微虚拟机性能优化关键技术研究

(申请清华大学工学博士学位论文)

培 养 单 位： 计算机科学与技术系

学　　科： 计算机科学与技术

研　究　生： 贾 越 凯

指 导 教 师： 陈　　渝　副教授

二〇二四年五月

Research on Key Technologies of Performance Optimization for Micro-Virtual Machines

Dissertation submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in

Computer Science and Technology

by

Jia Yuekai

Dissertation Supervisor: Associate Professor Chen Yu

May, 2024

学位论文公开评阅人和答辩委员会名单

公开评阅人名单

陈向群	教授	北京大学
陈文光	教授	清华大学

答辩委员会名单

主席	陈向群	教授	北京大学
委员	张福新	研究员	中国科学院计算技术研究所
	武延军	研究员	中国科学院软件研究所
	丁贵广	教授	清华大学
	任炬	副教授	清华大学
	陈渝	副教授	清华大学
秘书	向勇	副研究员	清华大学

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》及上级教育主管部门具体要求，向国家图书馆报送相应的学位论文。

本人保证遵守上述规定。

作者签名：贾越凯　　导师签名：

日　　期：2024.5.26　　日　　期：2024.5.26

摘 要

微虚拟机技术为现有通用操作系统的性能与安全优化提供了一个新的方向。其通过让应用程序实现部分操作系统的功能，而绕过原操作系统的低效实现，获得了出色的性能、灵活性与安全性。然而，现有的微虚拟机技术在调度优化、针对应用的专用化、敏感数据保护等方面存在一些不足。本文以微虚拟机及相关系统软件的性能优化为目标，主要研究内容与贡献如下：

（1）为利用微虚拟机优化通用操作系统上的任务调度性能，本文设计了 Skyloft 库操作系统，以支持应用程序自定义的调度策略。Skyloft 使用 Intel 最新硬件上的用户态中断技术，为用户态调度器提供了微秒级的抢占支持，以优化多样化负载下的尾延迟。Skyloft 通过让用户态调度器支持属于不同应用的线程切换，来实现多应用的统一调度，以提高 CPU 利用率。实验表明，Skyloft 比现有最先进的可由应用自定义调度策略的系统提高了 19% 的最大吞吐量，降低了 33% 的尾延迟。

（2）为方便针对每个应用定制微虚拟机操作系统以获得最佳性能，本文提出了一种组件化的操作系统设计方法 ArceOS，根据与操作系统设计理念的相关性进行组件划分，以降低耦合性并提升可重用性。ArceOS 基于 Rust 语言的特性设计了 API 快速路径，专门针对微虚拟机场景进行性能优化，同时也具有足够的兼容性。实验表明，ArceOS 的组件化设计能够方便构建出专用化的操作系统，API 快速路径的设计使小文件读写的性能提升了 50% 。ArceOS 同时兼容已有 C 应用程序，与 Linux 相比，使 Redis 的延迟降低了 33%。

（3）为解决现有方法在保护微虚拟机内敏感数据时的性能与灵活性不高的问题，本文提出了一种可信执行环境的设计方法 HyperEnclave，利用一个轻量的微虚拟机管理程序实现对微虚拟机的灵活隔离，来满足不同场景下的性能与安全需求。HyperEnclave 仅依赖如今广泛可用的硬件虚拟化与可信平台模块，而无需特定的安全硬件。HyperEnclave 提供了与现有可信执行环境兼容的编程模型与接口，能够支持已有应用的无修改运行。实验表明，HyperEnclave 引入的开销很小，在 SQLite 上的开销小于 5%，与现有硬件方法的性能与安全性相当。

关键词：操作系统；微虚拟机；内核旁路；单内核；可信执行环境

Abstract

Micro-virtual machines (microVMs) provide a new direction for optimizing the performance and security of existing general-purpose operating systems. By allowing applications to implement part of the OS functionality and bypassing the inefficient implementation of the original OS, microVMs have achieved excellent performance, flexibility, and security. However, existing microVMs have some shortcomings in terms of scheduling optimization, application-specific specialization, and sensitive data protection. This thesis aims to optimize the performance of microVMs and the related system software. The main research contents and contributions are as follows:

(1) In order to optimize task scheduling performance on general-purpose OSes using microVMs, this thesis designs the Skyloft library OS to support application-defined scheduling policies. Skyloft utilizes User Interrupts, a new hardware mechanism, to provide microsecond-scale preemption for the user-level scheduler and optimize the tail latency under various workloads. By enabling the user-level scheduler to support thread switching across different applications, Skyloft achieves unified multi-application scheduling and improved CPU utilization. Experiments show that Skyloft improves maximum throughput by 19% and reduces tail latency by 33% over existing state-of-the-art systems with application-defined scheduling policies.

(2) In order to facilitate customization of the microVM OS for each application to obtain the best performance, this thesis proposes a componentized OS design called ArceOS. It divides components based on their relevance to the OS design concept, reducing coupling and improving reusability. ArceOS designs an API shortcut based on the features of the Rust programming language, specifically optimized for microVM scenarios, while maintaining sufficient compatibility. Experiments show that ArceOS' componentized design enables easy construction for the specialized OS, and the API shortcut improves the performance of reading and writing small files by 50%. ArceOS is also compatible with existing C applications and reduces the latency of Redis by 33% compared to Linux.

(3) In order to address the performance and flexibility limitations of existing methods in protecting sensitive data within microVMs, this thesis proposes HyperEnclave, a trusted execution environment design. HyperEnclave utilizes a lightweight microVM monitor to achieve flexible isolation of microVMs, meeting performance and security re-

quirements in different scenarios. HyperEnclave relies only on widely available hardware virtualization and trusted platform modules, without requiring specific security hardware. HyperEnclave provides a programming model and API compatible with existing trusted execution environments and can run existing applications without modifications. Experiments show that HyperEnclave introduces only a small overhead, less than 5% on SQLite, and it provides comparable performance and security to existing hardware-based approaches.

Keywords: Operating System; Micro-Virtual Machine; Kernel Bypass; Unikernel; Trusted Execution Environment

目 录

插图和附表清单

符号和缩略语说明

ABI	应用程序二进制接口（Application Binary Interface）
API	应用程序编程接口（Application Programming Interface）
APIC	高级可编程中断控制器（Advanced Programmable Interrupt Controller）
BIOS	基本输入输出系统（Basic Input/Output System）
CFS	完全公平调度（Completely Fair Scheduler）
CPU	中央处理器（Central Processing Unit）
DMA	直接内存访问（Direct Memory Access）
DPDK	数据平面开发套件（Data Plane Development Kit）
ECALL	飞地调用（Enclave Call）
EPC	飞地页面缓存（Enclave Page Cache）
EPT	扩展页表（Extended Page Table）
FFI	外部函数接口（Foreign Function Interface）
FIFO	先进先出（First In First Out）
GPA	客户物理地址（Guest Physical Address）
HPA	主机物理地址（Host Physical Address）
IDT	中断描述符表（Interrupt Descriptor Table）
IOMMU	输入/输出内存管理单元（Input/Output Memory Management Unit）
IPC	进程间通信（Inter-Process Communication）
IPI	处理器间中断（Inter-Processor Interrupt）
KPTI	内核页表隔离（Kernel Page Table Isolation）
LTO	链接时优化（Link Time Optimization）
MMU	内存管理单元（Memory Management Unit）
MMIO	内存映射输入/输出（Memory-Mapped Input/Output）
MSR	模型特定寄存器（Model-Specific Register）
OCALL	外部调用（Outside Call）
OS	操作系统（Operating System）
PCR	平台配置寄存器（Platform Configuration Register）
POSIX	可移植操作系统接口（Portable Operating System Interface）
RAII	资源获取即初始化（Resource Acquisition Is Initialization）

RPC	远程过程调用（Remote Procedure Call）
RR	时间片轮转（Round-Robin）
RSS	接收端扩展（Receive-Side Scaling）
SDK	软件开发套件（Software Development Kit）
SGX	软件保护扩展（Software Guard eXtensions）
SLAT	二级地址翻译（Second Level Address Translation）
SLO	服务水平目标（Service Level Objective）
TCB	可信计算基（Trusted Computing Base）
TEE	可信执行环境（Trusted Execution Environment）
TLB	转译后备缓冲区（Translation Lookaside Buffer）
TLS	线程局部存储（Thread Local Storage）
TPM	可信平台模块（Trusted Platform Module）
UINTR	用户态中断（User Interrupts）
vCPU	虚拟处理器（Virtual CPU）
VFS	虚拟文件系统（Virtual File System）
VM	虚拟机（Virtual Machine）
VMX	虚拟机扩展（Virtual Machine eXtension）

第 1 章　引言

1.1　研究背景

操作系统是沟通应用程序与计算机硬件的桥梁，对应用的性能与安全性有着重要的影响。近年来，随着信息产业的飞速发展，诞生了许多新兴的计算场景，例如大模型训练、隐私计算、无服务器计算（serverless）、物联网等。同时，计算机硬件技术也随之发生了很大的变化，例如计算能力日益强大的图形处理器，以及人工智能加速器、智能网卡等新型硬件。然而，计算机操作系统的发展却相对滞后，目前使用的大多仍是几十年前开发的以 Linux、Windows 为代表的通用操作系统。这些传统的操作系统已经跟不上应用与硬件日新月异的发展，难以满足新场景下的性能与安全需求。

在性能方面，传统的操作系统以通用性为目标，希望以同一套架构或策略来应对大多数的场景。然而，这种通用性不仅让操作系统功能的实现变得复杂，还无法根据应用的特点进行针对性的优化，以致在简单的场景下反而表现不佳。例如，相关研究指出，使用 Linux 的网络栈通过万兆以太网卡发送小数据包，其延迟是网卡理论值的 4.8 倍，吞吐量仅为网卡理论值的 11%[1]。

在安全方面，由于通用操作系统需要尽可能多地考虑应用的场景与功能需求，使得代码变得十分臃肿而庞大。再加上传统的操作系统多使用不安全的 C 语言编写，使得更容易产生安全漏洞或软件缺陷。例如，Linux 内核目前已经拥有超过三千多万行代码[2]，每年都会曝出上百个安全漏洞[3]。此外，传统操作系统的宏内核（monolithic kernel）设计缺少细粒度隔离，也导致漏洞会在整个内核空间中传播，一个模块被攻破就影响整个内核。

另一方面，性能与安全也难以兼得，安全防护机制的引入必然会带来性能的开销。例如，传统的宏内核架构将应用程序与内核分别运行在不同特权级，会带来昂贵的系统调用开销。为防御熔断漏洞[4]引入的内核页表隔离（KPTI）机制[5]，更会造成高达 30% 的性能下降[6]。

为了提升通用操作系统上应用程序的性能与安全性，一种方法是对已有的操作系统进行改进。例如，引入新的系统调用接口[7-9]，或在内核的功能模块之间引入更轻量的隔离[10-12]等。然而，由于通用操作系统固有的复杂性以及模块间的紧耦合性，这种改进需要大量专家知识，消耗大量人力，而且这种改进也不够彻底，无法改变原有操作系统的架构与设计理念。此外，这种改进往往是临时性的，只能针对一种场景而不够通用，因此也不太容易合并进内核主线[13]，无法及时与主线

的更新同步。另一种方法是根据应用的需要从头开发一个新的操作系统，例如使用多内核[14]、微内核[15-17]等新的内核架构，或使用安全性更高的编程语言[18-23]等。然而，这不但需要更大的工作量，还会让已有的应用难以兼容。

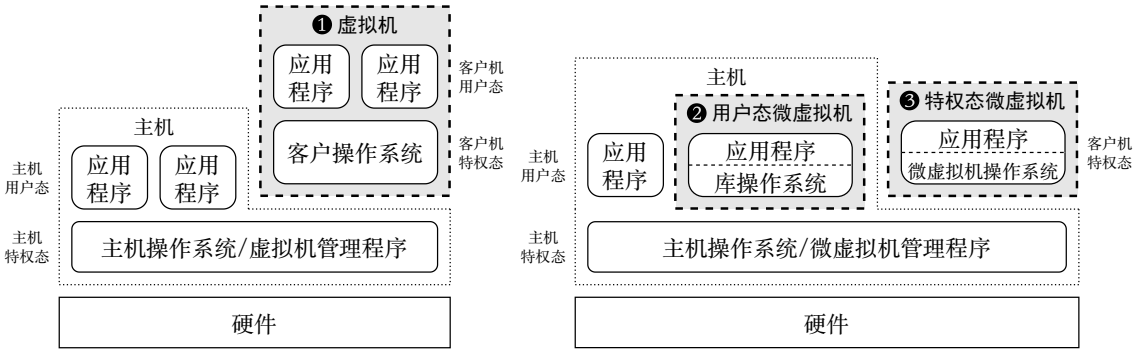
本文重点研究了除以上两种方法外的一种新方法，即**微虚拟机**技术。微虚拟机不单是指更轻量的虚拟机，还将部分操作系统的功能分解到应用程序与底层的管理程序中，并允许管理程序对微虚拟机进行灵活地管控，从而取得性能、安全性等方面的提升。微虚拟机技术在近几年越来越受到人们的关注与研究。

本文的研究内容将围绕微虚拟机技术展开。本章的剩余部分，先介绍微虚拟机技术的概念、优势与使用场景，然后给出利用微虚拟机技术优化通用操作系统时面临的一些问题与挑战，最后介绍了本文的主要工作与贡献。

1.2　微虚拟机技术概述

1.2.1　微虚拟机的概念

传统的虚拟化技术，通过将一台物理机的资源划分给多个虚拟机，来提供更好的资源利用率与更强的隔离性。传统虚拟化技术注重兼容性，一般要求不修改已有的操作系统与应用程序，就能在虚拟机中运行。因此，虚拟机需要尽量提供与物理机相似的硬件与设备访问接口，让客户操作系统与应用程序感知不到自己是运行在虚拟机中。如图 1.1(a) 所示，虚拟机中软件栈与物理机上的基本一致，包含一个完整的通用操作系统，其上可运行多个应用程序，一个具有更高特权级的虚拟机管理程序（hypervisor）负责管理所有虚拟机。



(a) 传统的虚拟机架构　　(b) 微虚拟机架构

图 1.1　虚拟机与微虚拟机的区别

实现完整的系统级虚拟化是一项重量级的工作，不仅需要支持大而全的通用操作系统，还需要在虚拟机管理程序中对硬件进行忠实地模拟。为了简化传统虚拟化架构中的软件栈，本文通过对已有工作进行归纳总结，提出了**微虚拟机**的概念，泛指一类更轻量的虚拟化技术。具体来说，微虚拟机具有以下特点：

- **轻量专用。**微虚拟机是对已有虚拟化架构的轻量化改造。它不再以运行大而全的操作系统、支持丰富的应用程序为目标，而是根据应用的需求，使用小而专的微虚拟机操作系统。因此，微虚拟机的操作系统一般只支持运行单个应用程序，并且是单地址空间单特权级的，从而可以方便地针对具体的应用进行优化。
- **功能解耦。**微虚拟机将部分操作系统的功能分解到应用程序与底层的管理程序中。例如，像 I/O 这样性能关键的功能就可以在应用程序中实现，而绕过原操作系统中的低效实现，并能减小系统调用的开销；像隔离这样安全关键的功能就在管理程序中实现。因此，应用与操作系统之间的界限变得模糊，微虚拟机中的软件既可以看做应用程序，又可以看做操作系统。
- **灵活管控。**微虚拟机轻量专用的特点，也为管理程序提供了更多的灵活性。微虚拟机的运行模式、不同微虚拟机间的隔离方式都可以被灵活配置，从而形成多种性能与安全级别供用户选择。例如，更高的性能级别可能会牺牲一定的安全性，而更高的安全级别往往伴随着更大的性能开销。这种灵活性能让用户更好地在性能与安全之间进行权衡。

微虚拟机的思想最早可以追溯到 1995 年 Engler et al. 提出的外核（Exokernel）技术[24]，不过该文只讨论了在用户态的场景。微虚拟机灵活管控的特点允许它运行在多种模式，图 1.1(b) 分别给出了运行在虚拟机（客户机特权态）与用户态的两种微虚拟机架构：

（1）特权态微虚拟机与传统的虚拟机架构更为接近，同样提供了一个机器级的执行环境，能够执行特权操作。只不过微虚拟机的操作系统更为轻量，而且与应用程序一起运行在客户机特权态，在客户机用户态不运行任何软件。这种方式为应用程序提供了足够的特权与灵活性，对底层管理程序的依赖较少。例如，应用程序可以自行管理客户机页表，处理中断或异常事件，在一些场景下性能提升明显[25]。然而，为提供机器级的执行环境，无论是软件还是硬件虚拟化都会带来一定的开销，而且硬件虚拟化在部分场景中无法使用。

（2）用户态微虚拟机在特权态微虚拟机的基础上进行了进一步地简化，提供了进程级的执行环境，整个微虚拟机作为主机操作系统的一个进程运行在用户态。此时的微虚拟机操作系统可视为是应用程序的一个依赖库，因此也被称为*库操作系统*（LibOS）。而主机操作系统同时作为微虚拟机管理程序，通过系统调用来支持库操作系统中的特权操作。该方式具有最小的硬件依赖，无需额外的虚拟化层，但对主机操作系统的依赖较多。此外，仍有一些特权功能无法在库操作系统中实现（例如缺页处理），需要以较大的开销调用主机操作系统，不适合需要频繁特权

操作的场景。

1.2.2　微虚拟机技术的优势

微虚拟机技术相比于传统的虚拟化甚至非虚拟化的场景，具有不少优势，下面列举了其中的几方面优势。

更轻量。微虚拟机的操作系统与应用程序高度相关，可以根据应用的需求进行定制，尽量只包含应用所需的功能，而不包含应用不需要或冗余的功能。因此，使用微虚拟机可以减小虚拟机镜像的大小，提升启动速度；还可以减少资源消耗，提高虚拟机的部署密度。

更灵活。微虚拟机的应用级资源管理能力，允许将主机操作系统的部分功能卸载到微虚拟机中，并由应用定义其实现方式。这使得可以针对不同应用与场景的需求，实现专用化（specialization）的操作系统。此外，操作系统的功能卸载也提供了开发与部署的灵活性，而无需修改原有的操作系统与重启机器。

更高效。由于微虚拟机单地址空间单特权级的特点，应用程序通过函数调用来管理底层资源，而无需系统调用的开销。单地址空间还允许应用程序与操作系统在编译时生成单个镜像，从而可以使用链接时优化（LTO）。微虚拟机操作系统的专用化优势，也方便针对应用的特点进行专属性能优化。此外，特权态微虚拟机允许应用程序管理更多特权资源，也有助于在特定场景下提升性能。

更安全。微虚拟机轻量化的特点，使得其操作系统具有较少的代码量，比通用操作系统小几个数量级。因此有利于减少安全漏洞，以及可被攻击者利用的代码片段。此外，特权态微虚拟机也提供了比进程更强的隔离性，同时在启动速度与资源消耗上与进程相当。

1.2.3　微虚拟机的使用场景

基于上文指出的几大优势，微虚拟机技术可以应用于不少场景中，在多方面带来优化，本节列举了几个典型的使用场景。

在云计算中，云提供商会将一台物理机器划分为多个虚拟机，提供给多租户使用。目前这些虚拟机中往往运行 Linux 这样的通用操作系统，但租户往往只会运行少部分应用，无需 Linux 的全部功能。此外，Linux 中的隔离与硬件抽象功能也会与云端虚拟机管理程序所提供的重复。在该场景下，使用微虚拟机代替传统的虚拟机，可以在保证多租户间相互隔离的情况下，大幅提升应用性能，同时也提高虚拟机的启动速度与部署密度[19,26-28]。

对于新兴的无服务器计算场景，需要能够快速启动用户提供的微服务。此时性能与安全往往难以兼得，使用容器技术具有较快的启动速度，但安全性较差；反

之，将微服务运行在虚拟机中具有较好的安全性，但其中的通用操作系统拖慢了启动速度。在该场景下，可以自然地使用微虚拟机来包装微服务，提供虚拟机级别安全性的同时，也具有良好的启动速度[29]。

在数据中心中，对应用的 I/O 性能（延迟与吞吐量）有着极致的要求，而 Linux 等通用操作系统的 I/O 栈性能不足。因此，以网络应用为例，数据中心通常使用用户态的网络栈[30]与网卡驱动[31]，来绕过内核的低效网络栈。此时，可以将这些用户态的网络技术实现到一个库操作系统中，并为应用程序提供易用的接口，使得应用既能通过这些技术获得性能提升，又不需要花费太多代价进行适配[32]。

在机密计算领域，需要一个可信基（TCB）小、安全性高的操作系统，为运行在可信执行环境中的应用程序提供支持。而微虚拟机操作系统能够方便针对应用进行定制，只包含应用必需的功能，从而减少 TCB 中的代码与攻击面[33-37]，与庞大的通用操作系统相比具有更好的安全性。

1.3　关键问题与挑战

本文希望利用微虚拟化技术，提升通用操作系统上应用程序的性能与安全性。为此，本节分别从微虚拟机对通用操作系统的性能优化、微虚拟机自身操作系统的性能优化、微虚拟机的安全防护性能这几个角度，梳理了微虚拟机技术中的三个关键问题，以及面临的挑战。

问题一：操作系统的调度功能难以卸载到微虚拟机中。由于微虚拟机功能解耦的特点，可以在应用程序，也就是用户态中实现部分需要特权的内核级功能。因此，如何在用户态高效实现这些功能就成了一个关键问题。目前已有不少工作，将网络[1,30-31,38-39]、存储[39-40]、文件系统[41-43]等多个内核子系统卸载到用户态微虚拟机中，并取得了不错的效果。对于同样性能关键的线程管理与调度功能，这种卸载将大大减少线程创建、同步等操作的开销，也能方便进行调度策略的定制以提升性能。然而，调度器的卸载面临抢占与多应用支持这两个挑战，因为它们依赖中断处理、CPU 状态切换等特权操作。这使得目前已有的利用微虚拟机优化的调度系统存在不少局限性，要么不支持抢占或抢占开销太大，导致短任务容易被长任务阻塞而降低性能[1,44]；要么不支持多应用，应用需要独占 CPU 而造成资源利用率不高[45]；要么为应用提供的灵活性不足，只能支持部分调度策略[46]。

问题二：为每个应用程序定制微虚拟机操作系统的代价大。微虚拟机的轻量专用特点，也是提升性能的一个重要途径。然而，这种专用化建立在能为每个应用都量身定制一个操作系统的基础上，需要大量的专家经验与开发成本。目前已有的一些方法以微库的形式提供一系列操作系统的功能模块，并通过对微库的组合

来形成针对应用需求的专用操作系统[28,47-48]。然而，这些系统仍使用 C 语言，使得微库在易用性与可重用性等方面存在不足。此外，现有系统为了兼容性，没有实现针对应用与内核间接口的专用化，大多仍采用传统的 POSIX 接口[27-28,49-50]，这在单地址空间的微虚拟机场景下存在许多冗余的调用路径，无法获得最佳性能。

问题三：微虚拟机敏感数据保护方法的性能与灵活性不高。微虚拟机的运行依赖底层的主机操作系统或虚拟机管理程序，但它们仍然是庞大的通用操作系统，存在众多潜在的安全问题。此外，管理程序与微虚拟机间的单向隔离，导致恶意的或被攻陷的管理程序可以随意访问或篡改微虚拟机内的敏感数据，这在具有更高安全要求的场景下不可接受。可信执行环境（TEE）技术是解决这一问题的常用方法，然而，现有的基于硬件的 TEE 技术[51-54]需要依赖特定的带有安全功能的 CPU，只在部分场景下可用，不具有通用性。而基于软件实现的 TEE 技术[55-57]则具有较大的开销，且不兼容已有 TEE 应用的生态。此外，现有的 TEE 技术也仅支持固定的隔离模式，无法同时支持用户态与特权态的微虚拟机，难以满足不同场景下的应用需求。

1.4　本文主要贡献

本文针对第 1.3 节指出的微虚拟机技术的三个关键问题，在相应的软件层开展研究。图 1.2 展示了本文研究内容所在的软件层次。系统的最底层软件是一个微虚拟机管理程序，支撑了上层通用操作系统以及特权态微虚拟机的运行①，通用操作系统之上还会运行普通应用程序与用户态微虚拟机。用户态微虚拟机可以绕过通用操作系统中性能关键的功能，而在库操作系统中进行实现。在特权态微虚拟机中也运行针对应用定制的专用操作系统。微虚拟机管理程序负责为上层的各软件提供安全隔离。本文的主要贡献如下：

（1）针对问题一，提出一种基于用户态中断的库操作系统设计，通过在用户态实现低开销抢占与多应用切换，优化通用操作系统上的任务调度性能。

为了方便又高效地支持应用程序自定义的任务调度策略，需要将内核的线程管理、调度、网络等功能都卸载到用户态，本文因此设计了 Skyloft 库操作系统。为解决将调度功能卸载到微虚拟机所面临的抢占与多应用支持这两大挑战，Skyloft 利用 Intel 最新硬件上的用户态中断技术[58]，在用户态实现了低开销的处理器间中断与时钟中断处理，从而为用户态调度器提供了微秒级的抢占支持，降低多样化负载下的任务处理延迟。同时，Skyloft 利用一个内核模块，允许用户态调度器切换属于不同应用的线程，从而实现多应用的统一调度，提高 CPU 利用率。Skyloft

① 该通用操作系统也可不依赖微虚拟机管理程序，直接运行在裸机上。

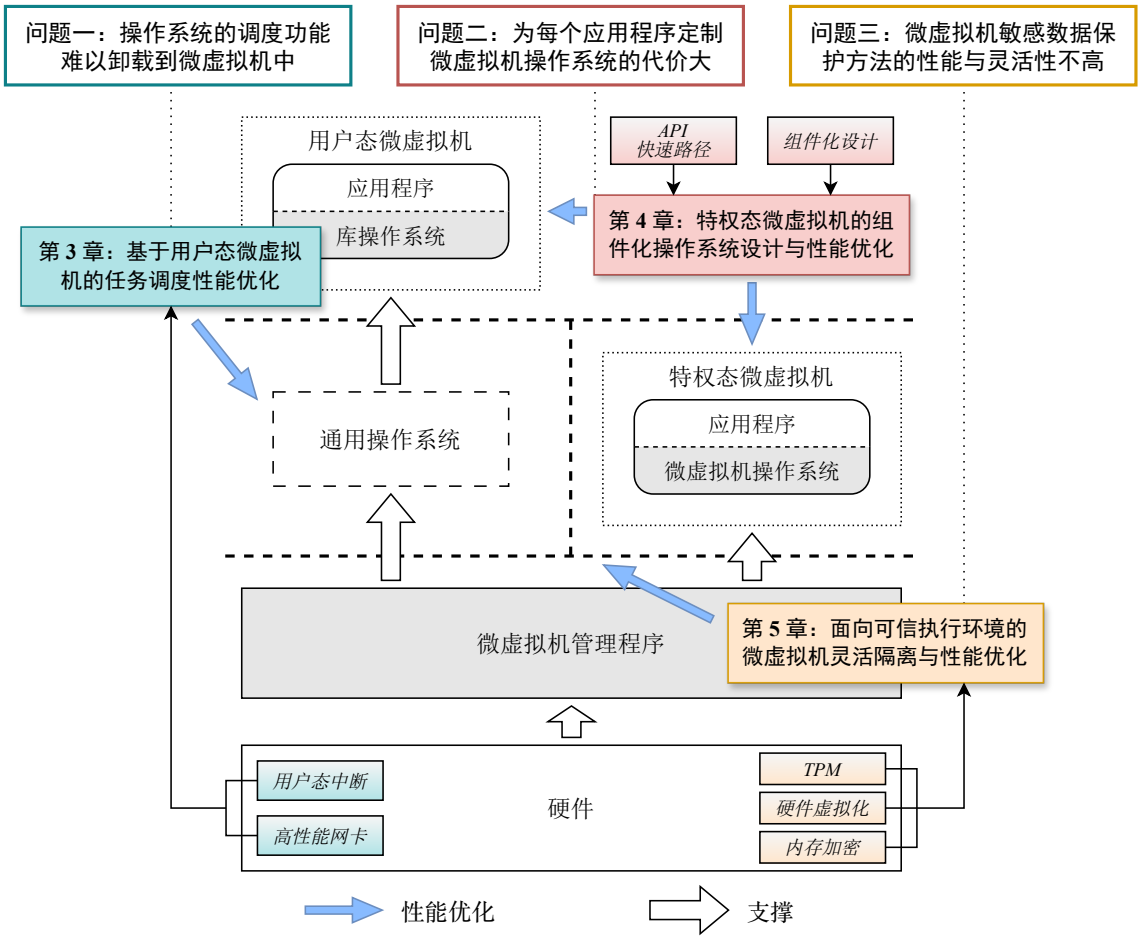


图 1.2　本文研究内容与主要贡献

也集成了基于内核旁路的用户态网络技术，使得网络栈能与调度器相互配合。实验结果表明，相比现有的可由应用自定义调度策略的系统，Skyloft 能将延迟敏感应用的最大吞吐量提高 19%。

（2）针对问题二，提出一种微虚拟机操作系统的组件化设计方法，通过实现针对应用的组件定制与 API 快速路径，优化特权态微虚拟机的性能。

高性能的微虚拟机操作系统需要能够方便地针对应用程序的需求进行专用化定制。为此，本文提出了一种组件化的操作系统设计方法 ArceOS，通过将组件按照与操作系统设计理念的相关性分为*操作系统相关*与*操作系统无关*两类，来降低耦合性，提升可重用性。ArceOS 还为组件提供灵活的定制功能，可以通过应用指定的配置文件，构建出 Unikernel、宏内核、虚拟机管理程序等多种形态的操作系统。此外，本文还提出了一种基于 Rust 语言特性的 *API 快速路径*设计，专门针对微虚拟机场景进行了性能优化，同时也具有足够的兼容性。实验结果表明，ArceOS 的组件化设计能够方便实现操作系统功能的定制。ArceOS 的 API 快速路径设计在小文件读写上比使用 POSIX 接口快了 50% 。ArceOS 同时兼容已有 C 应用程序，与

Linux 相比，使 Redis 的延迟降低了 33%。ArceOS 组件化操作系统目前已开源①，并被多家单位用于科研或二次开发。

（3）针对问题三，提出一种基于硬件虚拟化的可信执行环境设计，通过将微虚拟机运行在多种模式，优化微虚拟机安全隔离机制的灵活性与性能。

为解决已有可信执行环境技术通用性不足、性能开销大、隔离模式固定等问题，本文提出了一种可信执行环境的设计方法 HyperEnclave。HyperEnclave 仅依赖如今广泛可用的*硬件虚拟化扩展*[59]与*可信平台模块*[60]，而无需特定的安全硬件，方便在多种平台上实现，并以较小的开销解决了在没有安全硬件支持下的一些安全问题。HyperEnclave 利用轻量级的微虚拟机管理程序，提供*灵活的隔离模式*，同时支持用户态与特权态的微虚拟机，来满足应用在不同场景下对性能与安全的需求。HyperEnclave 为不同形态的微虚拟机提供了统一的、与现有 TEE 技术兼容的编程模型与接口，能够支持已有应用的无修改运行。实验结果表明，HyperEnclave 引入的开销很小（如 SQLite 的开销小于 5%），与现有硬件 TEE 方法的性能与安全性相当。HyperEnclave 灵活的隔离模式也在一些场景下取得了更好的性能。HyperEnclave 目前已开源②，并在蚂蚁集团落地商用。

1.5　本文组织结构

本文的其余章节组织如下：第 2 章介绍了微虚拟机技术的相关研究工作，包括性能优化与安全防护等方面。第 3 章介绍了基于用户态微虚拟机技术的 Skyloft 系统，用于优化通用操作系统的调度性能；第 4 章介绍了一种组件化的操作系统设计方法，用于优化特权态微虚拟机的性能。第 5 章介绍了面向可信执行环境设计的 HyperEnclave 系统，用于优化微虚拟机安全隔离机制的灵活性与性能。最后，第 6 章总结了全文，并展望了未来的研究方向。

① https://github.com/rcore-os/arceos
② https://github.com/HyperEnclave/hyperenclave

第 2 章　相关工作

本文研究的微虚拟机技术，是优化操作系统性能的一种重要方法。在本章，先介绍了目前已有的一些微虚拟机技术的相关研究，重点是如何设计在微虚拟机内运行的操作系统。然后，本章介绍了与微虚拟机相关的，在操作系统性能优化方面的一些其他技术。此外，由于微虚拟机让应用具有了更多资源管理的能力，对微虚拟机的安全防护也是不可忽视的，而且这些安全防护技术的开销也关系到系统整体的性能。因此，本章在最后介绍了可用于微虚拟机上的一些安全防护技术，以及它们各自的性能开销。

2.1　微虚拟机操作系统研究

微虚拟机具有功能解耦、轻量专用、灵活管控这三大特点。在目前的研究工作中，已有不少符合这三个特点的操作系统被提出，均可被称为“微虚拟机操作系统”①。根据它们的运行模式与使用场景，这里将它们分为库操作系统、Unikernel、数据平面操作系统三类分别进行介绍，如图 2.1 所示。

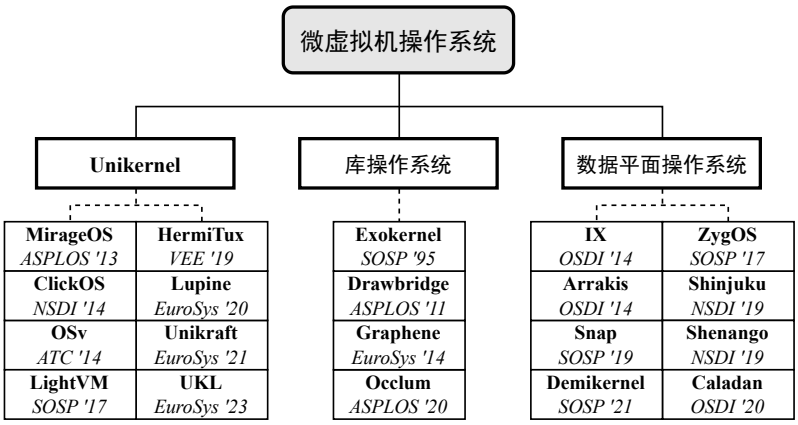


图 2.1　微虚拟机操作系统相关研究

2.1.1　库操作系统

库操作系统的概念最早在 Exokernel[24] 中提出，Exokernel 基于资源管理与保护分离的思想，通过一个小的内核将所有硬件资源暴露给用户态的库操作系统，内核只实现资源安全保护等少量工作。在库操作系统中实现了更适合应用需求的资源抽象与管理，从而提供了更好的性能与灵活性。

① 像 FreeRTOS[61] 这样的实时操作系统，虽然也符合前两个特点，不过它们直接运行在硬件的最高特权级，无底层管理程序，因此不算是微虚拟机操作系统，不在本文的讨论范围内。

由于库操作系统运行在用户态，具有更少的特权，近年来被更多地用于安全领域。Drawbridge[62] 将商用的 Windows 操作系统改造为一个库操作系统，以实现轻量级沙箱、应用跨机迁移、系统快速演化、兼容旧应用等功能。类似地，LKL[63] 将 Linux 的部分子系统以库的形式提供给应用程序，方便应用重用 Linux 的代码。gVisor[64] 通过在库操作系统中实现大部分系统服务，并减少对主机操作系统的访问，以减少对内核的攻击面。Graphene[65] 通过分叉（fork）主机操作系统上的进程，在库操作系统内实现了多进程支持。Occlum[36] 面向机密计算场景，通过库操作系统为 SGX 飞地内的应用程序提供了更好的兼容性，并利用软件故障隔离[66]（SFI）提供多进程间的保护。

2.1.2　Unikernel

Exokernel 与库操作系统的思想，在今天通常也基于虚拟化技术来实现，此时应用程序与库操作系统都运行在虚拟机的客户机特权态，具有一定的特权访问能力（即特权态微虚拟机），这时的库操作系统也被称为 Unikernel（单内核）。

为了使 Unikernel 尽量轻量，一些工作从头开始进行实现。MirageOS[19] 使用 OCaml[67] 语言统一编写 Unikernel 以及应用程序，在取得微虚拟机性能优势的同时，也利用高级语言的特性提供了安全性。ClickOS[26] 使用 Unikernel 托管与硬件高度相关的网络中间件，既便于管理，又能保证性能与启动速度。LightVM[29] 通过 Xen[68] 虚拟机管理程序与 Unikernel 的协同优化，将微虚拟机的启动速度提升至进程级别。Unikraft[28] 以微库的形式提供一系列操作系统的功能模块，并通过对微库的组合来形成针对应用需求的专用 Unikernel。从头开发的 Unikernel 也面临应用兼容性的问题，HermiTux[49] 利用二进制重写与共享库替换等技术实现了对 Linux 旧应用的二进制兼容。

另一些 Unikernel 重用了其他成熟操作系统的代码。Rump[69] 引入了“任意内核”（anykernel）的概念，可让应用程序重用 NetBSD 的部分代码（如设备驱动）。OSv[27] 提供了一个更适合云计算场景的 Unikernel，并针对自旋锁与网络 API 进行了优化，其网络栈也来自于 FreeBSD。

