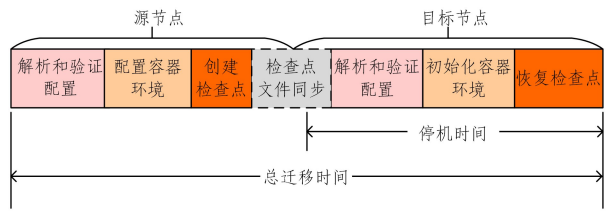

图 4 Singularity 检查点流程

Fig. 4 Singularity checkpoint steps

在实际使用过程中，容器的热迁移功能需要考虑两个重要的性能指标：停机时间和总迁移时间。二者都是整个迁移过程的重要性能指标。停机时间指动态迁移过程中应用程序暂停运行的时间。对于实时性要求较高的服务类应用而言，停机时间是一个重要的性能指标。而在高性能计算系统上，

运行的大多是计算任务，不需要对外提供服务，因此可以允许较长的停机时间。总迁移时间指从迁移指令下达到迁移完成所需的时间。在高性能计算系统中，总迁移时间是一个更重要的性能指标。在容错场景下，需要在机器故障预测到机器故障发生之间的窗口期内完成迁移，这就需要尽可能缩短总迁移时间，以防止突发状况导致迁移失败。通过优化总迁移时间，可以提高容器迁移的可靠性和稳定性。

CRIU 在生成检查点和恢复时，需要进行大量的文件系统 IO 操作，包括对检查点文件的创建、读取和写入等操作。这些操作需要不断地访问和操作并行文件系统，会受到网络、磁盘 IO 速度和文件系统性能等各方面因素的影响。对于内存占用较大的应用程序而言，CRIU 在生成检查点和恢复时所需的时间会更长。因此，可以针对检查点文件的读写和同步耗时，来进行总体迁移时间的优化。

在 CRIU 生成的检查点文件中，pages. img 占比最大，它包含了进程的内存地址空间内容。这些镜像文件在迁移过程中需要创建和加载内存检查点文件，因此会产生大量的共享存储的 IO 操作。如何优化这些 IO 操作成为容器迁移过程中需要解决的一个重要问题。针对这个问题，CRIU 开发了 page server 的机制，以解决传输 pages. img 所产生的 IO 开销。如图 5 所示，page server 有一个运行在目的节点的后台进程，它接收源节点传输而来的内存数据文件，避免了源节点在本地文件系统进行 IO 的开销[35]。通过 page server 的机制，CRIU 能够大大减少容器迁移过程中镜像文件传输的时间和 IO 开销，提高容器迁移的效率。然而，在共享存储系统中，目的节点仍然需要进行文件 IO 操作，因此容器迁移过程中可能会出现一定的性能开销。为了避免这种情况，需要在目标节点上设置一个高速的本地存储，来实现对 pages. img 文件的接收。其中，基于内存的文件系统 tmpfs[36] 是较为合适的一个选择。tmpfs 是一种内存文件系统，它将文件存储在 RAM 中，因此访问速度非常快。在容器迁移中，使用 tmpfs 来存储内存文件能够显著提高迁移的效率，减少容器停机时间和总迁移时间。

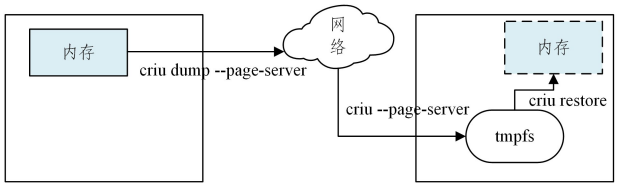

图 5 CRIU page server 工作流程

Fig. 5 CRIU page server workflow

针对剩余的检查点文件，可以选择将其压缩打包成一个大文件，然后传输到目标节点的 tmpfs 上，恢复容器时再将其解压缩。最终，结合 page server 和内存文件系统后的优化方案如下：

1）目的节点启动新的容器，容器运行 CRIU 的 page server，等待内存文件传输。

2）在源节点上，在容器内运行 CRIU 的生成检查点，CRIU 会将内存检查点文件传输到目标节点，其余检查点文件保存在本地的 tmpfs 上。

3）源节点会将剩余的检查点文件打包压缩并传输到目标节点。

4）目标节点此时已经收集完成所有的检查点文件，在容器内启动 CRIU 的恢复流程，重启先前的工作进程。

4 集成系统实现

4.1 Singularity 检查点功能拓展

本文对 Singularity 容器软件的源代码进行修改，添加了一些功能选项。这些修改充分考虑到高性能计算集群的多用户环境，可以让普通用户安全地执行容器的检查点生成，能够实现容器迁移过程中的状态保留、网络连接保持等功能，并且允许用户在另一个节点上重启恢复容器。

Singularity 于 2021 年加入了 Linux 基金会，并且改名为 Apptainer，本文对 Singularity 的修改基于 Apptainer 1. 0 版本。本文拓展和修改了 Singularity 检查点管理、容器启动、容器重启等功能命令。Singularity 软件由两个程序组成：命令行客户端和容器运行时 starter-suid。命令行客户端是 Singularity 命令，作为用户接口，负责解析用户输入的指令，完成相应的功能，主要包括镜像管理和容器相关操作。容器运行时 starter-suid 并不是独立的工具，只会被命令行 singulairty 所调用，负责实际和容器生命周期相关的功能，包括启动容器和在容器内调用进程。从名字可以看出，starter-suid 被设置了 setuid 位，在运行过程中允许执行必要的特权操作。本文针对 Singularity 的功能扩展的设计结构图如图 6 所示，其中蓝色模块为集成检查点功能新增或修改的模块。

