基于云预存储技术的 Docker 在线迁移方法

赵 旭 [1] 李艳梅 [1] 罗 建 [1] 罗金梅 [1]

摘 要 针对基于 Docker 容器的分布式云计算下出现负载不均衡问题, 有必要将较高负载服务器中的 Docker 容器进程迁移到其他相对空闲的服务器上. 而传统的容器迁移算法忽视了容器本身的特征, 从而导致在迁移过程中传输效率低下. 基于此, 利用第三方管理平台和数据预存储阈值机制, 提出一种 Docker 容器动态迁移预存储算法 PF-Docker. 首先将 Docker 容器内部进程运行相关文件和流动数据预存至云端存储器, 然后通过预存储阈值机制减少流动数据的无效传输, 最后在停机传输阶段将流动数据和冗余数据传输给目的服务器. 实验表明, 该方法在 Docker 容器迁移中能有效地降低迁移时间, 减少数据传输量, 提高容器的容错率.

关键词 Docker 容器, 预存储, 动态迁移算法, 云计算, 容错率

引用格式 赵旭, 李艳梅, 罗建, 罗金梅. 基于云预存储技术的 Docker 在线迁移方法. 自动化学报, 2023, **49**(11): 2426−2436

DOI 10.16383/j.aas.c180777

Docker Online Migration Method Based on Cloud Pre-Storage Technology

ZHAO Xu[1] LI Yan-Mei[1] LUO Jian[1] LUO Jin-Mei[1]

Abstract In order to solve the problem of load imbalance in distributed cloud computing based on Docker container, it is necessary to migrate Docker container processes in high load servers to other relatively idle servers. However, the traditional container migration algorithm ignores the characteristics of the container itself, which leads to low transfer efficiency in the migration process. Based on this, this paper proposes a Docker container dynamic migration pre-storage algorithm PF-Docker by using third-party management platform and data pre-storage threshold mechanism. Firstly, relevant files and flowing data of Docker internal process operation are pre-stored to cloud storage, and then invalid transmission of flowing data is reduced through pre-storage threshold mechanism. Finally, flowing data and redundant number are reported to the destination server in the shutdown transmission stage. Experiments show that this method can effectively reduce the migration time, reduce the amount of data transmission and improve the fault tolerance rate of Docker container migration.

Key words Docker container, pre-storage, dynamic migration, cloud computing, fault tolerance

Citation Zhao Xu, Li Yan-Mei, Luo Jian, Luo Jin-Mei. Docker online migration method based on cloud pre-storage technology. *Acta Automatica Sinica*, 2023, **49**(11): 2426−2436

云服务[1] 是一种当下热门的互联网模型, 它将软硬件资源进行整合, 以虚拟环境和裸机环境两种方式为用户提供对象存储、容量型、性能型等存储和计算功能. 一方面, 云平台可以隐藏复杂硬件特征, 用户无需具备专业计算机软、硬资源知识即可使用, 这种特性大大降低了个体运维成本; 另一方面, 虚拟技术为用户应用提供了独占资源环境, 可避免由资源共享带来的干扰[2−3]. 然而, 随着部署于云中的应用数量以及规模的急剧扩张, 云服务面临着一个难以忽视的高能耗问题. 在这种场景下, 在线迁移技术应时而生, 它可将应用在云中适时聚合与分离, 实现能耗可控性, 进而成为当下的研究热点之一. 深入研究有效的云端管理平台将对控制理论的发展和各种实际应用起到积极推动的作用[4−5].

与裸机在线迁移技术不同, 云进程在线迁移成本主要受两方面的影响: 虚拟执行环境和任务执行环境. 虚拟执行环境由应用执行所必须的虚拟机构成; 任务执行环境则包括任务自身执行状态, 所需输入、输出数据等. 当前云服务平台最主流的几种

收稿日期 2018-11-21 录用日期 2019-07-17
Manuscript received November 21, 2018; accepted July 17, 2019
国家自然科学基金 (61731330), 教育部产学协同育人项目 (201801154055, 201801246016, 201802003022), 四川省教育厅自然科学重点项目 (18ZA0468, 14ZA0123), 西华师范大学创新团队项目 (CXTD2014-11), 西华师范大学博士启动项目 (13E005), 西华师范大学英才基金项目 (17YC155, 17YC157) 资助
Supported by National Natural Science Foundation of China (61731330), Cooperative Production and Learning and Collaborative Education Project of the Ministry of Education (201801154055, 201801246016, 201802003022), the Key Natural Science Foundation of the Education Department of Sichuan Province of China (18ZA0468, 14ZA0123), the Innovation Team Project of China West Normal University (CXTD2014-11), the Doctoral Foundation Project (13E005), and the Meritocracy Research Fund of China West Normal University (17YC155, 17YC157)
本文责任编委 鲁仁全
Recommended by Associate Editor LU Ren-Quan
1. 西华师范大学计算机学院 南充 637009
1. School of Computer Science, China West Normal University, Nanchong 637009

执行环境为 KVM[6], VMware[7] 和 Xen[8]. 然而在迁移场景下, 由于程序运行局部性和磁盘 I/O (Input/output) 读写局限性原理, 导致记录磁盘的写操作会有较多的冗余, 它们高昂的管理开销、繁冗的管理层次更成为阻碍其发展的主要缺陷[9−10]. 鉴于虚拟机环境上述特征并不适于为大数据应用提供服务, 虚拟执行环境自身开销过高严重影响了在线迁移效率, 而轻量级容器利用自身低开销可以降低迁移系统自身的数据成本, 满足当前企业需求——高效、节能. 因此, 基于轻量级容器的迁移策略成为云服务相关技术中所急需的研究内容之一.

Docker[11] 是目前热门的容器技术之一, 如图 1 所示, 根据最新 RightScale[12] 云计算调查报告显示 Docker 的采用率从 2017 年的 35% 增长到了 2018 年的 49% (增长率为 40%). 作为一款轻量级开源容器引擎, 有能力简化和快速部署应用隔离环境, 完成云平台不同企业应用之间的“隔离”需求.

在图 2 中, 分别对比了 KVM 和 Docker 的执行开销, 具体的实验是以内存带宽基准测试程序 Stream 的 Copy, Scale, Add 和 Traid 四个组件为例. 横坐标分别代表内存占用为 20 ~ 800 MB, 纵坐标分别代表 Stream 四个组件随着脏页速率变化各自的数据处理能力. 实验结果表明, 对于内存密集型程序而言, KVM 自身执行环境比 Docker 自身执行环境多出了 20% 以上的开销. 内存计算型任务的迁移成本是影响在线迁移效率的另一个重要因素, 将庞大的应用数据迁入内存, 降低与外设交互所引发的开销是实现大数据应用高效执行的重要手段. 然而, 在动态迁移的场景下, 庞大的内存数据迁移会带来高昂的迁移开销.