还有一些工作尝试将现有的通用操作系统改造为 Unikernel，在取得性能提升的同时也保证兼容性。Kernel Mode Linux[70]（KML）通过一个内核补丁让 Linux 上的应用程序运行在内核态来避免上下文切换，不过没有针对应用的特点进行其他优化。Lupine[50] 使用 Linux 的配置系统裁剪掉应用程序不需要的功能，并利用 KML 消除系统调用开销，将 Linux 改造为了一个 Unikernel，取得了性能、启动速度等多方面的提升。UKL[71] 认为光是消除系统调用的硬件开销还不够，通过一系列方法缩短系统调用的执行路径，绕过许多在 Unikernel 中不必要的工作，进一步

提升了性能。

2.1.3　数据平面操作系统

数据平面操作系统（dataplane OS）致力于利用微虚拟机的性能与专用化优势，来为 I/O 密集型应用提供极致的性能。这些系统既有运行在特权态的，也有运行在用户态的。它们的核心思想是将数据平面与控制平面分离，让高性能的微虚拟机作为数据平面，处理 I/O 数据；主机操作系统作为控制平面，用于对资源进行管理与保护。

一些早期的数据平面操作系统基于 Dune[25] 来实现。Dune 利用硬件虚拟化，将应用程序运行在客户机模式，给予应用程序访问特权资源的能力，但应用程序仍通过系统调用（被替换为超级调用）访问主机操作系统。IX[1] 利用 Dune 让应用程序能直接管理高性能网卡，并通过批处理、零拷贝、多队列网卡与接收端扩展[72]（receive-side scaling, RSS）等技术，大幅提升了网络应用的性能与多核可扩展性。Arrakis[39] 利用单根 I/O 虚拟化[73]（SR-IOV）技术，构造多个虚拟的 I/O 设备并允许应用程序直接访问，除了网络外也对存储栈进行了优化。Demikernel[32] 整合了数据平面开发套件（DPDK）、远程直接内存访问（RDMA）等目前最先进的内核旁路技术，并为应用程序提供了一套统一的 API，使用户可以方便地使用这些技术得到 I/O 性能的提升。

还有一些工作在数据平面操作系统的基础上，专门针对任务调度进行了优化，以降低网络请求的排队延迟。ZygOS[44] 针对 IX 进行了负载均衡的优化，通过消除队头阻塞进一步提升了网络应用的性能。Shinjuku[45] 在 IX 的基础上，利用虚拟机 Posted-Interrupt 技术支持周期低至 5μs 的抢占，提升了重尾分布负载下网络应用的性能。Concord[74] 利用编译器插桩技术实现了近似的抢占功能，比中断方法具有更低的开销，进一步降低了尾延迟，此外 Concord 也允许让分发器能处理任务，提升了吞吐量。

然而以上这些数据平面操作系统，通常需要独占部分核心来运行一个应用，即使在负载较低时也占用了计算资源，使得资源利用率不高。一些工作希望能在保证延迟敏感应用性能的同时，又尽量提高后台尽力而为应用的吞吐量，以实现高 CPU 能效（efficiency）。Shenango[75] 基于应用间重新分配核心的方法实现了以上目标，它以 5μs 的周期检测应用的队列拥塞情况，相应地给应用增加或减少核心。Caladan[76] 将 Shenango 中的拥塞检测指标扩展为内存带宽、缓存占用等信号，进一步避免了后台内存密集型应用对延迟敏感应用的干扰。

2.2　微虚拟机相关的性能优化技术

微虚拟机技术通过避免上下文切换、绕过内核、轻量化等方法提升了性能。这些方法也是常用的性能优化方法，被广泛用于其他系统，如图 2.2 所示。本节将对这些方法进行介绍。

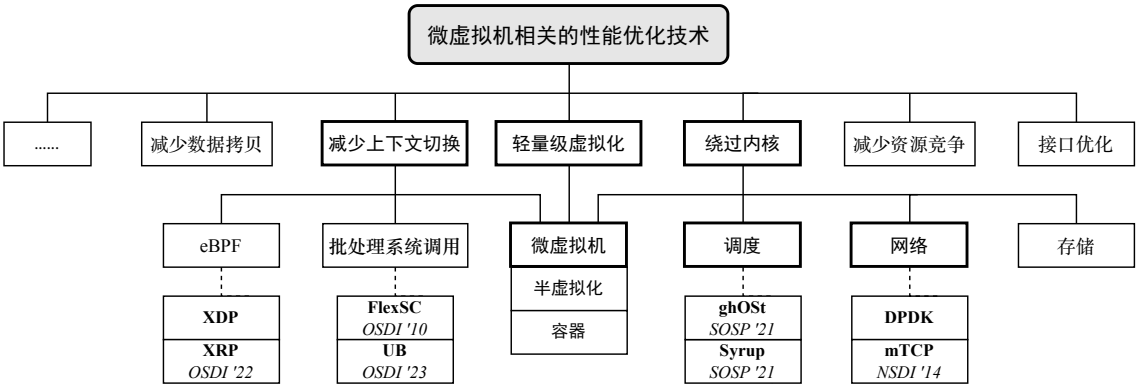


图 2.2　微虚拟机相关的性能优化技术

2.2.1　减少上下文切换

通用操作系统一般采用宏内核的架构，这使得应用程序与内核相互调用时需要进行昂贵的上下文切换。上下文切换的开销包含特权模式切换、CPU 状态的保存与恢复、地址空间切换（开启 KPTI[5]）等直接开销，还有缓存、TLB 缺失等因素导致的间接开销[7]，可达上千个时钟周期。当应用程序需要进行频繁系统调用时（如 I/O 密集型应用），就会显著影响性能。微虚拟机正是通过将应用程序与内核运行在同一地址空间来避免上下文切换，实现性能的提升。本节介绍其他一些减少上下文切换开销的方法。

批处理系统调用。一些工作采用了批处理的思想，通过一次上下文切换，将连续多个系统调用在内核中同时进行处理，来平摊总的时间开销。FlexSC[7] 设计了一套异步的系统调用接口，用户通过一块与内核共享的内存提交系统调用请求，内核会使用专门的线程进行处理。Linux 近几年引入的 io_uring 机制[9] 也采用了类似的思想。然而，这些方法要求应用使用异步的编程模型，对应用的修改较大。Privbox[77] 将应用中系统调用密集的事件循环放入内核态运行，并通过基于 SFI 的沙箱来保证安全性，虽然不改变应用的编程模型，但仍需进行少量修改。Userspace bypass[78] 无需修改应用，可在运行时决定是否需要将一段系统调用密集的应用代码片段移入内核，不过这也导致运行时的开销较大。

eBPF。扩展伯克利包过滤器[79]（extended Berkeley Packet Filter, eBPF）允许在内核的指定位置运行一段用户提供的代码，既满足了一定的灵活性，又无需频繁进出内核带来上下文切换的开销。例如，eBPF 已被用于优化网络[80]、存储[81]、

调度[46]、同步原语[82]等多个内核子系统。但是，eBPF 的使用场景有限，例如只能在内核的一些固定位置运行，而且要求用户代码经过验证，不能有循环、浮点等操作，也有运行时间的限制。

2.2.2　绕过内核

由于传统操作系统的通用性与复杂性，一些内核子系统为通用场景而设计，在专用场景下表现不佳。微虚拟机正是利用了绕过内核的思想，在应用程序中实现部分内核的功能，使得可以进行针对性的优化。本节介绍的是另一些绕过内核的方法，也被称为内核旁路（kernel-bypass）技术，这在内核的各个子系统上均有不少研究，

网络。数据平面开发套件[31]（DPDK）通过将网卡的设备地址空间映射到用户空间，使得可以在用户态高效实现网卡驱动。DPDK 还使用基于轮询的驱动模型（Poll Mode Driver, PMD），避免了中断处理的开销。不过 DPDK 只提供了收发数据包的接口，而不提供完整的网络协议栈。mTCP[30] 在 DPDK 之上实现了用户态的 TCP 协议栈，并专门针对多核可扩展性进行了优化。

存储。与 DPDK 类似，SPDK[40] 提供了用户态的高性能 NVMe 驱动，并为应用程序提供块设备的抽象。此外，也有一些在用户态实现的文件系统[41-42]。存储子系统的优化不是本文关注的重点，这里不作过多介绍。

调度。最近一些工作尝试绕过内核的默认调度策略（如 CFS），通过让用户自己指定调度策略，以针对应用的特点进行优化。ghOSt[83] 将内核中线程的状态变化通知给一个用户态的代理，在代理程序中根据用户自定义的策略进行调度决策，并通过系统调用交给内核去执行，但这也使得用户态代理需要与内核进行频繁通信，开销较大。Syrup[84] 将调度抽象为一个匹配问题，可在智能网卡、网络协议栈、线程三个层次配置自定义的调度策略。

调度子系统需要与网络等子系统配合才能更好地发挥其优势，如本章第 2.1.3 节讨论的数据平面操作系统。此外，现有的绕过内核的调度方法并没有很好地解决抢占与多应用支持这两个问题，本文将在第 3 章详细分析这些问题，并给出本文的用户态调用器设计。

2.2.3　轻量级虚拟化

从虚拟机的角度看，微虚拟机是对虚拟化软件栈的一种简化。本节介绍其他一些通过简化虚拟化软件栈，优化虚拟机内应用性能的方法。

半虚拟化。传统的虚拟化技术需要由虚拟机管理程序实现对硬件的忠实模拟，以支持运行无修改的客户机操作系统。半虚拟化（paravirtualization）技术通过使

用一套简化的软硬件接口，让客户操作系统与虚拟机管理程序协同来完成某项功能，以取得更高的性能。例如 Xen[68] 提供了半虚拟化的内存管理，Virtio[85] 提供了半虚拟化的设备驱动。半虚拟化与微虚拟机都通过修改客户机操作系统，并与虚拟机管理程序协同的方式来提高虚拟化的性能，但半虚拟化对原操作系统的修改很小，而微虚拟机操作系统采用了极简化的设计，改动程度更大，性能提升的空间也更大。

容器。容器技术利用命名空间与 cgroup 机制[86]，用软件方法实现了对内核资源的隔离，以提供一种更轻量的操作系统级的虚拟化，例如 Docker[87]、LXC[88] 等。与微虚拟机类似，容器也具有比传统虚拟化更快的启动速度与更低的资源占用。但是，容器中不提供操作系统功能的实现，完全依赖托管它的主机操作系统，这可能成为性能与安全瓶颈。而微虚拟机可以自己实现部分操作系统的功能，绕过原系统的低效实现以提升性能。同时，这也使得微虚拟机与主机操作系统间的接口也更少，比容器具有更小的攻击面。运行在客户机模式的特权态微虚拟机，更是在隔离性与性能上都优于容器[29]。

2.2.4　其他性能优化技术

除了以上介绍的与微虚拟机相关的性能优化技术外，对操作系统以及应用程序的性能优化方向还有很多，本节将简要介绍其中的几种。

减少资源竞争。多核系统中，由于共享资源的存在，往往会出现资源竞争的问题，从而影响性能。Corey[89] 将操作系统的所有资源默认为每核心私有的，而让应用程序来显式指定需要共享哪些资源，从而减少了资源竞争，提升了多核可扩展性。多内核架构[14]（multikernel）通过在每个核心上运行独立的内核，完全消除了内核中的共享，只靠消息传递进行内核间的通信。Clements et al. 定义了接口的可交换（commutative）规则[90]，一对可交换的接口在多核同时调用时不会发生冲突。基于可交换规则实现的 sv6 操作系统[90] 与 ScaleFS 文件系统[91] 避免了多核的竞争，具有良好的多核可扩展性。

接口优化。应用程序需要通过一组明确定义的接口才能调用内核提供的功能，而不能随意地调用内核函数。因此，一次功能调用往往涉及参数的多次转换，带来不少的开销。MegaPipe[8] 提供了每核心、支持批处理的通道（channel）用于进行应用与内核间的网络请求，以代替内核原有的基于文件描述符的套接字（socket）抽象，而不用经过繁琐的 VFS 层，提升了网络应用的性能与多核可扩展性。本文第 4 章也针对微虚拟机的特点进行了接口优化。

2.3　微虚拟机相关的安全防护技术

微虚拟机一般不对其内部运行的应用程序与内核加以隔离，而通过底层管理程序提供安全保障，即强化微虚拟机间隔离，弱化微虚拟机内隔离。具体地，管理程序需要保证微虚拟机不能访问管理程序（与其上运行的普通应用）的内存，不同的微虚拟机之间也不能相互访问。对于用户态微虚拟机，这是通过页表隔离实现的；对于特权态微虚拟机，这是通过硬件虚拟化提供的嵌套页表实现的。还有一些场景具有更高的安全级别，不允许管理程序访问微虚拟机的内存，就需要使用更强的安全防护技术。此外，也可以在微虚拟机内实施一些轻量级的隔离，在不对微虚拟机性能造成太大影响的情况下，提供一定程度的安全防护，例如在微虚拟机内支持多进程的应用。

不同的安全防护机制具有不同的开销，从而影响微虚拟机整体的性能。因此，本节将介绍一些与微虚拟机相关的安全防护技术，包括软件隔离、硬件隔离、可信执行环境等，并分析它们各自的性能开销。图 2.3 给出了一些具有代表性的安全防护工作。

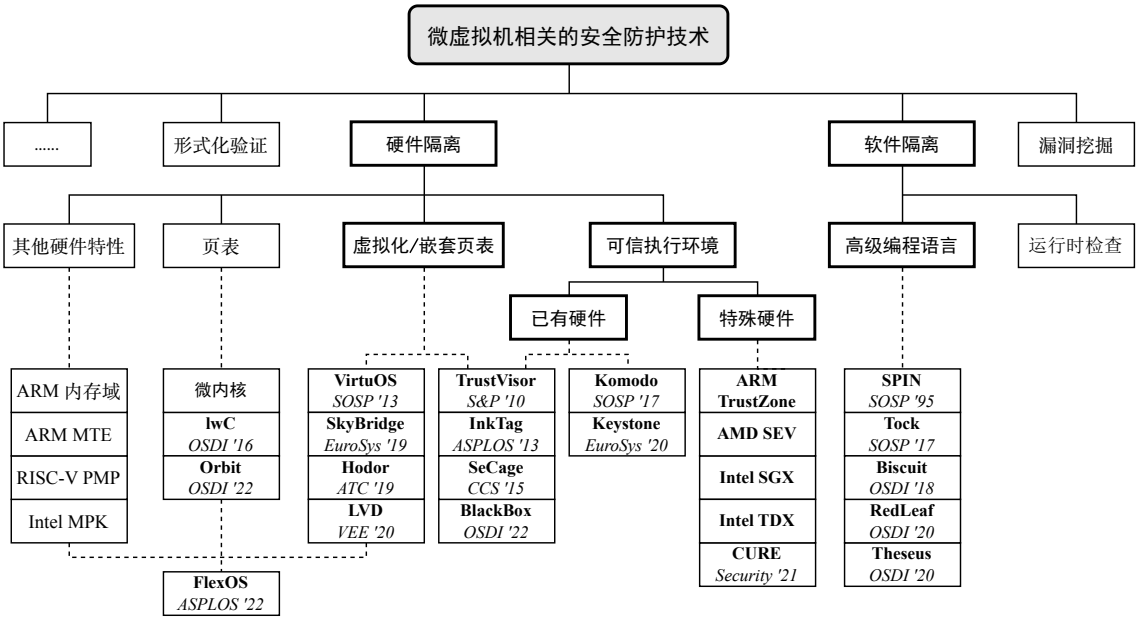


图 2.3　微虚拟机相关的安全防护技术

2.3.1　软件隔离

已有的系统软件大多使用 C 语言编写，由于 C 语言不提供类型与内存安全机制，可以通过指针随意访问内存，因此也导致了大量与内存相关的安全问题。为此，一些方法利用运行时或编译时的安全检查，阻止内存越界访问等非法行为，来实现隔离的效果。软件隔离的方法非常适合微虚拟机场景，能为处于同一地址空间的各模块提供一定程度的安全防护。

运行时安全检查。此类方法利用编译器插桩等技术，在程序的特定位置插入检查代码，使得能在运行时阻止程序的非法行为。软件故障隔离[66]（Software Fault Isolation, SFI）通过对内存访问的地址进行检查，来限制程序的内存访问范围。控制流完整性[92]（Control Flow Integrity, CFI）通过对间接跳转的目标进行检查，来防止攻击者劫持程序的控制流。在 Linux 内核中，也存在 KASAN[93] 这样的工具，用于动态检测内存访问越界、use-after-free 等缺陷。这些方法对已有软件的支持较好，无需进行修改，不过问题是运行时的开销较大。

高级编程语言。还有一些方法使用类型或内存安全的高级语言，通过强制让程序员写出符合安全规范的代码，希望从源头上消除软件的安全问题。

在一些早期的系统中，SPIN[94] 使用类型安全的 Modula-3[95] 语言编写，以避免进行随意访问内存，并支持模块的动态装载以提供一定的扩展性。Singlularity[18] 使用内存安全的 Sing# 语言（C# 的变种）编写，提出了“软件隔离的进程”的概念以代替硬件隔离。不过以上这两种语言都已经很少被使用。MirageOS[19] 采用 OCaml[67] 语言编写，为 Unikernel 提供了类型安全。最近的研究尝试还使用 Go[96] 语言编写操作系统，如 Cutler et al. 提出的 Biscuit[20]，并充分利用了闭包、通道、接口、垃圾回收等 Go 的高级语言特性。

近年来，Rust[97] 语言受到了人们的广泛关注。由于其具有类型、内存、并发等诸多安全特性，以及不输于 C 的性能，同时也具有对底层代码的控制能力，非常适合用于编写操作系统。Tock[98] 在没有硬件保护机制的嵌入式场景中，利用 Rust 实现了故障隔离与内存保护。RedLeaf[23] 完全利用 Rust 提供的类型与内存安全特性，用于操作系统功能单元的细粒度隔离，并以此实现了设备驱动的端到端零拷贝与故障隔离。Theseus[22] 提出了一种“语内”（intralingual）设计方法，最大程度利用编译器检查代码的安全性与正确性，并通过最小化操作系统功能单元间的状态溢出（state spill），实现了在线升级与故障恢复。Linux 内核从版本 6.1 起也支持使用 Rust 语言编写驱动程序[99]。本文在第 4 章也利用了 Rust 实现微虚拟机操作系统，并在组件化设计、接口快速路径等方面进行了优化。

然而，使用高级语言也无法提供绝对的安全，例如 Rust 语言无法避免使用 `unsafe` 代码块来绕过编译器的安全检查，进行裸指针访问或调用汇编代码等底层操作，此时安全性就与 C 语言一样了。而且使用高级语言需要重新编写应用，代价过大。

2.3.2　硬件隔离

利用硬件机制限制软件的可访问内存范围，可避免运行时检查的开销，并且难以被软件方法绕过，也是一种常用的安全防护技术。不过由于硬件隔离域的切

换开销较大，一般被用于实现微虚拟机间的隔离。本节列举了几种使用不同硬件机制的隔离方法。

页表隔离。页表用于将虚拟地址映射到物理地址，不同的页表代表了不同的地址空间，通过让两个地址空间的物理地址范围不重叠即可以实现隔离，是最常见的隔离机制。页表通常被用于进程（或用户态微虚拟机）间的隔离，例如微内核[15-17]将内核的功能模块以用户态进程的形式提供，通过页表隔离防止一个模块被攻破后影响到另一个模块，以提高系统的安全性与健壮性。也有一些方法在一个进程中维护了多个页表，以实现进程内功能模块的隔离。lwC[100] 可在进程中创建一个隔离的轻量上下文，用于保护进程内的敏感数据。Orbit[101] 利用页表实现了一种新的地址空间抽象来运行辅助任务，同时满足对辅助任务的强隔离性与可观测性要求。

然而，页表隔离的主要问题在于切换的开销较大，因为切换页表后会导致 TLB 表项的大量失效，当应用需要频繁进行 IPC 等跨地址空间通信时，性能会受到显著影响。

嵌套页表隔离。页表隔离只能作用于用户态，特权态的软件可以随意修改页表使得隔离失效。为此，基于硬件虚拟化的嵌套页表（nested page table）提供了更强的隔离性，例如 Intel 虚拟化扩展中的 EPT[102]。嵌套页表也被称为二级地址转换（SLAT）机制，一般用于实现虚拟机（或特权态微虚拟机）之间的隔离。客户机软件使用的*客户虚拟地址*（GVA），先通过客户操作系统维护的客户机页表转换为*客户物理地址*（GPA），再通过嵌套页表转换为*主机物理地址*（HPA）。

一些工作利用一个轻量的虚拟机管理程序维护嵌套页表，使得能够隔离具有特权的操作系统模块。例如 VirtuOS[103] 利用 Xen hypervisor 来隔离 Linux 中的一些模块，使内核具有一定的容错能力。LXD[10] 也利用嵌套页表来隔离 Linux 的设备驱动。

近年来，由于 Intel 在其虚拟化扩展中引入了 VMFUNC 机制[104]，使得可以在客户机用户态快速切换嵌套页表（甚至比切换普通页表还快[105-106]），促进了一系列基于嵌套页表隔离的工作的研究。SkyBridge[105] 利用嵌套页表隔离微内核的进程，并使用 VMFUNC 在用户态快速切换进程地址空间，大大减小了 IPC 的开销。LVD[11] 在 LXD 的基础上利用 VMFUNC 加速了跨域调用。

虽然使用嵌套页表可以利用 VMFUNC 减少切换开销，不过二级地址转换还存在一定的运行时开销，即 TLB 缺失时的代价较大。在客户机页表与嵌套页表均为 4 级时，最坏情况下会产生 25 次内存访问[107]。

轻量级硬件隔离机制。上述两种常用的硬件隔离机制能提供以页为粒度的强

大隔离，但主要问题是性能开销较大。近几年涌现出不少轻量的硬件隔离机制，提供了快速的隔离域切换。例如 Intel MPK[108]、ARMv7 内存域[109]、ARMv8.5 MTE[110]、RISC-V PMP[111]、新的硬件体系结构 CHERI[112] 等，这里不一一赘述。

快速的隔离域切换也使得这些机制适合微虚拟机内的细粒度模块隔离，例如使用 Intel MPK 对 Unikernel 的隔离[113-114]。FlexOS[115] 尝试用一套统一的接口，在一个 Unikernel 中同时支持这些机制，并可根据需求进行灵活配置。

2.3.3　可信执行环境

微虚拟机的运行依赖底层的主机操作系统或虚拟机管理程序，但它们仍然是庞大的通用操作系统，存在众多潜在的安全问题。恶意的或被攻陷的管理程序可以随意访问或篡改微虚拟机内的敏感数据，这在机密计算等具有更高安全要求的场景下不可接受。本节介绍一种新兴的安全级别更高的防护技术——可信执行环境（TEE），也是本文第 5 章要重点研究的对象。

TEE 旨在确保敏感数据在一个隔离且可信的环境中进行存储、处理和保护，该环境一般被称为飞地（enclave）。该场景具有较强的威胁模型，攻击者往往具有特权，TEE 不但需要抵御操作系统级别的软件敌手，还需要防止对硬件平台进行的恶意物理访问。为此，TEE 一般需要以下两大功能，来保证飞地的机密性、完整性，以及代码提供者身份的真实性：

（1） 隔离。TEE 的核心是内存隔离，来确保飞地内的代码、数据和运行时状态不被不受信任的软硬件访问或篡改，包括其他软件、固件、BIOS 或设备等。此外，TEE 一般还会用内存加密等技术防止物理攻击。

（2） 认证。TEE 在运行前需要通过远程证明（remote attestation）来生成一个认证报告（quote），其中包含了对运行在各层级的软件状态的度量（measurement），并使用硬件维护的认证密钥进行签名。远程用户可通过检查签名（反映硬件的身份）和度量结果（证明软件的状态）来验证报告的有效性。

可信执行环境的隔离功能需要由硬件提供保证。为此，要么修改硬件增加适用于 TEE 场景的新机制，要么在已有硬件机制的基础上再通过软件方法加以改造，本节将分别介绍这两类方法。

基于特定硬件。现有的大多数 TEE 都依赖特定的硬件，例如目前已在各大主流 CPU 上商用的工业级 TEE。Intel SGX[51] 将飞地作为进程的一部分，运行在用户态。Intel TDX[53] 和 AMD SEV[52] 等基于虚拟化的 TEE 将飞地作为一个独立的虚拟机，可运行具有特权功能的完整操作系统。ARM TrustZone[54] 则将飞地运行在一个独立的安全世界中。也有工作为 IBM PowerPC 架构提供了 TEE 的支持[116]。最近在 ARMv9-A 中还引入了基于领域（realm）的机密计算架构[117]（CCA）。

以基于进程的 Intel SGX 为例，应用在编写时就被分为可信与不可信两部分。可信部分（即飞地）的代码和数据存储在一块特殊的经过加密的物理内存中，被称为*飞地页面缓存*（Enclave Page Cache, EPC），无法被不可信的软硬件访问。EPC 内存的分配与页面映射仍由操作系统管理，因此飞地与应用不可信部分共用一份页表。为了防止不可信的操作系统随意修改页面映射而造成某些攻击，SGX 会在硬件中为每个 EPC 页面维护一个反向页表，即*飞地页面缓存映射*（Enclave Page Cache Map, EPCM），用于记录页面的虚拟地址、访问权限以及所属飞地，并在 TLB 缺失时进行额外的安全检查[118]。在飞地创建、初始化、远程证明完成后，可让控制流在应用的可信与不可信部分之间进行切换，以实现函数相互调用。其中应用不可信部分对可信部分的调用被称为 ECALL，反之，应用可信部分对不可信部分的调用被称为 OCALL，两者统称为边缘调用（edge calls）。Intel SGX 目前已在多个领域被广泛应用，但主要问题是需要修改应用程序以拆分出可信部分。因此目前常在 SGX 飞地中运行一个库操作系统来为旧应用提供兼容性[33-36]。

另一些研究型的可信执行环境需要对已有的硬件或固件进行修改，目前只有基于 FPGA 等可编程硬件实现的原型系统。CURE[119] 修改了 CPU 与系统总线以支持飞地标识符（eid），使得可以根据 eid 来对内存和外设进行访问控制，同时 CURE 也支持在不同的特权模式运行飞地。Penglai[120] 基于 RISC-V 实现，重点优化了内存保护的可扩展性，以支持上千个飞地的并发运行。

对于硬件实现的 TEE，不同的硬件厂商提供了不同的 TEE 设计，使得更新与演化都非常缓慢，而且很难实现一个适用于各种平台的统一软件架构。此外，硬件厂商也对 TEE 采用了不同的认证流程，导致很难在不同厂商提供的硬件 TEE 上进行数据的协同处理，因为不同厂商之间相互不信任。

基于已有硬件。以上提到的几种 TEE 技术存在依赖特定硬件、演化缓慢、与硬件厂商绑定等问题。另一些工作基于已有的可广泛获取的硬件，使用软件方法实现了 TEE，并提供与硬件实现相当的安全级别。

为了利用已有硬件机制实现 TEE 的隔离功能，常用的方法是使用硬件虚拟化与嵌套页表，通过一个轻量的虚拟机管理程序来实现 TEE 的功能。TrustVisor[55] 将应用程序中可信的代码（被称为 PAL）放在的虚拟机中进行保护，不过需要用户手动进行代码划分。InkTag[56] 利用一种半验证（paraverification）的方法，简化了虚拟机管理程序对不可信操作系统内存映射的检查，让管理程序变得更轻量更安全。SeCage[121] 利用程序分析技术自动分解应用中的敏感与非敏感代码，分别放入不同的虚拟机中运行，并利用 VMFUNC 机制进行高效跨域通信。BlackBox[122] 基于 ARM 虚拟化技术实现了 TEE，用于保护容器中的应用免受来自主机操作系

统的攻击。虚拟化技术也被用于实现工业级的 TEE，如 AWS Nitro Enclaves[123]、微软 Credential Guard[124] 和 Hyper-V Shielded VM[125]等。本文第 5 章也是基于此类方法，利用虚拟化实现可信执行环境用于保护微虚拟机，具有硬件方法难以提供的灵活性。

由于已有硬件的安全检查并不全面（例如没有 EPCM 等基于硬件的安全检查），以上方法容易受到内存映射攻击，但是现有方法在解决这些安全问题时会带来较大的性能开销。例如，TrustVisor 通过将 PAL 的页表设置为只读来防止此类攻击，不过在 PAL 注册和切换时会引入较大的开销。此外，这种方法还会导致对页表访问位与脏位的更新也触发嵌套页表缺页，从而带来巨大的性能下降[126]。本文第 5 章采用了让 hypervisor 管理飞地页表的方法来消除此类开销，同时也能避免内存映射攻击与基于页表的攻击[127-128]。

还有一些工作基于其他硬件机制实现了隔离功能。Komodo[129] 利用 ARM TrustZone 实现了类似 SGX 的飞地保护模型，并进行了形式化验证。Keystone[130] 利用 RISC-V PMP 机制实现飞地物理内存的隔离，而无需二级地址翻译的开销，不过能创建的飞地数量也受到了 PMP 的限制（最多 16 个）。

基于可信执行环境的微虚拟机操作系统。由于现有可信执行环境提供了与一般的应用不同的编程模型与接口，常常使用微虚拟机技术为 TEE 中的应用程序提供兼容性。例如 Haven[33] 是第一个支持在 Intel SGX 中运行无修改二进制程序的微虚拟机操作系统。SCONE[34] 利用 SGX 为 Linux 的容器提供安全加固。Graphene-SGX[35] 和 Occlum[36] 为 SGX 中的微虚拟机操作系统提供了多进程支持。OP-TEE[37] 是一个用于 ARM TrustZone 的微虚拟机操作系统。

2.3.4　其他安全防护技术

除了软硬件隔离机制提外，还有一些其他技术可用于微虚拟机的安全防护。例如利用静态或动态分析工具检测软件中的缺陷[131-133]，利用形式化验证技术在数学上证明软件的安全属性等等[16,134]。由于这些技术与本文的研究内容关联不大，这里不再展开介绍。

2.4　本章小结

本章介绍了目前的一些微虚拟机操作系统研究，以及与微虚拟机相关的性能优化与安全防护技术。然而，目前的微虚拟机技术在一些场景下存在局限性，性能也仍有不少可提升的空间。用于微虚拟机的安全防护技术，也难以兼顾通用性、安全性与性能。本文将在之后的章节对这些问题进行深入研究。

第 3 章　基于用户态微虚拟机的任务调度性能优化

3.1　本章引言

任务调度是操作系统的核心功能之一，调度策略的好坏关系到应用程序的尾延迟、最大吞吐量、资源利用率等指标[45,75-76]。以现代的云计算场景为例，谷歌数据中心将单个应用拆分为多个分布式服务，并使用远程过程调用（RPC）进行相互通信，RPC 的尾延迟会显著影响整个应用的服务质量，而为了降低尾延迟就需要高效的调度系统来保证[135]。由于应用与负载的多样性，只靠一个单一的调度策略难以在所有情况下都有好的表现[44-45,74,136-137]。因此，本章希望能够针对应用与负载的特点，对操作系统的调度策略进行灵活定制。

然而，在 Linux 这样的宏内核中，很难以应用为粒度实现调度器的定制。Linux 的调度子系统与其他子系统紧耦合，使得新调度策略的开发、调试、测试、维护都十分困难，需要专家级的知识与经验[83,138]。而且部署新的调度策略又需要重新编译内核与重启系统，难以应用于生产环境中[83,138]。虽然最近也有一些工作，通过用户态代理[83]、热更新[138]、伯克利包过滤器[46]（BPF）等方式为开发内核调度策略提供了一定的便利性，但是它们在性能开销、灵活性等方面不尽如人意。

另一方面，Linux 内核线程管理的固有开销，也会影响调度策略的实际表现。例如，由于上下文切换的开销，Linux 的任务抢占时延为毫秒级，难以满足高性能调度器所需的微秒级抢占需求[45]。此外，线程的创建与同步等操作也需要系统调用而花费不少时间，难以做到细粒度的并发与频繁的线程同步[139-140]。

用户态微虚拟机为调度策略的定制提供了另一种思路。最近的内核旁路技术将设备驱动[31,40]、网络栈[30,38]等内核子系统卸载到用户态，取得了巨大的性能提升。将调度子系统卸载到用户态，也是一个极具吸引力的方向。然而，在用户态实现通用调度框架的主要挑战是，难以同时支持*微秒级抢占*与*多应用调度*。

微秒级抢占是实现微秒级尾延迟的关键。数据中心中广泛使用运行到完成的先来先服务（First Come First Served, FCFS）调度策略，但只在轻尾负载下（例如 Memcached 的 USR 负载[141]）表现良好，在重尾负载下会因队头阻塞（head-of-line blocking）现象导致尾延迟恶化。例如，对于 RocksDB 等数据库服务，GET/SET 请求的处理时间只需几微秒，而范围查询的处理时间高达毫秒级，容易使短请求被长请求阻塞。此时就需要使用抢占式的处理器共享（Processor Sharing, PS）策略[136]。一些用户态的调度系统，例如 ZygOS[44] 与 Shenango[75] 不支持单个进程内的抢占式调度，而另一些用户态调度器依赖 Linux 信号机制来实现抢占[142-143]，会带来

高达数百微秒的开销。

另一方面，支持多应用也对提高 CPU 的资源利用率至关重要。在低负载时，调度器需要将延迟敏感的应用（latency critical, LC）未使用的核心分配给一些尽力而为的应用（best effort, BE），并在发生突发负载时将核心重新分配给 LC 应用。Shinjuku[45] 与 Concord[74] 系统虽然分别利用 Posted-Interrupts 与编译器插桩技术实现了微秒级的抢占式调度，针对轻尾与重尾负载优化了尾延迟与吞吐量，但它们让单个应用独占了所有核心，不支持在低负载时与其他应用分享核心，浪费了 CPU 资源。

为此，本章提出了基于用户态微虚拟机的通用调度框架 Skyloft，致力于实现以下目标：

灵活性：Skyloft 能提供灵活的调度策略定制，并支持多应用调度。这既包括了抢占式与非抢占式的策略，也包括了应用内与应用间的线程调度。为此，Skyloft 提出了一系列调度原语，帮助在用户态开发多种调度器。

高效率：Skyloft 支持低开销的抢占以满足微秒级尾延迟。为此，Skyloft 利用了 Intel 最近在 Sapphire Rapids 系列处理器中新推出的用户态中断技术[58]，将中断的处理委托到用户态。这些中断不仅可以由用户线程手动发送产生，也可以由硬件时钟自动产生。同时，Skyloft 也实现了轻量级的上下文切换、中断管理、资源分配与同步原语，用于高效的线程管理与调度。

兼容性：Skyloft 不仅兼容已有的基于内核旁路的高性能 I/O 框架，也提供了 POSIX 接口以兼容已有应用。Skyloft 能与 Linux 共存，应用可自由选择使用 Linux 原生的调度器还是 Skyloft 的调度器。

实验表明，Skyloft 能够灵活地支持分布式、集中式等多种调度模型。在 schbench[144] 测试中，Skyloft 比实现了同样分布式调度策略的 Linux 原生调度器的唤醒延迟降低了几个数量级。在合成负载下，Skyloft 比实现了同样集中式调度策略的 ghOSt[83] 系统的最大吞吐量高 19%，同时尾延迟低 33%。在 RocksDB 这样的真实应用中，支持抢占的 Skyloft 的吞吐量达到了 Shenango 的 1.9 倍。

本章的主要贡献如下：

1. 基于微虚拟机的思想，利用新硬件机制，设计了一个通用的用户态调度框架，同时支持微秒级抢占与多应用调度。
2. 提出了一套在用户态开发多种调度器的通用调度原语，能够方便用户灵活定制调度策略。
3. 实现了一个原型系统，并移植了典型的应用，在多种负载下都取得了比已有工作更好的性能。

3.2　研究动机

本节先整理与分析了现有的各类调度系统采用的调度模型，然后指出将它们实现在用户态面临的几大挑战，以及现有工作对这些挑战的解决情况与其局限性，最后给出了本章提出的 Skyloft 系统的设计目标。

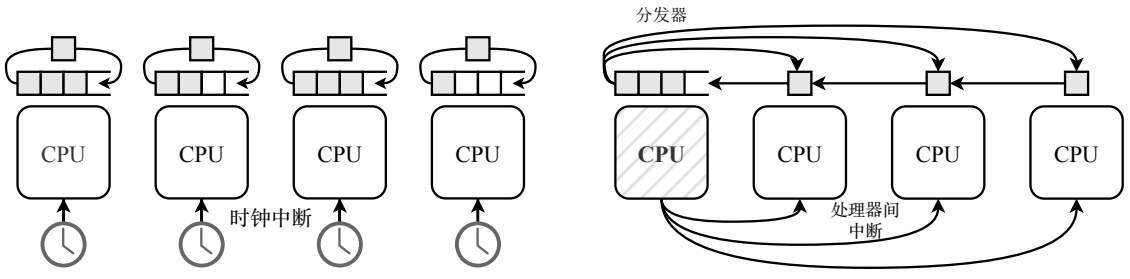
3.2.1　各种各样的调度模型

目前已经有许多任务调度系统被提出，大致可以按如下两个维度进行分类：

（1）根据**任务切换时机**，可以分为运行到完成、协作式、抢占式三类调度模型。对于运行到完成的模型，任务生命周期里始终不会切换到其他任务；协作式模型可以在任务执行过程中切换到其他任务，不过是任务自己主动切换，时机确定；而抢占式模型需要在某一个信号到来时进行被动切换（如中断），时机不确定。

