P1

各位评委、导师以及参赛选手们，下午好。我是来自华中科技大学的郑云鲲，今天我代表我们的团队为大家介绍我们的项目——vKernel：虚拟内核场景下面向轻量级虚拟化的优化技术。本项目旨在解决容器隔离性不足的问题，同时在容器化环境中实现高效性能与安全性的平衡。

P2

今天我的介绍分为四个方面来阐述，分别为背景、现存问题、系统设计、评估。

我们进入第一部分，

P3 （gvisor 之类的不是介绍了隔离性不足才提出存在这些方案吗，但是不能完全解决问题？顺序是不是要往后调）

容器技术作为一种操作系统级虚拟化技术，在云计算中带来了重大的变革。相比传统的虚拟机技术，容器通过共享主机操作系统的方式实现了轻量化和快速启动的特性，从而可以快速部署容器化应用程序，与硬件虚拟化相比，容器的内核虚拟化是一种更加轻量级的虚拟化方法，具有明显的性能优势。

但是现在的容器技术存在一些问题。

第一，只依赖内核提供的机制支持容器化应用间的物理资源限制，无法实现完全的隔离，容易产生 CPU 资源争用，内存资源泄露等问题。

第二，无法满足应用的多样化调度需求。为了保证运行过程中的安全性和隔离性，通常情况下仅允许其内应用使用单一策略且不可配置参数，无法满足应用的多样化调度需求，这会对容器内应用造成严重的性能损失。

可以说，共享内核一方面为容器带来高效性，但是另外一方面也给容器带来了诸多的安全隐患和性能损耗。这些问题成为制约容器发展和应用的主要技术瓶颈。

针对容器共享宿主内核导致的隔离性不足、无法个性化定制内核机制的问题，近年来出现的解决方案总的来说有如下方向：

（1）用户级内核隔离： 用户级别的内核隔离将对主机内核的请求重定向到在用户级别实现的特定于应用程序的内核。重定向用户级请求的关键和开销的主要

来源是拦截对主机内核的请求。 例如 gVisor，拦截应用程序系统调用以创建系统接口，类似于主机内核，无需硬件虚拟化。然而，请求拦截不可避免地会导致过多的上下文切换，从而带来巨大的开销。

（ 2）基于 虚拟机的 的内核隔离：比较典型的是基于虚拟机的安全容器方案 Kata， Kata 容器为每个容器运行私有的轻量内核，并通过硬件虚拟化提供容器隔离。但它需要虚拟机监视器 (VMM) 将虚拟化硬件公开给 guest kernel，与原生容器相比仍然会产生不可忽略的开销。

（ 3）基于 linux 内核现有机制隔离：利用操作系统内核中现有 的资源管理和安全机制，例如 cgroup、 namespace、 capabilities、 seccomp 和 apparmor ，为容器提供系统资源的隔离视图，限制它们对系统调用、特权函数和敏感文件的访问。虽然这种方法由于与主机内核紧密耦合而实现了接近本机的性能，但它无法提供足够的隔离或允许应用程序自定义内核配置或策略。

P5

总的来说，这三种解决方案都有各自缺陷，无法完全解决上述隔离性不足、个性化定制化不足、安全性不足的问题。

只依赖内核提供的机制支持容器化应用间的物理资源限制导致了容器隔离性不足。

而不同应用对内核的个性化需求不能满足，也限制了应用的发挥。

P6

安全性也是现有容器技术不容小觑的问题。

例如，2019 年发现的“Dirty Cow”漏洞 (CVE-2016-5195) 利用了 Linux 内核的一个竞争条件漏洞，允许本地用户提升权限。攻击者可以从一个容器中逃逸并获得宿主机的控制权，从而影响所有容器的安全。

P7

具体到 CPU 调度和内存管理的层面，
CPU 调度上
默认情况下容器并不具有策略和参数的配置权限，这就导致了不可配置且与主机

共享的参数无法满足应用多样化的调度需求。
而要为应用配置 CFS 以外的策略时，需要提升容器具有的权限，这时可以通过添加运行参数来实现。但这也会引入安全性威胁和隔离性被破坏的问题。

内存管理上，容器缺少独立的虚拟内存管理机制。

在多容器环境下，由于不同容器应用共享宿主内核机制，容器缺少独立的虚拟内存管理机制，使得容器应用无法个性化定制自身虚拟内存管理策略。更为严重的是，在宿主机限制系统虚拟内存总量时，所有容器共享这些虚拟内存限制，如果存在恶意容器过度占据虚拟内存，会严重影响应用的性能与稳定。