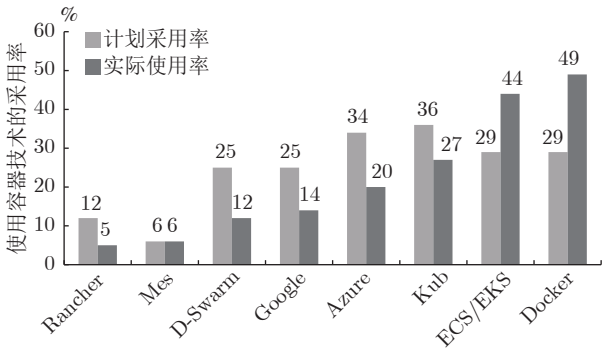

图 1　2018 年 RightScale 云计算调查报告

Fig. 1　RightScale cloud survey report of 2018

总的来说, 云端虚拟机迁移的成本主要源于两方面, 即虚拟机自身开销和用户应用所占据的资源开销. 单纯对某一方面进行管控, 都无法明显降低迁移成本. 根据对比 Docker 和 KVM 测试 Stream 发现, Docker 技术已显著减少了虚拟机自身开销, 但仍然无法满足内存密集型或者大数据应用动态迁移的成本需求. 实际上, 诸如华为、阿里等主流云服务提供商对于动态迁移所产生的时刻和所迁往的目标都有明确的管理方法, 只要科学采用这些信息, 充分利用云平台的闲置资源, 提前将一部分必须且稳定的数据放在云端存储上, 那么将显著降低宿主机的数据规模, 减少宿主机在后续迁移过程中的开销. 此外, 待迁移开始之后, 利用网络并行性, 宿主

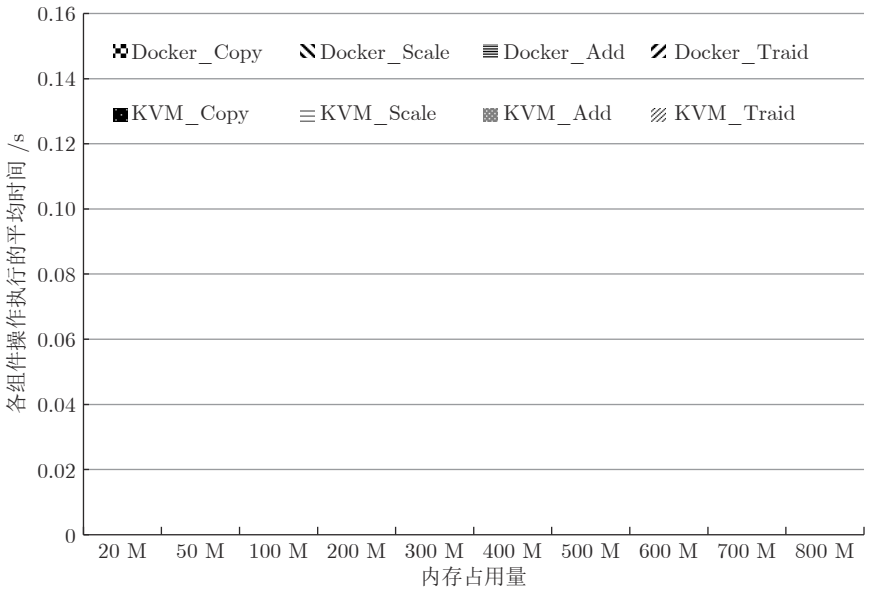

图 2　Stream 测试 KVM 和 Docker 开销

Fig. 2　Stream testing for KVM and Docker overheads

机和云端存储同时将各自数据迁移至指定目的主机, 在线迁移将同时获益于宿主机数据规模缩小以及网络并行性.

本文提出一种新的基于云预存储技术的 Docker 在线迁移方法: PF-Docker, 该方法将预存技术与成熟的 Pre-copy 技术相结合, 充分利用云平台的网络和存储资源, 实现高效在线迁移. PF-Docker 在应用程序平稳运行时, 将动态搜集用户数据行为, 并将稳定数据预存至闲置的网络存储资源上. 待在线迁移启动, 宿主机将对剩余数据实施基于 Pre-copy 的在线迁移, 而云存储中的数据将同时向目标机进行传输. PF-Docker 明显受益于缩小的宿主机数据规模以及云平台中丰富的网络存储并行性. 实验结果说明, 相对于现有容器迁移方法, 本文方法在不同类型的服务负载下均能正确执行. 对于空负载的 Docker 容器, 本方法对比文献 [13] 中的迁移方法降低了 25% 的数据成本; 在高负载写密集环境下, 本方法降低了 15% 的数据成本迁移和减少了总传输时间 11%; 在服务降级率方面, 本文 Docker 动态迁移框架比 KVM 动态迁移更加稳定.

本文主要贡献如下:

1) 云服务和预存储结合的动态迁移策略. 在迁移命令下达之前, 在云端寻找空闲服务器作为数据管理平台, 宿主机在特定条件下将容器动态迁移的稳定数据预存储到云端存储; 在迁移命令下达之后, 云端存储与宿主机利用网络并行性同时向目的机传输数据.

2) 轻量级容器 Docker 动态预存储迁移框架. 采用预存储、迭代传输和检查点恢复机制相结合的方式, 减少迭代次数与传输时间, 实现轻量级容器 Docker 动态预存储迁移.

3) 引入限定预存储阈值触发策略. 设定服务器预负载的上限, 判断运行 Docker 容器的服务器是否满足预存储的条件, 防止预存储无效占用云端资源.

1 相关工作研究

迁移技术一直是云计算研究的重点, 针对云端虚拟机环境动态迁移, 目前国内外已经有了许多相关的研究和方案. Collective 项目比较早地支持了虚拟机在线迁移, 主要是面向无实时性和不确定性的服务, 为需要迁移计算程序的用户提供了便利[14]. Kozuch 等[15] 提出了一种采用 Freeze-and-Copy 的虚拟机在线迁移方法. Clark 等[16] 提出了一种采用 Pre-copy 内存迁移算法的虚拟机动态迁移机制, 目前基于 Pre-copy 算法的虚拟机动态迁移已经得到广泛使用, 当前流行的虚拟化工具如 KVM、Xen 和 VMware 都提供基于 Pre-copy 算法的虚拟机动态迁移机制. 清华大学周佳祥等[17] 提出采用自适应双阈值进程动态迁移负载平衡系统. 北京大学张彬彬等[18] 提出了一种三阶段全系统动态迁移算法 TPM (Three-phase migration). 中国科学技术大学邵曦煜[19] 提出了非共享存储的 Ceph 虚拟机动态迁移系统. 吉林大学赵佳[20] 提出了一种新的高效迁移机制——HMDC 和 FMC. 北京航天航空大学的吕小虎等[21] 也提出了一种基于虚拟机磁盘的动态迁移机制.

虽然业界对于虚拟化动态迁移技术已经有了很多的研究成果, 但针对容器动态迁移的相关研究仍处于起步阶段. Zap 作为一个进程迁移的典型系统, 其设计目标是在网络计算体系结构下支持可迁移的计算环境, 但本身不支持动态迁移[22]. OpenVZ 是目前容器迁移技术中相对成熟的技术之一, 通过修改内核来达到虚拟化的目的, 利用检查点恢复机制支持容器的动态迁移. 但 OpenVZ 本质是基于 Linux 内核和作业系统的操作系统虚拟化技术, 和主流容器技术有很大的区别, 动态迁移效率和传统虚拟机差别不大, 且主要问题在于 OpenVZ 只能使用经过特定补丁的内核. 相比之下, Docker 容器技术直接从 Linux 内核进行虚拟化, 具有很大的优势. 针对容器迁移问题, 可以借助于进程组迁移原理来解决, 已有一些相关的开源工具和项目支持容器进程组迁移, 具体如 CRIU[23], P.Haul[24]. Docker 官方目前只支持离线迁移, 对于无状态容器, 迁移的过程为备份和恢复. 电子科技大学禹超[25] 提出基于 Linux 容器的同步迁移文件系统机制 FFSAS, 根据监控并记录 Linux 是容器对文件系统的修改, 将更改信息通过五元组的形式保存在高速缓存中, 最后同步到目的机, 减少整个迁移过程中的传输数据量, 但是这种机制很大程度上仅仅针对文件系统进程, 对于高负载写密集环境的云应用并不适用. 中国科学院大学胡丹琪[13] 提出基于容器内部检查点恢复机制的 Docker 容器整体迁移框架, 通过 runC 调用 CRIU 来实现相应功能, 避免了 P.Haul 等工具迁移完成后需要重启 Docker 守护进程的限制, 但是忽视了容器内部系统数据, 导致传输的数据量和总传输时间依旧很高.