通用操作系统的调度器一般是抢占式的，以防止某个应用一直占用 CPU。用户级线程库一般采用协作式调度，因为在用户态难以实现低开销抢占。对于执行时间较短的 RPC 请求（如 Memcached[145] 等键值存储的 GET/SET 请求），一般采用运行到完成的模型来避免切换开销，同时还有助于提高数据局部性[1]。然而，对于非抢占式调度，在重尾分布负载下容易导致队头阻塞而无法保证短任务的延迟。抢占频率越高，短任务就越不容易被长任务阻塞，不过过高的抢占频率也会使得 CPU 时间被大量浪费在状态的保存与恢复上，反而降低吞吐量。因此抢占周期一般与短任务的处理时间相当。

抢占除了可以使用时钟中断进行直接支持外，也可以使用处理器间中断（IPI），即由一个核心模拟时钟，按一定的频率向其他核心发送 IPI 来实现。这两种方式分别对应下文介绍的分布式与集中式调度模型，如图 3.1 所示。



(a) 分布式调度　　(b) 集中式调度

图 3.1　分布式与集中式的调度模型

（2）根据**任务分发方式**，可以分为分布式与集中式两类调度模型。对于分布式调度，每个 CPU 核心都会运行调度程序，根据相应的策略从运行队列中取出任务来运行，并且可以通过 CPU 本地的时钟中断来支持抢占（图 3.1(a)）。对于集中式调度，有一个专门的 CPU 核心执行调度决策（被称为分发器），不断给其他核心分

发任务，其他核心只运行任务而不干预调度，同时分发器也可以向其他核心发送 IPI 来支持抢占（图 3.1(b)）。

在实现具体的调度策略时，分布式调度一般需要在每个核心上维护一个运行队列，以避免多核同时操作全局运行队列造成数据竞争。不过这也容易导致负载不均衡而使性能下降，因此一般需要辅以一定的负载均衡策略[44,146]。集中式调度的单队列模型相对更容易做到负载均衡，不过由于分发器需要遍历所有核心，当核心数量较多时也容易成为性能瓶颈。此外，由于分发器核心一般无法运行任务，因此与使用同等数量核心的分布式调度相比，集中式调度的最大吞吐量更低。

通用操作系统以及一些早期的数据平面操作系统多采用分布式的调度模型，如 Linux 调度框架（包括 CFS[147] 和最近引入的 EEVDF 调度器[148-149]）、IX[1]、ZygOS[44] 等。近年来一些专门优化尾延迟的调度系统多采用集中式的调度模型，如 Shinjuku[45]、Concord[74] 等。

3.2.2　用户态调度的挑战

在用户态微虚拟机中支持第 3.2.1 节所述的几种调度模型并不容易。因为与设备驱动、网络栈等子系统不同，调度子系统是操作系统的核心功能，依赖中断、处理器状态切换等需要特权才能进行的操作。具体来说，运行在用户态的调度器需要解决*微秒级抢占*与*多应用调度*这两个问题，同时不能对性能造成太大影响。

微秒级抢占。抢占要求能在任意时刻打断任务的执行，并在之后恢复。在内核级调度器中，抢占可以利用硬件中断轻易实现，无论是抢占当前处理器的时钟中断，还是抢占其他处理器的 IPI。然而，为在用户态接收与处理抢占事件，现有的用户级线程一般通过*信号机制*[142-143,150]，但是抢占周期只能做到几毫秒，因为每次信号处理都需要 4 次用户态与内核态之间的切换，开销较大。而且信号处理还存在可扩展性问题，当多个核心上的线程同时收到信号时，会因数据竞争而需要更多时间来处理[143]。

多应用调度。为了让调度器能够调度多个应用各自的线程，首先需要解决不同应用的线程切换问题。在内核看来，不同的应用属于不同进程，具有不同的地址空间，如果切换到另一个应用的线程需要同时切换地址空间。内核级调度器可以轻易进行地址空间切换，但这些特权操作难以在用户态执行。此外，调度器还需要能够看到所有线程的状态信息（可能属于不同应用），如线程的优先级、时间片等，以提供给相应的调度策略，因此需要为用户态的调度器提供一定的数据共享支持，以获得调度的全局视图。

3.2.3　已有工作的局限性

目前已有一系列工作致力于让应用程序来定制调度策略，既包括用户态的调度器，也包括内核态的调度器。表 3.1 总结了它们与 Skyloft 的对比。

表 3.1　Skyloft 与已有调度优化系统的对比

方法	调度单元	微秒级抢占	多应用调度
用户级线程、IX[1]、ZygOS[44]	用户线程	✘	✘
Shinjuku[45]、LibPreemptible[151]	用户线程	✔	✘
Shenango[75]、Caladan[76]	用户线程	✘	✔
ghOSt[83]、Enoki[152]	内核线程	✔	✔
Skyloft	用户线程	✔	✔

用户级线程。一些运行时库[140,153-156]，或 Rust、Go 等支持异步协程的编程语言[96,157]，提供了 M:N 线程模型，将 M 个用户线程映射到 N 个内核线程。用户线程比内核线程更轻量，也无需内核干预就可进行各种线程操作，使得用户级线程的创建、切换、同步开销都很小。然而，正如上文所述，用户级线程难以支持低开销抢占与跨应用调度。因此，再好的调度策略，也只能作用于单个应用内，而无法统筹调度多个应用间的任务，又因为单个应用的任务类型比较单一（均为 LC 或均为 BE），难以发挥先进调度策略的优势。此外，由于内核的调度器无法感知用户线程的调度，可能会无意挂起持有锁的用户线程，从而导致无限期的自旋而浪费 CPU 资源[158-159]。

数据平面操作系统。一些工作通过将任务调度框架与 DPDK[31] 等内核旁路技术相结合，实现了一个数据平面操作系统，例如 IX[1] 与 ZygOS[44]，然而它们的任务都是运行到完成的，不支持抢占。Shinjuku[45] 与 Concord[74] 分别利用了 Posted-Interrupt 与编译器插桩技术实现了微秒级抢占，LibPreemptible[151] 同样使用了用户态中断。不过这些系统将应用程序的逻辑与调度器耦合，让多种调度策略的适配与多应用的支持变得很复杂。同时，以上所有系统也只能让单个应用独占 CPU，导致资源利用率不高。

应用间重新分配核心。一些系统致力于实现应用间的性能隔离[75-76,146,160-161]，既满足 LC 应用的尾延迟，又尽量提高 BE 应用的吞吐量。它们一般以这样一种模式工作：按一定的频率，根据不同的指标定期检查应用程序的拥塞情况，然后决定是否为应用添加或删除处理器核心，同时在应用内进行任务的负载均衡。例如 Arachne[161] 的重分配频率为 50ms，而 Shenango[75] 和 Caladan[76] 的重分配频率

低至 5μs。McClure et al. 分析了不同的负载均衡与重分配触发策略对应用尾延迟的影响[146]。然而，这些系统都采用集中式的模型进行应用间的调度，需要独占一个核心，而且限制了调度策略的通用性。同时以上系统也不支持抢占。

应用自定义的内核调度器。还有一些系统仍使用内核原有的线程管理和调度器，但将调度决策委托给应用程序。ghOSt[83] 通过将内核线程的状态变化通知用户态的代理，在用户态实现自定义的调度策略，但需要与内核进行频繁通信，开销较大。Plugsched[138] 通过实现内核调度器的热更新，可以不用重编译内核与重启系统就进行调度策略的替换，不过仍需基于复杂的内核调度框架编写策略。Linux 最近的 `sched_ext` 补丁[46]可通过用户编写的 BPF[79] 程序来控制内核的调度决策，然而 BPF 程序出于安全限制表达能力不强（如不能使用循环、浮点数），使得难以实现复杂的调度策略。Enoki[152] 让用户使用安全的 Rust 语言编写内核的调度策略，而无需与用户态代理程序的通信开销，但与以上其他方法一样，仍使用内核级线程，在线程的创建与同步操作时存在系统调用的开销，性能不如用户级线程。

3.2.4　本章设计目标

Skyloft 旨在提供一个灵活又高效的通用调度框架，方便用户根据应用需求适配各种先进的调度策略与模型，而提出新的调度策略不在本文的讨论范围内。本章在设计 Skyloft 系统时考虑了以下几点要求：

- **灵活性。** Skyloft 需要提供一套通用的调度接口以实现各种调度策略，包括第 3.2.1 节提到的分布式与集中式、协作式与抢占式模型等。此外，Skyloft 不应局限于调度单个应用内的线程，还应支持多个应用间的线程调度。例如，当一个应用中的所有线程都被阻塞时，应将处理器分配给其他应用，以提高 CPU 利用率。这就要求 Skyloft 的调度器需要能看到所有应用的线程信息，以便做出全局性的调度决策。
- **高效率。**为支持延迟敏感型应用，Skyloft 必须以最小的开销实现微秒级抢占。抢占可以通过 CPU 本地的时钟中断产生（图 3.1(a)），或由一个专门的分发器向其他核心发送 IPI 来实现（图 3.1(b)）。Skyloft 还应限制与内核的交互，将尽可能多的功能在用户态实现以减少上下文切换的开销。例如，采用用户级线程来最小化线程操作开销，在用户态处理中断来减少抢占开销。
- **兼容性。**Skyloft 应该只对内核进行最小程度的修改，以确保与现有 Linux 系统的兼容性。它还需要为应用提供 POSIX 兼容的线程与网络 API，并允许应用灵活选择使用 Skyloft 还是 Linux 原生的调度器。此外，Skyloft 还应允许应用程序也能利用现有的内核旁路 I/O 框架来提高性能（例如 DPDK[31]）。

3.3　系统设计

本节先介绍 Skyloft 系统的整体架构，然后介绍 Skyloft 的设计细节，包括在用户态支持基于时钟中断与处理器间中断的抢占，跨应用线程调度的设计，为支持丰富的调度策略而提供的通用调度原语，以及对用户态 I/O 框架的集成。

3.3.1　整体架构

作为微虚拟机，Skyloft 运行在一个宿主操作系统（即 Linux）之上。为避免 Linux 内核调度器对 Skyloft 调度的影响，当机器启动后，会隔离出几个 CPU 核心专门运行 Skyloft 及其应用程序（本文称它们为隔离核心），其余核心仍运行宿主操作系统的调度器与普通应用程序（本文称它们为普通核心）。如图 3.2 所示，Skyloft 主要由两部分组成：运行在用户态的 *Skyloft 库操作系统*，用于实现绕过内核的调度功能，并为应用提供 POSIX 接口；以及运行在内核态的 *Skyloft 内核模块*，用于实现不同应用间的线程切换与其他需要特权的操作。

Skyloft 库操作系统的核心功能是管理用户线程这一基本的调度单元。用户态的调度器会根据用户指定的调度策略 ❶，操纵在所有应用间共享的运行队列并做出调度决策。当需要进行线程切换时，调度器会根据切换前后的线程是否属于同一应用，决定是在用户态直接切换用户线程 ❷，还是通过内核模块切换相应的内核线程 ❸（第 3.3.3 节）。调度器还负责处理用户态中断以支持微秒级抢占 ❹（第 3.3.2 节）。此外，Skyloft 库操作系统也集成了绕过内核的用户态 I/O 栈 ❺（第 3.3.5 节）。

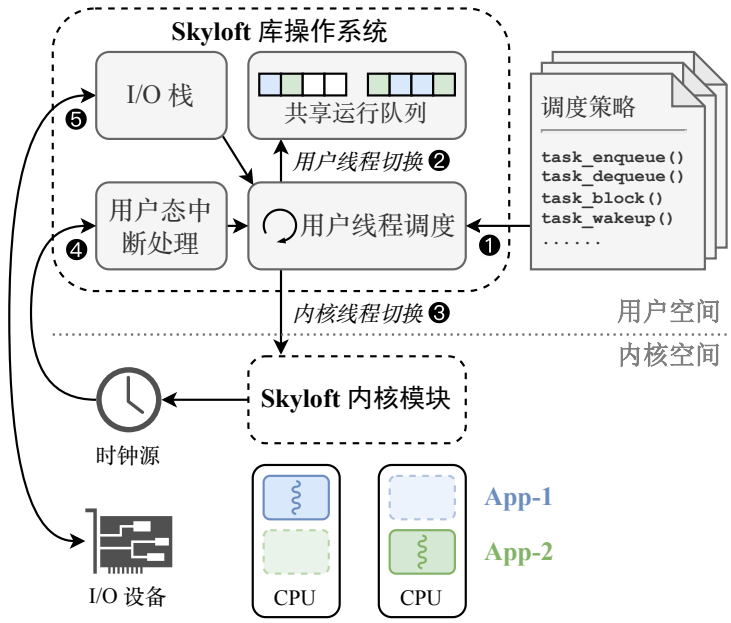


图 3.2　Skyloft 系统架构

3.3.2　用户态抢占支持

为了能在用户态支持微秒级的抢占式调度策略，Skyloft 利用了 Intel 用户态中断技术（UINTR）。在 Skyloft 中，抢占信号可以通过两种方式产生：向其他核心发送处理器间中断，或直接由硬件产生时钟中断。本节先简要介绍 Intel 用户态中断的相关背景，然后给出在 Skyloft 中支持这两种抢占方式的设计。

背景：Intel 用户态中断。用户态中断允许将中断直接交给用户态的处理例程来处理，而无需进出内核进行昂贵的特权级与地址空间切换，以提供一种低开销的事件分发与进程间通信机制。该特性已在自 Sapphire Rapids 起的服务器 CPU 中可用，Intel 也已经为 Linux 内核提交了用户态中断的补丁[162]。

在 Intel 的用户态中断机制中[58]，有一个特殊的向量号用来标识用户态中断，通过 UINV（User Interrupt Vector）寄存器配置。每个接收者线程会注册一个用户态中断处理例程（通过 UIHANDLER 寄存器），并在内存中维护一个用户态中断发布描述符（User Posted-Interrupt Descriptor, UPID）用于存放必要的中断信息。例如 UPID 中的 PIR（Posted-Interrupt Requests）字段保存了已由发送者发布但还未被接收的中断，只有当 PIR 非空时处理器才会认为发生了一个用户态中断。在进入处理例程之前，处理器还会根据 PIR 来设置 UIRR（User Interrupt Request Register）寄存器的相应位（以位图形式保存 64 个中断编号）。

用户态中断发送。用户态中断一般通过 `SENDUIPI` 指令产生。每个发送者会在内存中维护一个用户态中断目标表（User Interrupt Target Table, UITT），每个表项包含接收者的 UPID 地址与用户态中断编号。`SENDUIPI` 指令将 UITT 的表项索引作为参数，向对应的接收者发送对应编号的用户态中断。硬件会将该中断号写入目标线程 UPID 的 PIR 字段，并通过 APIC 向其所在的 CPU 发送一个向量为 UINV 的 IPI。

用户态中断处理。用户态中断的处理流程可分为以下几步：（1）识别：当处理器收到一个中断 V 时，只有当 V 等于 UINV 时才会进行下一步，否则 V 会被认为是一个普通中断，执行原本的 IDT 中断处理流程。（2）通知：硬件根据 UPID 的 PIR 字段的每一位设置 UIRR 寄存器，以标识待处理的用户态中断。（3）投递：一旦处理器进入了用户态，同时 UIRR 被设置，则会执行用户态中断的投递：将当前的栈指针、指令指针等寄存器压栈，然后跳转到用户态中断处理例程。（4）处理：中断处理例程进行其余上下文的保存与中断的处理，完成后恢复上下文，并执行 `UIRET` 指令回到之前被打断的位置。

使用处理器间中断进行抢占。这种方式适用于图 3.1(b) 所示的集中式调度模型。Skyloft 会让分发器同时扮演时钟的角色，每隔一定的时间检查是否需要打断

其他工作核心上的任务，如果需要就向其发送一个用户态 IPI，使其陷入用户态中断处理例程并切换到其他任务。这种方式的一个明显优势是具有较强的灵活性，例如可以根据负载的变化动态调整抢占通知的频率[151]。然而，这种方式需要一个专门的核心作为分发器，不能用于处理常规任务，限制了最大吞吐量。此外，分发器还需要持续监控所有工作核心，这可能会成为瓶颈，特别是在具有大量核心的系统上，从而影响可扩展性[45,74]。

使用时钟中断进行抢占。为消除专门的分发器核心，可采用如图 3.1(a) 所示的分布式调度模型，此时的中断需要由硬件自动产生（例如每 CPU 本地的时钟中断）。然而，如何将时钟、外设等硬件直接产生的中断委托到用户态是一个重大挑战。截止本文撰写时，无论是 Intel 手册[58] 还是其他相关研究都没有涉及该问题。

Skyloft 采用以下方法来支持在用户态处理时钟等硬件中断。首先是将 UINV 设置为时钟中断向量，使得处理器能将时钟中断识别为用户态中断。然而，由于该中断不是通过 `SENDUIPI` 指令产生的，目标线程的 PIR 不会在每次时钟事件时被自动设置，这就无法进入用户态中断的通知及后续流程。又因 UPID 是内核维护的结构，在用户态不可见，如果直接将其内存映射到用户空间进行修改容易产生安全隐患。为了能够在用户态更新 PIR，一种方法是执行一条额外的 `SENDUIPI` 指令，但这会导致不必要的 IPI 开销。幸运的是，UPID 中提供了一个 SN 位（suppress notification）可以阻止真实 IPI 的产生。于是，Skyloft 可以在设置 SN 位后执行一次目标为自身的 `SENDUIPI` 指令，从而实现在用户态高效修改 PIR 而不会触发真实的 IPI。

算法 3.1　Skyloft 对用户态硬件中断的支持

```
1  function uintr_handler():
2      SENDUIPI;                          /* 再次设置 UPID.PIR (3) */
3      处理中断;
4  function uintr_setup(handler):
5      UIHANDLER ← handler;               /* 注册用户态中断处理例程 */
6      UINV ← 硬件中断向量;
7      UPID.SN ← 1;                       /* 设置 SN 位 (1) */
8  function main():
9      uintr_setup(uintr_handler);        /* 该函数需在内核态实现 */
10     SENDUIPI;                          /* 设置 UPID.PIR (2) */
11     STUI;                              /* 取消屏蔽用户态中断 */
12     执行任务逻辑;
```

算法 3.1 给出了 Skyloft 支持用户态硬件中断（包括时钟、外设）的具体流程：（1） 在初始化时，设置 UINV 为硬件中断向量，同时设置 UPID 的 SN 位。（2）

然后先执行一次 SENDUIPI，将 UPID 的 PIR 初始化为非空（可使用任意中断编号），使得首次硬件中断能够在用户态处理。（3） 在中断处理例程里再次执行 SENDUIPI 指令，使得下一次硬件中断也在用户态处理。通过以上步骤，Skyloft 成功将本地 APIC 时钟中断委托到了用户态以支持分布式抢占调度。

3.3.3　跨应用线程调度

为了能在用户态支持多应用的调度，一种方法是实现 “用户级进程” 的抽象，即在用户级线程的基础上，增加地址空间、文件描述符表等进程级资源。然而，这会大大增加库操作系统的复杂性，而且这些资源的管理还需要大量特权操作，难以高效地在用户态实现。Skyloft 的基本想法是尽量重用内核已有的进程与线程抽象，利用进程代表应用的地址空间、文件等全局资源，利用内核线程代表应用的 CPU 资源。

内核线程管理。 Skyloft 的应用仍作为宿主操作系统的一个进程存在，其会为每个隔离核心创建一个对应的内核线程，并绑定在该核心上。当存在多个应用时，每个隔离核心上会绑定多个内核线程（每个应用一个）。在这些内核线程中，只有处于可运行状态的才会被内核的调度器选中去执行，其余内核线程要么已退出，要么被挂起或阻塞，不在内核调度器的运行队列里。本文分别将它们称为活跃的与非活跃的内核线程。为了防止内核调度器干扰用户线程的调度，Skyloft 对这些内核线程的状态做出以下规定：

单一绑定原则： 不能同时将两个及以上的活跃内核线程绑定到同一隔离核心上。

基于以上原则，在应用的运行期间，内核的调度器将被完全绕过，不会进行非预期的活跃线程切换。一个核心上的唯一活跃内核线程对应了当前正在其上面运行的应用，一个应用的所有活跃内核线程对应了该应用目前正在哪些核心上运行。如图 3.2 中，App-1 和 App-2 各自有一个活跃的内核线程，分别占据不重叠的隔离核心。

一些不归 Skyloft 管理的应用仍可能通过手动设置 CPU 亲和性而绑定到隔离核心上，且处于可运行的状态。但这些应用不与 Skyloft 共享状态，不会干扰 Skyloft 的正常运行。不过 Skyloft 应用可能会因与之共用同一核心而降低性能，所以也需要尽量避免这些非 Skyloft 的应用占据隔离核心。

用户线程切换。 Skyloft 会将所有应用的所有用户线程放入统一的运行队列中进行管理与调度，根据调度模型的不同，运行队列可以是每核心私有的，或是全局唯一的。因此，同个运行队列中可能包含属于不同应用的用户线程，运行队列需要在所有应用间共享。

在用户线程的运行过程中，可能会主动或被动（被抢占）地切换到其他用户线程。下一个要运行的用户线程会根据调度策略从共享的运行队列中取出。如果下一个线程与当前线程同属一个应用（同一地址空间），则可直接在用户态完成切换，无需内核干预，如图 3.3 中 App-1 的线程 ***A*** 切换到线程 ***B***。否则需要进行应用的切换，这是通过切换当前核心上两个应用对应的内核线程来实现的。具体来说，将当前应用的活跃内核线程设为非活跃的（挂起），并将目标应用的内核线程设为活跃的（唤醒），如图 3.3 中 App-1 的线程 ***B*** 切换到 App-2 的线程 ***C***。这两步操作需要在内核态同时完成以符合单一绑定原则。由于不同应用间的线程切换开销要显著高于相同应用内的线程切换开销（第 3.5.5 节），所以尽量减少应用间的线程切换，也是调度策略需要考虑的一个优化指标。

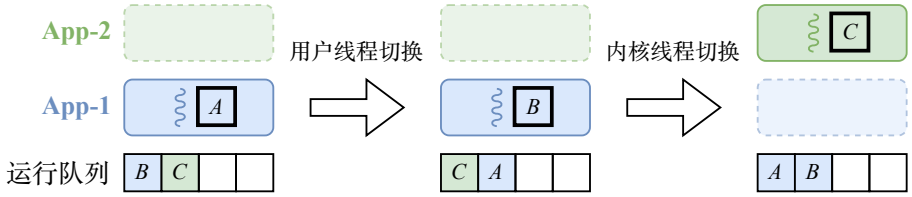


图 3.3　Skyloft 跨应用的用户级线程切换

应用启动与退出。当新应用启动时，需要创建内核线程并进行绑核，以及创建用户线程并更新运行队列（图 3.4 ❶）。新创建的内核线程显然是可运行的，然而如果之前已有其他 Skyloft 应用在运行，隔离核心上就会已存在活跃的内核线程，此时直接将新内核线程绑定到隔离核心上就会违反单一绑定原则。为此，内核需要提供一个操作，将当前内核线程绑核的同时，让其挂起以变为非活跃状态，并在该操作完成之前不能切换到其他活跃的内核线程，以确保单一绑定原则始终成立。在该操作完成后，新创建的内核线程将处于非活跃状态，等待其他应用的活跃线程将其唤醒（图 3.4 ❷）。

当应用的所有用户线程都执行完毕后，需要退出该应用（或者应用主动调用 `exit` 退出函数），终止该应用的所有内核线程。如果其中还有活跃的线程，需先唤醒其所在的核心上的另一线程（图 3.4 ❺），否则在该线程退出后，其所在的核心上将没有活跃的线程，且这些非活跃线程将永远无法被唤醒。此外，由于内核线程的退出需要回收资源，这只能由该线程自己完成，所以在每个内核线程真正退出之前仍需将其变为可运行状态。为了防止违反单一绑定原则，Skyloft 会先将待退出的线程都绑定回普通核心上（图 3.4 ❹），再让它们自行退出（如果是活跃的），或向它们发送终止信号（如果是非活跃的）。

如上所述，内核需要提供几个操作，来同时完成一些内核线程活跃状态的转换或 CPU 亲和性的设置，以遵循单一绑定原则。Skyloft 将这些操作都实现在一个

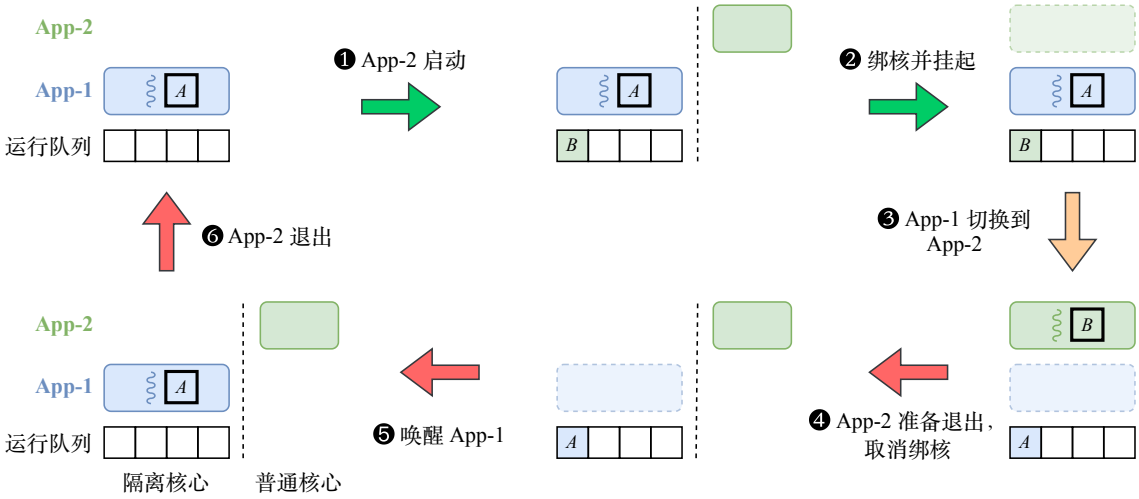


图 3.4　Skyloft 应用的生命周期

Linux 的内核模块中，并通过 `ioctl()` 接口提供给用户态的 Skyloft 调度器（详见第 3.4.2 节）。

3.3.4　通用调度原语

Skyloft 定义了一套通用的调度原语，用于在用户态实现第 3.2.1 节中提到的两种典型的调度模型。如表 3.2 所示，这些原语包括对运行队列和任务状态的操作，以及一些事件的回调。例如，`sched_timer_tick()` 会在时钟中断处理例程里被调用，并决定是否需要抢占当前线程。基于这些底层原语，可实现 pthread 函数等更高层的线程原语。

还有一些原语与特定的调度模型相关。例如，对于具有多个运行队列的分布

表 3.2　Skyloft 通用调度原语

调度原语	描述
`void task_init(struct task_t* task)`	初始化任务结构
`void task_terminate(struct task_t* task)`	任务运行完成，释放任务结构
`void task_enqueue(struct task_t* task)`	将任务加入运行队列
`struct task_t* task_dequeue()`	选择下一个要运行的任务并从运行队列中移除
`void task_block()`	挂起当前任务
`void task_wakeup(struct task_t* task)`	唤醒指定任务并将其加回运行队列
`void sched_init(void* data)`	初始化调度器
`bool sched_timer_tick()`	（在每次时钟中断时）更新调度器状态并返回是否需要重新调度（当前任务被抢占）
`void sched_balance()`	对运行队列进行负载均衡（仅分布式调度）
`void sched_poll()`	轮询全局运行队列并将任务分发给工作核心（仅集中式调度）

式调度来说，队列是否平衡会严重影响性能[44,75,146]。为此，Skyloft 提供了一个 `sched_balance()` 原语，用于实现各类负载均衡策略（例如工作窃取[163]）。而单队列的集中式调度自然就是平衡的[45]，无需以上操作。相反，其分发器需要调用 `sched_poll()`，来将全局运行队列中的任务分配给每个工作核心。

3.3.5　集成用户态 I/O 框架

Skyloft 采用与已有数据平面操作系统类似的设计[1,44-45,75]，集成了绕过内核的高性能 I/O 框架。以网络为例，Skyloft 使用 DPDK[31] 在一个专门的 I/O 核心（非隔离）上轮询网卡队列，每当收到一个数据包时，会根据接收端扩展（RSS）哈希值将数据包分发给相应的隔离核心（运行 Skyloft 库操作系统），两者之间通过共享的环形缓冲区（ring buffer）进行数据交换与事件通知。在 Skyloft 库操作系统中集成了轻量级的用户态 TCP 与 UDP 协议栈，用于解析网络数据包并为应用程序提供 POSIX 兼容的套接字（socket）接口。当用户线程因网络请求而阻塞时，调度程序会将其挂起并切换到其他用户线程。Skyloft 的调度器也会在空闲时检查是否有新到达的数据包，并根据需要创建一个新的用户线程来处理。

3.4　系统实现

本节将介绍 Skyloft 的实现细节。Skyloft 基于 Linux 实现，其中库操作系统包含 10,920 行 C 代码（近一半代码是基于 DPDK 实现的网络栈），与应用程序链接在一起，作为一个进程运行在用户态。Skyloft 内核模块包含 325 行 C 代码。此外，Skyloft 在 Linux 的 UINTR 补丁[162]上进行了 163 行修改，以支持用户态时钟中断。

3.4.1　Skyloft 库操作系统

调度器初始化。在系统启动时，会通过 Linux 的 `isolcpus` 命令行参数划分出隔离核心以供 Skyloft 及其应用使用，Linux 默认不会将任务调度到这些核心上。虽然 Linux 仍会在这些核心上运行一些后台任务[83]，不过不会干扰 Skyloft 的线程切换，对性能也几乎无影响。

在所有应用启动之前，会有一个 0 号应用先启动。该应用作为 Skyloft 的后台进程，不会创建任何用户线程，只是负责对应用间的共享状态进行初始化，以及执行一些全局操作。该应用启动后，会使用 pthread 库创建隔离核心数量的内核线程，并使用 `sched_setaffinity()` 绑定到相应的隔离核心上。对于后续启动的应用，其初始化流程与 0 号应用基本一致，只是将绑核操作换成了 Skyloft 内核模块提供的操作（第 3.4.2 节），可以同时完成绑核与挂起。

每个应用启动后，会先在每个内核线程上创建一个空闲用户线程，负责运行主调度循环直到找到其他可运行的用户线程。此外，当要从一个用户线程切换到其他应用的用户线程时，需要先切换到当前应用的空闲用户线程，再进行跨应用切换，从而将当前用户线程放回运行队列。

共享内存。不同的应用在宿主操作系统看来是不同的进程，它们之间使用共享内存来交换数据。每个应用都需要链接一份完全一样的 Skyloft 库操作系统，使得无论当前运行的是哪个应用，都使用相同的一套调度器代码与共享内存结构。需要在应用间共享的内存包括应用元数据、运行队列、共享数据结构的内存池等。在应用的元数据中主要记录了每个应用的每个内核线程的编号（通过 `gettid()` 获取），用于标识内核线程，以便被其他应用唤醒。在共享的运行队列中保存了当前所有的处于可运行状态的用户线程，调度器会根据相应的策略从队列中取出接下来要运行的用户线程。在用户线程的结构中也需要包含一些对其他应用可见的字段，如线程状态、所属应用编号以及具体调度策略所需的线程信息，使得无论是哪个应用在运行调度器，都能看到一致的调度信息。为此，Skyloft 在共享内存中维护了一块内存池，专门用于这些在应用间共享的数据结构。这块内存池在所有应用中都被映射到了相同的虚拟地址，使得能够在不同地址空间下仍使用同一地址进行访问。此外，用户线程的结构中还包含一些私有的、其他应用不可见的字段，如用户线程的寄存器上下文、运行栈等。

POSIX 接口。 Skyloft 为应用程序提供了 POSIX 兼容的接口，使得已有应用程序无需修改源码，仅需重新编译即可在 Skyloft 上运行以获得性能提升。这些接口分为两类：线程操作接口与网络操作接口。线程操作接口包括创建、睡眠、退出等对用户线程生命周期的管理，以及互斥锁、条件变量等同步原语。网络操作接口包括对 TCP、UDP 套接字的管理与读写。这些操作都绕过了宿主操作系统，直接在用户态实现，而无需昂贵的系统调用开销。

用户态中断配置。 Linux 的 UINTR 补丁[162]已经实现了对用户态 IPI 的支持，并以系统调用的形式为应用程序提供了几个接口，如表 3.3 所示。于是，无需额

表 3.3　Linux 用户态中断相关系统调用

系统调用	说明
`uintr_register_handler(handler, flags)`	注册成为一个接收者
`uvec_fd = uintr_vector_fd(vector, flags)`	返回一个接收者的文件描述符，以提供给发送者
`uipi_idx = uintr_register_sender(uvec_fd, flags)`	注册成为一个发送者，并与接收者建立连接

外的内核代码就可通过用户态 IPI 实现抢占式调度。此时需要有一个单独的分发线程作为发送者，向其他核心（作为接收者）发送抢占信号。分发线程通常会独占一个核心，没有高性能调度的需求，所以 Skyloft 会将其绑定到普通核心而非隔离核心上，使得能够有更多的隔离核心用于运行任务负载。由于发送者与接收者可能属于不同应用，一方打开的文件对另一方是不可见的。为此，发送者需要使用 `pidfd_get()` 与接收者共享打开的用户态中断相关文件描述符（`uvec_fd`），才能与接收者建立连接并允许发送 IPI。每个应用在每个隔离核心上的内核线程都是一个接收者，发送者需要与它们一一建立连接。

Skyloft 也能利用用户态时钟中断来进行抢占（第 3.3.2 节），而不需要专门的分发线程，但这并没有被 UINTR 补丁支持。为此需要修改用户态中断相关的 MSR（Model Specific Register）寄存器，并允许应用程序配置时钟频率。

用户态中断处理。如图 3.5 所示，Skyloft 为分布式与集中式调度策略实现了一个全局的用户态中断处理例程。在中断处理过程中，如果是时钟中断，首先需要重置 UPID.PIR 以允许下一次时钟中断（第 3.3.2 节）。然后处理例程会调用 `sched_timer_tick()` 原语更新调度器状态，并在需要抢占时进行重新调度。

```
1  void  __attribute__((interrupt))
2  uintr_handler(struct __uintr_frame *ui_frame, unsigned long long vec) {
3    if (is_timer_uintr(vec)) {
4      _senduipi(uintr_idx); // 重置 UPID.PIR 以允许下一次时钟中断
5    }
6    bool is_preempted = sched_timer_tick();    // 更新调度器状态
7    if (preempt_enabled() && is_preempted) {   // 检查是否需要重新调度
8      sched_enqueue(current);                  // 将当前任务放回运行队列
9      schedule();                              // 进入主调度循环
10   }
11 }
```

图 3.5　Skyloft 全局用户态中断处理例程

3.4.2　Skyloft 内核模块

Skyloft 内核模块会作为一个杂项设备文件挂载到 `/dev/skyloft`。它为用户态的调度器提供了两类操作：原子地调用一些内核的服务，用于实现内核线程的状态转换；或是执行一些特权操作，用于用户态中断的配置。这些操作都通过 `ioctl()` 接口提供给用户态的程序。

表 3.4 的上半部分列出了用于内核线程状态转换的几个操作。如第 3.3.3 节所述，它们会在应用启动、切换、退出时被调用。这些操作都可通过 Linux 为内核模块提供的 API 组合而来，包括设置当前线程的 CPU 亲和性、让当前线程挂起、唤醒其他线程等，而无需额外修改内核代码。

表 3.4　Skyloft 内核模块提供的接口

类别	接口	说明
应用生命周期管理	`skyloft_park_on_cpu(cpu_id)`	将当前内核线程绑定到指定核心，同时让其挂起
	`skyloft_switch_to(target_tid)`	让当前内核线程挂起，同时唤醒目标内核线程
	`skyloft_wakeup(tid)`	唤醒指定的内核线程
用户态中断配置	`skyloft_timer_enable()`	在当前核心上启用用户态时钟中断
	`skyloft_timer_set_hz(hz)`	设置当前核心上的时钟频率

表 3.4 的下半部分列出了配置用户态时钟中断所需的几个操作。如第 3.3.2 节所述，通过配置 UINV 与 UPID.SN 可以启用用户态时钟中断；通过配置本地 APIC 寄存器可以修改时钟中断的频率。当用户态时钟中断被启用后，该核心上的所有 APIC 时钟中断都将被委托到用户态来处理，而不会进入内核的处理例程，可能导致内核无法在这些核心上接收一些周期事件而造成系统故障。为此，Skyloft 通过 Linux 的命令行参数 `nolapic_timer` 禁用 APIC 时钟，而使用 PIC 等其他时钟源来触发周期事件，以保证宿主操作系统的稳定运行。

3.5　实验与评估

本节将通过一系列实验全面评估 Skyloft 调度框架的性能，并回答以下问题：

- Skyloft 对用户态时钟中断的支持会给分布式的调度策略带来多少收益？
- Skyloft 能否支持最近一些专门优化尾延迟的集中式调度策略？
- 与现有的通用调度框架相比，Skyloft 对多应用的调度性能如何？
- Skyloft 在具有不同类型负载的真实世界应用上的性能如何？
- Skyloft 的用户态抢占开销，以及各种线程操作的开销有多大？