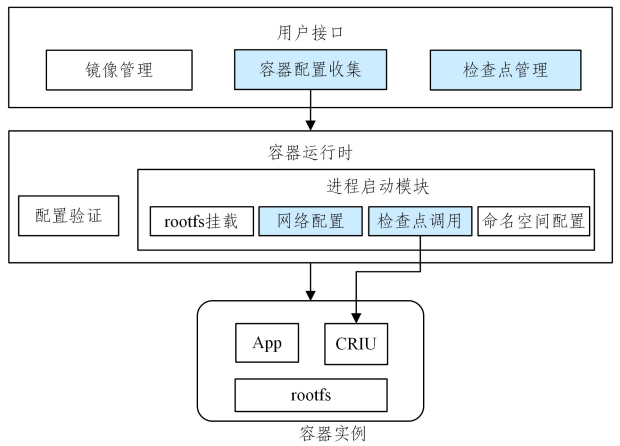

图 6 Singularity 检查点功能扩展的设计结构图

Fig. 6 Design architecture of CR-extended Singularity

1）容器配置收集用于整理用户提供的参数，对于重启的容器会从检查点目录去查找导出的配置并加载；

2）检查点管理用于维护保存检查点的目录，默认创建在分布式文件系统上，通过设置可以将 CRIU 生成的检查点存储到内存文件系统；

3）网络配置用于在启动容器时设置网络命名空间，创建 macvlan 虚拟设备以及配置网络信息；

4）检查点调用用于在检查点生成和恢复时在容器内启动 CRIU，需要提前绑定挂载 CRIU 相关文件到 rootfs，并且控制 CRIU 进程的身份和权限。

接下来将从用户操作命令的角度详细介绍高性能计算容器集成检查点功能的工作流程。

1）检查点目录管理

在容器运行时，为了确保容器内生成的检查点数据被存储到主机上并且不会随着容器生命周期的结束而消失，这个目录需要被绑定挂载到容器的 rootfs 上。在恢复容器时，会依据该目录中的配置信息来恢复容器，并将该目录绑定挂载，以便从 CRIU 检查点中恢复进程。这样可以确保在容器恢复期间，所有必要的信息都能被正确地加载和使用。因此，Singularity 需要提供检查点目录管理的功能。拓展的命令如表 1 所列，包含创建和删除检查点两个命令。

表 1　管理检查点目录相关命令

Table 1　Manage related commands of checkpoint directory

命令	功能
apptainer checkpoint create [--mem-dir] 〈name〉	创建检查点目录
apptainer checkpoint delete〈name〉	删除检查点目录

Singularity checkpoint create 命令在 Singularity 架构中的工作原理如图 7 所示。用户首先使用用户接口程序 Singularity 传递创建检查点目录的命令，并提供一个单独的检查点名称。这个命令会在用户的 home 目录下的默认路径～/.apptainer/checkpoint/criu/〈name〉创建一个目录。这个目录是一个广义的检查点目录，容器生成检查点时，会将容器配置导出到这个目录下，CRIU 也会将检查点读写在这个目录，容器启动时也会将工作进程的进程号写入这个目录，以供 CRIU 读取。如果指定了--mem-dir（memory directory 的缩写）参数，则选择在许多 Linux 发行版中默认提供的 tmpfs 挂载点/dev/shm 下创建目录，以加快检查点读写的速度。

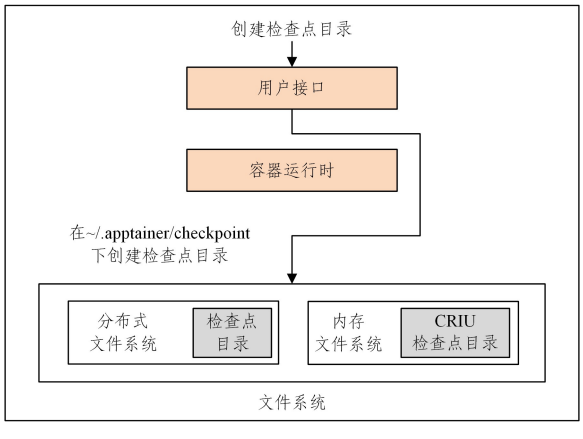

图 7　创建检查点目录

Fig. 7　Create checkpoint directory

Singularity checkpoint delete 命令比较简单，会根据提供的检查点名字来删除 create 子命令创建的目录，包括容器检查点目录和 CRIU 检查点目录，其中存储的数据也都会被删除。

2）容器启动

Singularity 还提供了一种启动实例（Instance）的方式。实例本质上是一个在后台运行的进程，是在后台运行的容器镜像的持久化和隔离版本。持久化意味着 Singularity 会保留这种启动的容器的相关信息，并对其进行管理，例如允许打印信息容器基本信息或者强制杀死容器进程。因此，Singularity 的实例模式启动是添加检查点功能的最佳模式。本文对容器实例启动命令的修改如表 2 所列。为了不影响原始的容器实例启动流程，额外新增了 criu-launch 参数，表示启动一个后续允许进行检查点操作的容器实例。其参数值为 checkpointname，是预先通过 checkpoint create 命令创建的检查点名字。