从降低开销的角度考虑容器动态迁移问题, 减少数据传输和优化迁移算法都是正确的选择之一[26–27], 但不同的应用场景和环境可能需要采用不同的策略和方法. 以内存程序为例: 不同源主机容器中运行程序的内存文件所占比例并不可控, 因此内存程序的传输能耗标准并不一致, 选择合适的方法并不容易. 对于本文中提出的 Docker 容器动态迁移策略, 实验发现减少源主机的传输数据量能直观感受到数据传输开销的降低. 采用云服务和预存储相结合的方式有利于降低整个动态迁移过程的性能影响、开销和提高数据传输效率.

2　PF-Docker 基于预存储的动态迁移方法

本文提出基于云预存的 PF-Docker 算法, 用于解决云服务器中的高能耗、高负载和管理难等问题, 算法采用动态预存技术和成熟的 Pre-copy 技术, 以减少 Docker 容器动态迁移过程中宿主机的传输数据量, 并充分利用云服务网络和存储等资源来实现高效的 Docker 容器在线迁移. 具体流程如图 3 所示. PF-Docker 主要由两部分构成: 动态云预存和动态迭代迁移. 当服务器和内部应用程序的运行状态趋于稳定时, PF-Docker 自动进入到动态云预存阶段, 此阶段会定时动态搜集两种必要信息: 应用程序数据行为和服务器运行数据状况. 根据对运行在不同宿主机上的 Docker 容器应用程序和宿主机本身运行状态进行采样分析, 重点针对 Docker 容器自身挂载文件和内部应用程序输出数据等稳定的数据文件, 结合宿主机的 CPU 使用率、带宽传输比例和内存占比, 将符合条件的稳定数据文件 (流动数据) 动态预存至指定的处于空闲状态的网络存储资源上; 将不符合条件的数据 (冗余数据) 留在各自的宿主机上. 这两类数据呈正交, 彼此互斥, 共同组成了完整的程序运行数据空间. 当用户需要对服务器进行维护时, PF-Docker 根据相关的命令进入到动态迭代迁移阶段, 它同时驱动宿主机和云端存储器向目标主机进行在线迁移. 网络目标机将对流动数据和部分冗余数据实施网络并行传输, 宿主机将对剩余冗余数据实施基于 Pre-copy 的动态迭代迁移, 以此保证云服务网络和存储的利用最大化, 实现高效的 Docker 容器的动态迁移.

2.1　动态云预存

与传统数据预存技术[28] 不同, 动态云预存技术通过阶段性对宿主机内部 Docker 应用程序的执行状态进行学习和分析, 选择符合某种特性的稳定数据, 将其在动态迁移之前预先传输至云端存储. 如图 4 所示. 根据对容器动态迁移进行实验分析, 发现在迁移过程的每次迭代时间开销主要由遍历、压缩和传输三部分构成. 其中遍历是指对应用程序所占内存页状态的遍历时间开销; 压缩是指在迭代传输期间一次迭代中压缩脏页和其他已变化信息的时间开销; 传输是指在迭代传输期间一次迭代中传输压缩数据的时间成本. 在容器动态迁移的每次迭代过程中, 对冗余数据的遍历结果重点分为已变化和无变化两种, 而传输阶段所传输数据主要为脏页和进程输出信息等已变化的运行数据, 进一步分析发现, 在这三个阶段中遍历和压缩的时间开销是影响

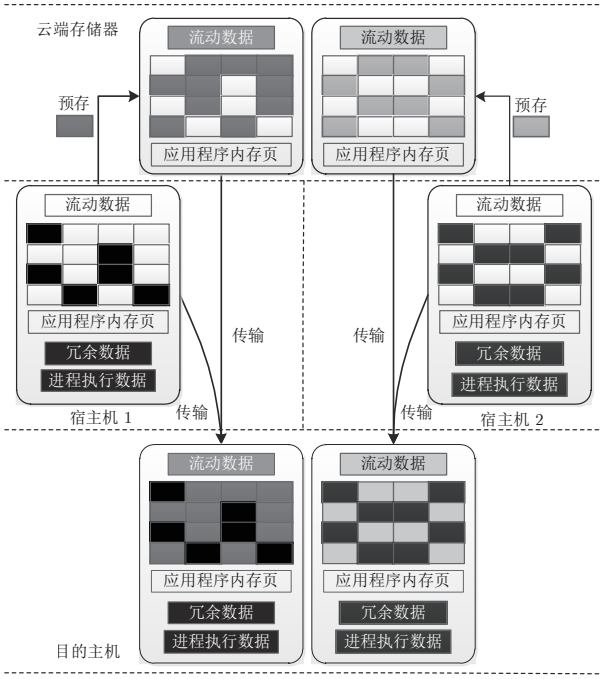

图 3　基于预存储的容器动态迁移框架: PF-Docker

Fig. 3　Prestorage-based dynamic migration framework for containers: PF-Docker

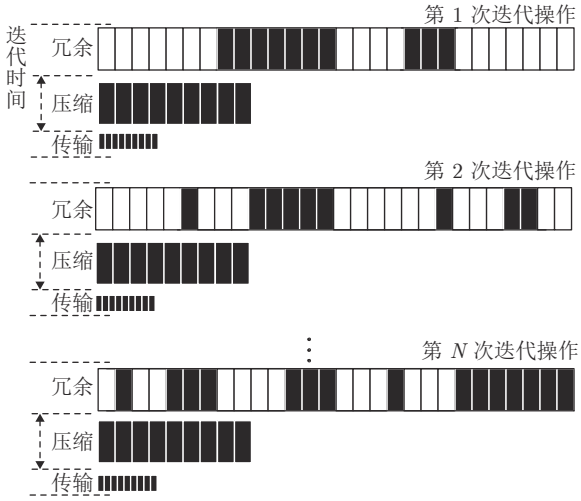

图 4　迭代传输开销分析

Fig. 4　Iterative transmission overhead analysis

迭代周期设置的重要因素. 而对于大数据应用而言, 遍历与压缩的成本会严重增加迭代周期, 进而影响容器的整体迁移时间. 在已有的研究成果中[29], 采取将压缩算法结合入迁移过程. 目前, 采用诸如 LZ4[30] 等最佳压缩算法可以将数据压缩到 1%. 这个压缩比能大大减少迭代数据的传输量, 以此迅速完成数据的传输. 本文后续将使用 LZ4 作为标准压缩算法. 在压缩算法的配合下, 传输开销的分量将明显降低, 甚至可被忽略. 其中迭代迁移的开销计算

方式如下

$$T_{\text{iter}} = T_{\text{travs}} + T_{\text{comp}} \tag{1}$$

其中, T_{iter} 表示迭代传输阶段时间间隔; T_{travs} 表示在迭代期间一次遍历所有内存页的时间成本, 与进程所占内存大小密切相关; T_{comp} 表示在迭代期间一次迭代中压缩脏页的时间成本, 与进程所占内存大小相关. 基于上述分析, 为了保证高效的网络传输和数据的均衡性, 本文根据对程序运行过程中的数据更改情况进行学习分析, 如式 (2) 所示, 在特定周期内, 数据修改频率大于这段时间的迭代次数, 那么将这种数据定义为稳定数据, 执行预存传输.

$$\eta = \sum_{i=1}^{n} C_i > \frac{T}{T_{\text{iter}}} = \frac{T}{T_{\text{travs}} + T_{\text{comp}}} \tag{2}$$

为了描述这一问题, 本文对预存数据进行如下定义: 宿主机容器和内部进程数据单次修改频率为 C, 那么 n 次数据修改频率为 $\eta = \sum_{i=1}^{n} C_i$, 当迭代传输次数小于数据修改率 η 时, 满足预存数据的定义, 进行有效的预存传输.