3.5.1　实验设置

评估的调度器。为了方便与现有调度优化系统进行对比，本实验基于 Skyloft 实现了与这些系统策略相同的调度器。表 3.5 列出了这些调度器以及在 Skyloft 或相应系统上进行实现所需的代码行数。在 Skyloft 上实现这些调度器只需几百行代码，远小于与之比较的系统，这也体现了 Skyloft 在用户态开发各种调度器的优势。另一方面，之前一些实现在用户态的调度器通常与其他功能紧耦合，缺乏针对不同策略的灵活性。例如，Shinjuku[83] 拥有大约 3,900 行代码并严重依赖 Dune[25]。

表 3.5　实验评估的调度器及其代码行数

调度器	代码行数	说明
Linux CFS (kernel/sched/fair.c)	6,592	Linux 完全公平调度
Linux RT (kernel/sched/rt.c)	1,939	Linux 实时调度（时间片轮转）
ghOSt Shinjuku	710	基于 ghOSt 实现的 Shinjuku 策略[83]
ghOSt Shinjuku-Shenango	727	基于 ghOSt 实现的 Shenango 策略[83]
Skyloft Round-Robin	141	基于 Skyloft 实现的时间片轮转（3.5.2 节）
Skyloft CFS	430	基于 Skyloft 实现的完全公平调度（3.5.2 节）
Skyloft Shinjuku	192	基于 Skyloft 实现的 Shinjuku 策略（3.5.3 节）
Skyloft Shinjuku-Shenango	444	基于 Skyloft 实现的 Shenango 策略（3.5.3 节）
Skyloft Work-Stealing	150	基于 Skyloft 实现的抢占式工作窃取策略（3.5.4 节）

实验环境。本实验使用一台支持用户态中断的 Sapphire Rapids 机器作为服务器，CPU 型号为 Intel Xeon Gold 5418Y，主频为 2.0 GHz，共有 2 个 CPU 插槽，48 个物理核心（96 个逻辑核心），128 GB 内存，并且关闭了 TurboBoost 与动态频率调节。为测试网络应用，客户端使用一台拥有 32 个物理核心的 Intel Xeon E5-2683 v4 机器，主频为 2.1 GHz，内存为 128 GB。服务器与客户端均配有 Intel 82599ES 10 Gbps 网卡以进行网络传输。如无特殊说明，服务器运行 Ubuntu 22.04 操作系统，内核版本为 Linux 6.0.0（已针对用户态中断进行了一定修改）。网络测试所用的 DPDK 版本为 22.11。

3.5.2　分布式调度

为了评估 Skyloft 将时钟中断委托到用户态的收益，本实验实现了几种分布式调度策略，并与 Linux 内核中集成的几种调度器进行对比，包括 `SCHED_RR` 与 `SCHED_BATCH`，分别为时间片轮转（RR）与完全公平调度（CFS）策略。

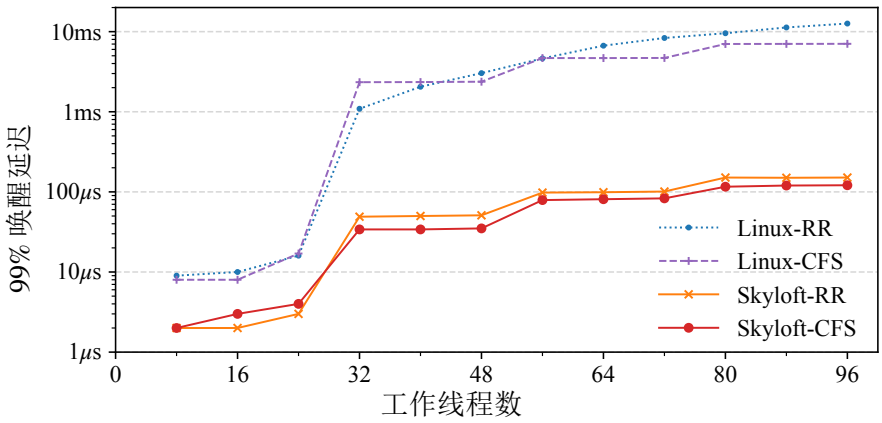
实验使用 schbench[144]（v1.0）这一调度器基准测试工具，该工具常被用于评估 Linux 等通用操作系统的调度器性能[164]。schbench 模拟了一种典型的网络应用，其会创建 M 个消息线程与 T 个工作线程，消息线程会不断唤醒工作线程，让其执行一些模拟的任务（矩阵乘法）。处理完后工作线程会进入睡眠状态，等待消息线程将其再次唤醒以执行下一个请求。

本实验将 Skyloft 配置为使用 24 个隔离核心。对于 Linux，也通过 `taskset`

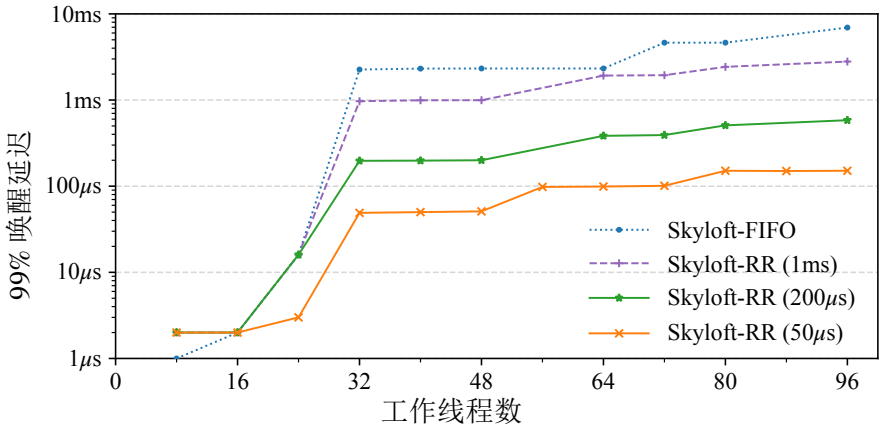
命令将 schbench 绑定到相同数量的核心上，并使用 `chrt` 命令让其使用 RR 调度策略。在 Linux 上无论使用 RR 还是 CFS 策略，schbench 都被设置为最高优先级。schbench 采用默认的参数（一次请求时间约为 2300μs），只创建 1 个消息线程，并通过不断增加工作线程的数量使 CPU 逐渐饱和。当工作线程数大于 CPU 数量时，工作线程的唤醒延迟将受到排队时间的影响，可体现调度器的性能。

图 3.6(a) 给出了 Linux 与 Skyloft 上共四种调度器的唤醒延迟。其中，基于 Skyloft 实现的两种调度器均使用了 50μs 的时间片，因为其能以极低的开销处理时钟中断并进行抢占。而 Linux 的抢占开销较高，只能支持毫秒级的时间片（CFS 与 RR 分别默认为 6ms 与 100ms），因此在 Skyloft 上的唤醒延迟要比 Linux 低几个数量级（100μs 对比 10,000μs）。此外，无论是 Linux 还是 Skyloft，CFS 策略的性能要略优于 RR，因为其能对阻塞的线程进行补偿[147]。

本节还评估了使用不同长度的时间片（即抢占频率）对唤醒延迟的影响。如图 3.6(b) 所示，schbench 的唤醒延迟大致与时间片长度相当，时间片越短（抢占频



(a) Skyloft 与 Linux 调度器的性能对比



(b) 配置了不同时间片的 Skyloft RR 调度器的性能对比①

图 3.6　使用 schbench 评估各种分布式调度策略的性能

① Skyloft-FIFO 表示无限长的时间片，即不启用抢占。

率越高），唤醒延迟也越小。

3.5.3　集中式调度

集中式的调度模型常用于优化尾延迟。本实验评估在 Skyloft 上实现 Shinjuku[45] 这一最近提出的集中式调度策略的表现。在 Shinjuku 中，有一个专门的核心运行分发器，并维护了一个全局的先进先出队列。对于每个传入的请求，分发器会将其分配给其他空闲的工作核心，并要求在一定的时间限制内完成处理。一旦超时，该请求就会被其他请求抢占，并重新加回请求队列。本实验的比较对象为以下四种系统：

（1） 原始 Shinjuku 系统，利用 Posted-Interrupt 支持微秒级抢占。该系统运行在 Linux 4.4 上，因为 Dune 无法在新版本内核上编译。

（2） 基于 ghOSt 实现的 Shinjuku 策略，利用一个全局代理向内核提交事务并做出调度决策[83]。该系统运行在 Linux 5.13 上，因为将 ghOSt 移植到新版本内核需要花费很大代价。

（3） 基于 Skyloft 实现的 Shinjuku 策略，利用用户态 IPI 支持微秒级抢占。

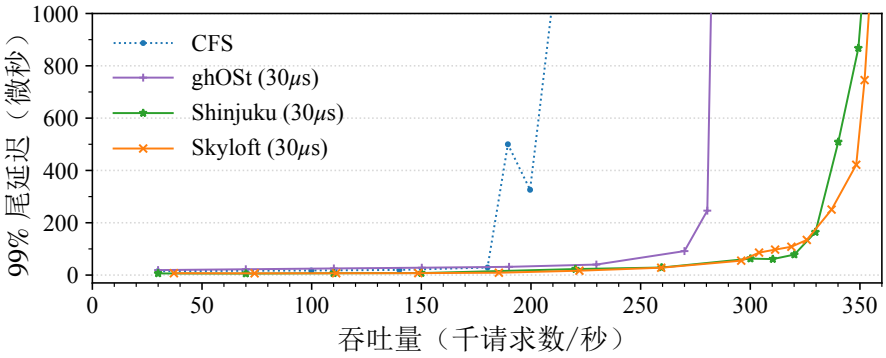
（4） 使用 Linux CFS 进行调度，不支持微秒级抢占。

单应用场景。本节的实验先评估只运行一个延迟敏感应用时的调度性能。实验实现了与 ghOSt 相同的开环负载生成器以及模拟的延迟敏感应用（让处理器自旋一定的时间）。其余配置也与 ghOSt 论文相同：使用 20 个核心作为工作核心，一个核心作为负载生成器与请求分发器。负载分布为 99.5% 的短请求与 0.5% 的长请求，处理时间分别为 4μs 与 10ms，符合泊松到达过程。

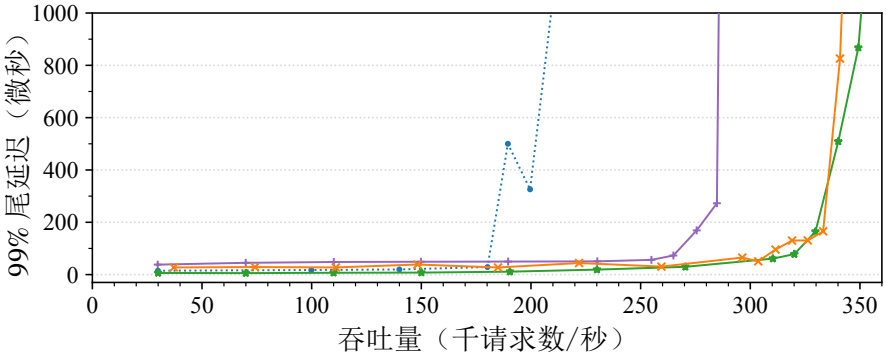
该负载的离散程度较高，因此不同的抢占频率会对尾延迟以及最大吞吐量造成较大的影响。本实验发现在该负载下使用 30μs 的抢占周期效果最好。当使用更高的抢占频率时，虽然能进一步降低尾延迟，但也使得中断处理的开销变大而降低最大吞吐量。

图 3.7(a) 给出了在该合成负载下各系统的性能对比。与 Skyloft 相比，Linux CFS 的最大吞吐量仅为 58.7%，这是因为其基于公平性的设计不适合对延迟敏感的应用。Skyloft 与 Shinjuku 的性能相当，因为二者均使用了低开销的抢占机制。对于 ghOSt，其性能不如前两者，最大吞吐量仅为 Skyloft 的 80.1%，在低负载下的尾延迟也为 Skyloft 的 3 倍以上。这是因为 ghOSt 仍基于 Linux 内核线程实现，在抢占时需要进行昂贵的上下文切换。此外，ghOSt 的性能还受到用户态代理与内核间频繁通信的影响。

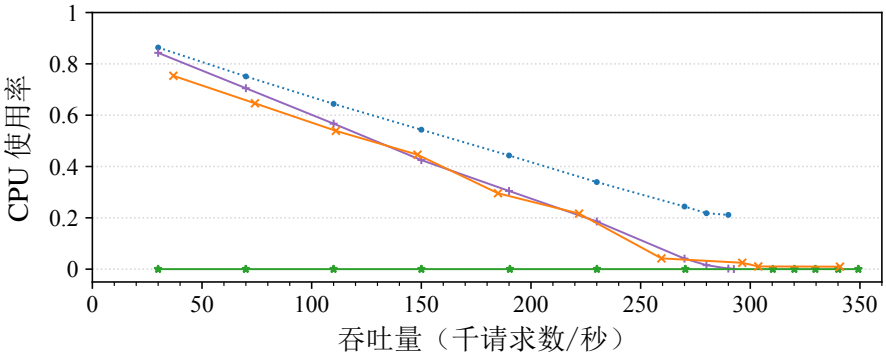
多应用场景。一个通用的调度框架还需支持多应用间的调度以提高 CPU 利用率。为了演示 Skyloft 对多应用调度的支持，本节的第二个实验在一台机器上同时



(a) 单应用场景下的尾延迟



(b) 多应用场景下的尾延迟



(c) 多应用场景下批处理应用的 CPU 使用率

图 3.7　使用合成负载评估各种集中式调度策略的性能

运行一个高优先级的延迟敏感应用，以及一个低优先级的批处理应用（CPU 密集）。

本实验基于上文的集中式调度器额外实现了 Shenango 提出的一种在多应用间分配核心的策略[75]：以任务排队时间作为依据，定期检测全局的请求队列是否拥塞，如果是就向运行批处理应用的核心发送抢占信号，使其让出 CPU 以切换到延迟敏感应用。该策略可以在延迟敏感应用的请求不繁忙时，让大部分核心都去执行批处理应用，以充分利用 CPU 资源，而在峰值负载到来时又能快速响应，以保证低尾延迟。实验的其他配置仍与 ghOSt 以及上文相同，使用同样的合成负载。

图 3.7(b) 显示，即使在后台运行批处理应用，Skyloft 仍能让延迟敏感应用保持与原始集中式策略相近的低尾延迟。与 ghOSt 相比，Skyloft 的最大吞吐量提高了

19%，同时在低负载下的尾延迟也降低了 33%。更重要的是，图 3.7(c) 显示，Skyloft 能在不同的负载压力下使批处理应用具有与 Linux 和 ghOSt 类似的线性 CPU 使用率。相比之下，Shinjuku 无法在多应用间分配资源，导致延迟敏感应用独占了所有 CPU，而使批处理应用的 CPU 使用率始终为零。

3.5.4　真实应用性能

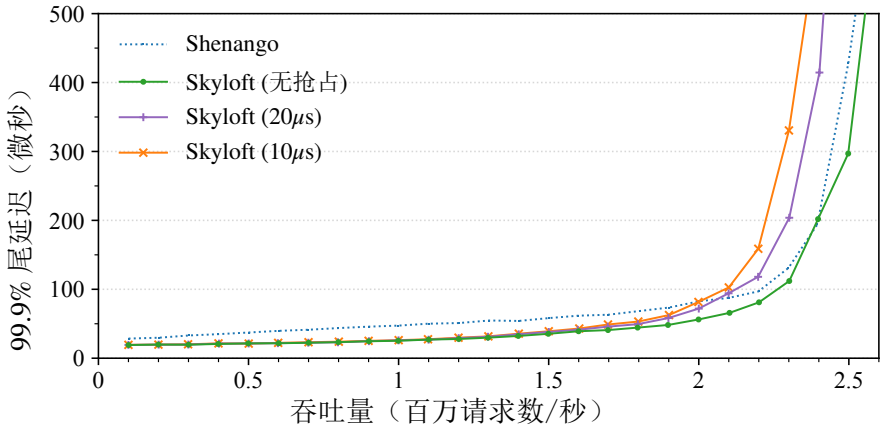
本节将综合评估 Skyloft 在真实世界应用上的调度与 I/O 性能。本实验选用两个流行的应用：Memcached[145]（v1.5.6）与 RocksDB[165]（v6.16.5），并将 Skyloft 与 Shenango 调度系统[75] 进行对比。两种应用的负载类型不同，Memcached 是内存型键值存储，其请求的处理时间短而均匀，符合轻尾分布；RocksDB 是持久型键值存储，不同请求的处理时间差异较大，符合重尾分布。此外，Memcached 已集成了网络后端（使用 TCP 协议），而 RocksDB 只是以库的形式提供，本实验为其包装了一个基于 UDP 协议的网络后端。Skyloft 使用第 3.3.5 节的方法实现了绕过内核的网络功能。实验另使用一台机器作为客户端（见第 3.5.1 节的环境配置），其上运行一个开环负载生成器，按照泊松到达过程生成不同类型的请求负载。

Memcached。对于 Memcached，客户端生成 USR 负载[141]，约有 99.8% 的 GET 请求与 0.2% 的 SET 请求，请求的服务时间均在几微秒左右。针对这样的轻尾负载，Skyloft 实现了与 Shenango 类似的分布式调度策略：在每个核心上进行 FIFO 调度，在核心之间使用工作窃取进行负载均衡，即空闲核心会从繁忙核心的运行队列中窃取一半的任务到自己的队列中。为了使服务器饱和，实验为 Skyloft 与 Shenango 均配置了 4 个工作核心。

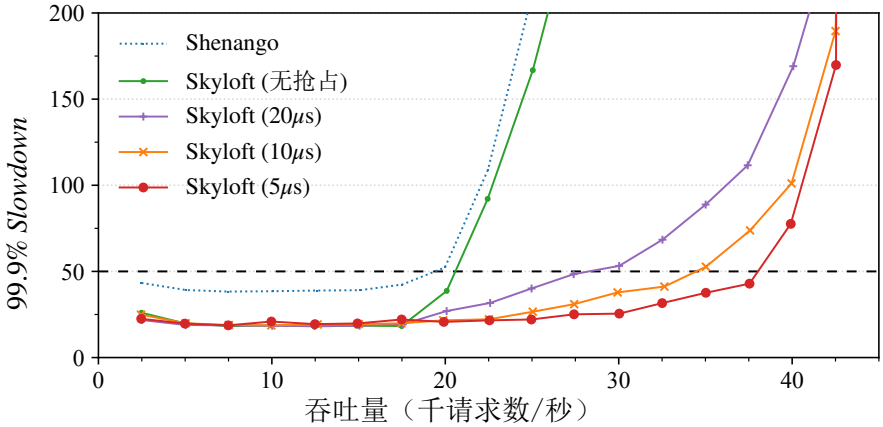
实验结果如图 3.8(a) 所示，Skyloft（无抢占）与 Shenango 的性能相当，最大吞吐量相差不过 2%。在低负载下，Skyloft 比 Shenango 的尾延迟略低，这是因为 Shenango 即使在单应用场景下也需要频繁地让出与唤醒核心，以调整核心分配。

实验还评估了在该负载下启用抢占的效果，这只需在原始调度器中增加对时钟中断的处理而无需其他修改。由图 3.8(a) 可知，在这样的轻尾负载下启用抢占反而会带来负面收益，导致最大吞吐量下降，而且抢占频率越高下降也越严重。这是因为在轻尾负载下不存在长请求阻塞短请求的情况，抢占只会增加额外的开销而无法提高性能。

RocksDB 服务器。对于 RocksDB，客户端生成双峰负载，约有 50% 的 GET 请求与 50% 的 SCAN（范围查询）请求，处理时间分别约为 0.95μs 和 591μs。在该负载下，启用抢占能有效改善短请求的尾延迟。本实验使用 99.9% $Slowdown$ 代替 99.9% 尾延迟作为性能指标，$Slowdown$ 表示请求的总响应时间（包括网络与排队延迟）与原始服务时间的比率。使用尾 $Slowdown$ 而不是尾延迟，能够以统一的



(a) Memcached（USR 负载：99.8% GET，0.2% SET）



(b) RocksDB 服务器（双峰负载：50% GET，50% SCAN）

图 3.8　Skyloft 运行不同类型负载下的真实应用的性能

服务水平目标（SLO）来反映重尾负载下的服务质量，因为请求的绝对延迟差异可能很大。

实验结果如图 3.8(b) 所示，可见该负载下提高抢占频率能有效降低尾延迟并提高最大吞吐量。Skyloft 能够支持低至 5μs 的抢占周期，在给定的 SLO（50×）下的吞吐量为不支持抢占的 Shenango 的 1.9 倍。

3.5.5　微基准测试

用户态抢占开销。本实验先测量了利用用户态 IPI 进行抢占的开销，并且与几种传统的异步通知机制进行对比。实验将发送者和接收者线程绑定到不同的核心上（同一插槽），接收者的事件处理例程设置为空操作。信号以及内核态 IPI 机制最终都需要访问本地 APIC 来发送中断，本实验将 APIC 配置为 x2APIC 模式（通过 MSR 访问而不是 MMIO）。待测的数据包括发送者花费在发送操作上的时间，接收者花费在事件处理上的时间（包含了上下文的保存与恢复），以及事件从发送到被接收的投递时延，结果如表 3.6 所示。

表 3.6　几种抢占机制的时间开销

机制	发送（时钟周期）	接收（时钟周期）	事件投递时延（时钟周期）
信号	1,224	6,359	5,274
内核态 IPI	437	1,582	1,345
用户态 IPI	167	661	1,211
`setitimer`	—	5,057	—
用户态时钟中断	—	642	—

在这几种机制中信号的开销最大，发送者需要进行一次 `kill` 系统调用，进入内核后再向接收者所在的核心发送一个 IPI。信号的接收者在收到 IPI 后会先陷入内核态的中断处理例程，再回到用户态的信号处理例程，处理完成后调用 `sigreturn` 再次陷入内核来返回之前被打断的位置，共需要 4 次用户态与内核态间的切换。内核态 IPI 与用户态 IPI 的开销都很小，虽然这两者的硬件路径基本相同，发送与接收均不涉及特权级的转换，但用户态 IPI 将对 APIC 的访问实现到了硬件中，因此开销更低。

实验还测量了利用用户态时钟中断进行抢占的开销，并与传统的基于信号的用户态定时器（`setitimer`）进行对比。此时抢占事件由当前核心的硬件时钟自动产生，而无需由另一个核心手动发送 IPI 产生。表 3.6 的下半部分给出了实验结果。与 IPI 类似，基于信号的定时器需要多次进出内核，开销远大于用户态时钟中断。值得注意的是，如第 3.3.2 节所述，为接收用户态硬件中断需要在中断处理例程里额外执行一条 `SENDUIPI` 指令（此时 UPID.SN = 1），这大概需要 123 个时钟周期。但即使多执行了一些操作，用户态时钟中断的接收开销仍比用户态 IPI 还少，本文推测其原因可能是时钟中断不需要跨核通信的开销。

用户级线程开销。 Skyloft 通过用户态的线程库实现了低开销的线程操作。本节评估了在 Skyloft 中进行几种常用的线程操作的时间，并与 Linux 原生内核线程（pthread）以及 Go[96] 语言的用户级线程进行比较。实验将所有线程都绑定到同一核心上，以获取准确的上下文切换时间。实验结果如表 3.7 所示，对于不涉及跨应用的大部分线程操作，Skyloft 用户级线程相比其他机制具有最小的开销。

对于跨应用的线程切换，Skyloft 在内核模块中实现。但是由于需要操纵内核的运行队列，即挂起与唤醒内核线程以遵循第 3.3.3 节的单一绑定原则，这比在 Linux 中直接切换两个分属不同应用但都处于可运行态的内核线程稍慢（1,905ns 对比 1,124ns）。但如果 Linux 的线程切换需要唤醒另一线程（例如进程间通信），

表 3.7　几种线程库的线程操作时间

方法	线程创建（纳秒）	互斥锁（纳秒）	条件变量（纳秒）	同应用线程切换（纳秒）	不同应用线程切换（纳秒）
pthread	15,418	28	2,532	898	1,124
Go	503	25	262	108	—
Skyloft	191	27	86	37	1,905

实验测得其切换时间达到了 2,471ns，与 Skyloft 的跨应用切换时间相当。跨应用的切换时间可以通过在内核中提供直接的内核线程切换操作而避免操纵内核运行队列来优化，不过需要对内核进行大量修改。

另一方面，由于 Skyloft 的应用内线程切换开销极低，可以通过合适的调度策略，尽量减少应用间的线程切换，来均摊整体的开销。而对于内核线程，无论切换双方是否属于同一应用，开销的差别均不大。Go 等用户级线程库不支持跨应用切换用户线程。

3.6　讨论

本节简要讨论 Skyloft 的局限性与未来工作。

共享内存安全性。在用户态调度多个应用需要在不同进程之间共享数据，这可能会引发安全问题。例如，恶意的应用程序可以篡改共享的运行队列，从而干预调度决策。不过，由于用户线程的上下文和运行栈仍是应用私有的，不会在应用间共享，因此无法对其他应用造成内存相关的攻击。这一问题可以通过轻量级的硬件隔离机制（例如 Intel MPK[108]）进一步隔离应用与调度器来解决。

用户态时钟重置。目前，Skyloft 对用户态时钟频率的配置是在应用运行之前进行的，这限制了应用只能使用固定的时钟频率而不能随负载变化而动态调整[151]。一个方法是将本地 APIC 的 MMIO 地址空间映射到用户态（需使用 xAPIC 模式），允许应用随时调整时钟频率而无需进出内核。然而，一个潜在的风险是，应用程序也因此可以直接使用 APIC 的其他功能，例如向其他核心发送 IPI，这在非特权模式下是不允许的。为此，可以引入一条新的硬件指令让应用直接修改 APIC 的时钟计数器（例如 `IA32_TSC_DEADLINE` MSR）。

用户态外设中断。 Skyloft 提供了将硬件中断委托给用户态的统一方法（第 3.3.2 节），这也包括了由外设产生的中断。在用户态处理 I/O APIC 或 MSI（Message Signaled Interrupts）中断，可为现有的用户态设备驱动提供中断支持，避免使用轮询的方式而浪费 CPU 资源。

3.7　本章小结

本章提出了 Skyloft 库操作系统，基于用户态微虚拟机，将主机操作系统的线程管理、调度、网络等功能都卸载到用户态并进行高效地实现，同时支持由应用程序来灵活定制调度策略。Skyloft 利用 Intel 用户态中断技术，实现了低开销的处理器间中断与时钟中断处理，为用户态调度器提供了微秒级的抢占支持，从而优化重尾分布下的尾延迟。Skyloft 通过让用户态调度器支持跨应用的线程切换，来实现多应用的统一调度，从而提高 CPU 利用率。实验结果表明，Skyloft 能针对不同的应用场景，方便地配置不同的调度策略，性能也优于已有系统。

第 4 章　特权态微虚拟机的组件化操作系统设计与性能优化

4.1　本章引言

除了绕过内核、消除系统调用的上下文切换开销外，专用的操作系统也是微虚拟机提升性能的重要途径。专用化（specialization）通过对功能进行按需实现，以及针对不同的场景选用不同的实现，形成极简的、可定制的操作系统，以获得最佳性能。

然而，现有的微虚拟机操作系统，在专用化的效果与效率上存在不足。它们一般采取以下三种方法：一是在已有通用操作系统的基础上进行裁剪与改造[50,62-63,70-71]，但是因为原系统模块的紧耦合性，专用化程度不够高；二是从头开发新的专用操作系统[19,26,166-167]，但是需要大量的专家知识与开发成本，效率不高；三是采用组件化的设计方法对功能进行组合[22,28,47-48]，但是在组件的易用性与可重用性上仍有待提高。

另一方面，现有微虚拟机操作系统在接口设计上难以同时兼顾性能与兼容性。POSIX 接口为应用程序提供了最大程度的兼容性，但在单地址空间的微虚拟机场景下存在许多冗余的调用路径，无法获得最佳性能。而使用针对性能进行优化的接口，又会导致难以兼容已有应用。

本章认为，是编程语言阻碍了操作系统的专用化发展。现有的通用操作系统或微虚拟机操作系统，大多采用 C 语言编写，导致功能模块紧耦合，难以裁剪、改造、替换与重用。近年来新兴的 Rust 语言，具有诸多高级语言的特性，以及与 C 相媲美的性能，非常适合用于实现专用化的微虚拟机操作系统。然而，目前虽然也有不少用 Rust 语言实现的微虚拟机操作系统[22,168]，但是它们并没有充分发挥 Rust 以及微虚拟机的优势，在性能上反而不如用 C 编写的系统。

为了更好地利用专用化提升微虚拟机的性能，本章提出了 ArceOS 微虚拟机操作系统。ArceOS 利用 Rust 语言的特性，针对微虚拟机的特点，在组件化设计与接口设计两方面进行了优化。在组件化设计中，ArceOS 通过将组件划分为操作系统相关与操作系统无关两类，来减少组件对操作系统设计理念上的依赖，以降低耦合性，提升可重用性（第 4.3.2 节），同时也借助 Rust 语言提供的机制实现了操作系统功能的灵活定制（第 4.3.3 节）。ArceOS 还利用组件化设计带来的灵活性，不仅可以作为一个 Unikernel 运行于特权态微虚拟机中，还能形成宏内核、虚拟机管理程序等多种内核形态（第 4.3.5 节）。

在接口设计中，ArceOS 借用了 Rust 标准库的接口，提供了快速路径，避免了

微虚拟机场景下使用 POSIX 接口带来的冗余路径，提升了性能。同时还能直接支持已有的 Rust 应用，提供了足够的兼容性（第 4.3.4 节）。

目前，ArceOS 提供了 30 多个可重用组件与 10 多个松耦合模块，实现了多核、多线程、网络、文件系统、图形显示等基本内核功能，支持 x86_64、RISC-V、AArch64 等主流处理器架构，QEMU/KVM、x86 服务器、树莓派、黑芝麻等多种硬件平台，以及 Virtio、Intel 82599 万兆网卡等设备驱动（均可根据应用需求进行灵活定制），总代码量达 3 万多行。实验结果表明，ArceOS 的组件化设计方便了操作系统功能的定制，ArceOS 的 API 快速路径设计比使用 POSIX 的接口快 50% 以上（文件读写）。ArceOS 同时兼容已有 C 应用程序，与 Linux 相比，使 Redis 的延迟降低了 33%。

本章的主要贡献如下：

1. 提出了一种基于设计理念相关性的组件化操作系统设计方法，能够降低组件间的耦合性，并提供良好的可定制性与可重用性。
2. 提出了一种基于 Rust 语言特性的 API 快速路径设计，专门针对微虚拟机场景进行了性能优化，同时也不失兼容性。
3. 实现了一系列可重用的组件，能够快速搭建出满足应用需求的多形态操作系统，支持已有的 C 或 Rust 典型应用。
4. 通过实验展示了组件化以及 API 快速路径的优化效果，并与 Linux 以及已有的类似工作进行了细致的对比。

4.2　研究动机

专用化是提升操作系统性能的重要途径。本章先介绍利用微虚拟机实现专用化的基本方法，以及面临的挑战，然后讨论了如何利用组件化的设计方法以及新型编程语言来应对这些挑战，最后给出了本章系统的设计目标。

4.2.1　操作系统专用化

微虚拟机只运行一个应用程序，而且可以在应用中实现部分操作系统的功能，非常适合根据应用与场景的特点，对操作系统的功能进行深度定制，从而获得最大的性能。这种专用化主要体现在两方面：

（1）极简性，即只实现必须要实现的功能，去除任何不需要的功能。这不仅可以减小镜像大小、提升启动速度、减小攻击面，还能避免过多地考虑通用性，简化实现方式，带来更多优化机会。例如，对于只运行单个应用的微虚拟机，应用与内核的特权级隔离、地址空间隔离都是多余的；对于事件循环驱动的单线程服务器

应用，抢占与调度是多余的；对于基于 UDP 的低延迟网络应用，完整的 TCP 协议栈是多余的，而且还可以通过消除 VFS 层的多态性来缩短调用路径[71]。

（2）可定制性，即提供同种功能的多种实现。即使是同一功能，在不同的场景下，使用不同的算法或实现方式，也会有不同的性能表现，这使得应用程序能够根据需要选择最佳的实现。例如，本文第 3 章重点关注的任务调度，在不同的任务负载下有各自适合的调度策略；不同的内存分配算法也对应用的性能有很大影响[28]；对于设备驱动，也可以根据应用的需要进行配置，在性能更高的轮询模式与 CPU 利用率更高的中断模式之间进行取舍。

4.2.2　实现专用化的挑战

专用化的终极目标是能够以很小的代价为每个应用快速生成出一个专用的操作系统。然而，即便是在微虚拟机这样的轻量场景下，实现以上目标也绝非易事。本节分析了其中的一些关键挑战。

操作系统模块紧耦合。为了实现专用化，操作系统的功能模块需要能被方便地移除或替换。然而，在以宏内核为代表的传统操作系统设计中，各个功能模块间存在错综复杂的函数调用与数据访问关系[28]。如果要删除其中一个模块，或替换为另一种实现，必须仔细考虑这些依赖关系，确保其他模块不受影响。因此难以对它们进行专用化改造。

现有的一些微虚拟机操作系统基于对已有通用操作系统的改造[50,62,70-71]，但因为原系统模块的紧耦合性，专用化程度不够高，例如只能做到去除特权级分离与多地址空间，其余大部分功能仍保留原系统的设计。

操作系统模块难以重用。另一条实现专用化的途径是从头开发新的操作系统。但是操作系统的开发门槛较高，不仅需要掌握相关软硬件技术，还要考虑诸多实现细节，这对于只为了运行某个应用的普通用户来说难以接受。另一方面，虽然开发操作系统的工作量大，但是过程具有重复性。例如，几乎所有的操作系统都要实现页表操作、内存分配等基本功能，或是链表、平衡二叉树等基础数据结构。在开发新操作系统时理应可以重用其他系统中的相关代码，以减少开发代价。

然而，由于传统操作系统的自包含性，其功能模块只考虑了为自己服务，很难被其他软件重用。再加上传统操作系统模块的紧耦合性，如果要将一个模块拆出给其他软件使用，就不得不处理模块间复杂的依赖关系。

已有一些微虚拟机操作系统，重用了成熟操作系统的部分代码[27,63,69]，但是这种重用的粒度较大，如完整的网络协议栈或驱动程序，使得其中仍保留大量原系统的影子，专用化程度不够高。还有一些工作采用组件化的设计方法，通过对内核组件的组合来形成满足需求的专用操作系统[22,28,47-48]，从而减少开发代价，但

是这些组件与原系统的设计较为耦合，难以被广泛重用。

接口设计难以兼顾性能与兼容性。另一个阻碍专用化效果的因素是接口的设计。现有的许多微虚拟机操作系统[27-28,49-50]，为了能够兼容已有的应用程序，选择使用 POSIX[169] 这样的原通用操作系统中的接口。然而，这些接口原本是为宏内核而设计的，考虑了诸多宏内核下的特点，如地址空间隔离，在微虚拟机场景下存在冗余的调用路径，从而限制了从应用到内核端到端的功能专用化。但如果使用面向性能的专用接口[19,22,26,32]，又会面临应用兼容性的问题。

4.2.3　编程语言的考虑

为解决以上挑战，采用组件化的操作系统设计方法仍是一个值得深入研究的方向。为了更好地实现一个组件化的操作系统，需要重新思考操作系统的架构设计，乃至使用的编程语言。先前的工作难以实现专用化，或存在一些不足，在很大程度上可以通过使用更现代的编程语言代替原来的 C 语言而解决。近年来，Rust[97] 语言越来越流行，而且在编写操作系统等底层软件上很有吸引力[21-23,98-99,168]。本文认为 Rust 语言非常适合用于实现微虚拟机场景下的专用操作系统。本节先分析 C 语言在操作系统专用化方面的不足，然后介绍 Rust 语言在该方面的优势。

C 语言的不足。C 语言具有简洁、高效、灵活等特点，被认为是编写操作系统的首选语言。然而，也正是因为 C 语言的高度灵活性，导致这些操作系统中存在许多混乱的、不利于维护的代码，例如类型强制转换、随意访问其他模块的函数或数据等。正是这种过度的灵活性导致了软件模块的紧耦合，特别是对于操作系统这样的大型软件来说。此外，C 语言也没有提供一些现代高级语言的特性，如面向对象、模块封装、泛型、包管理器等，使得用 C 语言编写的功能模块难以被其他软件重用。C 语言的简单数据类型也限制了模块间接口的设计。更不必说 C 语言在类型、内存、并发等方面的不安全性，容易导致缓冲区溢出、use-after-free、数据竞争等种种安全漏洞[170]。

Rust 语言的优势。 Rust 语言最具特色的机制是基于所有权的内存安全管理，使得无需垃圾回收等运行时开销，就能实现自动内存管理与安全检查，从而提供与 C/C++ 相当的性能。此外，Rust 还具有许多高级语言的特性，如类型安全、泛型与接口、函数式、枚举模式匹配等等，可以带来极大的开发便利性。同时，Rust 对底层代码的控制能力，使得它非常适合用于编写操作系统等底层软件。下面主要介绍 Rust 有助于实现专用的微虚拟机操作系统的几大特性。