针对上述问题，本项目拟进行的工作是：实现既保证隔离性又拥有近似于原生 Docker 容器性能的容器技术。可对现有轻量级虚拟化技术进行优化，或提出新的轻量级虚拟化方法并实现。
要做这样的工作，首先要选择两个基本的技术方案：第一个是内核扩展方式的选择，第二个是实现内核函数重定向到虚拟内核的方式。

P9

首先，要实现内核功能拓展，我们横向对比了以下几种方式：

1.通过修改内核扩展 cgroup：高效实现定制化策略，但需要重新编译内核，对系统的稳定性和可维护性会造成影响

・2.可加载内核模块 LKM:优势在于可以在运行时动态加载，无需重新编译内核，易于部署，且可加载模块同样位于内核态

・ 3. eBPF（扩展伯克利包过滤器）：在内核空间中运行可编程的虚拟机，依赖于 kprobes 机制，性能开销较大,安全性更强

P10

因此，在系统设计上，我们使用了 LKM 的方式，容器与自身对应的虚拟内核直接进行交互，虚拟内核中通过对容器依赖的最小化内核代码与数据实现容器私有化。

P11

第二个问题是，如何实现重定向，将容器对内核部分函数的访问重定向至虚拟内核，从而使个性化定制的内存配置策略能顺利地起作用。

在传统内核中，存在两种重定向技术，一种是基于陷阱（trap）的方法，另一种是基于跳转的方。Ftrace 是一种高效的内核执行流跟踪与重定向技术。Ftrace 包括一整套内核跟踪与分析工具，我们这里只关注其底层内核执行流重定向机制。ftrace 基于跳转的方式比基于 trap 的方式性能更加出色，但是在实际应用中仍然存在一些不可避免的开销。

因此我们采用了一种简化的基于跳转的重定向技术 inline hook。Inline hook 技术的设计方式是通过在函数入口处插入一个 jmp 指令，并直接指向自定义函数的入口地址。当内核调用该函数时，会首先跳转到自定义函数的入口处执行自定义的代码，完成操作后再跳转回原函数继续执行。

P12

我们的对现有的轻量级虚拟化技术进行优化的预期目标是

1. 高性能： 不同于 gVisor 和 Kata 等方案以牺牲性能为代价换取隔离性， 本项目希望能够实现接近原生 Docker 容器的负载性能

高安全性： vkernel 系统部署在虚拟机环境中， 只需要对虚拟机内核进行修改，不会对主机内核的安全性造成影响

可定制： 用户可以自定义 vkernel,在特定容器中支持自定义调度策略

轻量级： 在实现较强的隔离性和安全性的前提下， 能够实现接近 Docker 容器的启动延迟， 不会带来额外的开销

P13

接下来我从 CPU 调度、内存管理策略、系统调用、日志隔离四个方面详细介绍系统设计。

对 CPU 调度的虚拟化从两个方面进行设计，第一个是独立任务调度，第二个是参数视图隔离

独立任务调度，就是局部调度器上的 CPU 资源的分配应当不受到全局调度器上配置的影响， 自定义调度策略。

参数视图隔离，就是局部调度器不再复用全局调度器上的参数配置， 而具有其独立的参数列表。

P14

独立任务调度的方面，首先要考虑策略接口，然后考虑具体的自定义策略的定义。

首先要考虑本项目采用策略作用于整个控制组的方案。从线程， 运行队列，控制组角度考虑

1. 如果从线程的角度，设计实现过于复杂
2. 如果策略应用于单个队列，同一个控制组在不同 CPU 上的运行队列使用的策略不同。将任务固定在同一个 CPU 上，影响任务运行，浪费硬件资源。
3. 如果策略作用于整个控制组，容器本身依赖控制组实现资源控制， 与局部调度器具有良好的适配性。大量的接口可以降低对策略配置实现的复杂性， 便于实现

因此，本项目采用策略作用于整个控制组的方案，同时增加"real_policy" 变 量，

指示局部调度器实际策略类型。

关于 Real_policy 变量，会在 对 参 数 文 件“cpu.real_policy”进行读写时， 同时更新所属调度组的对应变量。

（可详细描述看篇幅决定）

自定义策略方案采用 FIFO，FIFO 调度策略下先进先出的特性使其并不需要复杂的红黑树结构，取而代之的是简单的链式结构。

P17

运行队列考虑了轻量性和兼容性。

轻量性上，参考 RT 调度器中对 SCHED_FIFO 的实现，同时进一步简化， 用单链表方式实现。投入的成本和引入的判定过程并不复杂。

兼容性上，整体复用 CFS 调度器的结构， 只在最关键的选择和放置逻辑处， 即 task_timeline 处增加一个链式运行队列 tasks_list，只修改了调度实体 sched_entity

在 cfs_rq 上的放置方式

P18

运行队列的维护上，分为入队、出队、选择（根据篇幅决定是否详细描述）

时机的判断，在周期性调度中， 仅更新任务运行信息。而在在主动调度中， 两个调度实体属于同一个容器即控制组时， 可直接跳过抢占检查，而其它情况下按照原生 CFS 调度器的判断方式来进行.