迁移总数据包含预存数据和宿主机数据, 而预存数据量与宿主机数据量成反比, 预存数据量越多, 则留在宿主机的数据量越少, 网络传输和云存储的利用率也就越高. 如图 5 所示, 预存数据分为 Docker 容器环境依赖和进程的数据文件, 其中 Docker 容器环境依赖包括基础镜像、子系统文件和挂载数据, 进程的数据文件包括应用的内存页、程序输出数据和进程执行信息. 分析发现 Docker 容器的环境依赖文件基本不会发生变化, 可以直接传给云端存储器; 而进程的数据文件会随着程序的运行发生改变, 这时需要采用动态预存算法对该部分数据文件进行

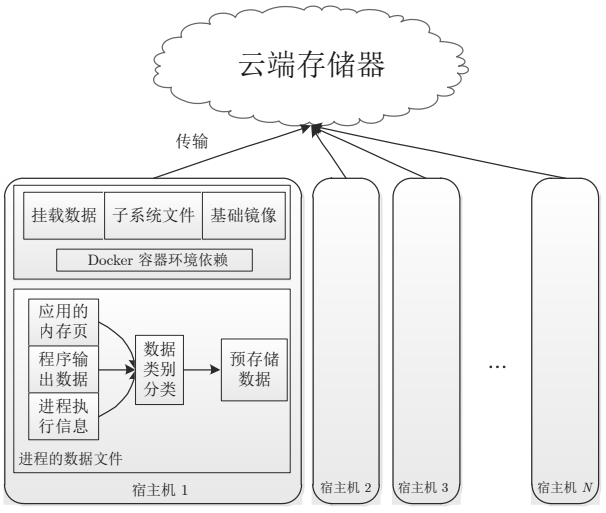

图 5 动态云预存传输流程图

Fig. 5 Dynamic cloud pre-storage transfer flowchart

分类处理, 寻找出符合条件的预存数据, 再传给云端存储器. 具体的预存传输过程分为以下两部分:

1) 在宿主机和云端存储平台之间建立 socket 通信管道, 用于预存数据的传输;

2) 利用 Docker 本身提供的 inspect 接口实现相关函数的调用, 获取与容器相关的数据信息, 将 Docker 容器运行所需的基础镜像、挂载数据、子系统文件和其他稳定的数据信息预存储到云端存储平台, 等待容器动态迁移命令下达.

在利用云预存储技术实现容器动态迁移的前提下, 为避免对宿主机容器进程的重复采样分析影响进程的自身执行情况, 本文采用限定阈值的方法来确定宿主机是否满足预存储的条件. 基于限定阈值的 PF-Docker 迁移方法需要综合考虑宿主机 CPU、带宽、内存资源的利用率.

宿主机的 CPU 使用率为

$$P_{\text{cpu}} = \frac{\sum\limits_{i=1}^{n} D_{\text{cpu}}\left(\sum\limits_{i=1}^{n} V_{d_i}\right)}{S_{\text{cpu}}} \tag{3}$$

其中, V_{di} 表示单个 Docker 容器在单个 CPU 中的使用率, $D_{\text{cpu}}(\sum_{i=1}^{n} V_{d_i})$ 表示单个 CPU 在该物理节点所有 Docker 容器 d_i 中使用率, S_{cpu} 表示该物理服务器总的 CPU 使用率.

宿主机的带宽使用率为

$$P_{\text{bw}} = \frac{\sum\limits_{i=1}^{n} D_{\text{bw}}(i)}{T_{\text{bw}}} \tag{4}$$

其中, $D_{\text{bw}}(i)$ 表示 Docker 容器 d_i 在该物理节点的带宽使用情况; T_{bw} 表示该物理节点带宽流量的最大值.

宿主机的内存使用率为

$$P_{\text{mem}} = \frac{\sum\limits_{i=1}^{n} D_{\text{mem}}(i)}{A_{\text{mem}}} \tag{5}$$

其中, $D_{\text{mem}}(i)$ 表示 Docker 容器 d_i 在该物理节点使用的内存大小; A_{mem} 表示该物理节点总的内存大小. 云服务器节点的 CPU、带宽、内存等资源的使用率可以用一个向量集合 $P_{\text{coc}} = (P_{\text{cpu}}, P_{\text{mem}}, P_{\text{bw}})$ 表示.

通过预存储阈值设定算法的检测, 重点获取当前服务器中 Docker 容器内部任务的运行状况, 主要具有以下两点优势:

1) 动态调控. 目前主流的任务类型主要分为: 计算密集型、内存密集型、多线程执行等. 用户首先可以根据 Docker 容器内部执行任务的类型来设定预存阈值的大小; 其次在保证任务正常运行的情况

下, 根据主机之间的资源利用情况来调控预存阈值的大小, 保证资源的充分利用.

2) 预存防积. 在 Docker 任务实际运行过程中, 针对计算和内存密集型等相关变化过快的类型, 为了防止频繁地触发预存, 影响 Docker 任务运行. 本文通过定时收集 Docker 任务的运行全局信息, 并在一定周期之内对全局数据比较并分析, 判断 Docker 任务的变化, 获取恰当的预存阈值参数, 防止预存积累, 减少资源损耗.

采集云服务器节点的 CPU、带宽、内存等资源的物理数据时, 必须保证采集的数据具有准确性, 同时不能对节点运行的 Docker 容器产生影响. 在确定预存储的判定条件之后, 用户根据任务类型设定一个预存储的阈值 T_{pre}. 在一定周期 T 时间内, 如果服务器的 P_{coc} 值小于设定的阈值 T_{pre}, 则该物理节点未达到预转储触发条件, 不用考虑预存储问题. 除此之外, 当 $P_{\text{cpu}} > T_{\text{pre}}(P_{\text{cpu}})$, 即 CPU 使用率超过设定的 CPU 阈值时, CPU 满足触发条件; 当 $P_{\text{bw}} < T_{\text{pre}}(P_{\text{bw}})$, 即应用服务所需带宽占比低于带宽阈值时, 带宽满足触发条件; 当 $P_{\text{mem}} > T_{\text{pre}}(P_{\text{mem}})$, 即内存使用率超过设定的内存阈值时, 内存满足触发条件. 以上各种情况均属于该物理节点满足预存储条件, 在针对不同类型任务的情况下, 采取不同的预设阈值. 具体触发流程的伪代码见算法 1.

算法 1. 预存储阈值触发判断算法

输入. $P_{\text{coc}} \leftarrow [P_{\text{cpu}}, P_{\text{bw}}, P_{\text{mem}}]$数组.

输出. 判断是否启动预存.

```
PRE_STORE ← True
PRE_SEND ← True
T_pre ← {T_bw, T_mem, T_cpu}
handle = pre_file () // get container information
function PreSTore (PRE_SEND, P_coc)
  while PRE_STORE do
    if P_coc < T_pre then
      PRE_STORE ← False
    else
      if P_coc.P_wb < T_pre.T_wb then
        handle.prestore (PRE_SEND)
        send_data (prestore_data)
        PRE_STORE ← False
      else
        if P_coc.P_cpu > T_pre.T_cpu || P_coc.P_mem > T_pre T_mem then
          handle.prestore (PRE_SEND)
          send_data (prestore_data)
          PRE_STORE ← False
        end if
        PRE_STORE ← True
      end if
      PRE_STORE ← True
    end if
    PRE_STORE ← True
  end while
```

2.2　动态迭代迁移

综合考虑宿主机和目的主机之间没有共享存储硬件和软件环境, 以及 Docker 容器数据和系统文件的特殊性, 本文设计了一种基于预存储的 Docker 容器动态迁移框架, 采用 Pre-copy 算法结合检查点恢复操作的方法, 以此来实现云端 Docker 容器动态迁移.