表 2　启动容器命令

Table 2　Launch container command

命令	功能
apptainer instance start [options] -criu-launch 〈checkpoint name〉〈image path〉 〈instance name〉[args…]	启动需要检查点功能的容器

本文对 Singularity 原有的实例启动流程进行了修改。具体修改的内容如图 8 所示。

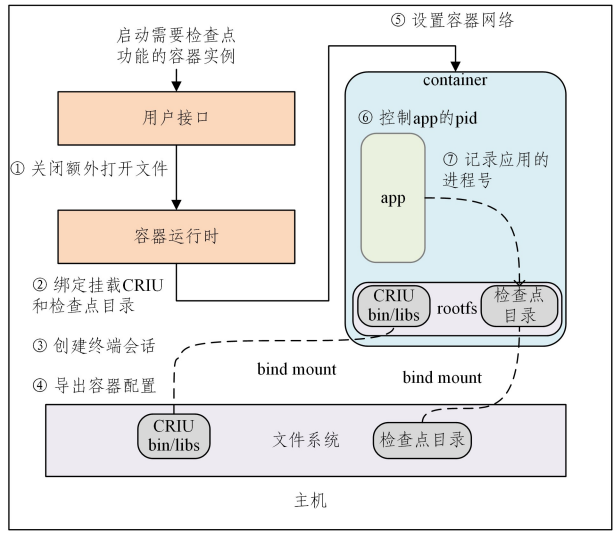

图 8　启动需要检查点功能的容器流程

Fig. 8　Start a container process requiring checkpoint capabilities

（1）关闭从父进程（通常是 shell 进程）继承来的打开文件。CRIU 执行 dump 操作时会获取进程打开文件表并访问这些打开文件。然而，由于这些文件在容器的 rootfs 内无法访问，这将导致 dump 操作失败。因此，为了确保操作成功，需要将这些打开文件关闭。

（2）在容器运行时 starter-suid 进行 rootfs 初始化时，需要绑定挂载 CRIU 相关的二进制文件，包括可执行文件和依赖的动态链接库，以及检查点目录。

（3）给容器内进程创建终端会话。在原始流程中，运行时 starter-suid 会让容器外的监控进程创建终端会话，它是容器内进程的 session leader。在执行 dump 操作时，由于 pid 命名空间的隔离，CRIU 无法获取 session leader 进程的信息导致操作失败，因此需要给容器内的 init 进程创建一个终端会话。

（4）导出容器配置。容器的配置由两部分组成，分别是启动容器时指定的参数选项和全局配置。这里只需导出启动容器时指定的参数即可，导出到容器检查点目录下，可供后续容器恢复时读取导入。

（5）设置容器网络。在容器技术中，为了让容器能够具有独立的 IP 地址并且与外部通信，需要设置容器网络。为此，

需要在容器内部创建网络命名空间，然后在其中创建 macvlan 设备，并为该设备配置 IP 地址、子网掩码、默认网关等网络信息。基于 macvlanCNI 插件实现了这些网络配置。

(6)控制容器内工作进程的 pid 号为一个较大值。这通过设置/proc/sys/kernel/ns_last_pid 文件实现。ns_last_pid 是 Linux 3.3 开始支持的特殊文件，包含内核分配的最后一个 pid。当内核需要分配一个新的 pid 时，默认分配 last_pid +1。容器内的 pid 从 1 开始分配，容器内有一些进程来进行初始化工作，会消耗固定数量的 pid。因此，当实际启动工作进程时，它的 pid 会是一个固定的值。为了避免容器重启时 CRIU 进程号和当前工作进程号冲突，需要控制当前工作进程号为较大值。

(7)记录应用的进程号。在后续进行 CRIUdump 操作时，将进程号作为参数传递给 CRIU。

3)检查点生成和恢复

接下来是在启动的容器内执行检查点的命令，如表 3 所列。对于执行检查点生成的命令，--criu 参数表示使用 CRIU 来生成检查点。该命令还使用了--page-server 选项，用于控制 CRIU dump 调用是否带有--page-server 参数。执行检查点恢复操作的命令，最终会在容器内调用 criu restore 来执行恢复动作。这个命令主要在无盘迁移流程中使用。

表 3 检查点生成和恢复相关命令

Table 3 Related commands for checkpoint generation and recovery

命令	功能
apptainer checkpoint instance -criu [--page-server] 〈name〉	执行检查点生成操作
apptainer checkpoint instance --criu -restore 〈name〉	执行检查点恢复操作