P19

在 CPU 调度策略的选择上采用自适应策略，具体的策略选择逻辑是（根据篇幅决定是否详细描述）

P20

CPU 调度虚拟化的第二个部分是参数隔离

对于参数隔离，第一个问题是，要隔离哪些参数？ProcFS 下有大量的参数与调度系统相关， 可以选择明确影响相关调度器运行结果的参数进行隔离。

本项目基于命名空间机制以 sched_wakeup_granularity_ns 为例实现了参数视图隔离

第二个问题是，参数视图隔离页是如何实现的？PidNS 和参数的隔离需求都是在每一个容器中存在且唯一，因此参数视图隔离页可以直接依赖 PidNS 实现

P24

系统设计的第二块是内存管理策略，为了有效解决容器个性化定制内存管理策略时的冲突问题，我们提出了基于完全隔离思想的容器内存管理机制，。包括独立的透明大页管理和独立的虚拟内存管理机制。

针对独立的透明大页管理机制，虚拟内核需要有独立的透明大页管理参数、缺页中断和 khugepaged 线程逻辑支持。

但是，为了降低独立管理机制的复杂度，虚拟内核的缺页中断和 khugepaged 透明大页支持逻辑可以复用宿主物理内核逻辑。而仅需要对容器的缺页中断进行重定向，缺页中断处理大页时处理决策基于虚拟内核管理参数，处理逻辑基于宿主内核逻辑。

针对虚拟内存管理机制，我们设置了内核可调参数，处理的逻辑是重定向内存管理策略相关函数、复用宿主管理逻辑。

P26

系统设计的第三个方面是系统调用虚拟化。我们移除了容器对全局系统调用表的访问权限， 每个容器拥有位于内核空间不同区域的独立系统调用表，实现了隔离性和可定制性。

P27

系统设计的第四个方面是内核日志隔离。主要是修改了日志结构，

第一，是使内核日志具备能够区分不同 pid_namespace 的身份标识

第二，修改内核日志读取函数， 增加 vkernel 进程识别，根据权限计算日志大小并复制到用户空间不同区域的独立系统调用表

P28

最后，我们对系统的性能做了全面的评估。

第一个，展示了容器日志隔离，限制了容器对内核日志的访问， 成功实现主机内核日志的隔离化

第二个，为容器定制 vkernel 内核模块， 每个模块为容器独享，容器间互相隔离

（描述一下图片）

启动效率方面，

从容器启动效率来看以相对比例启动时间为测量指标，发现基于 vkernel 容器不会对启动时间造成明显增加。

P30

CPU 调度虚拟化的有效性方面，

为了验证 CPU 调度系统虚拟化方案中 FIFO 策略的有效性，将 CPU 调度系统虚拟化中局部调度器上的 FIFO 策略与原生环境下 RT 调度器提供的 SCHED_FIFO 策略进行对比，其中选择特权容器和具有 CAP_SYS_NICE 权限的容器进行对比，结果表明，

P31

然后，在单容器运行环境中配置不同的策略进行测试。具体方式是在 CPU 调度虚拟化方案实施的新内核中运行一个普通容器，分别通过将参数“cpu.real_policy”设置为 0 和 1 来表示局部调度器上的调度策略配置为 SCHED_NORMAL 和 FIFO。

结果表明，整体运行测试结果与在宿主机中基本类似， 与原本使用单一的 CFS 调度策略相比， 在实体发送信息量较大的长任务场景中， 性能有了显著提升。

P32

总的来说我们项目更好的实现了容器的安全性和隔离性，同时又不会给原生容器添加过多额外的开销。创新性体现在以下几个方面：

1. vKernel 的系统框架设计
2. 实现了一种容器的 CPU 调度策略，可以支持每个容器独立配置参数，同时又兼顾了效率。
3. 实现了全局资源内核日志的隔离
4. 实现了可定制化的容器内存管理策略

以上就是我们项目的介绍，感谢各位评委老师的聆听!