在 Docker 容器动态迁移时, 主要在两个阶段进行容器内部进程运行的相关信息传输: 迭代拷贝和停机拷贝阶段. 前者主要传输内存页面信息, 首先将容器进程所有内存页面传输至目的主机, 再通过迭代循环实现脏页的更新, 当宿主机和目的主机之间的内存数据基本同步之后, 跳转至停机拷贝阶段; 后者将容器运行剩余脏页和状态信息传输至目的主机. 最后在目的主机上重启容器运行, 确保迁移的容器继续平稳运行.

具体流程如图 6 所示, 在宿主机、云端存储和目的主机三者之间分别建立 socket 通信管道, 用于数据的传输. 数据的传输主要分为初始传输、迭代传输和停机传输.

1) 初始传输. 如图 6 中①所示, 在迁移命令下

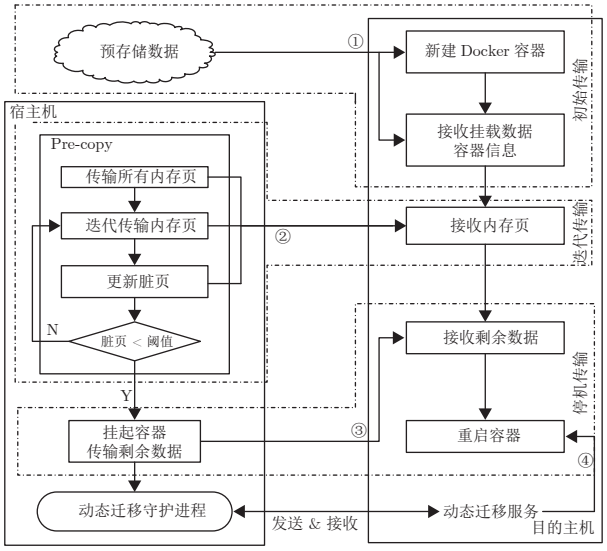

图 6　动态迭代迁移流程图

Fig. 6　Dynamic iterative migration flowchart

达之后, 宿主机给云端存储下达迁移命令, 云端存储将预存储数据传输的目的主机, 包括 Docker 容器重启所需的基础镜像、挂载数据和容器系统信息等相关数据.

2) 迭代传输. 如图中②所示, 在宿主机向云端存储下达迁移命令的同时, 宿主机对 Docker 容器内部进程执行检查点预转储操作, 然后利用 Linux 内存跟踪机制获取修改后的内存页, 并迭代执行数据传输. 最后在满足一定的循环终止条件后, 结束迭代传输进入停机传输.

3) 停机传输. 如图中③和④所示, 对宿主机上的 Docker 容器内部进程执行检查点转储操作, 挂起容器和容器内部运行的程序, 获取剩余的脏页和容器进程执行数据信息. 将检查点转储文件传输到目的主机上, 待目的主机完全接收到重启容器和进程所需的文件后, 执行重启容器操作. 如若重启成功, 则动态迁移正常结束, 删除宿主机上被挂起的 Docker 容器和进程, 并卸载相关文件目录, 删除云端存储上的所有预存数据文件. 反之容器与进程回到宿主机与云端存储, 恢复被挂起的 Docker 容器与进程.

基于预存储的 Docker 容器动态迁移, 物理机之间不需要共享存储硬件的支持, 降低了硬件环境的要求. 同时 Docker 容器本身开销相对于进程而言可以忽略不计, 利用 Pre-copy 算法进行迭代传输减少了迁移的总时间和停机时间.

3 实验设计与分析

3.1 实验设计

本节从两个方面对 PF-Docker 迁移开销进行评估: 应用类型与并行度. 应用类型对在线迁移的开销有明显影响, 本节选取了两种代表性应用类型, 分别为内存计算密集型和 I/O 型. 其次, 本节还对并行程序的迁移开销进行了评估, 主要选取了多线程 Linux 内核编译.

本文分别从迁移总时间、宿主机迁移数据量和服务降级率 3 个方面来检测迁移效率, 为此选择了 Loop 测试和 Stream 测试两种具有代表性的测试用例来检测 PF-Docker 动态迁移的性能指标. 具体如表 1 所示.

PF-Docker 框架的 3 台服务器操作系统均为 Ubuntu14: 04.1, 内核的版本为 4.2.0-36-generic, 安装相同版本 CRIU 和 LM-Docker, 其中 CRIU 为 2.12, LM-Docker 是在官方 Docker boucher 目录下 v1.9.0-experimental-cr.1 版本基础上进行修改. 实验时 Docker 容器中部署 ubuntu 操作系统, 采用基础镜像为 ubuntu14: 04.1, 大小为 788 MB. 文献 [13] 框架的服务器与本文的框架硬件和软件配置一样, 不同之处在于文献 [13] 框架采用的是两台服务器.

3.2 评估方法

本文采用了文献 [13] 和 [25] 的评测方法, 分别从迁移总时间、宿主机迁移数据量、服务降级率 3 个方面对 PF-Docker 动态迁移方法进行评估.

1) 迁移总时间: 是指迁移指令发起到整个迁移过程完成所耗费的总时间, 具体为

$$T_{\mathrm{PF}} = \sum_{i=1}^{n} T_{\text{pre-dump}}(i) + T_{\mathrm{rover}} + T_{\mathrm{dump}} + T_{\mathrm{res}} \quad (6)$$

其中, $\sum_{i=1}^{n} T_{\text{pre-dump}}(i)$ 是迭代传输的总时间, T_{rover} 为流动数据和部分冗余数据传输时间, T_{dump} 是停机传输时间, T_{res} 是目的机重启容器进程的时间, 以上各部分时间构成了 PF-Docker 迁移总时间. 由于云服务中动态迁移应用场景多在负载均衡和在线升级维护上, 为了防止因突发情况导致动态迁移失败, 要求动态迁移总时间尽可能低.

2) 总迁移数据量: 是指迁移指令发起到整个迁移过程完成所传输数据的总量, 具体为

$$D_{\mathrm{PF}} = \sum_{j=1}^{n} D_{\text{pre-dump}}(j) + D_{\mathrm{rover}} + D_{\mathrm{dump}} + D_{\mathrm{edy}} \quad (7)$$

其中, $\sum_{j=1}^{n} D_{\text{pre-dump}}(j)$ 是迭代传输数据量, D_{rover} 是流动数据量, D_{dump} 是停机传输数据量, D_{edy} 是部分冗余数据量, 以上各部分数据构成了 PF-Docker 迁移总数据量. 由于云服务中应用进程彼此之间对网络资源存在一定的竞争, 而传输数据占用的网络带宽与传输数据量的大小成正比, 为了减少服务器网络带宽的占用和降低对云服务网络 I/O 的影响, 动态迁移过程中传输的数据量越少越好.