总体来说，不同的命令最终都是在容器内调用 CRIU，只是给 CRIU 赋予的参数有所差别。实际调用 CRIU 的流程如图 9 所示。

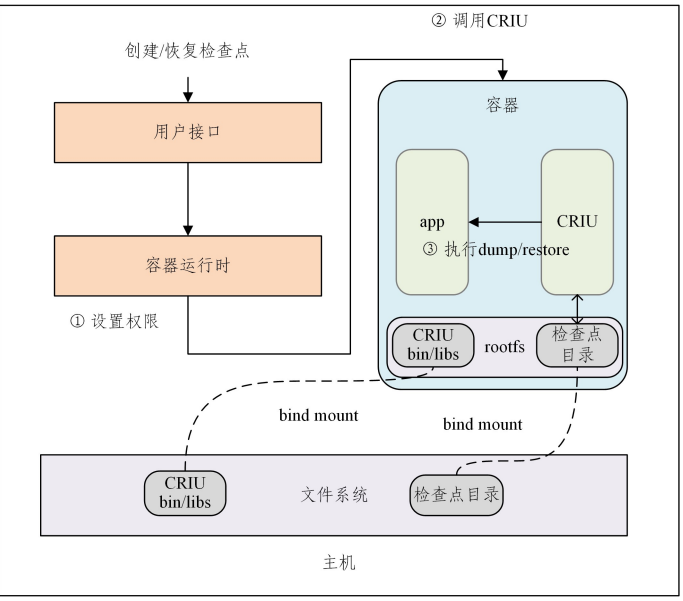

图 9 创建/恢复检查点流程

Fig. 9 Checkpoint creation/restoration process

(1)在容器运行时，为了保证成功调用 CRIU，需要在调用前给 starter-suid 设置适当的 capability，并将进程用户凭证切换成调用命令的普通用户。

(2) 利用 setns 来将 CRIU 加入容器所在的命名空间。

(3)最终，根据 Singularity 不同的命令选项来设置 CRIU 的参数并对其进行调用。

4)容器重启

容器重启的命令如表 4 所列。--criu-restart 用于指定当前开始的容器是容器重启的流程，这个流程区别于普通的容器启动，主要有两个重要环节：容器环境的恢复和 CRIU 的调用。—page-server 参数用于无盘迁移的流程，启动容器后首个进程为 CRIU page-server，以接收源节点发送来的内存数据。默认情况下会直接通过 CRIU restore 在容器内恢复进程。

表 4 容器重启相关命令

Table 4 Container restart-related commands

命令	功能
apptainer instance start [options] -criu-restart 〈checkpoint name〉 [--page-server] 〈image path〉〈instance name〉 [args…]	重启容器

重启容器和普通的启动容器大部分工作类似，都会进行启动容器最核心的工作：进行配置的收集和验证、设置rootfs，创建命名空间、启动容器进程并监控容器。它们的主要差异在于重启容器需要进行额外的操作来导入之前生成的检查点数据，并最终调用 CRIU。重启容器额外的工作流程如图 10 所示。

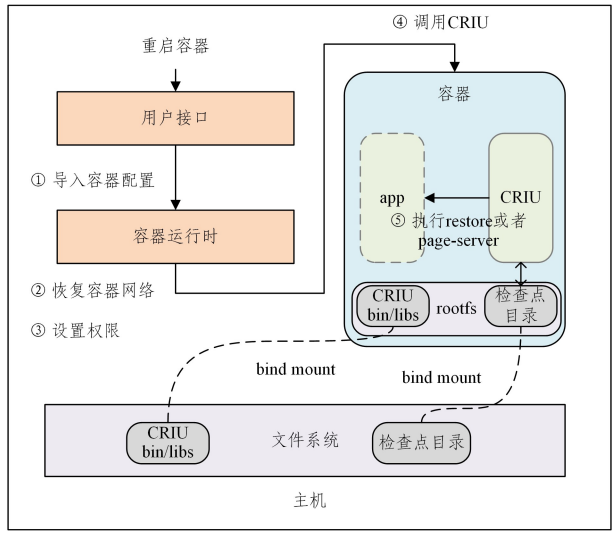

图 10 重启容器流程

Fig. 10 Restart container process

(1)在用户接口中，Singularity 会导入容器启动时导出的配置，保证重启的容器与启动时具有相同的配置。

(2)恢复容器网络。容器网络的恢复实际上是对其进行重建。与启动容器时类似，此过程涉及网络命名空间和 macvlan 虚拟设备的创建和配置，容器运行时会读取检查点目录中的配置文件，配置文件中包括了恢复容器所需的网络信息，如容器 IP、子网掩码、默认网关等。初始化后，容器将拥有与之前相同的 IP 地址、子网掩码和默认网关。

(3)与执行检查点操作时一样，需要在调用 CRIU 前给当前进程设置合适的 capability，彻底舍弃 root 身份，变为普通用户。

(4)在容器内调用 CRIU。CRIU 作为容器内的第一个工作进程启动，并根据是否指定--page-server 参数来调用不同的 CRIU 功能。

4.2 迁移原型工具 Migrator

为了验证对 Singularity 拓展的功能，本文还设计并实现了一个容器迁移原型工具 Migrator。Migrator 是一个 C/S 架构的软件，通过调用 4.1 实现的 Singularity，可实现无盘迁移以及保持网络状态，以验证本文工作。Migrator 由两个组件组成：客户端 Migrator-cli 和服务端 Migratord。其设计如图 11 所示。