（1）良好的包管理机制。Rust 通过包（crate）的形式组织代码模块，每个包都有一个配置文件，指定了其所依赖的其他包。开发者还可以从官方的包发布平台

crates.io① 中寻找与下载自己所需的包，或将自己开发的包发布出来供其他软件使用。目前 crates.io 中已有超过十万个包[171]。Rust 的包管理机制为组件化的操作系统设计提供了便利，可以将操作系统的功能模块以包的形式组织，从而显式指定它们的依赖关系，避免随意调用，降低耦合性。此外，包发布平台也方便组件的重用（第 4.3.2 节）。

（2）灵活的条件编译。Rust 可在配置文件中，为每个包指定一组特性（feature）。特性可在编译时选择是否启用，从而决定是否编译某段代码。特性还能用于配置包的依赖关系，或者通过包的依赖关系进行传递。对于专用操作系统，这种灵活的条件编译机制，既能方便根据需求选择性启用某些功能，又能方便在同种功能的多种实现间进行切换（第 4.3.3 节）。

（3）基于高级类型的库函数接口。Rust 在 `std` 标准库中提供了对操作系统级功能的访问，例如线程、文件、网络等。与 C 的文件描述符不同，Rust 的标准库接口使用专门的数据类型对这些系统资源进行封装，并通过资源获取即初始化（RAII）机制，由编译器保证这些资源在其生命周期结束时被释放。这种接口既避免了文件描述符转换的开销，又避免了资源泄漏或多次释放。然而，在 Linux 等通用操作系统上，Rust 仍会依赖 libc 库（使用 POSIX 接口）实现对系统级资源的访问，反而增加了开销。但在微虚拟机场景下，可以完全利用这套接口代替传统的 POSIX 接口，既能提供更好的性能与安全性，又能直接支持已有 Rust 应用（第 4.3.4 节）。

4.2.4　本章设计目标

本章的总体目标是能够快速定制出满足应用需求的高性能高安全微虚拟机操作系统。为此，本章认为，使用组件化的设计方法是实现这一目标的关键。为了解决已有组件化设计存在的问题，本章希望能够基于 Rust 语言的优势，结合微虚拟机场景的特点，重新思考组件化操作系统的设计方法。本章具体的目标包含以下三方面：

1. **组件可重用**：组件不应该与特定内核的设计和实现绑定，需要能够方便地被其他应用或内核重用。
2. **组件灵活定制**：应用程序需要能够对组件的功能与组合方式进行方便地配置，以实现专用化所需的极简性与可定制性。
3. **接口专用化**：应用程序调用内核的接口也需要专用化以提供最佳性能，但同时还要保证足够的兼容性。

① https://crates.io

4.3　系统设计

本节先介绍 ArceOS 微虚拟机操作系统的整体架构，然后介绍 ArceOS 的设计细节，包括基于操作系统设计理念相关性的可重用组件化设计、灵活的组件功能定制方法、基于 Rust 语言特性的 API 快速路径优化，最后演示了利用组件化设计实现的另一特色功能：构建多种形态的内核。

4.3.1　整体架构

ArceOS 以特权态微虚拟机作为基本形态，即 Unikernel，仅支持一个应用程序，应用与内核在构建时被链接在一起生成单个镜像，共同运行在客户机特权态，可以直接管理特权硬件资源，而依赖底层的微虚拟机管理程序进行安全隔离。通过不同的配置选项，ArceOS 可以构建为其他形态的内核，如宏内核或虚拟机管理程序，直接运行在裸机上（见第 4.3.5 节）。不失一般性地，ArceOS 的设计思想与优化技术也可用于用户态微虚拟机（如第 3 章介绍的 Skyloft 库操作系统）。

图 4.1 描述了 ArceOS 的系统架构。根据依赖关系，自底向上，ArceOS 可划分为以下几个层次：

- **元件层**：与操作系统的设计无关的部分，可方便被其他系统软件复用。
- **模块层**：与操作系统的设计相关的部分，它们与 ArceOS 的设计较为耦合，不太容易被其他系统软件复用。元件层与模块层集中了 ArceOS 的大部分组件，可认为是传统意义上的“内核”，用来管理硬件并为应用提供服务。
- **API 层**：将模块层的功能封装为 API 的形式，以供应用程序调用。ArceOS 为 Rust 和 C 语言编写的应用程序分别提供了不同的 API，并针对语言的特点做了快速路径优化。
- **用户库层**：通过对 API 层的进一步封装，提供对已有用户库的兼容（如 Rust 标准库、libc 库），以便应用程序的移植。
- **应用程序层**：目前 ArceOS 提供了对 Rust 和 C 这两种语言编写的应用程序的直接支持。

4.3.2　可重用组件化设计

传统的操作系统，虽然大多也采用“模块化”的思路进行功能单元的组织。但这种模块的划分粒度较大，并没有清晰的边界，使得模块间可以随意调用，耦合程度严重。而且这些模块也难以被其他软件重用。例如，为 Linux 实现的设备驱动大量依赖 Linux 的内核 API，如果要将其移植到其他系统，需要在其他系统中也支持这些 API，这几乎覆盖了 Linux 的所有子系统。

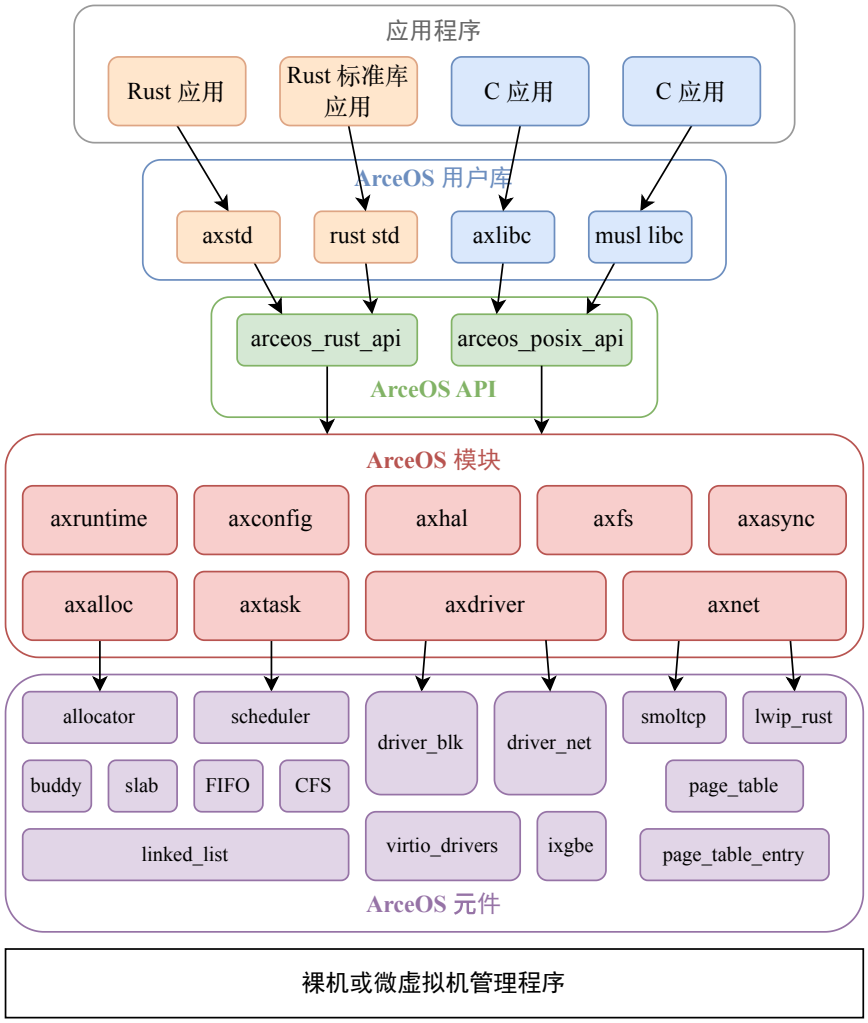


图 4.1　ArceOS 系统架构

为了让操作系统的功能单元具有更广泛的适用范围，需要从模块的划分方式入手重新设计。ArceOS 的基本设计思路是按照更小的粒度，以及是否与操作系统的设计理念相关这两个依据进行功能划分。本文将这些粒度较小的、为了可重用性设计的功能单元称为组件。

操作系统无关与操作系统相关。ArceOS 为提升组件可重用性的重要方法是根据与操作系统的设计理念的相关性进行组件划分。操作系统虽然种类繁多，实现各不相同，但是有的功能单元是几乎所有的操作系统都需要实现的，而且无论操作系统具有什么形态与设计理念，都不会对这些功能单元做太多调整，本文称它们为操作系统无关的组件。这些组件位于依赖关系的底层，对其他组件的依赖较少，但可能会有大量其他组件依赖它。例如，像链表这样的基础数据结构，或各种调度算法的实现。除此之外，本文称剩余的功能单元都是操作系统相关的组件，它们为某个特定的操作系统而打造，并与其设计理念绑定，当用于其他操作系统时可能需要进行重大修改，因此可重用性相对较差。这些组件位于依赖关系的较

高层，一般是对其他组件的组合，以实现更复杂的功能。例如，实时操作系统有自己特有的任务管理方式，以满足高实时性要求，但不一定适用于其他操作系统。表 4.1 总结了这两类组件的区别，并例举了一些组件示例。

表 4.1　ArceOS 组件类别

类别	操作系统相关/无关	依赖关系层次	可重用性	举例
元件	无关	底层	好	基础数据结构与算法 硬件相关操作 具体设备驱动 具体文件系统 网络协议栈
模块	相关	高层	差	系统配置与初始化 异常与中断处理 任务管理与调度 虚拟文件系统抽象 设备驱动层抽象

在 ArceOS 中，把操作系统无关的组件称为*元件*，把与操作系统相关的组件称为*模块*，元件层与模块层共同构成传统意义上的内核，提供操作系统的基本功能。ArceOS 利用 Rust 语言的包管理机制来实现组件化，组件都以 Rust crate 的形式提供，具有清晰的边界与依赖关系。此外，Rust 语言的单一所有权机制，以及对可变全局变量的使用限制（除非使用 `unsafe` 代码），使得状态都是组件私有的，不会在组件间共享，容易实现组件的拆分与替换。通过 Rust 的包管理平台 crates.io，可以方便实现组件的重用与被重用。

元件层在设计上需要更多地考虑可重用性，仔细设计对外的接口，减少对其他组件的依赖，并避免与 ArceOS 的设计产生耦合。ArceOS 的最终目标是让整个操作系统都是由可重用的元件构成的，然而，由于元件设计上的复杂性，一些功能暂时无法与操作系统的设计理念解耦，就需要暂时以模块的形式存在。因此，一种循序渐进的开发方式是，逐步将操作系统的功能从模块拆解到元件中，减少模块的数量，增加元件的数量。

有序的组件依赖。影响组件可重用性的另一因素是组件间的依赖关系。这种依赖关系包括在另一组件中实现的函数、定义的数据结构。由于在重用一个组件时，需要同时包含其依赖的所有组件，混乱的依赖关系容易导致引入过多无关紧

要的组件，不仅违背了专用化的极简性原则，也导致了“牵一发而动全身”的情况。例如 C 语言没有提供语言层面的组件概念，可以通过外部函数声明的形式调用任意函数，造成了无序的跨组件调用，难以重用。为此，ArceOS 规定组件间需形成有序的依赖关系，即遵循以下原则：

1. *明确定义依赖关系*。一个组件可调用的组件列表，以及可使用的功能（函数或数据结构）需要被明确给出，不能随意调用不在列表中的功能。
2. *无反向依赖*：下层组件不能反向调用上层组件（如元件层不能调用模块层）。
3. *无循环依赖*：组件不能相互依赖，整体的依赖关系需要形成有向无环图。

在 ArceOS 中，组件的依赖关系通过 Rust crate 的配置文件（Cargo.toml）指定，只能调用在依赖列表中的组件的功能（或 Rust 核心库 `core`），可以避免随意进行跨组件调用。ArceOS 通过规定元件只能调用元件，模块只能调用元件或模块来避免反向依赖，防止重用下层组件时需要同时包含上层组件。此外，Rust 的包管理机制也明确禁止了循环依赖，使得依赖关系清晰。

然而，由于操作系统固有的复杂性，有些时候仍然难以避免一些反向或循环的组件依赖。例如，自旋锁的实现与操作系统的设计关联不大，应该被放到元件层。一个实现正确的自旋锁需要在进入临界区前关中断或关抢占，并在出临界区时重新打开，以防当前线程在临界区被调度走而影响正确性。但是，“关抢占”这一操作无法在元件层单独实现，因为需要依赖模块层中的线程管理功能，而线程管理模块与操作系统的设计理念较为相关，不适合移入元件层中。

为此，ArceOS 专门提供了一个元件 crate_glue，用于在不得已时提供*接口定义明确*的反向或循环调用。该元件将依赖双方分为功能的调用者与实现者，位于下层的调用者会声明一套所需功能的接口，位于上层的实现者会对其声明的接口进行具体实现。两者只需依赖 crate_glue 元件，而无需直接形成反向或循环的依赖。这种方式，虽然在本质上还是存在组件的循环依赖或反向依赖，但对这些调用关系进行了特殊标记，避免了开发者的滥用与不经意的使用，有利于日后的分析与检查。表 4.2 列出了目前 ArceOS 使用 crate_glue 解决反向或循环依赖的情况。

4.3.3　组件灵活定制

专用化要求能根据应用的需求构建出最简的、可定制的操作系统。ArceOS 利用 Rust 的“特性”（feature）机制，提供组件的可选择性与可替换性，以实现专用化。

基于“特性”的组件定制。图 4.2 展示了通过特性对组件定制的一个例子。在 ArceOS 的 allocator 组件（内存分配）的配置文件中，可以指定三种特性，分别对应不同的内存分配算法（TLSF[172]、slab[173]、伙伴系统[174]）。内存分配算法的具体

表 4.2　ArceOS 组件间不可避免的反向或循环依赖关系

功能调用者	功能实现者	调用关系	功能描述
axlog（日志输出）	axruntime（运行时）	模块 → 模块	输出带时间的日志
axhal（硬件抽象层）	axruntime（运行时）	模块 → 模块	中断与异常处理例程
axhal（硬件抽象层）	应用程序*	模块 → 应用	宏内核形态的系统调用
axfs（文件系统）	应用程序*	模块 → 应用	应用自定义的文件系统
自旋锁	axtask（任务管理）	元件 → 模块	线程抢占的关闭与开启
互斥锁	axtask（任务管理）	元件 → 模块	线程的睡眠与唤醒

* 指运行于内核态的 ArceOS 应用程序层，而非传统的运行于用户态的应用程序（见第 4.3.5 节）。

实现在 allocator 的依赖组件中，并可通过启用不同的特性来进行选择。在 allocator 的代码实现中，也会根据特性来条件编译，分别使用相应的依赖组件实现不同内存分配算法间的切换。这些依赖组件都是可选的，只要不启用相应的特性就不会对它们进行编译，从而不被包含进最终的镜像中。此外，特性还可以通过组件的依赖关系进行传递，如在图 4.2 的第 5 行可以通过 allocator 组件的特性来启用其依赖的 rlsf 组件的特性。

```
1 [package]
2 name = "allocator"
3
4 [features]
5 alloc_tlsf = ["rlsf/foo"]
6 alloc_slab = ["slab_allocator"]
7 alloc_buddy = ["buddy_system_allocator"]
8
9 [dependencies]
10 rlsf = { version = "0.2", optional = true, features = ["foo"] }
11 slab_allocator = { path = "../slab_allocator", optional = true }
12 buddy_system_allocator = { version = "0.9", optional = true }
```

图 4.2　ArceOS 基于 “特性” 的组件定制

Rust 的编译工具链会根据所启用的特性，选择相应的组件（同时进行条件编译）进行组合，来形成最终的操作系统镜像。为了提供最佳性能，组件的选取与组合在编译期间静态完成，无运行时的开销。Rust 编译器会对软件栈上的所有组件进行编译，包括应用程序、用户库、被选择的模块与元件，生成一系列静态库（rlib 格式）。最后，由链接器将这些组件静态链接，生成最终的统一镜像，并支持链接时优化（LTO）。

基于组件灵活定制的专用操作系统演示。下面以最简单的 helloworld 应用程序为例，演示 ArceOS 是如何使用最少的组件，组装出一个仅支持输出功能的最简

```rust
fn main() {
    println!("Hello, world!");
}
```

```
[dependencies]
axstd = { path = "../../ulib/axstd" }
```

(a) 最简形式

```rust
fn main() {
    println!("{}",
        String::from("Hello,")
        + " world!");
}
```

```
[dependencies]
axstd = { path = "../../ulib/axstd", features = [
    "alloc",
    "alloc_slab", # "alloc_tlsf", "alloc_buddy"
] }
```

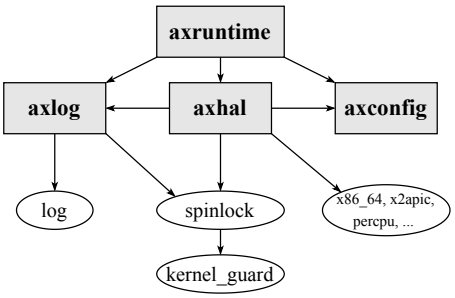
(b) 动态内存分配

```rust
fn main() {
    std::thread::spawn(|| {
        print!("Hello,");
    }).join().unwrap();
    println!(" world!");
}
```

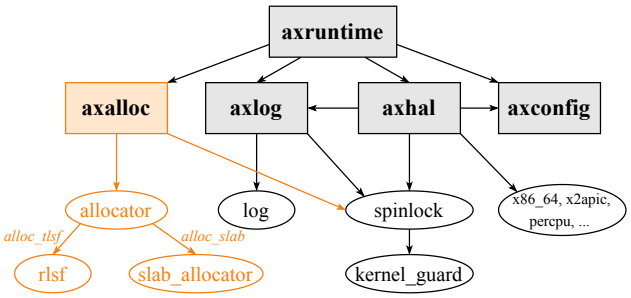
```
[dependencies]
axstd = { path = "../../ulib/axstd", features = [
    "multitask",
    "sched_cfs", # "sched_fifo", "sched_rr"
] }

```

(c) 多线程

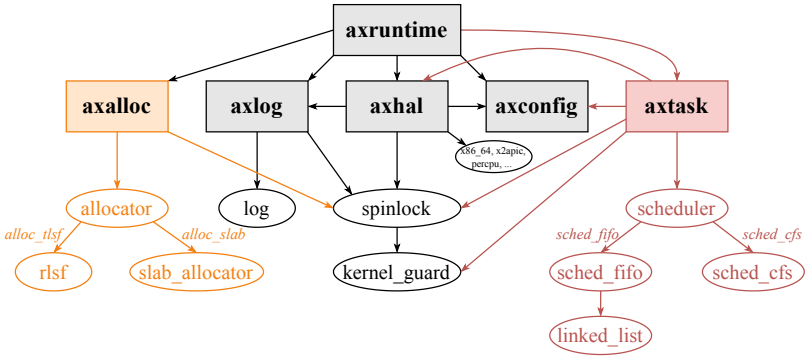
图 4.3　不同功能需求下的 helloworld 应用程序及其配置文件



(a) 最简形式



(b) 增加动态内存分配功能（启用 “alloc” 特性）



(c) 增加多线程功能（启用 “multitask” 特性）

图 4.4　不同功能需求下 helloworld 应用程序的组件依赖

操作系统。然后演示当应用的功能增加时，如何通过修改配置文件，快速构建满足应用需求的最简操作系统。

图 4.3(a) 是一个最简单的 helloworld 应用程序，仅完成输出一行字符串的功能。为了支持该应用程序的运行，需要针对运行的平台准备相应的执行环境，并提供 “输出” 的功能（如串口）。如图 4.4(a) 所示，ArceOS 最少需要四个模块层的组件来支持该应用的运行，这四个模块也是支持应用运行的最小组件集合。其中，最主要的是模块是 axruntime 与 axhal。axruntime 用于为应用建立执行环境，对引入的各组件进行初始化。axhal 是硬件抽象层，为各种硬件相关操作提供了统一的接口。此外，axconfig 定义了平台相关常量与内核参数，如物理内存范围、内核加载基地址、栈大小等。axlog 用于输出内核日志消息。

模块层的组件还会调用元件层的组件。如 axhal 会使用 x86_64、x2apic 这样的对具体硬件进行操作的组件；axhal 和 axlog 都会调用 spinlock 来使用自旋锁；spinlock 还会调用 kernel_guard 进行关中断或关抢占①。

当应用程序的功能需求增加时，会引入更多的组件，如图 4.3(b) 中的应用需要动态内存分配功能才能完成字符串拼接操作。为此，ArceOS 提供了一个 “alloc” 特性，应用在配置文件中启用该特性后，会引入一些额外的组件来实现动态内存分配功能（如图 4.4(b) 中的 axalloc 等）。类似地，如果应用需要使用多线程功能（图 4.3(c)），则可以启用 “multitask” 特性，并额外引入 axtask 等组件（图 4.4(c)）。其他可选组件还有设备驱动（axdriver）、网络（axnet）、文件系统（axfs）等。

应用程序通过指定不同的特性，来选择同一功能的不同实现。如图 4.3(b) 右侧，应用可在配置文件里启用不同的特性来指定相应的内存分配算法，并引入相应的组件依赖（图 4.4(b)）。图 4.3(c) 和 4.4(c) 也展示了应用对不同调度算法的配置及相应的组件依赖。

4.3.4　API 快速路径

ArceOS 利用了 Rust 的高级语言特性，为 Rust 应用程序调用内核的功能提供了快速路径，以缩短传统接口的调用路径，提高性能。

传统接口的不足。传统的通用操作系统，或一些微虚拟机操作系统，为了兼容已有应用，仍使用 POSIX 这样的接口作为应用程序调用内核的接口。POSIX 接口为宏内核而设计，考虑了诸多宏内核下的特点，但在微虚拟机场景下限制了灵活性，也存在许多冗余的调用路径。例如，宏内核的应用与内核处于不同地址空间不同特权级，应用需要通过系统调用访问内核。由于隔离性，系统调用的参数不能通过栈来传递，因此对参数的数量有限制（一般不超过 6 个），使得接口的参数

① 为了简洁性，这里省略了用于处理反向与循环依赖的 crate_glue 组件。

与数据类型不能太复杂。此外，宏内核对安全性的要求也使得内核不得直接访问存储在应用内存中的数据，因此内核在给应用返回一个对象时，需要使用一个整数编号作为抽象（如线程编号 pid 或文件描述符 fd），带来编号分配与查找的开销。另一方面，为减少系统调用次数，许多 POSIX 接口都提供了批处理的版本（如向量读写 `readv`、`writev`），这在微虚拟机场景下也是不必要的。反之，如果完全抛弃 POSIX 接口，重新设计一套针对微虚拟机优化的接口，又会导致无法兼容已有应用，带来不小的移植代价[19,22,26]。

以文件读写为例，如图 4.5(a) 所示，在 Linux 上 C 语言编写的应用程序要读取一个文件的内容时，使用文件描述符作为内核文件对象的中间表示。这不仅需要内核为每个进程维护一份文件描述符表，还会在文件描述符的分配与查找时带来不小开销，特别是在多核场景下严重影响可扩展性[8]。

```c
int fd = open("foo.txt", O_RDWR);
int len = read(fd, buf, 100);
close(fd);
```

(a) C

```rust
let mut f = File::open("foo.txt")?;
let len = f.read(&mut buf)?;
// drop(f);
```

(b) Rust

图 4.5　C 和 Rust 语言的文件操作接口

针对 Rust 应用的 API 快速路径。 ArceOS 希望能够提供一套接口，既能针对微虚拟机的特点进行优化，又能带来足够的兼容性。为此，ArceOS 借用了 Rust 的标准库 API，作为 Rust 应用访问内核服务的接口。如图 4.5(b) 所示，Rust 应用在进行文件操作时，采用了资源获取即初始化（RAII）风格的接口，在打开文件时会创建一个文件对象，并在其生命周期结束后自动释放以关闭文件。这不仅消除了文件描述符带来的间接开销，还让开发者无需自行考虑文件的关闭，避免了资源泄漏或多次释放。

在传统的宏内核下，难以将此类接口直接作为应用访问内核的接口。因此，现有的通用操作系统或微虚拟机操作系统，对 Rust 应用的支持都是通过 libc 实现的，Rust 标准库在内部仍会调用 libc 的库函数，使用传统的 POSIX 接口来访问内核，反而使 Rust 应用的调用路径比 C 还长。但微虚拟机让直接使用这样的接口成为可能。由于应用程序与内核处于同一地址空间，能够相互调度与访问各自的所有函数与内存，因此可以让 `File::open` 直接返回内核的文件对象，保存在应用的栈上，允许应用直接对其进行操作。

除了能消除 POSIX 接口带来的冗余路径外，使用 Rust 标准库的 API 还能利用 Rust 的众多高级语言特性。仍以图 4.5(b) 中的文件操作为例，Rust 的所有权机制确保了该文件对象只能在同一个线程中被访问，因此可以在编译期间检查出并发访问的缺陷，无需在内核中进行额外的运行时检查。此外，可以使用闭包、泛型等

高级数据类型作为接口的参数，以及使用带 async 关键字的异步函数[157]，为应用提供更多灵活性，并且无接口转换的开销。使用 Rust 风格的函数作为 API，还能实现从应用程序到用户库，再到内核的端到端 Rust 调用，中间不存在外部语言接口[175]（FFI）的转换，不会丢失参数的类型信息，便于 Rust 编译器进行全局优化与安全检查[22]。

使用 Rust 标准库的接口还能一定程度解决应用兼容性的问题，因为可以直接支持已有的 Rust 应用程序。由于现在越来越多的新应用开始使用 Rust 语言编写，也有不少旧应用使用 Rust 语言进行重写[176]，这种方式可以带来足够的兼容性。

针对 C 应用的 API 快速路径。除了面向 Rust 应用的快速 API 外，ArceOS 也支持 POSIX API，以兼容已有的 C 应用程序。ArceOS 为 C 语言也提供了一定程度的快速路径，但由于 POSIX 接口的限制，效果不如 Rust。

以线程创建为例，当 C 应用程序要创建线程时，在 Linux 或一些 POSIX 兼容的微虚拟机操作系统上都是通过 libc 的 pthread_create 接口实现的，其内部会先使用 sys_mmap 系统调用分配线程的私有内存（栈与 TLS），再调用 sys_clone 完成线程的最终创建。然而，这些系统调用为了通用性，其参数类型与 pthread_create 存在较大差异，使得需要对传入的参数进行层层处理，延长了调用路径，带来了额外的开销（见第 4.5.2 节的实验评估）。

ArceOS 通过使用库函数级别的接口（API）代替系统调用级别的接口（ABI），以缩短 C 应用程序对内核服务的调用路径。ArceOS 的用户库提供了与 pthread_create 接口一致的 sys_pthread_create，来直接创建一个线程，而不再使用 sys_clone 系统调用。库函数级别的接口还有助于简化用户库的实现。例如，对于 libc 中的 malloc 接口，可通过 sys_alloc 直接调用内核的

表 4.3　ArceOS API 与 Linux 系统调用的差异

	Linux	ArceOS C 应用	ArceOS Rust 应用
创建线程	sys_clone 创建内核线程，libc 仍需维护自己的线程结构	提供与 pthread_create 一致的接口，用户库无复杂实现	提供与 Rust 标准库一致的接口，用户库无复杂实现
动态内存分配	libc 自行维护分配器，通过系统调用扩充容量	提供与 malloc 一致的接口，用户库无复杂实现	注册全局分配器后，无需显式接口与用户库实现
互斥锁	sys_futex 创建内核 futex 结构，等待时需进内核查找对应结构	提供与 pthread_mutex_* 一致的接口，用户库无复杂实现	内核返回等待队列结构，等待时直接调用
打开文件/套接字	内核返回文件描述符	内核返回文件/套接字结构，用户库维护文件描述符表	内核直接返回文件/套接字结构
使用文件/套接字	系统调用使用文件描述符作为参数，内核查找对应结构	将文件描述符作为参数，用户库根据描述符查找对应结构	将文件/套接字结构直接作为参数

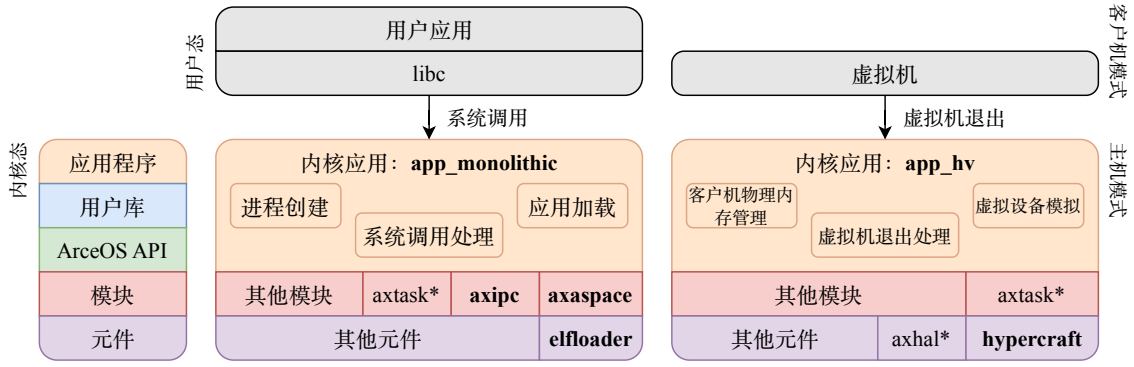
分配器实现动态内存分配，而无需在用户库中自行实现堆的管理（使用 `sys_brk` 或 `sys_mmap`）。

在本节的最后，总结了 ArceOS API（分别针对 C 与 Rust 应用）与 Linux 系统调用接口的几处典型差异，如表 4.3 所示。

4.3.5　多内核形态构建

组件化设计带来的灵活定制能力，可以让 ArceOS 不止是一个微虚拟机操作系统（Unikernel），还能形成其他形态的内核或系统软件，比如具有用户内核隔离的宏内核，或是能运行虚拟机的虚拟机管理程序。本节将对这两种内核形态的支持进行详细介绍。

由于微虚拟机单特权级的特点，ArceOS 采用以应用为中心的设计思想，通过在应用程序层（仍处于内核态）实现特权功能，以扩展为不同形态的特权软件，同时避免对原有架构与性能的破坏。为了便于区分 ArceOS 的应用程序层与宏内核中运行在用户态的应用程序，本节称 ArceOS 的应用程序层为*内核应用*（仍为内核的一部分），运行用户态的应用程序为*用户应用*。本节先分析这两种内核与 Unikernel 的差异，然后分别提出相应的内核应用设计方案。图 4.6 给出了利用 ArceOS 的组件实现的这三种内核的架构。



(a) Unikernel　(b) 宏内核　(c) 虚拟机管理程序

图 4.6　ArceOS 扩展为其他内核形态①

宏内核。 ArceOS 宏内核的目标是能二进制兼容已有的 Linux 应用程序。与 Unikernel 不同，宏内核*不信任应用程序*，因此需要采取种种措施，提供应用与内核以及应用间的隔离，以防应用程序破坏内核与其他应用。其次，宏内核一般也需要提供*多应用*的支持②。具体来说，宏内核与 Unikernel 的差异主要体现在以下几方面：

① 粗体的为新增组件，带 “*” 的为需少量修改的组件。
② 虽然这在一些 Unikernel[50,65,114,177] 中也得到了实现。

- *接口*。Unikernel 中，应用程序通过函数调用直接访问内核服务，而宏内核中需要通过系统调用，存在特权级的切换。使用系统调用接口的优势是容易实现对已有应用的二进制兼容，而不需要基于专门的用户库重新编译。不过也因此无法使用第 4.3.4 中介绍的 API 快速路径。
- *进程*。Unikernel 中没有进程的概念，或者内核和应用整体可视为一个进程，而宏内核一般提供多进程的抽象。进程可认为是共享相同资源的线程集合，多进程间需要有资源的隔离（例如地址空间、打开的文件等）。这些资源在 Unikernel 中可全局只维护一份，但在宏内核中需要为每个进程都维护一份。此外，宏内核还需提供进程间通信（IPC）的机制，例如信号、共享内存等。
- *地址空间*。宏内核的应用程序不再与内核共享同一地址空间，需要为每个进程创建单独的地址空间，并在进程切换时进行切换。多地址空间还影响应用程序的内存布局与系统调用时参数的传递，例如需要额外将用户参数拷贝到内核地址空间以防 TOCTOU（time-of-check to time-of-use）攻击。
- *应用程序加载方式*。为了支持运行多个应用，需要实现动态加载，而不能再将内核与应用静态链接在一起作为一个单独镜像。此外，在为新应用准备资源时，可能需要利用写时复制、延迟分配等虚拟内存技术来加速应用启动。

ArceOS 提供了一个内核应用 app_monolithic，其中包含了宏内核相比于 Unikernel 需要额外实现的部分，例如进程创建、用户应用加载、系统调用实现等（图 4.6(b)）。对于系统调用，由于宏内核无法利用 ArceOS 提供的 API 快速路径，需使用模块层提供的功能接口对 Linux 系统调用进行忠实实现。此外，原有的模块也需新增一些接口来实现与 Linux 兼容的系统调用，如增加线程克隆接口以实现 `sys_clone`。对于进程，ArceOS 宏内核沿用了 Unikernel 形态的线程管理组件（axtask），只是在线程控制块中多了一个额外的字段（以下称*资源控制块*），用于存放地址空间、文件描述符表等不与其他进程共享的资源（图 4.7），并在切换线程时进行切换。此外，还需新增一些组件用于实现宏内核特有的功能，如各种 IPC 机制、虚拟内存管理等。

虚拟机管理程序。本节讨论的是类型一的 hypervisor，即 hypervisor 管控所有硬件资源，拥有最高特权级，无需依赖主机操作系统。此时的 hypervisor 可认为是一种特殊的内核，只是需要额外考虑虚拟机的状态管理与事件处理，包括：

- *处理器状态*。hypervisor 通常使用传统内核中的线程来包装虚拟处理器（vCPU），以支持多核虚拟机。因此，需要在线程上下文中维护 vCPU 的状态，并实现相应的切换流程。
- *虚拟机抽象*。类似于宏内核中的进程，hypervisor 需要为每个虚拟机维护一

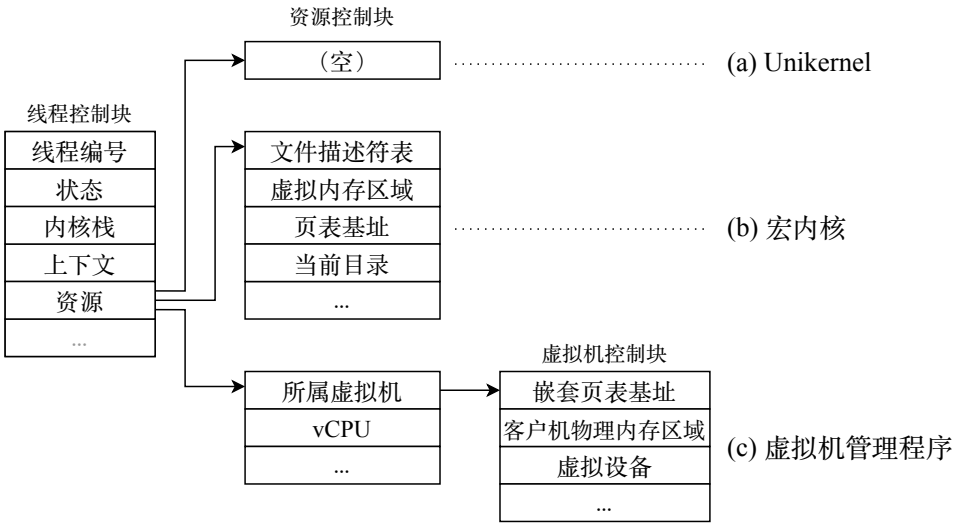


图 4.7　ArceOS 不同形态下的线程控制块结构

个数据结构，即同一个虚拟机使用的资源，可在该虚拟机的所有 vCPU 间共享，但在不同虚拟机间相互隔离。例如地址空间（GPA 到 HPA 的映射）、虚拟机设备列表等。

- 虚拟机退出的处理。类似传统内核的异常处理，hypervisor 需要注册一个虚拟机退出处理例程，该例程主要用于实现虚拟设备的模拟，包括 I/O 指令的模拟、中断注入等。

ArceOS 提供了一个元件层组件 hypercraft，其中封装了基本的虚拟化功能，例如不同架构下的 vCPU 抽象、嵌套页表实现、虚拟机退出例程的注册等。其余与 hypervisor 的功能紧密相关的部分实现在应用程序层（app_hv），例如客户机物理内存的管理、虚拟机退出的处理、虚拟机设备的模拟等（图 4.6(c)）。与宏内核类似，此时的线程控制块仍需维护额外的资源控制块，其中包含 vCPU 的状态、线程所属的虚拟机等不在虚拟机间共享的资源（图 4.7）。ArceOS 通过将资源控制块的具体结构交给其他组件来定义（这里指内核应用 app_monolithic 和 app_hv），使得可以不改模块层的 axtask 的代码，只改应用程序层的代码，就能扩展出满足不同内核需求的线程控制块。

4.4　系统实现

本节将详细介绍 ArceOS 中几个核心模块与元件的设计与实现。ArceOS 目前实现了多核、多线程、网络、文件系统、图形显示等基本内核功能，支持 x86_64、RISC-V、AArch64 等主流处理器架构，QEMU/KVM、x86 服务器、树莓派、黑芝麻等多种硬件平台，以及 Virtio、Intel 82599 万兆网卡等设备驱动（均可根据应用