3) 服务降级率: 指动态迁移等其他服务操作对

表 1 实验负载类型表

Table 1 Experimental load type table

负载类型	具体配置
Loop 测试	基础镜像为 Ubuntu14.04.1, 在 Ubuntu 操作系统下执行永真循环打印程序 (低负载写空闲)
Stream 基准测试	基础镜像为 Ubuntu14.04.1, 通过设置不同的内存读写速率进行测试 (高负载写密集)
多线程内核编译测试	在 Ubuntu14.04.1 上部署内核编译依赖环境, 生成镜像 zx/Ubuntu1, 在不同线程下测试 (并行程序写密集)

进程运行的影响, 具体为

$$S_{\mathrm{PF}}=\sum_{a=1}^{n}D_{\mathrm{checkpoint}}(a)+S_{\mathrm{lm}}+S_{\mathrm{res}} \tag{8}$$

其中, $\sum_{a=1}^{n}D_{\mathrm{checkpoint}}(a)$ 是检查点操作对容器进程服务影响, S_{lm} 是容器停机传输过程中对容器内部运行进程的影响, S_{res} 是重启容器服务对容器内部运行程序的影响. 因为动态迁移中传输数据会抢占整体云服务中的带宽资源, 频繁检查点操作还会占用服务器的 CPU 资源, 这都会对运行中的应用程序服务性能带来负面影响, 所以好的迁移方法在迁移过程中应尽量减少服务降级率.

3.3　实验结果分析

3.3.1　面向不同应用类型的迁移

3.3.1.1　Loop 场景测试

在 Loop 测试场景下对比本文和文献 [13] 两种 Docker 迁移框架的性能. 如图 7 所示, 由于容器内部应用程序和容器自身的脏页产生率较低和网络传输的延迟性, 导致动态迭代迁移过程中的迁移算法和压缩算法带来的收益无法实现, 因此在传输时间开销上, 本文 PF-Docker 迁移框架对比文献 [13] 的 Docker 迁移框架不占优势. 但是由于数据预存储使得宿主机向目的主机传输的数据量降低了 25%, 减少了宿主机的网络带宽使用, 降低了部分开销.

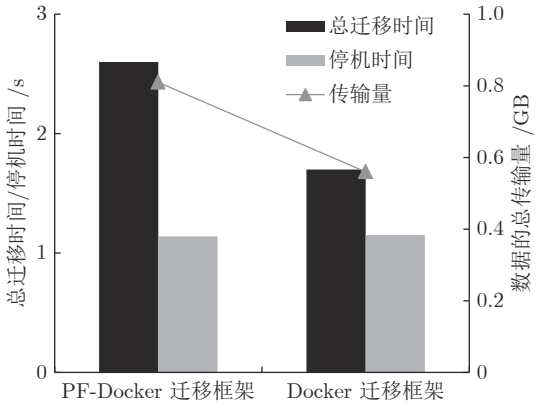

图 7　读写空闲场景下两种 Docker 动态迁移方法性能对比

Fig. 7　Performance comparison of two Docker dynamic migration methods in read-write idle scenarios

3.3.1.2　Stream 场景测试

Stream 测试是业界公认的内存带宽性能测试基准工具. 同 Loop 的读写空闲测试相比, Stream 基准测试可以选择读写密集空间的测试. 为了分析在不同读写密集空间脏页率下 PF-Docker 容器动态迁移性能, 分别选取内存数据大小为 20 MB, 50 MB, 100 MB, 200 MB, 300 MB, 400 MB, 500 MB, 600 MB, 700 MB, 800 MB, 900 MB, 1 000 MB, 1 500 MB 和 2 000 MB 14 组负载进行测试. PF-Docker 将分别从总迁移时间和总迁移数据量两个方向和文献 [13] 的 Docker 迁移方法进行性能对比, 从应用程序服务降级率和 KVM 动态迁移进行性能对比.

由于 Stream 测试进程的脏页更新率与 Stream 进程的占用内存成正比. Docker 动态迁移面临着随内存占用的增大, Stream 测试进程的脏页更新率不断增大以及网络带宽的堵塞, 导致动态迁移长时间内无法实现数据的收敛, 增加了动态迁移针对脏页传输的迭代次数, 从而导致整个动态迁移的总时间会不断增加. 而在本文 PF-Docker 迁移框架中, 由于提前将数据预存储, 减少了预传输阶段对源主机网络带宽的堵塞, 使得动态迁移算法可以更快地实现数据的收敛.

图 8 为 PF-Docker 迁移框架和文献 [13] 的 Docker 迁移框架在迁移总时间开销方面的对比, 分析发现当 Stream 的内存占用在 500 MB 之前时, 对比两种框架在总迁移时间并没有很大的变化, 然而在 500 MB 之后, 随着 Stream 内存占用的不断增大, 网络带宽堵塞的情况越来越严重, 导致容器动态迁移的数据收敛越来越困难, 而本文 PF-Docker 迁移框架在迁移总时间开销的效果就越来越明显, 总迁移时间相比于文献 [13] 的方法降低了 20%.

图 9 为 PF-Docker 迁移框架数据总传输量和 Docker 迁移框架数据总传输量. 其中, 根据 PF-Docker 迁移框架的特殊性, 为更好地分析数据传输来源, 将其又细分为宿主机传输数据量和预存储传输数据量. 实验对比中, PF-Docker 的数据总传输量始终低于 Docker 动态迁移框架, 并随着 Stream 内存占比的不断增加, 数据总传输量的优势越来越明显. 同时根据对 PF-Docker 的数据来源分析, 分为预存储和宿主机分别向目的主机传输, 而 Docker 迁移框架只采用宿主机向目的主机的传输方式, 对比分析发现在单一从宿主机向目的主机传输的数据量中, PF-Docker 的宿主机传输量低于 Docker 迁移框架的宿主机传输量, 随着数据的收敛和迭代传输次数的减少, 在宿主机总迁移数据也降低了 22%.

如图 10 所示, 实验测试内容包括 Docker、PF-Docker、KVM 和 LM-KVM, 分别代表 Docker 本地执行、PF-Docker 动态迁移、KVM 本地执行和 KVM 动态迁移. 通过实验对比 Docker、PF-Docker、KVM 和 LM-KVM 在 Stream 执行时间随脏页速率的变化. 可以看到, PF-Docker 动态迁移在脏页率较少时, 性能略差于 KVM 本地执行, 但随着脏页率增加, PF-Docker 方法表现越好. 从图 10 中还