用户通过 Migrator-cli 向 Migratord 发送迁移命令，Migratord 调用修改后的 Singularity，将容器的镜像和配置保存在主机的文件系统上，修改后的 Singularity 通过特定的命令由运行时 starter-suid 调用 CRIU，通过限制 CRIU 工具的工作范围和权限，满足非特权用户的正常使用，实现了 CRIU 在超算环境的安全使用，并且最大限度地保障了整体系统的可靠性。

检查点迁移的流程有两种可选模式：默认的流程将检查点文件直接创建在分布式文件系统上，无需额外同步工作；另一种为无盘迁移的流程，该流程将内存相关的检查点文件通过 CRIU 内置的 page-server 直接传送到目标节点的内存文件系统上，将其余检查点文件生成到本地内存文件系统，然后由 Migratord 压缩传输这些剩余文件。目标节点上的 Migratord 会接受来自源节点的请求完成恢复工作。默认迁移流程无需额外的同步检查点工作；而无盘迁移流程下，还需要接收除了内存数据外的检查点文件并存储到本地的内存文件系统。最终，目标节点上的 Migratord 调用 Singularity 重建容器环境，并通过 CRIU 从检查点中恢复工作进程。

保持网络连接的功能是由分布在各个节点的 migratord 来实现的。migratord 为容器迁移前后建立一致的基于 macvlan 的网络环境。

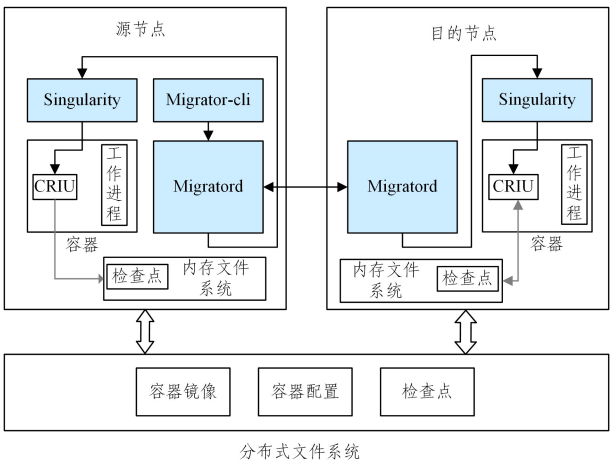

图 11 Migrator 设计架构

Fig. 11 Migrator design architecture

5 实验结果与分析

5.1 实验环境说明

为评估所设计的 Singularity 容器检查点与迁移方案的性能，本节对容器的迁移过程进行了一系列性能测试。测试环境是真实超算系统中的两个隔离的计算节点，并根据需求升级了操作系统内核。节点的具体配置如表 5 所列。

表 5 节点配置

Table 5 Node configuration

参数名	参数值
CPU	Hygon C86 7185 32-core Processor
内存/GB	128
操作系统	Centos 7.9
内核版本	Linux 5.4.224
磁盘带宽/(MB/s)	100
网络带宽 1/(Gbit/s)	1

5.2 性能测试

测试采用的程序是 HPL(High-Performance Linpack)，它通过高斯消去法求解稠密线性代数方程组来评测高性能计算机的实际持续峰值浮点性能。HPL 是全球超级计算机 TOP500 的重要依据。

本次测试选取的 HPL 版本为 2.3。在 HPL 中，可以通过设置一系列参数来定义计算规模，并通过调优得出最优计算性能。本次测试中，我们通过设置 *Ns* 参数来改变矩阵规模，从而改变计算负载，以评估在不同负载下的迁移性能。具体而言，本文根据实际测试需求设置了一系列参数，具体如表 6 所列。

表 6 HPL 参数设置

Table 6 HPL parameter settings

参数名	说明
矩阵数量 *N*	表示求解矩阵的数量，设置为 1
矩阵大小 *Ns*	表示矩阵阶数，本文通过设置 *Ns* 来改变计算规模
块大小数量	表示运行的块大小的数量，这里设置为 1
块大小 *Nbs*	表示矩阵分块的大小，设置为 256
进程网格数 *PxQ*	*P* 和 *Q* 分别为行和列方向的进程数，都设置为 1

1)内存和检查点大小的关系

本文选取了不同负载下的 HPL 容器实例，当其处于内存峰值状态时，进行检查点操作，并对检查点文件的大小以及内存大小进行分析，如图 12 所示。可以看出，生成的检查点文件大小与容器实例所占用的内存大小几乎一致。这一结果表明，CRIU 生成的检查点文件主要是内存数据，针对内存文件的迁移优化是一个值得研究的方向。

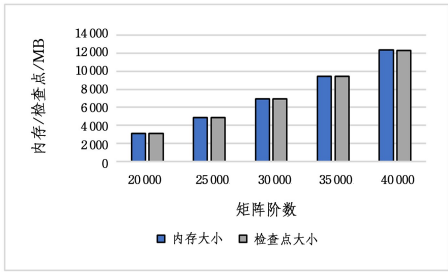

图 12 内存和检查点大小的关系

Fig. 12 Relation between memory and checkpoint size

2)总体迁移时间

接下来测试 HPL 在不同计算负载下,默认迁移和无盘迁移的总体迁移时间。测试结果如图 13 所示,随着计算规模 Ns 的增加,检查点大小逐步增加,总体迁移的时间也逐步提升。此外,实验也证实,与默认的磁盘迁移方案相比,无盘迁移方案的总迁移时间更短,缩减幅度在 8.2%～10.3%之间。性能优势的来源主要有几个:(1)基于内存文件系统。无盘迁移时,检查点的读写都位于内存文件系统,不论是 CRIU 的生成和恢复,还是检查点同步传输,都会节省一部分读写磁盘的开销,因此基于内存的文件操作能够显著减少迁移耗时。(2)Page-server 机制能够避免部分磁盘 I/O。无盘迁移的流程中,内存相关数据会直接发送到目标节点的内存文件系统;而默认的迁移流程中,内存相关检查点会先保存到本地文件系统,再发送到目标节点,最终目标节点从网络接收数据并保存到磁盘。Page-server 也能够通过减少磁盘 IO 来提升一部分性能。