表 4.4　ArceOS 组件列表

层次（组件总数）	组件	代码行数[b]	描述
元件层（33）[a]	linked_list	445	支持常数时间删除的链表[c]
	crate_glue	167	提供跨组件调用机制
	kernel_guard	175	创建关中断或关抢占的临界区
	spinlock	301	内核自旋锁
	percpu	213	CPU 私有数据的定义与访问
	scheduler	388	具有统一接口的多种任务调度算法
	allocator	551	具有统一的接口的多种内存分配器
	slab_allocator	423	基于 slab 的内存分配器
	page_table	401	针对不同硬件架构的通用页表结构
	page_table_entry	423	多种硬件架构下的页表项定义
	driver_virtio	316	各类 Virtio 设备的驱动程序
	axfs_vfs	404	虚拟文件系统接口
	axfs_ramfs	349	内存文件系统
	…	…	…
	小计	7,724	—
模块层（11）	axruntime	284	应用运行时
	axhal	3,318	硬件抽象层
	axalloc	208	全局内存分配器
	axtask	917	任务管理模块
	axsync	187	同步互斥模块
	axdriver	895	设备驱动模块
	axnet	1,203	网络模块
	axfs	1,529	文件系统模块
	…	…	…
	小计	8,882	—
API 层（2）	arceos_rust_api	532	为 Rust 应用程序提供的 API
	arceos_posix_api	3,662	为 C 应用程序提供的 API（POSIX 接口）
用户库层（2）	axstd	1,066	为 Rust 应用程序提供的精简版标准库
	axlibc	9,460	为 C 应用程序提供的精简版 libc 库
合计（48）	—	31,326	—

[a] 不包括其他开发者提供的已发布到 crates.io 中的组件。
[b] 不包括配置文件。
[c] Rust 核心库提供的链表不支持该操作。

需求进行灵活定制），总代码量达 3 万多行。表 4.4 给出了 ArceOS 目前在各个层次的核心组件列表及代码量[①]。

4.4.1　核心组件

本节从组件的角度，介绍 ArceOS 中一些最核心的、几乎会被所有应用都使用的组件（模块层）。

应用运行时（axruntime）。该组件负责对内核所启用的几个功能模块进行统筹管理与初始化，会在系统及硬件平台初始化完成后（由 axhal 实现），进入应用程序的主函数之前被执行。axruntime 会根据应用指定的特性，选择性依赖其他模块层组件，并通过条件编译选择性对它们进行初始化。此外，由于它作为所有应用都必须包含的模块，管控着所有其他模块，所以也会注册一些全局处理例程，如向其他模块分发中断与异常，注册 Rust 的崩溃（panic）处理例程等。

硬件抽象层（axhal）。该组件封装了不同平台的硬件相关操作，并向上提供了统一的接口，使得其他模块在调用时无需关心具体的硬件细节，只需实现硬件无关的代码。ArceOS 支持多种硬件平台，不同平台有着不同的启动流程、处理器状态以及平台相关设备[②]，均在 axhal 中完成配置与初始化。当系统启动后，首先会在 axhal 中执行硬件相关的初始化流程，再跳转到 axruntime 中执行硬件无关的初始化流程。此外，axhal 也提供了对架构相关指令与结构的封装，如开关中断、切换页表等指令，以及页表、异常与中断的向量表与上下文等结构，并实现了发生异常与中断时的状态保存与恢复。

由于硬件平台的多样性以及硬件操作的繁琐性，axhal 可能因此变得十分臃肿与复杂。幸运的是，其他开发者已经提供了众多封装好的硬件相关操作（通过 crates.io 发布），例如 x86_64、x2apic、aarch64-cpu 等 Rust 包，也包括 ArceOS 元件层的一些组件，axhal 可以重用这些组件而大大简化自身的实现。

动态内存分配（axalloc）。该组件主要包含了两个内存分配器：页分配器与字节分配器。其中页分配器只能返回一个或多个连续的且对齐的内存页（4KB），用于物理页帧或 DMA 内存的分配。字节分配器即一般的堆分配器，可以返回一块任意大小的内存。axalloc 的页分配器基于位图（bitmap）来实现，而字节分配器根据应用程序指定的特性，可选 slab、TLSF、伙伴系统等多种分配算法。此外，axalloc 还提供了查询各分配器已用的内存、剩余的内存等接口。

在初始化时，axruntime 会向 axalloc 提供所有可用的物理内存段，先将它们全部用于页分配器，并将字节分配器设为空。当后续按字节分配失败时，会请求页

① 基于该版本统计：https://github.com/rcore-os/arceos/tree/ba8f1fc7a18ee1ac4711afa4f0d984498534f114。
② 这里指串口、时钟、中断控制器等简单但必需的设备，而不包括网卡、磁盘等复杂且可选的设备。

分配器分配一段连续的物理页，将它们添加进字节分配器，然后再次尝试字节分配直到成功。字节分配器每次扩充的容量下限为当前容量，使得每次扩充都会让容量至少翻一倍。

axalloc 还负责将字节分配器注册为 Rust 的全局内存分配器，使得应用程序与内核共用一个分配器。其他所有用到动态内存分配或释放的操作（如使用 `Box<T>`、`Vec<T>` 等类型），均会被重定向到全局分配器中。

4.4.2　任务管理

ArceOS 的任务管理功能实现在 axtask 组件中。在微虚拟机操作系统的设计中，一般只有线程而没有进程的概念（事实上可将整个微虚拟机看做一个进程），ArceOS 将线程作为最基本的调度单元，并称为一个任务①。

该组件实现了多任务的调度以及任务生命周期的管理（创建、退出、睡眠、唤醒、切换），并支持通过特性指定不同的任务调度算法。最简单的是 FIFO 协作式调度，只有当任务主动让出时才会发生任务切换，适合基于事件循环的运行到完成式的应用。其他支持的调度算法还有时间片轮转、完全公平调度等，它们都是抢占式的调度算法，适合对延迟或公平性有要求的应用。该组件会从 axruntime 接收时钟中断，并决定是否要进行抢占。此外，在 axtask 中还实现了等待队列，用于为上层模块提供基本的线程同步机制。例如 axsync 基于 axtask 的等待队列实现了互斥锁、信号量、条件变量等更多高级同步原语。

axtask 也可用于本文第 3 章提出的库操作系统，作为 Skyloft 用户级线程库的实现，以进一步提高组件的可重用性。

4.4.3　设备驱动

ArceOS 的 axdriver 组件中实现了对设备驱动层的抽象。该组件提供了对各类设备的抽象与封装，目前 ArceOS 支持三种设备的抽象：网络设备、存储设备、图形显示设备，它们分别具有一组统一的接口，用于实现类型相同但型号不同的设备驱动组件。例如，对于存储设备，接口主要是数据块的读写，对于网络设备则是数据包的收发以及数据包缓冲区的分配与释放。

axdriver 会注册几种支持的设备型号，应用程序可通过特性在已注册的设备中选择要使用的具体设备。axdriver 只负责将选定的设备驱动进行组合，而驱动的具体实现会拆分到元件层的组件中。表 4.5 列出了 ArceOS 目前支持的设备型号，其中大部分是 Virtio 设备[85]。此外，应用程序也可通过特性指定设备的探测方式，如通过 PCI 总线还是设备树。

① ArceOS 的宏内核形态可基于任务扩展出进程结构（第 4.3.5）节。

如果应用程序指定了多个同一类型的设备，axdriver 需要将它们放入一个列表中，并通过统一的接口进行访问。然而，不同的设备在实现时会被组织为不同的数据类型，在进行功能调用时会涉及数据类型的动态分发[178]（例如通过虚函数表），带来一定的调用开销，且无法使用内联优化。为此，axdriver 提供了一种特性配置，可通过限制同一类型只能有一个设备，从而使用静态分发以提供最佳性能。如不启用该特性则仍为多设备动态分发。

表 4.5　ArceOS 设备驱动列表

设备类型	设备型号	描述
存储	ramdisk	内存模拟磁盘
	virtio-blk	Virtio 块设备
	bcm2835-sdhci	树莓派 SD 卡控制器
网络	virtio-net	Virtio 网卡
	ixgbe	Intel 82599 万兆网卡
显示	virtio-gpu	Virtio 显卡

4.4.4　网络栈

ArceOS 的网络功能实现在 axnet 组件中。该组件不仅是对网卡驱动与网络协议栈组件的组合，还为应用程序提供了基于套接字（socket）的网络接口。ArceOS 引入了已在嵌入式与实时系统中被广泛使用的 smlotcp[179]（Rust 编写）和 lwip[180]（C 编写）作为网络协议栈，两者都以元件层组件的形式提供。这两个组件都提供了各自的网卡设备抽象，通过为 axdriver 中的网卡驱动实现该接口，可将驱动与协议栈结合。axnet 在这两个协议栈提供的套接字接口上再封装了一层，使得接口统一且与 POSIX 接口更接近，例如提供 TCP 套接字的 `connect`、`bind`、`listen`、`accept`、`send`、`recv` 等操作。除了 TCP，axnet 还提供了 UDP 套接字、DNS 解析等多种网络功能。

为了给应用提供最佳性能，目前 ArceOS 的网络访问是基于轮询的。在每次调用套接字操作后，都会轮询网卡，将收到的数据包传入协议栈以推进状态机，或发送已在协议栈发送缓冲区的数据包。此外，网卡驱动也使用一组零拷贝的接口进行数据包的收发。例如，在接收数据包时，会预先分配一堆数据包缓冲区，并放入网卡的接收队列。当网卡收到数据包后，会直接将该缓冲区传递给协议栈以避免拷贝，协议栈处理完成后将缓冲区再次放入接收队列。

4.4.5　文件系统

文件系统也是操作系统中一个容易被解耦的功能，因此它们需要尽可能被实现为一个元件，便于被其他系统重用。ArceOS 通过一个元件层组件 axfs_vfs 定义了虚拟文件系统层，提供对文件系统、文件节点的接口定义。在实现具体的文件系统时，只需为其包装一层该组件提供的接口，就可被 ArceOS 使用。ArceOS 目前以这种方式实现了多种文件系统组件，例如内存文件系统 ramfs、设备文件系统 devfs、进程文件系统 procfs 等。此外，ArceOS 还通过重用他人编写的组件 rust-fatfs[181] 支持 FAT 文件系统。

在模块层中的 axfs 中，会根据应用程序提供的配置，静态选择所需的文件系统，进行初始化并挂载。与 ArceOS 的设备驱动一样，静态配置也减小了虚拟文件系统层的动态分发开销（见第 4.5.3 节的实验）。此外，axfs 在内核文件对象之上，为应用程序提供类似 Rust 的标准库接口，作为实现 API 快速路径的基础，如 `File::open`、`File::read` 等，

4.5　实验与评估

本节通过一系列实验评估基于组件化设计的 ArceOS 在几大内核基础功能上性能表现，包括线程管理、文件与网络操作，以及针对 Rust 与 C 应用的 API 快速路径的优化效果。最后，本节评估了 ArceOS 在真实世界应用上的性能。

4.5.1　实验环境

本章的实验在一台桌面计算机上进行，CPU 型号为 4 核 AMD Ryzen 3 5300G，主频为 4.0 GHz，内存为 16 GB，并且关闭了超线程与动态频率调节。实验机器运行 Ubuntu 18.04 操作系统，内核版本为 Linux 5.11.0，并使用 QEMU/KVM 作为虚拟机管理程序运行 ArceOS 及其他 Unikernel。

实验以 Unikraft[28]（v0.15.0）这一目前最先进的 Unikernel，以及 Linux 作为基线。其中 Unikraft 和 ArceOS 一样运行在虚拟机中，Linux 运行在裸机上，以演示微虚拟机技术相对于主机原有应用程序的性能提升。由于 Unikraft 依赖 musl libc 库[182] 才能支持大部分 POSIX 操作，为公平比较，实验在评估 Linux 上的应用程序时也链接了相同版本的 musl libc 库。

4.5.2　线程操作性能

本实验分别使用 C 语言和 Rust 语言编写应用程序，评估 ArceOS 与基线系统上的几种线程操作的用时，包括线程的创建、切换与同步，以及在这些操作中利用

API 快速路径的优化效果。

线程创建。由于线程创建对内存分配较为敏感，因此实验将所有系统上的线程栈大小均设为 4KB，并将 ArceOS 与 Unikraft 均配置为使用 TLSF[172] 分配算法，而 Linux 因难以修改内核的内存分配算法仍使用默认的。创建的线程均为空线程，不执行任何操作直接退出。

图 4.8 给出了在各个系统上使用 C 与 Rust 语言创建一个空线程所需的时间，以及时间的主要组成部分。在 Linux 与 Unikraft 上，C 应用通过 POSIX 接口 `pthread_create` 创建线程，其开销主要来自 `sys_mmap` 与 `sys_clone` 系统调用，分别用于分配线程私有内存（栈与 TLS）以及最终的线程创建，此外还有很大一部分用户库的开销，用于为这几个系统调用准备参数。

在 Unikraft 中，主要因为避免了系统调用时的上下文切换而使开销大大减小。然而，由于 Unikraft 注重对 POSIX 接口的兼容性，使得在该简单场景下仍有大量不必要的操作。图 4.8 中的 Unikraft-opt 为经过笔者优化的版本，消除了多余的内存清零与 POSIX 线程结构的初始化，可以使开销进一步降低。但由于仍使用低效的 `sys_clone` 接口以及实现上的问题，每次线程创建仍需多次内存分配与拷贝，导致开销依然可达数百纳秒。

此外，对于 Rust 应用，在 Linux 和 Unikraft 上均需依赖 libc 库以提供系统级功能支持，使其最终仍会调用 `pthread_create` 创建线程，比 C 还多了一部分 Rust 标准库的开销。

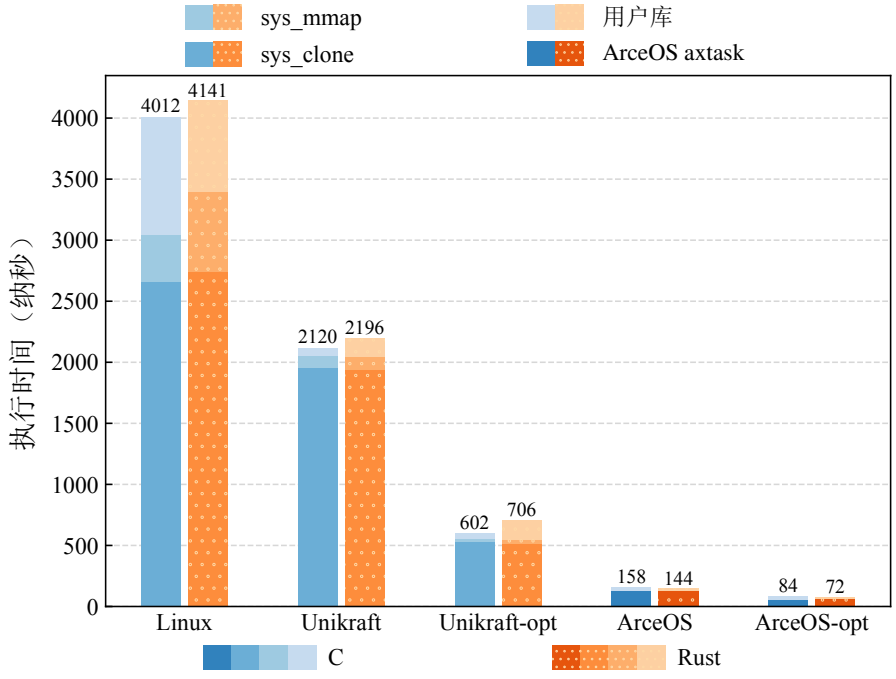


图 4.8　几种系统的线程创建性能剖析

ArceOS 的线程创建开销非常小，主要原因是使用了 API 快速路径，简化了用户库以及底层模块（axtask）的实现，而无需经过 `sys_clone` 的转换。同时，Rust 比 C 应用具有更小的开销，因为 Rust 用户库的接口与底层模块的接口更接近。此外，对于此类简单场景，ArceOS 还允许配置为不启用浮点数与 TLS，从而免去线程创建时浮点上下文与 TLS 结构的内存分配与初始化，使得线程创建的开销可以低至几十纳秒（图 4.8 中的 ArceOS-opt），与用户级线程[96,157]相媲美。

Unikraft 虽然提供了众多配置选项，但并不能关闭浮点数与 TLS。因此本实验也说明了，即使对于 Unikraft 这样的目前最先进的、能根据应用需求进行灵活定制的微虚拟机操作系统，其专用化程度仍然不够，仍具有较大的性能提升空间。在 Linux 中实现这种程度的定制则更为困难。

线程切换与同步。为了评估线程切换的开销，本实验创建了两个线程，让它们不断调用主动让出操作（yield），使这两个线程来回切换，并测量单次切换花费的时间。此外，实验还使用条件变量，测量了让当前线程睡眠，唤醒另一线程并切换过去的总时间，以评估线程同步的开销。

实验结果如表 4.6 所示。对于线程让出操作，由于 C 和 Rust 均提供无参数的简单接口，这两种应用的执行时间几乎一样，但 Unikraft 和 ArceOS 的开销比 Linux 低一半以上。对于条件变量操作，在 Linux 和 Unikraft 上，无论是 C 还是 Rust 均通过底层的 `sys_futex` 系统调用直接实现，因此这两种应用的执行时间也差不多，不像线程创建时 Rust 需要间接调用 C 库，而使开销更大。对于 ArceOS，可以使用 API 快速路径避免 `sys_futex` 带来的接口转换，因此开销比 Unikraft 更低。此外，将 ArceOS 配置为禁用浮点数与 TLS，能进一步提高这两类线程操作的性能，例如线程切换的时间减少了近 90%。

表 4.6　几种系统的其他线程操作性能对比

系统	线程让出（纳秒）		条件变量（纳秒）	
	C	Rust	C	Rust
Linux	408	408	1,070	996
Unikraft	199	199	279	274
ArceOS	179	178	212	204
ArceOS（无浮点）	107	109	140	132
ArceOS（无 TLS）	89	89	125	118
ArceOS（无浮点无 TLS）	19	19	45	39

4.5.3　文件操作性能

本节的实验同样对 C 和 Rust 应用分别进行评估。实验先创建了一个 1MB 大小的文件，然后分别测量打开、读取 1 字节与写入 1 字节的执行时间，为了单独评估文件操作的开销，避免磁盘访问的影响，本实验在所有系统上均使用内存文件系统（ramfs）。

表 4.7 给出了实验结果。对于 Linux 和 Unikraft，由于 Linux 系统调用的开销与实现的复杂性，其性能最差。Unikraft 大大优化了文件读写的性能，与 Linux 相比分别有 61% 和 79% 的性能提升，但对于打开文件操作的性能提升不大（12%）。此外，从使用的编程语言来看，Rust 在打开文件操作上的调用路径比 C 更长，速度稍慢，但在文件读写操作上与 C 几乎一样。

表 4.7　几种系统的文件操作性能对比

系统	打开文件（纳秒）		读取单字节（纳秒）		写入单字节（纳秒）	
	C	Rust	C	Rust	C	Rust
Linux	618	686	217	220	421	411
Unikraft	542	597	75	76	75	76
ArceOS	88	51	39	19	39	19

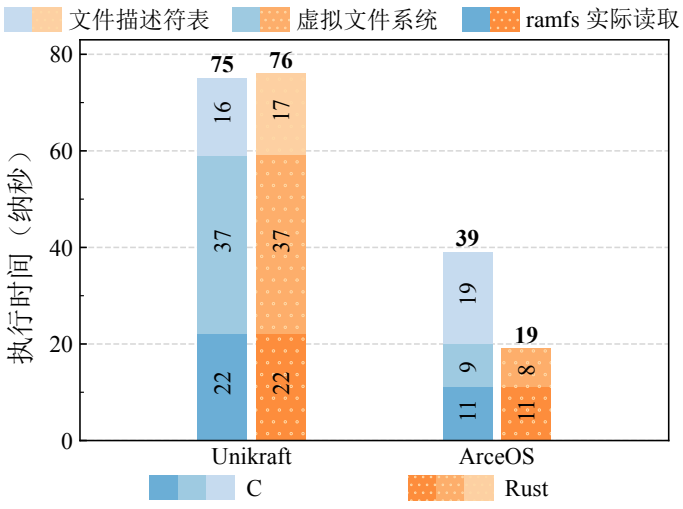


图 4.9　ArceOS 与 Unikraft 文件读取操作的性能剖析

ArceOS 在这三者中具有最佳的性能，相比于 Linux，这些文件操作的性能都提升了 90% 左右。此外，与前两者不同，ArceOS 对 Rust 应用的性能优化效果更明显，Rust 比 C 的读写性能快了 50% 左右。为了进一步分析 ArceOS 性能好的原因，

实验还将 ArceOS 与 Unikraft 的文件读取操作进行分解，分成三个阶段分别测量执行时间，分别为文件描述符表的查询、虚拟文件系统层的分发，以及内存文件系统的实际读取，如图 4.9 所示。可见 Unikraft 的主要开销花费在虚拟文件系统层，这是因为 Unikraft 的普通文件读写与使用 I/O 向量的文件读写共用一个接口，使得需要额外进行一些内存分配与拷贝操作，但在 ArceOS 中则没有这部分的开销。另一方面，对于 Rust 应用，ArceOS 还利用 API 快速路径完全消除了文件描述符表的开销。

此外，实验还评估了读写不同大小的数据块对性能的影响，如图 4.10 所示。对于小数据块（⩽ 1KB），在 ArceOS 中 Rust 相对于 C 的性能提升比较明显，可达 41.5% ∼ 58.5%。但对于大数据块（> 16KB），API 快速路径的优化效果会逐渐被数据拷贝所掩盖，使得 C 和 Rust 的性能差距逐渐缩小（小于 4%）。

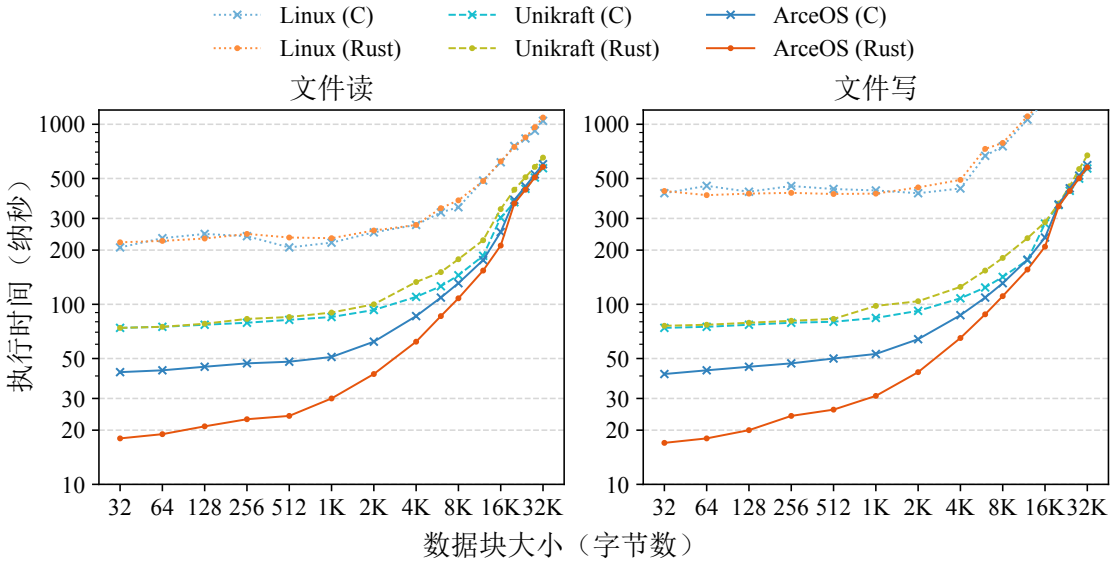


图 4.10　几种系统使用不同大小数据块的文件读写性能对比

4.5.4　网络操作性能

本节的实验在各系统上运行一个常用的网络性能评估工具 NetPIPE[183]。该工具会在客户端与服务器之间收发不同大小的 TCP 消息，测量单向传输延迟与相应的网络带宽。实验另使用一台机器作为客户端，CPU 型号为 32 核 Intel Xeon E5-2683 v4，主频 2.1 GHz。服务器与客户端均通过 Intel 82599ES 10 Gbps 网卡进行网络传输。

本实验的比较对象仍为 Linux、Unikraft 和 ArceOS。其中 Linux 运行在裸机上，使用内核的网络协议栈与 ixgbe 网卡驱动。Unikraft 运行在虚拟机中，由于其不支持物理网卡驱动，只能使用 Virtio 网卡，并通过 Linux 的网桥与物理网卡交换数据包。ArceOS 也运行在虚拟机中，不过可以利用组件化的优势方便重用他人编写的

物理网卡驱动，例如这里使用了 ixy.rs[184] 中的该型号网卡驱动。物理网卡会通过 Linux 的 VFIO[185] 机制直通给 ArceOS 虚拟机，以提供与裸机相当的性能。ArceOS 的网络协议栈使用 smoltcp，其余网络实现细节见第 4.4.4 节。

实验将待测系统分别运行在客户端或服务器端，具体配置以及实验结果如图 4.11 所示。比较 ArceOS 与 Linux，对于短消息，例如在 64B 下 Linux 与 ArceOS 的传输延迟分别为 29.0μs 与 8.0μs，Linux 为 ArceOS 的 3.6 倍，这是因为短消息的传输开销主要在于系统调用，ArceOS 利用微虚拟机的单地址空间优势、快速路径优化以及更精简的网络协议栈大大减小了这部分的开销。但对于长消息（> 64KB），ArceOS 的性能开始比 Linux 差，而且最大带宽仅达到 3.6Gbps。因为 ArceOS 虽然在驱动程序中尽量避免了拷贝，但目前在网络协议栈层面的优化不足，还未提供零拷贝接口，对长消息需要多次拷贝而导致性能不如 Linux，这也是 ArceOS 今后需要改进之处。

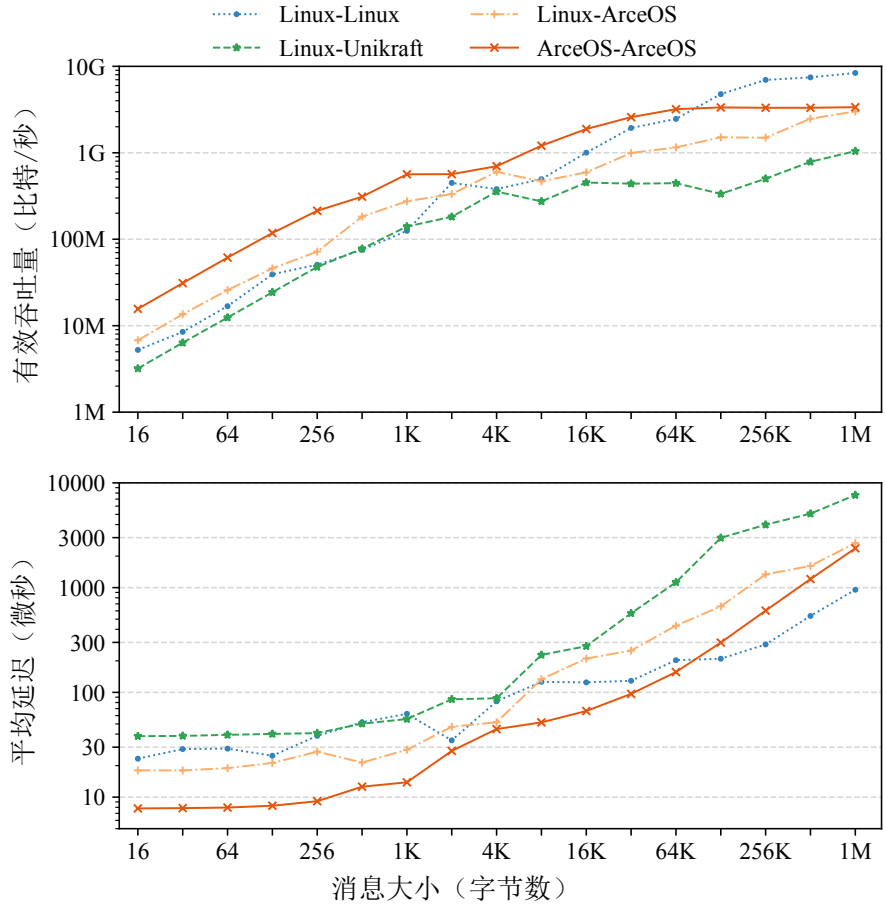


图 4.11　几种系统的网络传输性能对比

为比较 ArceOS 与 Unikraft，实验让客户端均运行 Linux，服务器运行相应的系统。无论是小数据包还是大数据包，Unikraft 比 Linux 的性能都差，这是因为其虽然也消除了系统调用的开销，并同样使用了轻量的 lwip 协议栈，但由于不支持

物理网卡驱动，仍需通过主机 Linux 与物理网卡交换数据包。此外，对于一端运行 Linux 而另一端运行 ArceOS 的情况，与两端都运行 ArceOS 类似，也是小数据包表现较好，大数据包表现较差，不过整体性能仍优于 Unikraft。

4.5.5　综合应用性能

ArceOS 除了直接兼容 Rust 应用外，也为 C 应用提供了基本的 POSIX 支持，能运行一大批已有的复杂 C 应用，并可利用组件化的优势根据场景进行针对性的优化。本节主要评估在 ArceOS 上运行目前被广泛使用的 Redis[186] 数据库（v7.0.12）的性能。为支持运行 Redis，ArceOS 需要依赖大部分模块层组件，包括任务管理、设备驱动、网络、文件系统等。

本实验采用与第 4.5.4 节的网络实验相同的配置。Redis 服务器被配置为内存数据库，客户端使用 Redis 自带的 `redis-benchmark` 工具向服务器发送 GET 请求，采用默认配置（不启用流水线，数据大小为 3 字节），然后不断增加客户端的并发连接数并测量相应的吞吐量与 99% 尾延迟。

实验结果如图 4.12 所示。运行在虚拟机中的 ArceOS 比裸机上的 Linux 具有更

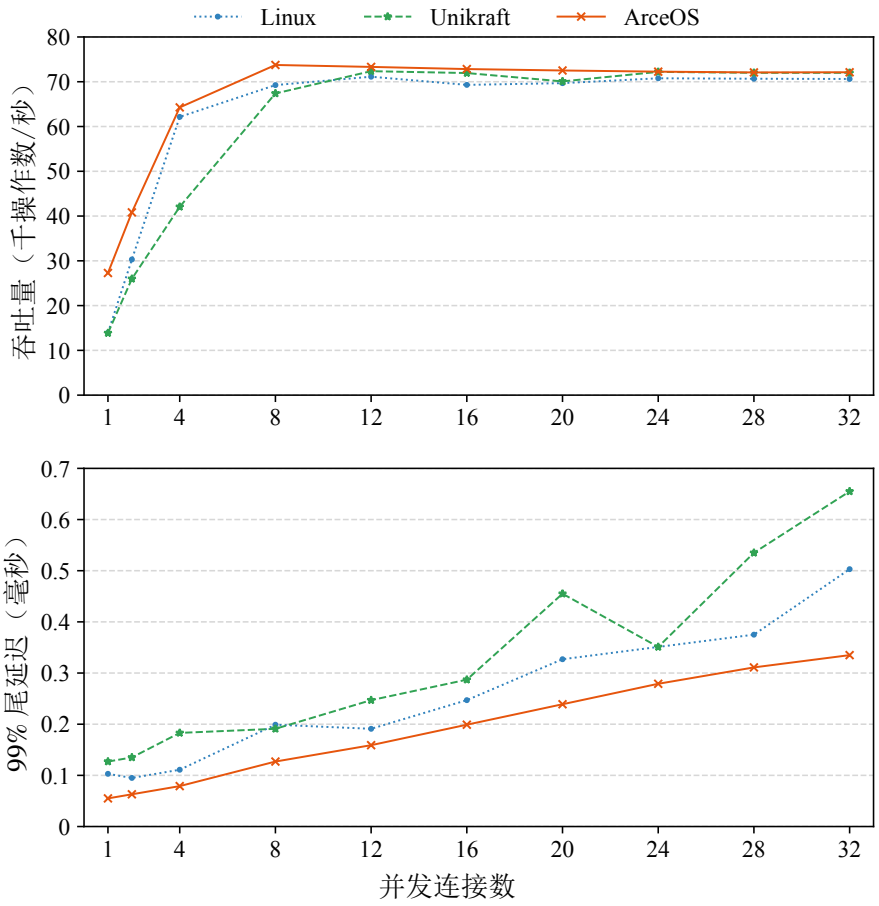


图 4.12　几种系统的 Redis GET 请求性能对比

高的吞吐量与更低的尾延迟，例如在连接数为 32 时比 Linux 的尾延迟低了 33.4%。因为 Redis 的请求都是小数据包，使得 ArceOS 在该场景下表现优异（正如第 4.5.4 节所述）。Unikraft 因不支持物理网卡驱动而表现较差。

本节的实验主要为说明 ArceOS 能根据已有应用的需求快速打造出合适的微虚拟机操作系统，短时间带来性能的提升。虽然 Linux 也可使用 DPDK[31] 等内核旁路技术提供高性能的网卡驱动①，但需要额外实现用户态网络协议栈，提供套接字接口才能运行已有的应用，例如本文第 3 章的 Skyloft 系统。相比之下，ArceOS 通过对组件的组合来支持 Redis 这样的复杂应用，同时具有高性能、兼容性与灵活性三大优势。既提供绕过内核的高性能网络栈，又兼容已有应用，还能方便驱动程序、协议栈等组件的定制与重用。

4.6　讨论

本节简要讨论 ArceOS 的局限性与未来工作。

彻底消除循环依赖。目前，ArceOS 只是利用 crate_glue 明确标记循环的组件依赖，但仍没有完全消除循环依赖（第 4.3.2 节）。当所有直接与间接的循环依赖被消除后，将有助于模块化的测试、调试与形式化验证。例如，一个组件中的缺陷，只可能影响所有依赖它的组件，而不会在整个系统中传播。为彻底消除循环依赖，一种方法是将大组件拆分为几个小组件的方式，不过这可能会使组件间的逻辑关系变得混乱，增大开发难度，也让代码可读性变差。因此，如何提供一种不存在循环依赖，而且不影响开发便利性的组件拆分方式，仍有待进一步研究。一个可能的思路是在编程语言层面引入一些新的特性。

微内核形态支持。第 4.3.5 节讨论了如何将 ArceOS 从 Unikernel 的基本形态扩展为宏内核与虚拟机管理程序。对于微内核形态的扩展，与宏内核基本类似，也需要提供应用程序与内核的地址空间以及特权级隔离，并实现 IPC 机制。不过，由于微内核的各种服务（如设备驱动、网络、文件系统）也运行在用户态，它们之间需要使用专门的 IPC 接口进行通信，而不像 Unikernel 与宏内核那样直接使用函数调用。因此，支持微内核的一个挑战是，如何在不对现有内核功能模块做太多修改的情况下，让它们同时支持 IPC 与函数调用接口，并可通过相应的配置选项进行切换。另一方面，IPC 也会成为微内核的性能瓶颈，需要针对 IPC 机制进行深度优化。

① Unikraft 也可利用 vhost-user 在虚拟机中使用 DPDK。

4.7　本章小结

高性能的微虚拟机操作系统需要能够方便地针对应用程序的需求进行专用化定制。为此，本章提出了一种组件化的操作系统设计方法 ArceOS，通过基于设计理念相关性的组件划分，能够降低组件间的耦合性，并提供良好的可定制性与可重用性，例如构建出多种形态的内核。ArceOS 利用 Rust 语言的标准库接口提供 API 快速路径设计，既获得了比 POSIX 接口更好的性能，又具有足够的兼容性。实验表明，ArceOS 能够为应用快速构建出满足需求的专用操作系统，而且具有比 Linux 等系统更高的性能。

第 5 章　面向可信执行环境的微虚拟机灵活隔离与性能优化

5.1　本章引言

微虚拟机的另一主要特点是需要依赖底层的管理程序。然而，由于底层管理程序一般仍是通用的操作系统（例如在上两章中都是 Linux）, 其庞大的代码量导致了众多潜在的安全漏洞。此外，管理程序与微虚拟机间的隔离是单向的，管理程序能够访问微虚拟机的所有内存，因此一旦被攻破，其上运行的所有微虚拟机都会受到影响，造成敏感数据泄露或被篡改等风险。

随着人们对数据隐私保护意识的不断提高，如何安全地在不受信任的环境中处理敏感数据，成为了一个重要的研究方向。一种通用的解决方法是可信执行环境（TEE）技术，如本文第 2.3.3 节所述，其通过将数据放在一块硬件保证的隔离分区中进行处理，阻止来自不可信的主机操作系统与其他实体的访问。目前，Intel SGX[51] 等可信执行环境已在各类桌面和服务器处理器中被广泛使用。利用 TEE 技术来保护微虚拟机，既能阻止不可信的管理程序对微虚拟机的恶意访问，又不影响管理程序对微虚拟机提供正常的管控与服务。

然而，现有的大多数 TEE 技术都需要特定的硬件或固件修改，使得它们只在部分场景下可用，而且修改与升级的周期较长。此外，闭源的硬件也使得难以审计其中的安全漏洞或后门。而一些使用软件方法实现的 TEE 需要较大的开销才能保证安全性[55-57]。