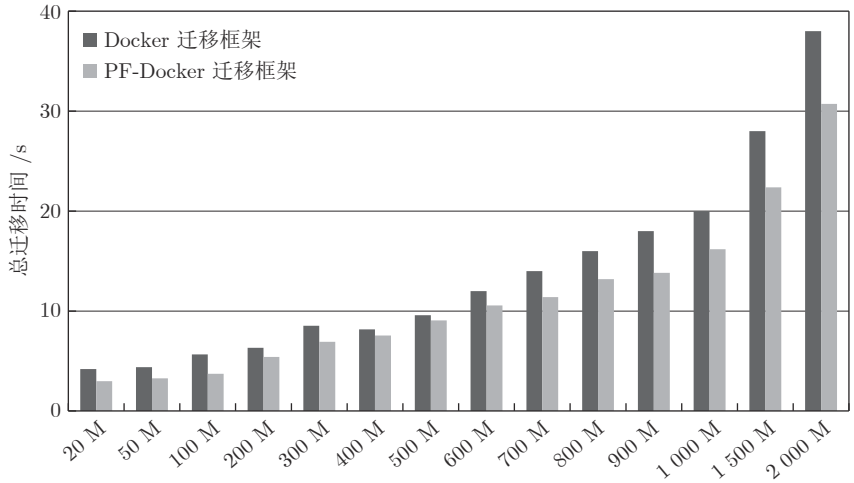

图 8 迁移总时间对比

Fig. 8 The comparison of total migration time

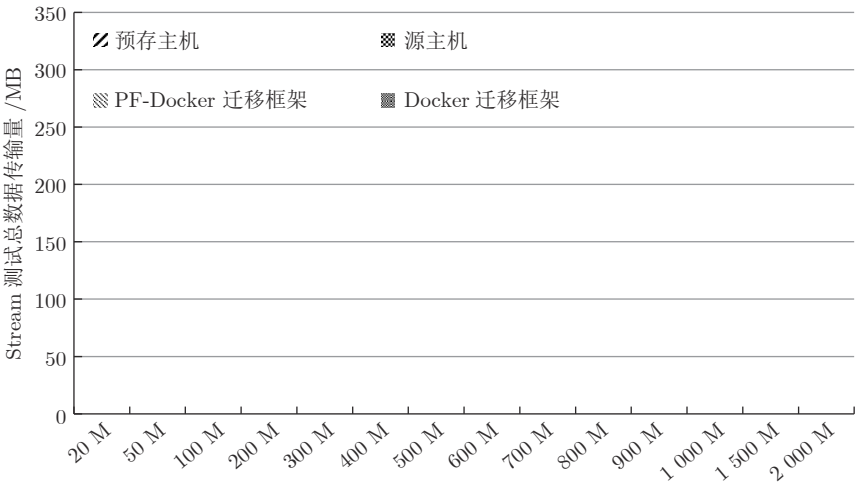

图 9 源主机迁移总数据量对比

Fig. 9 The comparison of total data volume of source host migration

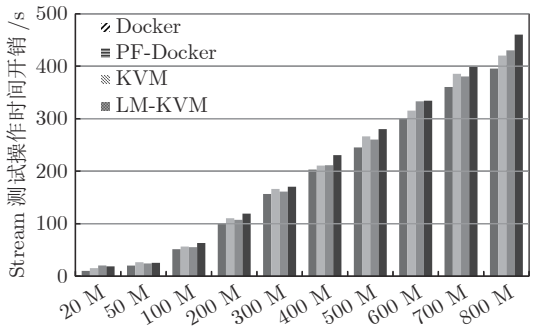

图 10 Stream 测试下 PF-Docker 与 LM-KVM 对比

Fig. 10 The comparison of PF-Docker and LM-KVM under Stream test

可以看出, 在该实验场景下, Docker 和 KVM 性能差异不是很大, 这是由于 Stream 测试中内存读写操作是连续的, 内核通过数据预取操作进行优化, KVM 虚拟内存和宿主机物理内存映射次数较少. 这也是本文选择 Stream 测试工具作为对比分析动态迁移性能差异的原因.

3.3.2 面向并行程序的迁移

以下对内核编译测试数据进行分析. 如图 11 所示, 对比 Docker 本地执行和 KVM 本地执行性能, 在 8, 4, 2, 1 四种多线程并发下本地 KVM 开销比 Docker 分别多耗了 33.3%, 38.1%, 17.9%, 18.5%. PF-Docker 动态迁移执行性能相比 Docker 本地分别下降了 4.8%, 3.6%, 2.4%, 1.8%. KVM 动态迁移执行性能相比 KVM 本地分别下降了 21.4%, 17.2%, 183%, 78%. KVM 动态迁移执行开销相比 PF-Docker 动态迁移分别多耗了 54.6%,

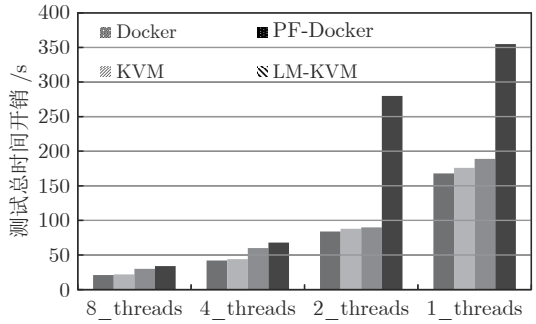

图 11　内核编译负载下 Docker 与 KVM 对比

Fig. 11　The comparison of Docker and KVM under the kernel compilation load

56.3%, 225%, 107%. 由此可知, 在多线程并发高负载下 PF-Docker 动态迁移带来的服务降级率比 KVM 要低.

3.3.3　面向风险迁移过程

针对计算密集型和内存密集型的进程, 在迁移过程出现失败的几率会大大增加, 结合在第 2 节提到的 3 个预存条件 (CPU、内存、带宽) 和执行多线程编译任务的实验数据进行动态迁移的风险迁移分析.

如图 12 所示, 预存阶段频繁使用 3 个单一的阈值触发判定条件会一定程度影响计算密集型和内存密集型任务的运行状态, 本文通过预存阈值的动态调控机制, 同等环境下对比任务自身运行状态升高了 7%, 但对比单一阈值判定条件对任务运行状态的影响降低了 5%, 采用预存阈值动态调控机制, 能一定程度降低对任务本身运行状态的影响.

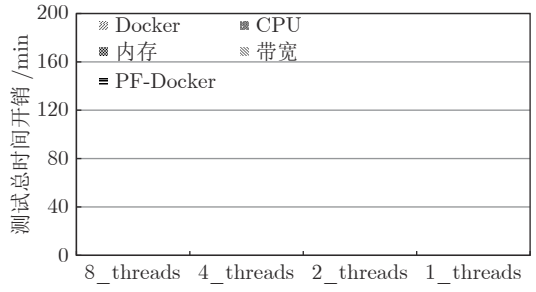

图 12　多线程内核编译在不同预存条件下对比

Fig. 12　The comparison of multithreaded kernel compilation under different pre-stored conditions

总的来说, 本文提出的基于 Docker 容器动态迁移预存储算法是在整个迁移过程中分别加入第三方数据管理平台、引入预存储阈值机制. 实验对比表明, 从总传输数据量、总迁移时间、服务降级率三个方面来看, 在 Loop 测试和 Stream 测试上, 本文框架比文献 [13] 中的框架从以下几个方面都有所降低, 其中包括数据量、传输时间等, 在 Stream 测试和内核编译测试上, 本文 Docker 迁移框架比 KVM 动态迁移服务性能更加优秀和稳定. 从而进一步提高了云中资源的利用率, 降低了部分服务器的高负载问题.

4　结束语

随着云计算技术的高速发展, 如何降低能耗是保证高效利用云资源的前提. Docker 作为目前业界使用率最高的容器引擎, 实现其动态迁移技术能有效降低能耗, 提高容器本身的容错率. 实验结果表明, 本文提出的基于 Docker 容器动态迁移预存储算法通过在容器迁移过程加入预存储机制和引入预存储阈值的操作, 使得容器迁移过程中减少了不必要的迁移, 从而提高了迁移的效率和降低了能耗, 可以帮助现有的云资源进行有效的利用和管理. 随着容器迁移技术的持续发展, 在后续研究中, 将会进一步考虑降低内存页的传输量和数据的存储开销, 研究并设计切实可行的相关算法, 最后从大规模任务、跨域云迁移等多种环境来证实算法的可行性, 使得容器迁移更加有效、节能.

References

1　Guo Gang, Yu Jiong, Lu Liang, Ying Chang-Tian, Yin Lu-Tong. Data migration model based on RAMCloud hierarchical storage architecture. *Journal of Computer Applications*, 2015, **35**(12): 3392−3397
(郭刚, 于炯, 鲁亮, 英昌甜, 尹路通. 内存云分级存储架构下的数据迁移模型. 计算机应用, 2015, **35**(12): 3392−3397)

2　Filani D. Dynamic data center power management trends, issues, and solutions. *Intel Technology Journal*, 2008, **12**(1): 59−67

3　Shieh A, Kandula S, Greenberg A G, Kim C, Saha B. Sharing the data center network. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. Boston, USA: USENIX Association, 2011. 309−322

4　Xia Yuan-Qing. Cloud control systems and their challenges. *Acta Automatica Sinica*, 2016, **42**(1): 1−12
(夏元清. 云控制系统及其面临的挑战. 自动化学报, 2016, **42**(1): 1−12)

5　Xia Yuan-Qing, Yan Ce, Wang Xiao-Jing, Song Xiang-Hui. Intelligent transportation cyber-physical cloud control systems. *Acta Automatica Sinica*, 2019, **45**(1): 132−142
(夏元清, 闫策, 王笑京, 宋向辉. 智能交通信息物理融合云控制系统. 自动化学报, 2019, **45**(1): 132−142)

6　Fenn M, Murphy M A, Martin J, Goasguen S. An evaluation of KVM for use in cloud computing. In: Proceedings of the 2nd International Conference on the Virtual Computing Initiative. 2008.