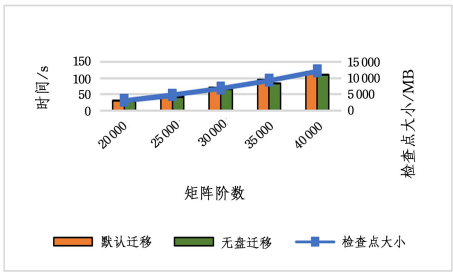

图 13 总体迁移时间

Fig. 13 Total migration time

3)迁移耗时分布

接下来,我们将分别对默认迁移和无盘迁移的时间消耗进行详细分析。这些时间消耗主要包括检查点同步的时间、CRIU 进行检查点生成和恢复的时间,以及 Migrator 内部的一些操作。对于默认迁移流程来说,Migrator 内部的操作包括:(1)Migrator-cli 和 Migratord 之间的 RCP 调用;(2)调用 Singularity 命令。调用 Singularity 命令主要的耗时在于创建容器或者在容器内执行命令时必须执行的一些检查和初始化工作。而对于无盘迁移来说,Migrator 内部操作耗时也包括:(1)Migrator 内部组件之间的 RCP 调用耗时;(2)每次执行 Singularity 命令带来的固定开销;(3)对 CRIU 生成的除内存页面外的其余检查点数据进行压缩、打包、传输和解压缩过程的开销。

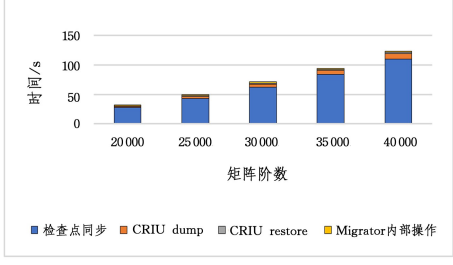

图 14 默认迁移流程耗时

Fig. 14 Time consuming of default migration process

测试结果如图 14 和图 15 所示,随着矩阵规模 Ns 的增大,两种迁移方式中,检查点同步和 CRIU dump、restore 耗时也相应增加,而 Migrator 内部操作所占用的时间都比较固定,没有随着矩阵规模 Ns 的增大而增加,这表明 Migrator 内部操作的时间开销与应用的计算规模基本无关。

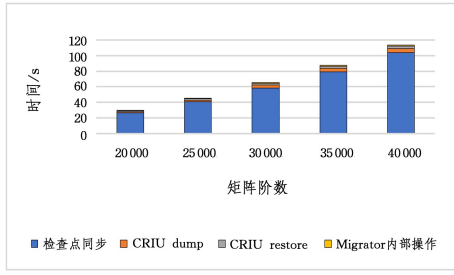

图 15 无盘迁移流程耗时

Fig. 15 Time consuming of diskless migration process

4)迁移对计算任务的影响

接下来测试迁移是否会影响应用的工作效率。从理论上来说,迁移意味着整个计算任务的进程状态的迁移。迁移前后可能会涉及硬件缓存和内核文件缓存失效的问题,但对于跨度较长的高性能计算任务来说,任务的耗时不会有明显差异。本研究对未进行迁移的任务的耗时进行了统计,并且和进行迁移的任务的耗时进行了对比。需要注意的是,迁移任务的耗时已经去除了总体的迁移时间。从统计结果图 16 中可以看到,针对不同计算规模的任务,在默认迁移和无盘迁移方式下,任务自身的耗时都没有显著变化。因此,本研究认为,在高性能计算任务中进行迁移不会对应用的工作效率产生明显的影响。

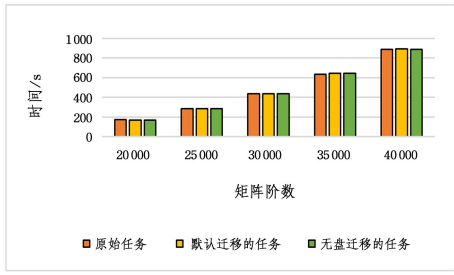

图 16 任务运行时间

Fig. 16 Time consuming of tasks

5.3 实际应用测试

本文采用实际应用对本文工作开展迁移测试。测试选择了以下 3 个应用:

1)GROMACS(GROningen MAchine for Chemical Simulations)[37]。GROMACS 是一款通用软件,用于对具有数百万颗粒子的系统进行基于牛顿运动方程的分子动力学模拟。它主要用于生物化学分子,如蛋白质、脂质等具有多种复杂键合相互作用的核酸分析。

2) LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator)[38]。LAMMPS 是一款经典分子动力学软件,包含的势函数可用于固体材料(金属、半导体)、软物质(生物大分子、聚合物)、粗粒化或介观尺度模型体系。

3)AutoDock Vina[39]。AutoDock Vina 是一款分子建模仿真软件,对蛋白质-配体对接特别有效,是速度最快、使用

最广泛的开源对接软件之一。

针对这 3 个实际应用，分别基于默认迁移和无盘迁移进行了容器迁移的测试，实验数据如图 17 所示。可以看到，无盘迁移相比默认迁移耗时更短。实验还表明本文实现的迁移方案已经具备了一定的实用性，可以针对实际的应用完成迁移工作。

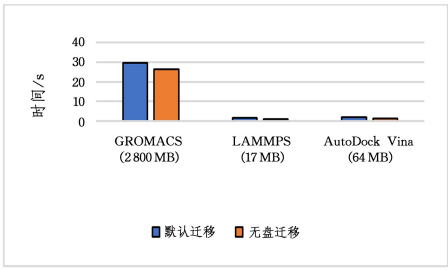

图 17 实际应用迁移时间

Fig. 17 Migration time of practical applications

结束语 本文对 Singularity 进行了二次开发，拓展了它的检查点功能。考虑到高性能计算环境的多用户使用场景以及 Singularity 自身的设计理念，该实现特别重视安全性，通过限制 CRIU 工具的工作范围和权限，满足了非特权用户的正常使用，并且最大限度地保障了整体系统的可靠性。基于 macvlan 虚拟设备，实现了针对迁移场景的容器网络虚拟化方案，使得容器能够在迁移前后保证网络状态，维持网络连接。该方案为在高性能计算并行应用的局部任务迁移过程中保持网络连接提供了可行性的技术方案。本文针对迁移的流程进行了优化，提出了基于 Singularity 的无盘迁移的方案，减少了迁移过程的消耗时间，实现一套 C/S 架构的迁移系统原型来验证迁移方案的可行性，为迁移方案未来集成进作业管理系统打下基础。

本文工作对 Linux 内核版本有一定要求，需要 4.0 以上内核版本才能工作，仍然存在一定的局限性。为了保证兼容性，国内主流超算系统依然是 3.10 版本的内核居多。本文提出的保持网络状态的方案基于 macvlan 虚拟设备，如何在非特权用户使用模式下安全实现网络虚拟化的构建将是后续的工作内容之一。本文实现的 Migrator 作为验证 Singularity 检查点功能的原型工具，还不具备迁移系统所需要的许多功能，比如故障预测、迁移节点调度、作业管理系统集成等。基于本文实现的 Singularity 检查点功能，后续工作将会设计出更完善的迁移系统，并将其与故障预测系统和作业管理系统等基础设施集成，以提高实用性。

参 考 文 献

[1] YANG X, WANG Z, XUE J, et al. The Reliability Wall for Exascale Supercomputing [J]. IEEE Transactions on Computers, 2012, 61(6): 767-779.

[2] BOZYIGIT M, WASIQ M. User-level process checkpoint and restore for migration [J]. ACM SIGOPS Operating Systems Review, 2001, 35(2): 86-96.

[3] PEARCE M, ZEADALLY S, HUNT R. Virtualization: Issues, security threats, and solutions [J]. ACM Computing Surveys, 2013, 45(2): 1-39.

[4] LAADAN O, NIEH J. Operating system virtualization: practice and experience[C]// Proceedings of the 3rd Annual Haifa Experimental Systems Conference. 2010: 1-12.

[5] XAVIER M G, NEVES M V, ROSSI F D, et al. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments [C] // 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. 2013: 233-240.

[6] LIU P N, GUITART J. Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study [J]. Journal of Supercomputing, 2021, 77(6): 6273-6312.

[7] ABRAHAM S, PAUL A K, KHAN R I S, et al. On the Use of Containers in High Performance Computing Environments [C]// IEEE 13th International Conference on Cloud Computing (CLOUD). 2020: 284-293.

[8] JAERYUN L, CHAE Y, TAK B. Comparative Analysis of Container for High Performance Computing [J]. Journal of the Korea Society of Computer and Information, 2020, 25(9): 11-20.

[9] TORREZ A, RANDLES T, PRIEDHORSKY R, et al. HPC container runtimes have minimal or no performance impact [C]// 1st IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). 2019: 37-42.

[10] YONG C H, LEE G W, HUH E N. Proposal of Container-Based HPC Structures and Performance Analysis [J]. Journal of Information Processing Systems, 2018, 14(6): 1398-1404.

[11] ZHANG J, LU X Y, PANDA D K, et al. Is Singularity-based Container Technology Ready for Running MPI Applications on HPC Clouds? [C] // 10th International Conference on Utility and Cloud Computing (UCC)/4th International Conference on Big Data Computing, Applications and Technologies (BDCAT). 2017: 151-160.

[12] MERKEL D. Docker: lightweight Linux containers for consistent development and deployment [J]. Linux Journal, 2014, 2014(239): 76-91.

[13] JARAMILLO D, NGUYEN D V, SMART R. Leveraging microservices architecture by using Docker technology [C] // Southeast Conference 2016. 2016: 1-5.