另一方面，现有 TEE 技术对微虚拟机的支持也不够。它们在设计上限制了受保护的代码与数据（被称为飞地）只能在一种固定的模式下运行，无法同时支持用户态与特权态微虚拟机，难以满足不同场景下应用对性能与安全的需求。例如，Intel SGX 的进程级飞地运行在用户态，无法访问特权硬件资源，在一些场景下的性能开销较大；而 AMD SEV[52] 和 Intel TDX[53] 提供的虚拟机级飞地需要在其中运行一个完整的客户机操作系统，具有相当大的可信计算基（TCB），在安全性与启动速度上不如进程级飞地。另一方面，现有支持用户态微虚拟机与特权态微虚拟机的 TEE，分别使用了不同的编程模型与应用接口，这也增加了应用的开发与适配难度。

为了解决这些问题，本章提出了一种基于硬件虚拟化的 TEE 设计方案 HyperEnclave。HyperEnclave 具有很少的硬件需求，仅依赖如今广泛可用的虚拟化扩展（用于实现隔离），以及可信平台模块（用于信任根与随机数生成），从而方便在多种硬件平台上实现。为了能同时支持用户态与特权态的微虚拟机，HyperEnclave 提

供了*灵活的隔离模式*，通过支持在不同硬件特权级下运行飞地微虚拟机，允许应用根据使用场景选择面向安全或面向性能的模式（第 5.3.5 节）。同时，HyperEnclave 为用户态与特权态的微虚拟机提供了统一的、基于进程的 TEE 编程模型与接口，并与 Intel SGX 的生态兼容（第 5.3.4 节）。

HyperEnclave 使用一个轻量的微虚拟机管理程序实现对飞地微虚拟机的管理，并利用硬件虚拟化提供的嵌套页表机制来保证飞地与不可信操作系统与应用间的隔离。为了防止软件 TEE 可能面临的基于页表的攻击[127]，HyperEnclave 通过让可信的代码管理飞地的页表以及处理缺页事件，避免不可信操作系统的介入，以减小性能开销。同时，这种设计也防止了一些来自飞地恶意软件的攻击（第 5.3.3 节）。

为减小微虚拟机管理程序的 TCB 与攻击面，同时支持 TEE 所需的认证功能，HyperEnclave 采用了一种*度量延迟启动*的方法：先启动主机操作系统，再由一个内核模块加载虚拟机管理程序，运行在最高特权级，同时将原来的主机操作系统降权到客户机模式。启动过程中的所有部件都会被度量，并利用可信平台模块（TPM）保证启动过程的安全性（第 5.3.6 节）。

本章在一台 AMD 服务器上实现并部署了 HyperEnclave，通过一个飞地开发套件可兼容已有的 Intel SGX 应用，几乎无需修改源代码。本章还为 HyperEnclave 移植了许多 SGX 的应用程序和库，包括 Rust SGX SDK[187] 和 Occlum 库操作系统[36]。微基准测试和真实应用测试结果表明，HyperEnclave 引入的开销很小，性能与原生 SGX 相当（例如 SQLite 的开销仅为 5%）。此外，在不同的场景下使用不同隔离模式的飞地，能提供更好的性能。

本章的主要贡献如下：

1. 提出了一种具有灵活隔离模式的 TEE 设计，能支持不同形态的微虚拟机，以更好地满足不同场景下应用对性能与安全的需求。
2. 设计了一种低开销内存隔离机制，让可信的代码管理飞地页表与处理缺页，以防止基于页表的攻击和飞地恶意软件的攻击。
3. 设计了一种度量延迟启动的方法，通过与基于 TPM 的认证机制相结合，减小了攻击面。
4. 提供了对 Intel SGX 生态的兼容，支持在非 Intel 平台上无修改运行已有 SGX 应用程序。
5. 基于最小的硬件需求（虚拟化和 TPM）进行了实现，并通过真实硬件和应用上的实验说明了该方案的低开销、通用性、灵活性优势。

5.2　设计目标与挑战

本章旨在利用可信执行环境技术保护微虚拟机内隐私数据的机密性与完整性，免受底层不安全的管理程序的攻击。然而，将现有的 TEE 用于微虚拟机的保护还存在一些不足。以下介绍了本章系统的设计目标，并分析了其中的挑战。

最小化硬件需求。现有的商用 TEE 技术[51-54]，大多需要依赖特定的安全硬件。除了应用场景受限以外，对硬件的修改与版本升级也更为缓慢。此外，这些硬件也都是闭源的，难以审计其中的安全漏洞或后门。这使得这些 TEE 技术在通用性方面不如基于公开算法与广泛可用硬件的替代方案（如同态加密）。

虽然目前也有不少在广泛可用硬件上利用软件方法实现的 TEE[55-57,121]，但它们往往需要在性能与安全性上做出不少权衡。例如，TrustVisor[55] 同样基于硬件虚拟化来隔离应用程序的敏感（被称为 PAL）与非敏感部分。TrustVisor 为了防止内存映射攻击（如图 5.2），通过去掉客户机页表的写权限，来让虚拟机管理程序拦截并验证对客户机页表的修改。这种设计为 PAL 的注册与切换都引入了不小的开销，且随 PAL 的大小线性增长（例如注册 64 KB 的 PAL 需要 435μs[55]）。更糟糕的是，由于客户机页表访问位和脏位的更新，在高内存压力下会触发许多嵌套页表缺页，从而带来巨大的性能下降[126]。

因此，本章希望能够使用尽量少的硬件需求来实现 TEE，使得其在多种硬件平台上都可用，同时解决已有软件方法的一些问题。

不同形态微虚拟机的支持。本文在第 1 章介绍了微虚拟机的两种基本形态：用户态微虚拟机与特权态微虚拟机。这两类微虚拟机都有各自的适用场景。例如，用户态微虚拟机不能访问页表等特权资源，处理中断与异常等特权事件，当需要时必须切换到管理程序中，带来不小的性能开销，还容易受到侧信道攻击[127]。在特权态微虚拟机中可以执行特权操作，例如可以通过管理页表加速垃圾回收等应用，或是实现更细粒度的隔离[25]，但这也会带来一定的虚拟化开销，例如与管理程序间的切换开销增大，而且使用二级地址转换还会导致 TLB 缺失的代价增大，在运行内存密集型任务时会带来显著的性能下降。

然而，现有的大多数 TEE 技术，只能支持在固定的模式运行，无法同时支持这两种微虚拟机。例如，硬件实现的 Intel SGX，以及软件实现的 TrustVisor，都将 TEE 作为应用程序地址空间的一部分，在用户态运行。而 AMD SEV[52] 和 Intel TDX[53] 只能将 TEE 以一个完整的虚拟机的形式运行。本章希望能够设计一种新的 TEE 抽象，同时支持这两种微虚拟机的运行，以更好地满足不同场景下应用对性能与安全的需求。

统一的编程模型。将微虚拟机作为 TEE 来运行，还需要提供合适的编程模型

与接口。这不仅包括应用程序的编写方式，还有 TEE 所需的额外安全服务（如远程证明与数据密封）。现有的 TEE 技术为保护不同形态的微虚拟机，需要使用不同的编程模型，使得用户在不了解底层硬件的情况下很难进行应用的开发与适配。例如，为支持网络等 I/O 访问，基于虚拟化的 TEE 一般会在其中实现完整的网卡驱动与网络协议栈；基于进程的 TEE 则不实现网络功能，而是调用主机操作系统提供的服务。

本章的解决思路是重用已有的 TEE 设计，为这两种微虚拟机提供统一的编程模型与接口，具体来说，即 Intel SGX 的基于进程的 TEE 模型[188]，因为其不单具有更小的可信基，也具有更好的生态支持。Intel SGX 是如今在云上最流行的 TEE 之一，包括谷歌云平台（GCP）、微软 Azure、阿里云等主要的云服务提供商都提供了基于 SGX 的机密计算服务[189-190]。并且人们已经为 SGX 投入了大量工作，开发了不少工具，做了不少优化，用于减轻应用移植代价、提高开发效率、改进性能与安全性等。例如用于 SGX 的库操作系统[35-36]、容器[34]、WebAssembly 微运行时[191]、自动分区与保护工具[192-193]。因此，兼容 SGX 的生态将大大减少旧代码的移植工作量。

在 SGX 的编程模型中，应用会被分别可信与不可信两部分，作为同一个进程，分别运行在飞地与主机操作系统的用户态。应用程序通过 ECALL 进入飞地运行，在飞地中通过 OCALL 调用不可信的应用或主机操作系统的服务（例如系统调用）。这种编程模型不仅适用于用户态微虚拟机，对特权态的微虚拟机同样适用，这时的特权态微虚拟机可视为一种 “具有一定特权能力的特殊进程”。

5.3　系统设计

本节先介绍 HyperEnclave 系统的整体架构与威胁模型，然后介绍 HyperEnclave 的设计细节，包括内存管理与保护中的挑战与解决方法，对 SGX 编程模型的支持与飞地生命周期的管理，飞地的灵活隔离模式，以及对可信启动与远程证明的支持。最后分析了 HyperEnclave 设计的安全性。

5.3.1　整体架构

HyperEnclave 上的软件运行在下列三种模式中：（1） *管理模式*，具有最高特权级，对应到 x86 硬件上即 VMX root 模式。（2） *普通模式*，运行不受安全保护的软件，包括应用的不可信部分与操作系统，对应到硬件上即 VMX non-root ring-0 和 ring-3。（3） *安全模式*，运行受安全保护的软件，即应用的可信部分（飞地），根据*飞地隔离模式*的不同（第 5.3.5 节），该模式分别对应了不同的硬件模式，包

括 VMX non-root ring-0 和 ring-3，以及 VMX root ring-3。

图 5.1 给出了 HyperEnclave 的系统架构，主要包含以下组件：

- **RustMonitor。**这是一个轻量级的微虚拟机管理程序，运行在管理模式，用于管理飞地内存，保证内存隔离，控制飞地状态变换。RustMonitor 只负责资源的管理与安全保障，而复杂的任务会被委托给上层的主操作系统来处理。
- **主虚拟机和主操作系统。**RustMonitor 会创建一个唯一的客户虚拟机（本文称之为主虚拟机），其上运行一个通用操作系统（如 Linux），本文称之为主操作系统。除了负责在普通模式运行应用的不可信部分，主操作系统还负责任务调度、I/O 设备管理等工作，但是均不被 RustMonitor 和飞地信任。
- **内核模块。**RustMonitor 会通过主操作系统上的一个内核模块被加载、度量与启动。同时，该内核模块也用于实现一些需要特权的飞地操作。
- **飞地开发套件。**为了方便应用的开发，HyperEnclave 提供了一套飞地开发套件，其 API 与 Intel 官方的 SGX SDK[194] 兼容。SDK 包含不可信运行时（SDK uRTS）和可信运行时（SDK tRTS）两部分。基于该套件，可使大量已有的 SGX 应用无需修改源码就在 HyperEnclave 上运行。
- **App。**本章是指应用的不可信部分，运行在主操作系统上。
- **飞地。**即应用的可信部分，运行在安全模式。飞地可以是一个微虚拟机，以提供更好的性能与兼容性。

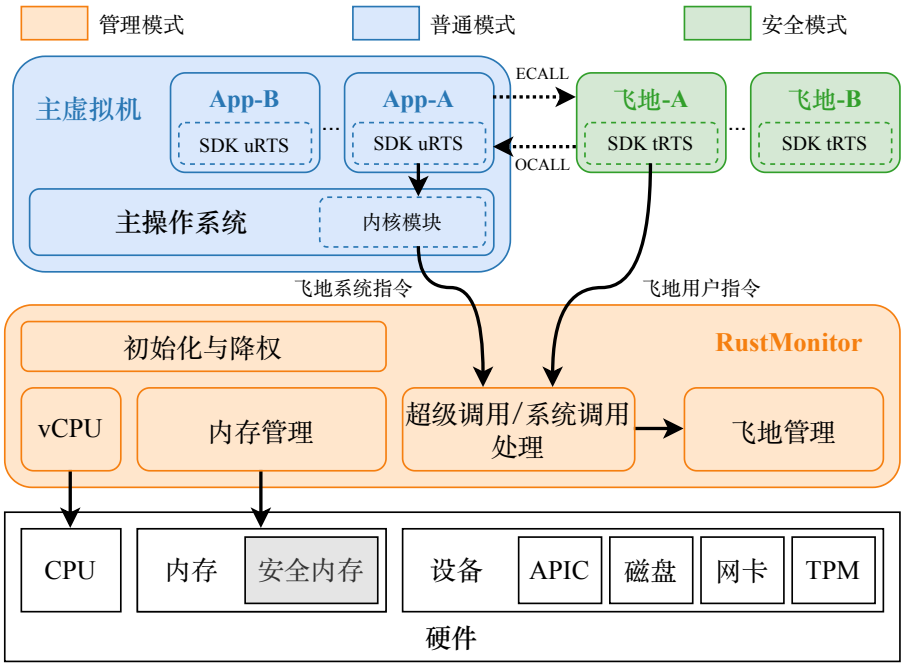


图 5.1　HyperEnclave 系统架构

5.3.2　威胁模型

与其他 TEE 的设计类似[130,195]，本文信任底层硬件，包括利用硬件虚拟化建立隔离的 CPU，以及用于建立信任根的可信平台模块。

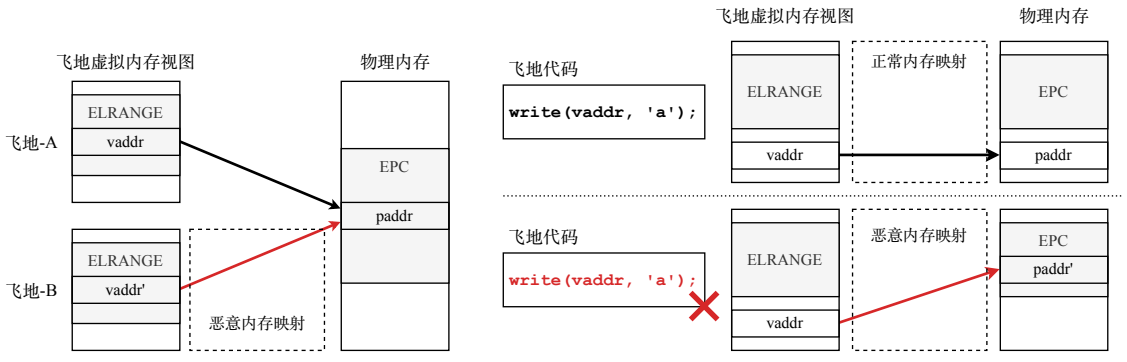
在系统启动过程中，本文假设主操作系统在早期阶段是可信的。因为在云场景下，攻击者很难在系统启动过程中进行物理攻击，并且可以通过硬件安全模块（HSM）等设备进行安全加固。

在 RustMonitor 启动后，主操作系统被降权为普通模式，本文假设此时的主操作系统是不被信任的，可能通过网络等外部访问被攻击者控制，进而对 RustMonitor 或飞地进行破坏，例如尝试直接访问受保护的内存或通过 DMA 进行攻击。HyperEnclave 也能通过硬件支持的内存加密，防止系统启动后的某些物理攻击（如冷启动与总线嗅探）。此外，本文也考虑到飞地可能因恶意代码或内存缺陷而被攻击者控制，本文的设计也能防止受损的飞地破坏其他飞地或 RustMonitor，以及针对主操作系统和应用程序代码的攻击[196]。

类似其他 TEE，本文不关注对拒绝服务攻击（DoS）和侧信道攻击的防护，例如缓存定时攻击和推测执行攻击[197]。

5.3.3　内存管理与保护

可信执行环境最关键的功能是隔离不可信应用与飞地的内存。HyperEnclave 不依赖五花八门的硬件隔离机制，而是使用如今广泛可用的硬件虚拟化（或嵌套页表）实现内存隔离。然而，使用虚拟化这样的软件方法实现 TEE 面临不少挑战，特别是在还需兼容 SGX 编程模型的情况下。本节先对这些挑战进行分析，并介绍 HyperEnclave 的内存管理与保护方案。



(a) 别名映射　　(b) 重映射

图 5.2　TEE 中几种可能的内存映射攻击

挑战。对于基于进程的 TEE，飞地运行在用户态而无法管理自己的页表。一种自然的做法是让不可信的操作系统管理飞地页表（例如 Intel SGX、TrustVisor[55]）。

然而，这种设计容易遭受内存映射攻击。如图 5.2 所示，两个不同飞地的虚拟地址不能被映射到同一物理地址（别名映射），一个飞地外的虚拟地址也不能被映射到飞地内（重映射）。为了防止此类攻击，SGX 的硬件扩展了页面缺失处理程序（Page Missing Handler, PMH），并引入了 EPCM 结构，用于在 TLB 缺失时进行额外的安全检查[118]。在没有安全硬件支持的情况下，一种常见的软件方法是*写保护页表*[55,198-199]，即通过配置嵌套页表，去掉飞地页表所在的页面的写权限，这样任何对飞地页表的更新都会陷入到 hypervisor，并由 hypervisor 验证修改的合法性。然而，在 x86 平台上，客户机页表项的访问位与脏位的更新也会导致嵌套页表缺页，陷入到 hypervisor 中，从而产生巨大的性能开销。更糟糕的是，由于将飞地的缺页交给不可信的操作系统来处理，还容易受到基于页表的侧信道攻击[127]。

另一方面，*飞地动态内存管理*（即 SGX 二代中的 EDMM[200]）也给 TEE 的设计带来更多挑战。EDMM 要求在飞地初始化完成后，还能够动态添加、删除飞地页面，或者修改飞地页面的属性或权限。在没有 EDMM 的情况下，需要在飞地初始化之前就配置好所有可能用到的物理页面，拖慢了启动速度。除了可以减少飞地的创建时间，使用 EDMM 还能在飞地中支持一些独特的功能，如栈与堆的按需分配、代码页的动态创建（用于支持 JIT 编译）等。在 SGX 二代平台上，飞地会通过 OCALL 调用向 SGX 驱动程序发出页面修改请求，然后由驱动程序完成实际的修改。由于驱动程序不受飞地信任，因此需要再次进入飞地进行安全检查才能让修改生效，从而导致频繁的飞地模式切换。

HyperEnclave 内存管理。上述挑战的根源在于传统方法让飞地的页表和缺页处理都由主操作系统来管理。因此，一种解决思路是，为飞地创建一个单独的页表，并交给可信的 RustMonitor 来管理以及处理缺页①，而不涉及主操作系统的参与。然而，这样的设计也面临着新的挑战：在 SGX 的编程模型中，飞地与不可信应用的隔离是*单向的*，飞地能够访问应用不可信部分的所有内存用于交换数据。为此，飞地的页表映射中包含了应用程序的整个地址空间，但这也使得当应用的页表映射发生变化时（例如由于页面换出），需要由 RustMonitor 同步更新飞地的页表映射，带来昂贵的同步开销。

为了避免页表同步，飞地与应用的地址空间应该各自独立，但需解决数据交换的问题。HyperEnclave 的做法是，在飞地与应用双向隔离的基础上，使用一块专门的共享内存用于数据交换。为此，在应用程序的地址空间中，会预先分配一块内存与飞地共享，本章称其为*编组缓冲区*（marshalling buffer）。编组缓冲区的地址映射在飞地的整个生命周期中是固定的，其目标物理地址不会发生改变，并且该内

① P-Enclave 可以管理自己的页表（见第 5.3.5 节）。

存区域不会被换出。这样一来，除了共享的编组缓冲区，飞地不再需要访问应用程序的地址空间，也就不需要在页表中包含其内存映射。这种设计还可以抵御一些针对 SGX 的飞地恶意软件攻击[196]，因为限制了飞地只能访问应用程序中一块受限的内存区域，而使攻击失效。关于编组缓冲区的更多安全分析详见第 5.3.7 节。

在这种设计下，RustMonitor 需要接管飞地的缺页处理与页面权限修改。例如，当飞地访问了一个未分配物理页面的虚拟地址时（例如由于页面换出或 EDMM），会触发缺页并陷入 RustMonitor 中，RustMonitor 会从飞地的内存池中取出一个空闲的页面，并在飞地的页表中添加新的映射，然后恢复飞地的执行。当飞地需要修改页面权限时，会发起一个超级调用（hypercall），由 RustMonitor 来修改飞地页表，并刷新相应的 TLB 条目①。

内存加密。为了防止针对内存的物理攻击，例如冷启动攻击和总线嗅探攻击，HyperEnclave 可以利用硬件内存加密引擎（例如 AMD SME[52] 和 Intel MK-TME[201]），以页面为粒度对部分物理内存进行加密。以 AMD SME 为例，通过设置页表项中的 C-bit，可实现对软件透明的内存加密，硬件会在向该页面写入数据时自动完成加密，并在读取时自动解密。

需要加密的内存包括 RustMonitor 自身使用的内存，以及预留给飞地的内存。而主操作系统与应用不可信部分的内存无需加密，这使得在运行非飞地应用时几乎无性能损失。

内存隔离。如图 5.3 所示，HyperEnclave 的内存隔离需要满足以下安全要求：

1. 主操作系统与应用程序不能访问属于 RustMonitor 和飞地的物理内存。
2. 飞地不能访问属于 RustMonitor 和其他飞地的物理内存，且只能访问不可信

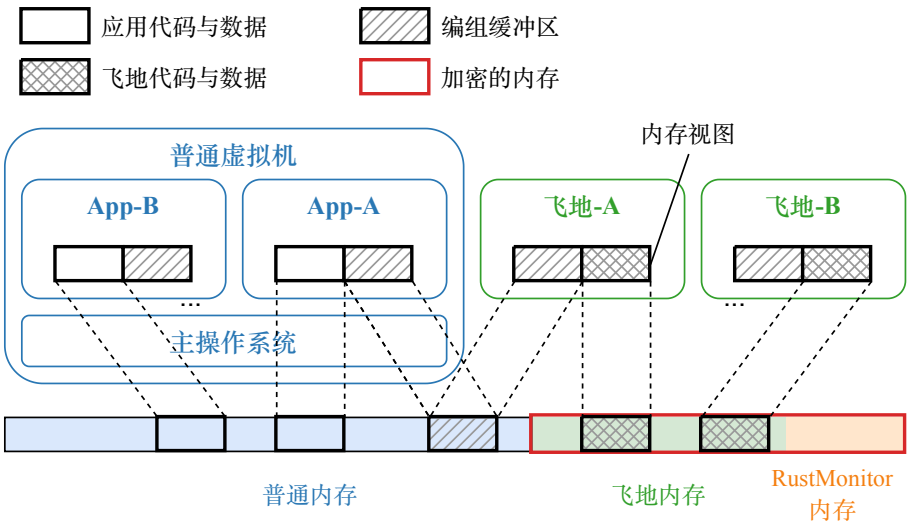


图 5.3　HyperEnclave 内存隔离

① P-Enclave 可以自行修改页面权限（见第 5.3.5 节）。

应用程序中一块受限的内存区域用于传递参数（即编组缓冲区）。

3. 恶意的外围设备不能通过 DMA 访问属于 RustMonitor 和飞地的物理内存。

HyperEnclave 通过在 RustMonitor 中为主虚拟机与飞地分别维护一份嵌套页表以满足前两个要求，同时也会维护飞地的普通页表以应对上文提到的几个挑战。为支持灵活的隔离模式（第 5.3.5 节），RustMonitor 为 HU-Enclave 只维护了普通页表而无需嵌套页表，以避免嵌套页表带来的 TLB 缺失开销。此外，为了满足第三个要求，RustMonitor 还配置了 IOMMU，以限制外围设备的物理内存访问范围。

5.3.4　飞地生命周期

为了给不同形态的微虚拟机提供统一的接口，且实现更好的兼容性，HyperEnclave 采用了与 Intel SGX 类似的编程模型与接口。SGX 的接口通过一些特殊指令来提供，根据指令所需的权限，可分为*飞地用户指令*（ENCLU）与*飞地系统指令*（ENCLS）两类。HyperEnclave 将这些指令在 RustMonitor 中进行了相应的实现，并通过超级调用（hypercall）接口暴露给上层软件[①]。飞地用户指令可直接在应用程序或飞地内调用；对于飞地系统指令，则是先通过 `ioctl()` 接口进入内核模块再进行调用。为了尽量兼容已有 SGX 应用，HyperEnclave 中涉及的大多数数据结构都与 SGX 类似（如 SIGSTRUCT 结构、SECS 页面、SSA 页面等），下文也使用 SGX 中的术语进行表述。表 5.1 列出了 HyperEnclave 支持的 SGX 指令，下面介绍使用这些指令对飞地生命周期的管理。

表 5.1　HyperEnclave 支持的 SGX 指令

用户/系统	类型	指令	描述
系统	控制	ECREATE	创建一个飞地
系统	内存	EADD	添加飞地页面
系统	内存	EEXTEND	扩展飞地度量值
系统	内存	EREMOVE	回收 EPC 页面
系统	控制	EINIT	初始化飞地
用户	控制	EENTER	进入飞地
用户	控制	EEXIT	退出飞地
用户	控制	ERESUME	重新进入飞地（AEX 处理）
用户	安全	EGETKEY	创建密码学密钥
用户	安全	EREPORT	创建密码学报告

① 对于 HU-Enclave 则是系统调用（见第 5.3.5 节）。

创建。当应用需要创建一个飞地时，通过内核模块向 RustMonitor 发起 ECREATE 请求。内核模块会准备一个页面用于存放飞地的元数据，即 SECS（SGX Enclave Control Structure）结构，其中包含飞地的虚拟地址范围（ELRANGE）、编组缓冲区大小等信息。不同的飞地需要使用不同地址的 SECS，因此 SECS 也用于标识不同的飞地。RustMonitor 根据 SECS 为飞地准备嵌套页表与普通页表，以及根据飞地隔离模式准备虚拟机或进程的执行环境（第 5.3.5 节）。

加载。新创建的飞地一开始是空的，需要将受保护的代码与数据逐页加载进飞地中。此时会先通过内核模块从 EPC 内存中分配一个页面，并调用 EADD 指令在 RustMonitor 中完成对飞地页面的添加。RustMonitor 会先对页面的地址进行合法性检查，例如确保不能有两个飞地共用同一个页面，然后在飞地的嵌套页表与普通页表中添加相应的映射，并将数据从普通内存拷贝到该 EPC 内存页中。此外，RustMonitor 还需要对页面进行度量，包括页面的内容与属性，并将度量结果保存到飞地对应的 SECS 页面中。

初始化。当飞地的所有页面都加载完毕后，通过调用 EINIT 接口来初始化飞地。应用程序会传入一个 SIGSTRUCT 结构，其中包含了飞地开发者预期的飞地度量值，以及开发者的签名。RustMonitor 会验证页面加载时计算的度量值是否与预期的一致，以及签名的有效性，以防飞地页面被篡改。此外，编组缓冲区也在飞地初始化时被添加到飞地的页表映射中。

进入与退出。在飞地初始化完成后，应用程序通过调用 EENTER 指令，将 CPU 从普通模式切换到安全模式（本文称之为模式切换），进入飞地中去执行。RustMonitor 需要保存普通模式下的 vCPU 状态，并填入目标飞地的状态，例如指令指针、栈指针、TLS 指针等。最后，将普通页表与嵌套页表切换为飞地的，并刷新 TLB。飞地通过 EEXIT 返回到应用程序，与进入时相反，RustMonitor 需要将 vCPU 状态、普通页表与嵌套页表都恢复为普通模式的①。以上切换过程只对部分 vCPU 寄存器进行了切换，而其余大部分控制寄存器都无需切换，让飞地与应用程序共用，因此模式切换的速度要高于完整虚拟机之间的切换。

飞地在运行过程中可能会发生异常或中断，即异步飞地退出（Asynchronous Enclave Exit, AEX）。在一般情况下，AEX 需要先进入主操作系统的处理例程，根据需要再进入飞地中的处理例程（即两阶段异常处理[202]）。在 HyperEnclave 中，飞地的 vCPU 会被配置为拦截所有异常与中断，因此 AEX 会先陷入 RustMonitor，RustMonitor 将飞地上下文保存到一个 SSA（State Saving Area）结构中，清空通用寄存器以防敏感信息泄露，最后向主虚拟机注入相应的异常或中断。在主操作系

① 对于 HU-Enclave，切换的是物理 CPU 而非 vCPU 的状态（见第 5.3.5 节）。

统中完成 AEX 的处理后，应用程序调用 ERESUME 指令，从 SSA 恢复上下文并继续飞地的执行。本文在第 5.3.5 节提出的 P-Enclave 支持直接在飞地内处理异常与中断，避免多次模式切换，大大减少了开销。

销毁。当完成飞地内的计算后，由应用程序调用 EREMOVE 接口来销毁飞地。RustMonitor 会回收飞地的所有 EPC 页面，从飞地的普通页表与嵌套页表中删除相应的映射，并将页面清零以防信息泄露。

5.3.5　灵活的飞地隔离模式

为了同时支持用户态、特权态微虚拟机的运行，从而满足不同场景下应用对性能与安全的需求，HyperEnclave 可为飞地选用不同的隔离模式，它们在性能与安全等级上各有所长。

如图 5.4 所示，现有 TEE 设计一般只支持将飞地运行在一个固定的模式，例如 Intel SGX 在（主机）用户态运行飞地，TrustVisor[55] 在（客户机）用户态运行受保护的代码（PAL）。而 HyperEnclave 支持飞地在以下三种模式中运行：① GU-Enclave 运行在客户机用户态；② P-Enclave 运行在客户机特权态（以及可选的客户机用户态）；③ HU-Enclave 运行在主机用户态。

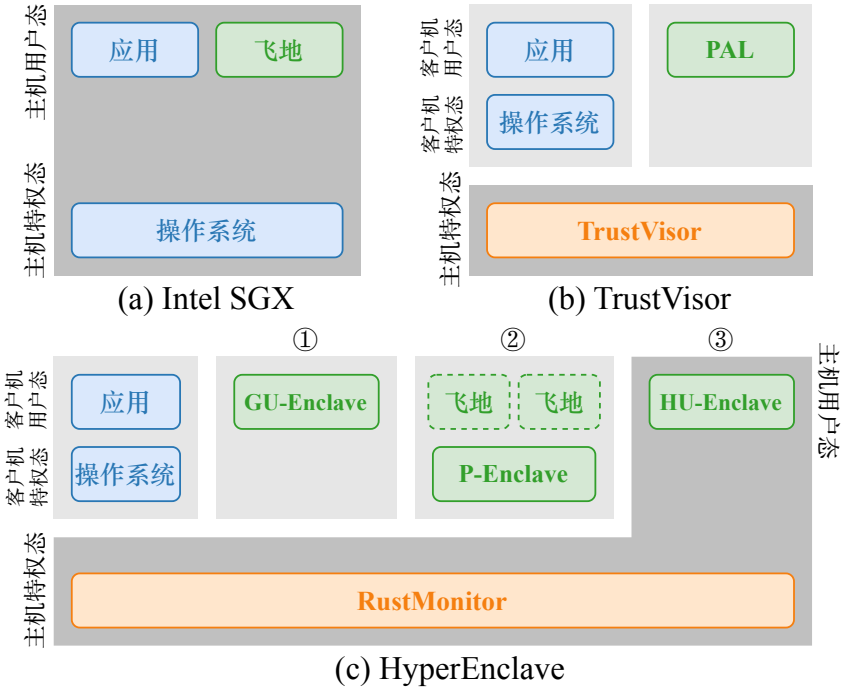
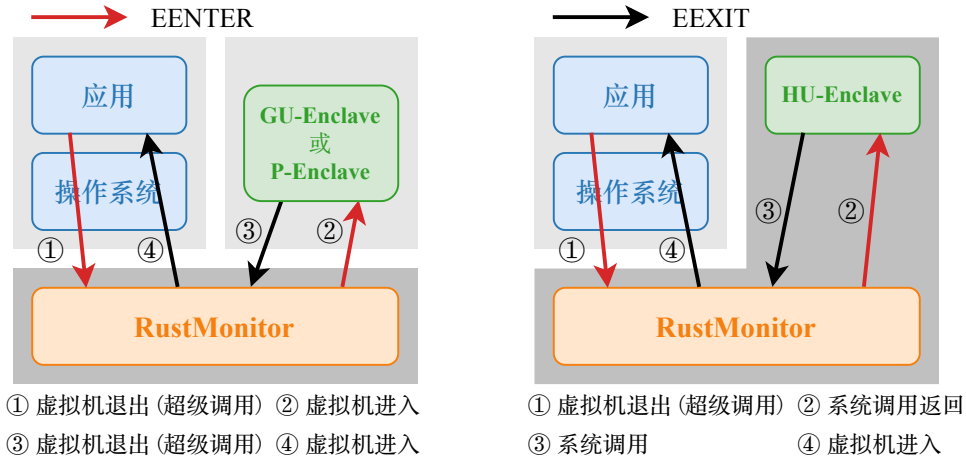


图 5.4　HyperEnclave 不同飞地隔离模式的比较

不同的飞地隔离模式使用不同的上下文切换代码，有着不同的切换时间。具体来说，如图 5.5(a) 所示，GU-Enclave 和 P-Enclave 使用超级调用和虚拟机退出来进入和退出飞地；在图 5.5(b) 中，HU-Enclave 使用系统调用和系统调用返回来进



(a) GU-Enclave 与 P-Enclave　　(b) HU-Enclave

图 5.5　HyperEnclave 不同飞地隔离模式下的切换分析

入和退出飞地，具有更低的切换开销。

客户机用户态。 GU-Enclave（guest user enclave）是最基本的飞地模式，通常用于运行计算密集型任务。在这种模式下，飞地在客户机用户态运行（如 VMX non-root ring-3），利用嵌套页表进行隔离。

GU-Enclave 使用类似虚拟机的运行环境，RustMonitor 会为其准备 vCPU 上下文，以及客户机页表与嵌套页表。而飞地与主虚拟机间的切换通过虚拟机进入/退出指令触发，然后由 RustMonitor 切换相应的 vCPU 上下文以及这两份页表。

主机用户态。 HU-Enclave（host user enclave）运行在主机用户态，主要通过将虚拟机模式的切换（约 880 个时钟周期）优化为 CPU ring 的切换（约 120 个时钟周期），来提供更快的切换性能（图 5.5）。此外，HU-Enclave 还消除了 GU-Enclave 中因虚拟化带来的额外开销，如 vCPU 上下文的切换，以及二级地址翻译的 TLB 开销。根据第 5.5.7 节中的评估，HU-Enclave 可显著提升 I/O 密集型应用的性能。GU-Enclave 与 HU-Enclave 均可支持用户态微虚拟机。

在加载 HU-Enclave 时，RustMonitor 会为其准备一个进程上下文而非 vCPU，同时只需维护普通的一级页表而无需嵌套页表。在进入飞地时，RustMonitor 切换的是物理 CPU 的状态，并通过系统调用返回指令（x86 的 `SYSRET`）来进入 HU-Enclave。相应地，在从飞地中退出时，HU-Enclave 通过系统调用指令（x86 的 `SYSCALL`）陷入 RustMonitor。其他飞地用户指令（如 `EGETKEY`、`EREPORT`）会被当成系统调用在 RustMonitor 中进行实现。对于 HU-Enclave 内产生的中断和异常，是经过 IDT 的处理流程陷入 RustMonitor，而非虚拟机退出。

特权态。 HyperEnclave 可在客户机特权态运行 P-Enclave（privileged enclave），以支持特权态微虚拟机。在这种模式下，飞地可以访问 GDT、IDT 和一级页表，这

会给一些应用程序带来收益，正如 Dune[25] 中所展示的。其中一个例子是垃圾回收（例如 Java 和 Go 应用）。垃圾回收器需要经常修改页面权限以触发缺页，来跟踪页面的访问情况。对于用户态的飞地（如 GU-Enclave 和 HU-Enclave），需要借助主操作系统来处理缺页，更新页表，从而需要频繁的模式切换而导致巨大的性能开销。P-Enclave 可以配置自己的异常处理例程（如缺页处理），使得可以完全在飞地内处理异常而无需切换出来。RustMonitor 会将一个异常类型的白名单提供给 P-Enclave，而将其他异常仍转发给主操作系统。此外，P-Enclave 还支持基于页表的飞地内隔离方案，例如对不受信任的第三方库进行沙箱化。

另外，当飞地有了接收中断的能力后，P-Enclave 还可以通过计算中断频率来检测不正常的中断事件，然后请求 RustMonitor 将它们转发到主操作系统。因此，现有的基于中断的侧信道攻击[203-208]可以被检测与防御。

P-Enclave 与 GU-Enclave 类似，也需要 RustMonitor 为其准备 vCPU 的上下文。不过在模式切换时，P-Enclave 需要多切换一些 vCPU 特权状态，使得切换速度比 GU-Enclave 稍慢。

不同隔离模式的选择。应用程序可以在本节提出的三种隔离模式中进行灵活选择，来满足不同场景下的性能与安全需求。

在性能方面，HU-Enclave 提供了最快的切换时间与最低的运行时开销，其次是 GU-Enclave，然后是 P-Enclave，不过 P-Enclave 可以在一些场景下通过对特权硬件资源的直接访问来减少切换的次数，从而获得更好的性能。