7　Feng X J, Tang J X, Luo X, Jin Y H. A performance study of live VM migration technologies: VMotion vs XenMotion. In: Proceedings of the Asia Communications and Photonics Conference and Exhibition. Shanghai, China: IEEE, 2011. 1−6

8　Liu Z B, Qu W Y, Yan T, Li K Q. Hierarchical copy algorithm for Xen live migration. In: Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. IEEE, 2010. 361−364

9　Travostino F, Daspit P, Gommans L, Jog C, De Laat C, Mambretti J, et al. Seamless live migration of virtual machines over the MAN/WAN. *Future Generation Computer Systems*, 2006, **22**(8): 901−907

10　Bradford R, Kotsovinos E, Feldmann A, Schiöberg H. Live wide-area migration of virtual machines including local persistent state. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. San Diego, USA: ACM, 2007. 169−179

11 Docker. Make better, secure software from the start [Online], available: http://www.docker.com/, March 12, 2018

12 Rightscale-2018-state of the cloud report [Online], available: https://www.rightscale.com/li/state-of-the-cloud, January 25, 2018

13 Hu Dan-Qi. Docker Container Dynamic Migration Framework Based on Cloud Computing, Academy of Sciences [Master thesis], University of Science and Technology of China, China, 2017.
(胡丹琪. 基于云计算的 Docker 容器动态迁移框架 [硕士学位论文], 中国科学院大学, 中国, 2017.)

14 Chandra R, Zeldovich N, Sapuntzakis C, Lam M S. The collective: A cache-based system management architecture. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation-Volume 2. Boston, USA: USENIX Association, 2005. 259−272

15 Kozuch M, Satyanarayanan M. Internet suspend/resume. In: Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications. Callicoon, USA: IEEE, 2002. 40−46

16 Clark C, Fraser K, Hand S, Hansen J G, Jul E, Limpach C, et al. Live migration of virtual machines. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation-Volume 2. Boston, USA: USENIX Association, 2005. 273−286

17 Zhou Jia-Xiang, Zheng Wei-Min, Yang Guang-Wen. Adaptive dual-threshold dynamic load balancing system based on migrating. *Journal of Tsinghua University (Science and Technology)*, 2000, **40**(3): 121−125
(周佳祥, 郑纬民, 杨广文. 一种基于进程迁移的自适应双阈值动态负载平衡系统. 清华大学学报 (自然科学版), 2000, **40**(3): 121−125)

18 Zhang Bin-Bin, Luo Ying-Wei, Wang Xiao-Lin, Wang Zhen-Lin, Sun Yi-Feng, Chen Hao-Gang, et al. Whole-system live migration mechanism for virtual machines. *Acta Electronica Sinica*, 2009, **37**(4): 894−899
(张彬彬, 罗英伟, 汪小林, 王振林, 孙逸峰, 陈昊罡, 等. 虚拟机全系统在线迁移. 电子学报, 2009, **37**(4): 894−899)

19 Shao Xi-Yu. Optimization of Non-shared Storage Virtual Machine Live Migration System Based on Ceph [Master thesis], University of Science and Technology of China, China, 2018.
(邵曦煜. 基于 Ceph 的非共享存储虚拟机动态迁移系统的优化 [硕士学位论文], 中国科学技术大学, 中国, 2018.)

20 Zhao Jia. Research on Live Migration of Virtual Machine [Ph.D. dissertation], Jilin University, China, 2013.
(赵佳. 虚拟机动态迁移的关键问题研究 [博士学位论文], 吉林大学, 中国, 2013.)

21 Lv Xiao-Hu, Li Qin. Research and implementation on migration of virtual machine including VM-disk. *Computer Science*, 2009, **36**(7): 256−261
(吕小虎, 李沁. 虚拟机磁盘迁移技术研究与实现. 计算机科学, 2009, **36**(7): 256−261)

22 Osman S, Subhraveti D, Su G, Nieh J. The design and implementation of Zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 2002, **36**(SI): 361−376

23 CRIU. Checkpoint/restore [Online], available: https://criu.org/Checkpoint/Restore, March 12, 2018

24 CRIU. P. Haul [Online], available: https://criu.org/P.Haul, March 13, 2018

25 Yu Chao. Research on the Live Migration Mechanism of Linux Containers [Master thesis], University of Electronic Science and Technology of China, China, 2015.
(禹超. Linux Containers 热迁移机制研究 [硕士学位论文], 电子科技大学, 中国, 2015.)

26 Breitgand D, Kutiel G, Raz D. Cost-aware live migration of services in the cloud. In: Proceedings of the 3rd Annual Haifa Experimental Systems Conference. Haifa, Israel: Association for Computing Machinery, 2010. Article No. 11

27 Jaikar A, Huang D D, Kim G R, Noh S Y. Power efficient virtual machine migration in a scientific federated cloud. *Cluster Computing*, 2015, **18**(2): 609−618

28 Mogul J C, Farkas K I, Ranganathan P, Pinheiro E S. System and Method for Energy Efficient Data Prefetching, U.S. Patent 7437438, October 14, 2008

29 Krishnan N, Baron D. A universal parallel two-pass MDL context tree compression algorithm. *IEEE Journal of Selected Topics in Signal Processing*, 2015, **9**(4): 741−748

30 LZ4 [Online], available: http://lz4.github.io/lz4, March 25, 2018

赵 旭 西华师范大学计算机学院硕士研究生. 主要研究方向为云计算, 虚拟化平台.
E-mail: zhaouxu_xu@163.com
(**ZHAO Xu** Master student at the School of Computer Science, China West Normal University. His research interest covers cloud computing and virtualization platform.)

李艳梅 西华师范大学计算机学院副教授. 2013 年获得电子科技大学计算机科学与工程学院博士学位. 主要研究方向为计算机视觉, 云计算与大数据处理. 本文通信作者.
E-mail: liyanmei76@126.com
(**LI Yan-Mei** Associate professor at the School of Computer Science, China West Normal University. She received her Ph.D. degree from the School of Computer Science and Engineering, University of Electronic Science and Technology of China in 2013. Her research interest covers computer vision, cloud computing, and big data processing. Corresponding author of this paper.)

罗 建 西华师范大学计算机学院副教授. 2009 年获得电子科技大学计算机科学与工程学院硕士学位. 主要研究方向为计算机视觉, 云计算与大数据处理.
E-mail: luojian@cwnu.edu.cn
(**LUO Jian** Associate professor at the School of Computer Science, China West Normal University. He received his bachelor degree from the School of Computer Science and Engineering, University of Electronic Science and Technology of China in 2009. His research interest covers computer vision, cloud computing, and big data processing.)

罗金梅 西华师范大学计算机学院硕士研究生. 主要研究方向为计算机视觉和云计算.
E-mail: luojinmei_w@163.com
(**LUO Jin-Mei** Master student at the School of Computer Science, China West Normal University. Her research interest covers computer vision and cloud computing.)