[14] MICHAEL K. namespaces(7) — Linux manual page [EB/OL]. (2021-08-27) [2023-06-02]. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[15] MICHAEL K. cgroups(7)—Linux manual page [EB/OL]. (2021-08-27) [2023-06-02]. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[16] LEITE L, ROCHA C, KON F, et al. A survey of DevOps concepts and challenges [J]. ACM Computing Surveys (CSUR), 2019, 52(6): 1-35.

[17] DEBAB R, HIDOUCI W K. Containers Runtimes War: A Comparative Study [C] // Proceedings of the Future Technologies Conference. Springer, 2020: 135-161.

[18] KURTZER G M, SOCHAT V, BAUER M W. Singularity: Scientific containers for mobility of compute [J]. PLOS ONE, 2017, 12(5): e0177459.

[19] Shane Canon and Douglas Jacobsen Revision. shifter[EB/OL].

[2023-6-22]. https://shifter.readthedocs.io/en/latest/.
[20] REID P, TIM R. Charliecloud: unprivileged containers for user-defined software stacks in HPC[C]// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery, 2017: 1-10.
[21] BENEDICIC L, CRUZ F A, MADONNA A, et al. Sarus: Highly Scalable Docker Containers for HPC Systems [C]// IEEE International Conference on High Performance Computing, Data, and Analytics. 2019: 46-60.
[22] BRIAN A, WAHID B, TINA B, et al. 2014 NERSC Workload Analysis [EB/OL]. (2014-11-05)[2023-06-02]. https://portal.nersc.gov/project/mpccc/baustin/NERSC_2014_Workload_Analysis_v1.1.pdf.
[23] AUSTIN B. NERSC-10 Workload Analysis (Data from 2018) [EB/OL]. (2020-04-01)[2023-06-02]. https://portal.nersc.gov/project/m888/nersc10/workload/N10_Workload_Analysis.latest.pdf.
[24] MICHELLE M. Enabling User Defined Software Stacks with Singularity [EB/OL]. (2021-11-15)[2023-06-02]. https://www.nas.nasa.gov/hecc/support/kb/enabling-user-defined-software-stacks-with-singularity_637.html.
[25] Pi supercomputing cluster user documentation [EB/OL]. (2020-08-15)[2023-06-02]. https://docs.hpc.sjtu.edu.cn/container/index.html.
[26] FRANCESCO D, ENRICO M U, LUCA C, et al. AWS ParallelCluster [EB/OL]. https://github.com/aws/aws-parallelcluster.
[27] CHRISTIAN K. HPC Container Engine State-of-Art [EB/OL]. (2021-02-06) [2023-06-02]. https://containers-on-pcluster.workshop.aws/.
[28] ALIBABA-CLOUD. Elastic High Performance Computing [EB/OL]. https://www.alibabacloud.com/product/ehpc.
[29] ALIBABA-CLOUD. E-HPC: Use high-performance container applications [EB/OL]. https://help.aliyun.com/document_detail/102579.html?spm=5176.21213303.J_6704733920.37.5dab3eda2NILb8&scm=20140722.S_help%40%40%E6%96%87%E6%A1%A3%40%40102579.S_0%2Bos0.ID_102579-RL_singularity-OR_helpmain-V_2-P0_6.
[30] Voluntary registry of Singularity installations [EB/OL]. https://docs.google.com/spreadsheets/d/1Vc_1prq_1WHGf0LWtpUBY-tfKdLLM_TErjnCe1mY5m0/pub?gid=1407658660&single=true&output=pdf.
[31] ANSEL J, ARYA K, COOPERMAN G. DMTCP: Transparent checkpointing for cluster computations and the desktop[C]// IEEE International Symposium on Parallel & Distributed Processing. IEEE, 2009: 1-12.
[32] PICKARTZ S, EILING N, LANKES S, et al. Migrating LinuX containers using CRIU[C]// International Conference on High Performance Computing. Springer, 2016: 674-684.
[33] ZHAO Y, XIA N, TIAN C, et al. Performance of container networking technologies[C]// Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems. 2017: 1-6.
[34] ZHOU D, TAMIR Y. Fault-tolerant containers using nilicon [C]// 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2020: 1082-1091.
[35] STOYANOV R, KOLLINGBAUM M J. Efficient live migration of linux containers [C]// High Performance Computing: ISC High Performance 2018 International Workshops. Springer, 2018: 184-193.
[36] SNYDER P. tmpfs: A virtual memory file system: [C]// Proceedings of the autumn 1990 EUUG Conference. 1990: 241-248.
[37] GROMACS development team. gromacs [EB/OL]. https://manual.gromacs.org/documentation/2019/index.html.
[38] THOMPSON A P, AKTULGA H M, BERGER R, et al. LAMMPS—a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales[J]. Computer Physics Communications, 2022, 271: 108171.
[39] TROTT O, OLSON A J. AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading[J]. Journal of Computational Chemistry, 2010, 31(2): 455-461.

CHEN Yiyang, born in 1997, postgraduate. His main research interests include high performance computing and cloud service.

WANG Xiaoning, born in 1981, Ph. D, associate professor, Ph. D supervisor, master supervisor. Her main research interests include high performance computing, grid computing and cloud service.

(责任编辑:杨雪敏)