在安全方面，虽然很难实现绝对的安全，但可以从隔离方式与攻击难度角度来评估相对的安全性。GU-Enclave 与 P-Enclave 均使用嵌套页表进行隔离，并且运行在客户机模式而非主机模式，对攻击者来说，能比 HU-Enclave 提供更深的防御。此外，拥有更多特权的 P-Enclave 以及运行在主机用户态的 HU-Enclave，可能会向恶意的飞地暴露更多攻击面，相对更容易逃逸到运行 RustMonitor 的主机特权态。

5.3.6　可信启动与远程证明

可信执行环境的另一个重要功能是对硬件平台与其上运行的软件进行认证。本节先介绍 HyperEnclave 的另一硬件需求——可信平台模块的相关背景，然后介绍 HyperEnclave 的可信启动流程与远程证明机制。

背景：可信平台模块。可信平台模块（Trusted Platform Module, TPM）是一个安全密码处理器，既是行业标准[60]，也是 ISO/IEC 标准[209]，几乎被用于所有的个人计算机与服务器中。

TPM 是 TEE 实现远程证明的关键一环。其具有一组平台配置寄存器（PCR），可用于在启动过程中度量启动代码。PCR 会在系统重新启动或开关机时被重置为

零。在系统启动过程中，PCR 只能被更新为新的度量值（称为 PCR 扩展），而不能被设置为任意值。

每个 TPM 芯片都配备了一个唯一的非对称密钥，称为认可密钥（Endorsement Key, EK），由制造商作为信任根嵌入其中。TPM 可以生成一个 PCR 值的报告，并使用 TPM 身份认证密钥（Attestation Identity Keys, AIK）进行签名，其中 AIK 是在 TPM 内部生成的，并使用 EK 进行认证。对启动代码的任何修改都会反映在报告中。远程方在收到报告后，可以验证签名是否来自一个可信的 TPM，以及 PCR 的摘要报告是否被篡改。

度量延迟启动。HyperEnclave 基于 TPM 实现可信启动，其流程如图 5.6 所示，本文称之为*度量延迟启动*（measured late launch）。在系统启动过程中，最先执行的是一段静态且不可变的代码片段，被称为核心度量信任根（Core Root of Trust for Measurement, CRTM），用于为后续启动的固件和软件建立度量链，包括 BIOS、grub、主操作系统内核以及初始内存文件系统（initramfs）。每个启动组件的度量结果都会被存储到 TPM PCR 中，因此任何修改都会在最终认证报告中被反映出来。

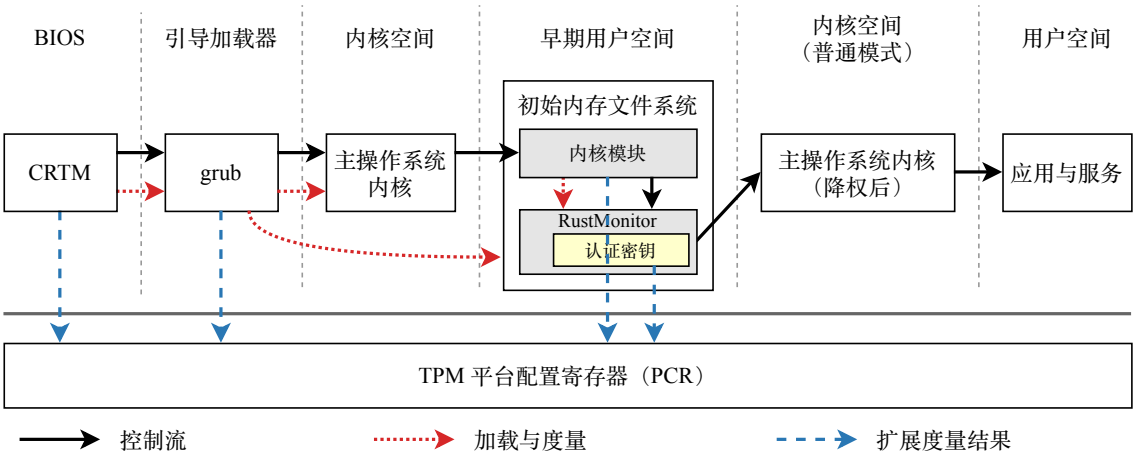


图 5.6　HyperEnclave 启动流程：度量延迟启动

为了减少来自主操作系统的攻击面，HyperEnclave 将 RustMonitor 的镜像放入 initramfs 中。主操作系统的内核启动后，会在一个内核模块中对 RustMonitor 镜像进行度量，并将结果扩展到 TPM PCR 中，然后在*早期用户空间*（early userspace）启动 RustMonitor。此时依赖磁盘、文件系统、网络的用户程序还未开始执行，难以被攻击者利用。通过此方式对启动过程进行度量，可以保证加载 RustMonitor 时的软件状态是可信的。

当 RustMonitor 加载完成后，控制流会跳转到预定义的入口。RustMonitor 会配置自己的执行环境，包括栈、页表、IDT 等，并为每个 CPU 准备 vCPU 的状态（利用保存的主操作系统的上下文）。之后，RustMonitor 会启动主虚拟机，并将主

操作系统降权为普通模式。最后，控制流返回到内核模块，内核在普通模式继续后续的启动流程，不会感知到 RustMonitor 的存在。

HyperEnclave 基于以上启动方法，使得 RustMonitor 在加载时类似类型二的虚拟机管理程序（如 KVM[210]），但在运行时类似类型一的虚拟机管理程序（如 Xen[68]），借用了通用操作系统的部分启动流程，减小了可信基。通过这种方式，一旦主操作系统被降级到普通模式，RustMonitor 就不再需要信任主操作系统。

远程证明。在度量延迟启动的流程中，所有启动的组件都会被度量并扩展到 TPM 中。当 RustMonitor 完成启动后，需要将度量值扩展到飞地中。为此，RustMonitor 会派生一对认证密钥公私钥对（hypervisor attestation key），用于对飞地度量结果进行签名。然后，RustMonitor 会将派生出的公钥扩展到 TPM PCR 中，而私钥受到内存隔离与加密的保护，永远不会离开 RustMonitor。

在创建飞地时，所有添加到飞地中的页面（包括页面内容、页面类型以及读写执行权限）都会被 RustMonitor 度量，从而生成整个飞地的度量结果。这个中间的度量结果会被存放在 RustMonitor 的内存中，对飞地和主操作系统都是不可见的。

与 TPM 和 Intel SGX 类似，HyperEnclave 采用了 SIGn-and-MAc（SIGMA）认证协议来进行远程证明。如图 5.7 所示，HyperEnclave 的认证报告主要由飞地度量结果以及平台度量结果（均为签名后）组成。其中飞地的度量结果会使用 RustMonitor 的认证密钥进行签名，形成飞地度量签名（ems）。RustMonitor 认证密钥的公钥也包含在认证报告中（hapk）。而平台的度量结果由 TPM 操作生成（TPM_Quote），其中包含了所有启动代码（包括 CRTM、BIOS、grub、内核、initramfs 和 RustMonitor）的度量结果（PCR 值的摘要），以及使用 TPM 认证密钥（tapk）进行的签名。远程用户在收到认证报告后，可以通过比较平台和飞地的度量结果，以及生成签名的证书链，来验证报告的真实性和完整性。

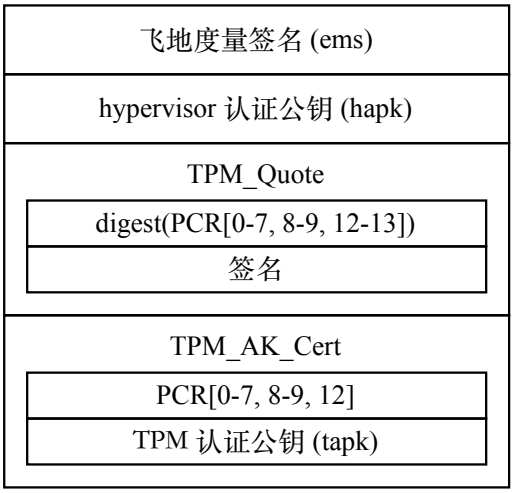


图 5.7　HyperEnclave 认证报告的结构

安全密钥生成。当 RustMonitor 首次初始化时，会利用 TPM 的随机数发生器生成一个根密钥 K_{root}，并通过 TPM 的密封操作（seal）将其存放在 TPM 之外。在系统重新启动后，RustMonitor 会使用 TPM 的解封操作（unseal）解密 K_{root}。只有完全相同的 TPM 芯片和匹配的 PCR 配置才能正确解封 K_{root}。此外，RustMonitor 还会在完成启动后将 PCR 填充一个恒定的值，以防主操作系统恶意获取 K_{root}。所有其他的密钥，包括飞地的密封密钥和报告密钥，都是从 K_{root} 和飞地的度量结果中派生而来的。

5.3.7　安全分析

信任建立。HyperEnclave 依赖对启动过程进行度量来建立对 RustMonitor 的信任，这是 TEE 设计中的常见方法（如 TrustZone[54] 和 Keystone[130]）。在启动过程中，所有的组件（包括 CRTM、BIOS、grub、内核、initramfs 和 RustMonitor）都被度量并扩展到 TPM 的 PCR 中。因此，对启动代码的任何篡改都将通过远程证明得到反馈。

HyperEnclave 会在受控的环境中部署（例如云计算场景中的数据中心），以限制攻击者对平台的物理访问。为了减少攻击面并最小化可信基，HyperEnclave 将 RustMonitor 的镜像放入 initramfs 中，并在早期用户空间加载和度量 RustMonitor。在该阶段，主操作系统的内核不会接受来自用户的外部输入，并且禁用了诸如网络之类的外设。通过度量延迟启动的方法（第 5.3.6 节），在主操作系统降级为普通模式后，RustMonitor 不再需要信任主操作系统。

飞地内存隔离。如第 5.3.3 节所述，飞地的内存和页表都由 RustMonitor 维护，主操作系统无法访问它们。同时，TLB 会在模式切换时被清空，以防止使用过时的 TLB 条目进行非法内存访问。RustMonitor 通过从主操作系统的嵌套页表中删除相应的映射，来防止主操作系统访问保留的受保护物理内存。RustMonitor 还配置了 IOMMU 以防未经授权的设备访问保留的物理内存。

HyperEnclave 的设计可以防止内存映射攻击，因为主操作系统无法干预飞地的地址映射。飞地需要与不可信的应用共享一块特殊内存（编组缓冲区），为了防止应用使用一个精心构造的地址（例如覆盖飞地的内存）来劫持编组缓冲区，RustMonitor 会在将该区域的映射添加到飞地页表之前，确保其地址范围处于飞地地址范围之外。虽然在极端情况下，攻击者可能会控制编组缓冲区，但这也不会导致新的安全问题，因为编组缓冲区在设计上就是不被信任的，开发人员需要自行验证通过缓冲区传递的数据是可靠的（与 SGX 的威胁模型一致）。

防范恶意飞地。先前的工作指出，飞地恶意软件可能会窃取机密数据、劫持飞地外应用程序的控制流[196]。HyperEnclave 的设计可以抵御以下几种潜在的恶意

飞地行为：

- *防止对应用程序的任意内存访问*。在 SGX 中，飞地可以访问应用程序的整个地址空间，这可能导致密钥或栈保护字（stack canaries）被泄露，或者代码指针被篡改从而使控制流被劫持。在 HyperEnclave 中，由于飞地只能访问自己的内存以及用于传递参数的编组缓冲区，此类攻击将得到避免。
- *防止执行 EEXIT 后的任意跳转*。通过在执行 `EEXIT` 指令之前设置 `RBX` 寄存器的值，SGX 允许在飞地中跳转到任意地址，从而给飞地恶意软件提供攻击面[196]。在 HyperEnclave 的设计中，由于 `EEXIT` 指令被 RustMonitor 模拟，可以很容易地防止此类攻击，只需在调用 `EEXIT` 时添加相应的检查。

物理攻击。硬件内存加密技术（如 AMD SME）可以用于保护飞地免受物理攻击，如冷启动攻击和总线嗅探攻击。通过内存加密，数据将始终在内存或内存总线上进行加密，并且只有在 CPU 内部被解密。内存加密的密钥在系统启动时会随机生成并存储在 CPU 中，无法被软件显式访问。

侧信道攻击。与 SGX 相比，HyperEnclave 可以防止某些类型的侧信道攻击。由于飞地的页表和缺页都由 RustMonitor 处理而无需不可信的主操作系统的参与，后者将无法进行基于页表的攻击[127-128,211]。本文将基于微架构的攻击的防护留作未来工作（例如推测执行攻击[197]）。

5.4　系统实现

本章在支持硬件虚拟化和内存加密的 AMD 平台上实现了 HyperEnclave。在目前的实现中，RustMonitor 主要由大约 7,500 行 Rust 代码编写而成，运行在主操作系统上的内核模块有大约 3,500 行 C 代码。此外 HyperEnclave 还对 Intel 官方的 SGX SDK（v2.13）进行了约 2,000 行的代码修改。

5.4.1　微虚拟机管理程序

RustMonitor 作为微虚拟机管理程序，运行在最高特权级并保证飞地的隔离，因此需要非常高的安全性。为了减少内存和并发缺陷带来的安全风险，RustMonitor 几乎完全使用内存安全的 Rust 实现（正如其名），仅少量上下文切换代码使用汇编。与现有的一些通用虚拟机管理程序相比（如 KVM 和 Xen），RustMonitor 代码量更小，也更容易进行形式化验证。

当系统启动时，会通过 Linux 命令行参数将一块物理内存区域保留，专门留作 RustMonitor 和飞地使用。RustMonitor 主要对表 5.1 中的 SGX 指令进行了实现，并负责管理飞地的页表与处理缺页。

5.4.2　内核模块

内核模块在系统启动时由主操作系统加载。然后，内核模块会完成对 RustMonitor 的加载、度量和启动，并将度量结果扩展到 TPM PCR 中，作为 TPM 报告的一部分。当内核模块加载时，会创建一个设备文件，并挂载到 `/dev/hyper_enclave`。应用程序可以通过打开该设备文件，使用 `ioctl()` 来调用飞地系统指令。此外，内核模块还负责对 EPC 页面的分配，但页面映射的合法性检查以及让映射最终生效仍在可信的 RustMonitor 中完成。

5.4.3　飞地开发套件

为了方便 HyperEnclave 上应用程序的开发，同时复用 Intel SGX 丰富的生态，HyperEnclave 提供了一个飞地开发套件，与 Intel 官方的 SGX SDK[194] 在 API 层面兼容。因此，原来为 Intel SGX 编写的应用，无需修改源码，只需切换 SDK 并重新编译就能在 HyperEnclave 上运行。为了支持更多应用程序，HyperEnclave 还移植了 Rust SGX SDK[187] 和 Occlum 库操作系统[36]。HyperEnclave 的飞地开发套件在 Intel 官方 SGX SDK 的基础上改进而来，主要修改如下：

SGX 指令替换。如第 5.3.4 节所述，RustMonitor 将 SGX 接口以超级调用的形式提供给上层软件。为此，需要将官方 SDK 中原有的 SGX 飞地用户指令进行替换（`EENTER`，`EEXIT`，`ERESUME` 等），根据飞地的不同隔离模式，替换为超级调用或是系统调用。同时让这些调用的参数语义与顺序均与 SGX 保持一致，以保证兼容性。

使用编组缓冲区传递参数。在 HyperEnclave 中，飞地只能访问自己的地址空间，以及一块与应用程序共享的内存用于传递参数（编组缓冲区）。为了实现编组缓冲区的初始化，需要修改 SDK 的不可信运行时库（`libsgx_urts.so`）。编组缓冲区在飞地初始化过程中通过 `mmap()` 分配而来，并设置了 `MAP_POPULATE` 标志，用于直接分配物理内存并建立映射，而不使用按需分配。此外，SDK 还会通过一个 `ioctl()` 调用来要求主操作系统在飞地的生命周期内不要压缩或换出这些物理页面（在内核模块中实现）。

当应用程序调用 `EINIT` 接口初始化飞地时，会将编组缓冲区的基地址和大小传递给 RustMonitor，从而在飞地的页表中添加编组缓冲区的映射。编组缓冲区的基地址和大小也会传递给可信运行时库，以便在飞地中访问编组缓冲区内的数据。编组缓冲区的大小可以在飞地配置文件中进行配置。

对于边缘调用（edge calls），SGX SDK 在实现 OCALL 时，会在飞地内部调用 `sgx_ocalloc()` 函数，在不可信应用的栈上分配一块区域，用于跨飞地内外

的数据传输，因为原来的 SGX 设计允许飞地可以访问应用的整个地址空间（包括栈）。HyperEnclave 对 OCALL 的支持非常容易，只需要修改 `sgx_ocalloc()`，将该区域的分配从栈上改为编组缓冲区上。而为支持 ECALL 时的参数传递，则需要将数据拷贝到编组缓冲区中。为此，HyperEnclave 修改了 SDK 的 Edger8r 工具（使用 OCaml 语言编写），能自动生成将传输数据按一定格式存储到编组缓冲区中的代码，使其对开发人员来说是透明的。

SGX 的编程模型还支持使用 `user_check` 属性的参数。对于此类参数，Edger8r 工具不会生成检查地址范围和执行数据移动的代码。由于在 SGX 的飞地中可以访问应用的整个地址空间，一些飞地为了性能，可能会使用 `user_check` 属性的指针直接操作飞地外部的数据，从而无需跨飞地边界拷贝数据的开销。为了支持此类应用，HyperEnclave 为开发人员提供了一个接口，可以手动在编组缓冲区中分配一个内存块并随意使用，以达到类似 `user_check` 属性参数的效果。

远程证明。在 HyperEnclave 中，远程证明的流程与 SGX 类似，遵循相同的 SIGn-and-MAc（SIGMA）协议。HyperEnclave 扩展了 SDK 中的 `sgx_quote_t` 结构，让其包含 HyperEnclave 的认证报告信息（图 5.7）。这些修改对飞地开发者来说都是透明的。

5.5　实验与评估

本节将通过一系列实验来全面评估 HyperEnclave 的性能，并回答以下问题：

- HyperEnclave 引入的虚拟化层会对普通应用产生多大的开销？
- HyperEnclave 引入的不同隔离模式分别有怎样的性能？
- HyperEnclave 引入的 P-Enclave 会在什么场景下带来多大的收益？
- HyperEnclave 引入的编组缓冲区会产生多大的开销？
- HyperEnclave 的内存加密性能如何？
- HyperEnclave 在真实世界应用中的性能表现如何？

5.5.1　实验环境

本章采用的实验平台配有两颗 AMD EPYC 7601 CPU（共 64 个物理核心，128 个逻辑核心）和 512 GB 内存，其中配置了 2GB 保留内存用于 RustMonitor，24 GB 用于 EPC 内存（飞地页面）。主操作系统为 Ubuntu 18.04，内核版本为 Linux 4.19.91。为了与 Intel SGX 进行比较，本节还使用一台配有 Intel Xeon E3-1270 v6 CPU 和 64GB 内存的机器进行了相同的实验。HyperEnclave 和原生 SGX 使用的 SDK 版本均为 2.13。所有程序均使用 GCC 7.5.0 编译，并开启相同的优化等级。

本节采取了一些方法，尽量屏蔽不同的硬件平台对比较性能带来的影响。首先，除非明确说明，所有实验的内存使用均没有超过 EPC 的大小，以避免昂贵的页面交换。其次，所有测试都是在单线程模式下运行的。对于微基准测试（表 5.3 和表 5.4），实验使用相同的 SGX SDK 版本来比较 AMD 硬件上的 HyperEnclave 和 Intel 硬件上的原生 SGX，并且使用 CPU 时钟周期数作为单位，以避免 CPU 频率的影响；对于真实世界应用，实验将基准线设置为没有安全保护的 SDK 仿真模式，而去比较 HyperEnclave 和原生 SGX 引入的相对开销。因此，不同机器上的绝对性能结果是不可直接比较的。

所有在 HyperEnclave 上的性能测试均开启了内存加密。然而，由于第一代 SGX 和 HyperEnclave 的内存加密方式有所不同，SGX 采用的是 Merkel 树和 AES-CTR，而 HyperEnclave（AMD SME）采用的是 AES-XTS（图 5.10 给出了内存加密开销的评估），这可能解释了 HyperEnclave 在内存密集型工作负载下性能比 SGX 好的原因。此外，HyperEnclave 的模式切换更快（特别是 HU-Enclave），这解释了其在 I/O 密集型工作负载下性能好的原因。

5.5.2　虚拟化开销

HyperEnclave 将主操作系统运行在虚拟机中，可能会引入一些开销，本节的实验将对此进行评估。实验分别在启用和不启用 RustMonitor 的情况下，在主操作系统中运行 LMBench[212]、Linux 内核（v5.15）构建和 SPEC CPU 2017 INTSpeed 基准测试[213]。表 5.2 与图 5.8 的结果均显示，在大多数基准测试中，虚拟化的开销低于 1%。这表明即使 HyperEnclave 引入了额外的虚拟化层，运行在主操作系统上的非 SGX 应用程序的性能也几乎不受影响。这得益于 HyperEnclave 将大多数设备直通到主虚拟机，并尽可能在嵌套页表中使用大页，以减轻 TLB 的压力，从而大大减少了虚拟机的退出次数。

表 5.2　HyperEnclave 虚拟化开销：LMBench 和 Linux 内核构建

	LMbench （微秒）						内核构建（秒）
	null call	fork	ctxsw	mmap	page fault	AF UNIX	
原生	0.1195	196.3	3.13	66,125	0.2433	5.73	1,410
启用虚拟化	0.1192	197.9	3.22	66,407	0.2461	5.69	1,417
开销	-0.25%	0.82%	2.88%	0.43%	1.15%	-0.70%	0.50%

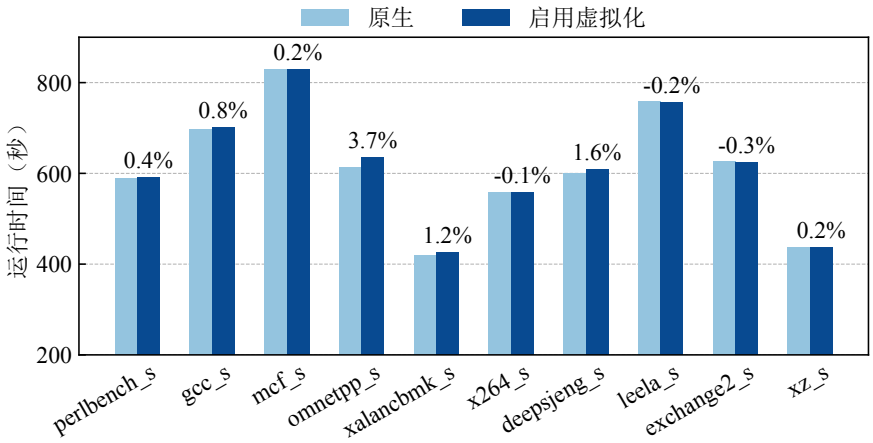


图 5.8　HyperEnclave 虚拟化开销：SPEC CPU 2017

5.5.3　模式切换开销

HyperEnclave 提供了不同隔离模式的飞地，本节将评估它们各自的切换性能。实验在 HyperEnclave（不同的飞地隔离模式）和 Intel SGX 上分别测量了 SGX 边缘调用（ECALL 和 OCALL）的执行时间。测试代码运行了 1,000,000 次没有参数的空边缘调用，并取中位数。实验还测量了在 HyperEnclave 上实现的 `EENTER` 和 `EEXIT` 操作的指令级延迟。不过由于实验所用的 SGX 平台不支持在飞地内执行 `RDTSCP` 指令，因此无法测量 SGX 上的指令级延迟。

表 5.3 给出了测试结果。可见 HU-Enclave 具有最佳的边缘调用性能，因为它将虚拟机模式的切换（约 880 个时钟周期）优化为 CPU ring 的切换（约 120 个时钟周期）。而 P-Enclave 比 GU-Enclave 稍慢，因为 P-Enclave 需要切换更多的特权状态。所有结果均与 Intel SGX 相当。

表 5.3　HyperEnclave 和 Intel 上执行 SGX 原语所需的 CPU 周期数

	`EENTER`	`EEXIT`	ECALL	OCALL
Intel SGX	—	—	14,432	12,432
HU-Enclave	1,163	1,144	8,440	4,120
GU-Enclave	1,704	1,319	9,480	4,920
P-Enclave	1,649	1,401	9,700	5,260

5.5.4　飞地异常处理开销

为了展示 P-Enclave 在特定场景下的优势，本节评估了飞地内异常处理的性能。实验首先使用未定义指令异常（`#UD`），测试代码通过在飞地中执行未定义指令来触发异常，异常处理例程仅做指令指针的推进就直接返回。对于 P-Enclave，异常可

完全在飞地内部捕获和处理，不发生模式切换。而对于 GU-Enclave 和 SGX，异常会触发一次 AEX，先切换到不可信的操作系统，然后执行两阶段的异常处理[202]，具有较长的调用路径。表 5.4 的第一行显示，在 P-Enclave 内部处理异常，速度分别比 GU-Enclave 和 Intel SGX 快约 68 倍和 110 倍。

表 5.4　HyperEnclave 处理飞地内异常所需的 CPU 周期数

	Intel SGX	GU-Enclave	P-Enclave
#UD	28,561	17,490	258
#PF	—*	2,660	1,132

* 实验所用的 SGX 一代平台不支持在飞地初始化完成后修改页面权限。

实验还模拟了一个典型的垃圾回收场景。测试代码首先分配了一块大的内存块，然后通过更改飞地的页表映射来取消对内存块的写权限；之后，飞地访问内存块以触发缺页异常（#PF）；在异常处理例程中，会将写权限恢复。表 5.4 的第二行显示，P-Enclave 比 GU-Enclave 大约快 2.3 倍，因为 P-Enclave 可以在飞地内自行处理缺页异常并更新页表，而 GU-Enclave 需要陷入 RustMonitor 来更新页表。

5.5.5　编组缓冲区开销

为了衡量 HyperEnclave 引入的编组缓冲区的开销，本节构建了一个不使用编组缓冲区的 HyperEnclave 版本作为基线。实验以边缘调用时传输的数据大小，以

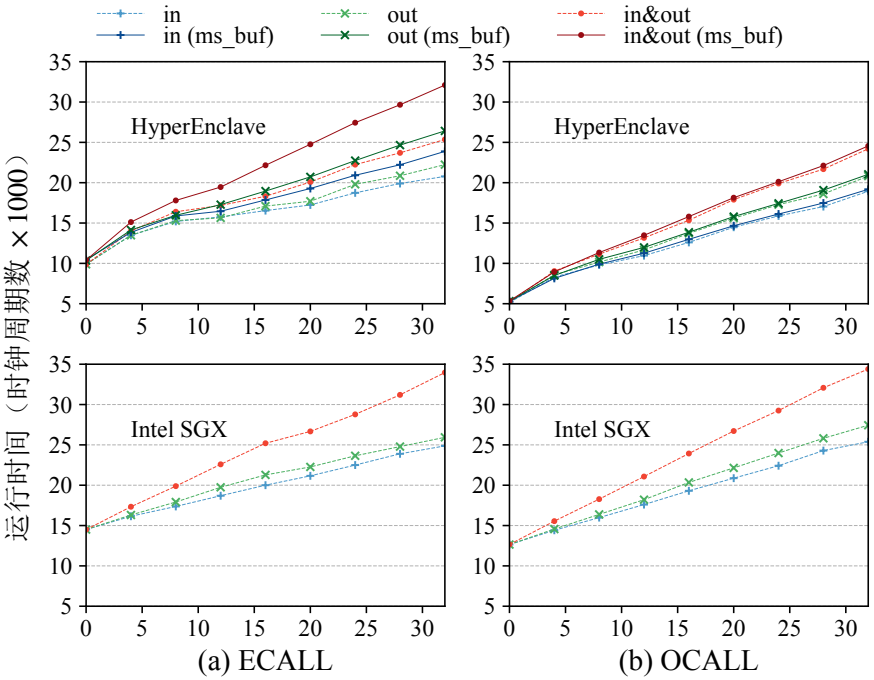


图 5.9　HyperEnclave 编组缓冲区开销

及数据的移动方向作为自变量，分别在 HyperEnclave（GU-Enclave）和 Intel SGX 上测量了带参数的 ECALL 和 OCALL 调用的开销。在数据传输完成后，会通过 CLFLUSH 指令确保数据不被缓存。

图 5.9 给出了实验结果，可见开销几乎随传输数据的大小线性增长。对于 ECALL，HyperEnclave 在传输 16KB 数据时，“in”、“out” 和 “in&out” 方向的开销分别为 8%、11% 和 21%，这是由于 ECALL 时需要一次额外的内存拷贝。对于 OCALL，由于在实现过程中没有额外的内存拷贝（第 5.4.3 节），开销与基线几乎一致。在许多实际工作负载中，ECALL 的数据传输往往只占总处理时间的很小一部分，特别是对于计算密集型和内存密集型任务。因此，编组缓冲区的开销对实际应用的影响很小。

5.5.6　内存加密开销

HyperEnclave 基于 AMD SME 实现硬件内存加密，本节将对比 HyperEnclave 与 Intel SGX 的内存加密开销。实验在顺序访问和随机访问模式下，测量启用和不启用加密时的内存访问延迟。其中缓冲区大小从 16KB 逐渐增大到 256MB。图 5.10 给出了在 HyperEnclave 和在 SGX 上的结果。当缓冲区大小超过最后一级缓存大小时（8 MB），对于顺序访问和随机访问，HyperEnclave 上的开销分别约为 2.4 倍和 25 倍；而在 SGX 上分别约为 3 倍和 30 倍。当缓冲区大小超过 EPC 大小时（93 MB），由于 EPC 页面交换，SGX 上顺序访问和随机访问的开销分别超过 45 倍

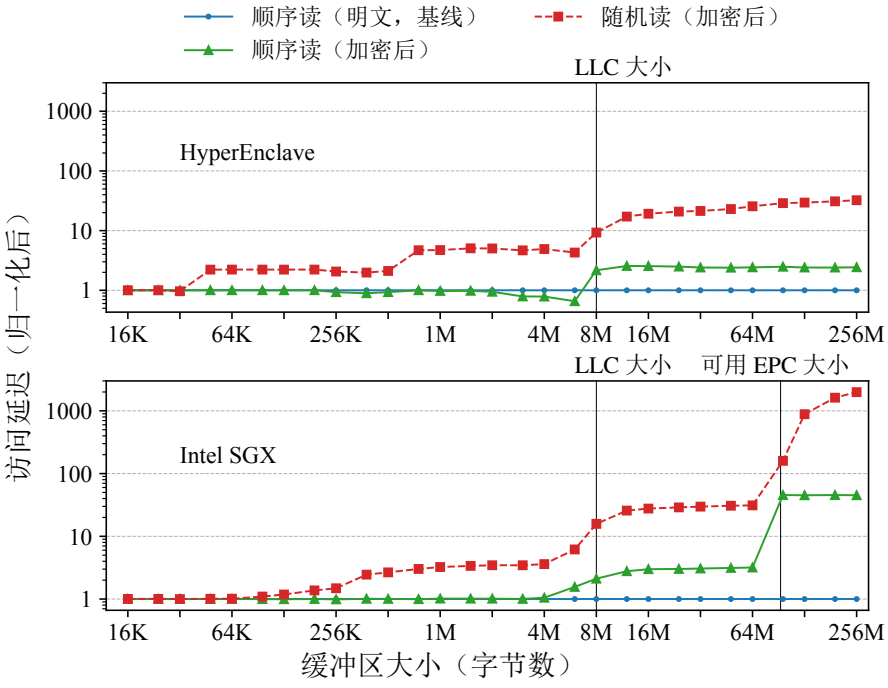


图 5.10　HyperEnclave 和 Intel SGX 上访问加密内存的开销

和 1000 倍，而在 HyperEnclave 上，由于保留了 24GB 内存作为飞地内存以避免页面交换，使得开销仍小于 30 倍。

5.5.7　真实应用性能

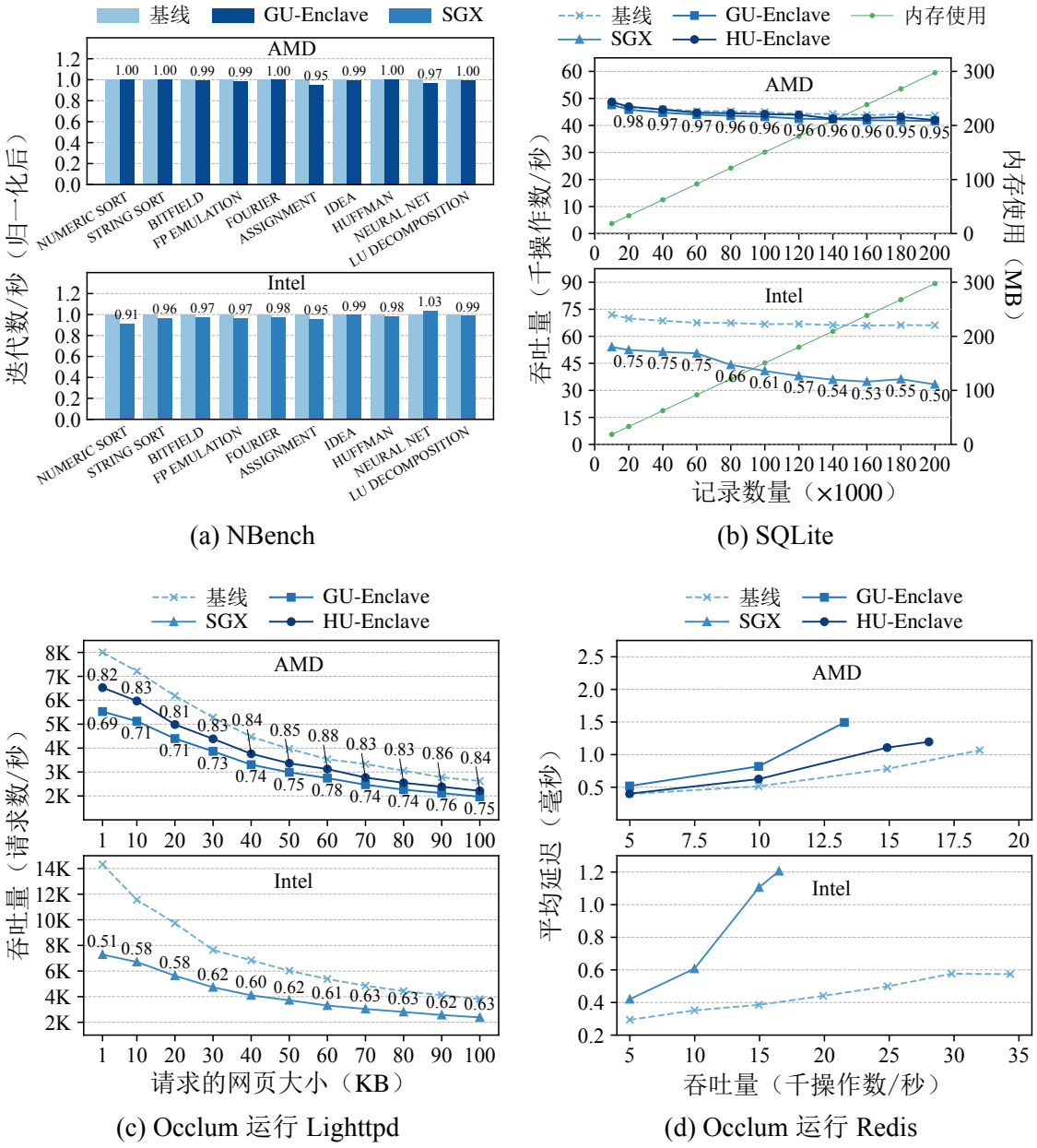
最后，本节将评估在 HyperEnclave 上运行几个真实世界的应用程序的性能，并与 Intel SGX 进行对比。实验选取了四个真实应用：一个算法基准测试套件 NBench[214]，一个轻量级网页服务器 Lighttpd[215]（v1.4.40），两个很受欢迎的数据库软件 SQLite[216]（v3.19.3）和 Redis[186]（v6.0.9），代表了典型的 CPU 密集型、I/O 密集型和内存密集型任务。对于 NBench 和 SQLite，它们都基于 SGX SDK（或 HyperEnclave 的飞地开发套件）进行移植。对于 Lighttpd 和 Redis，它们都运行在 Occlum 库操作系统[36]（v0.21）上，而无需额外的移植工作。实验将相同的应用代码在 SDK 仿真模式下编译以作为基线（不提供安全保证）。

NBench。 NBench 用于测量系统的 CPU、FPU 和内存的性能，不涉及 I/O 和系统调用。实验使用了一个基于 SGX SDK 移植的 NBench 版本 SGX-NBench[217]。如图 5.11(a) 所示，HyperEnclave 和 SGX 引入的开销分别约为 1% 和 3%，可见 TEE 对计算密集型任务的支持较好。

SQLite。该实验重点关注内存性能，因此数据均存储在内存中，并将客户端与数据库链接为同一个应用以避免 I/O 操作。实验使用 YCSB[218] 工作负载 A（50% 读取，50% 更新）进行测试，逐渐增大数据记录的数量，并测量执行 100,000 次数据库操作所需的时间。结果如图 5.11(b) 所示，在 SGX 上，当内存使用量较低时，吞吐量约为基准的 75%。当内存使用量超过 EPC 大小（约为 90 MB）时，由于页面交换，性能下降到 50%。在 HyperEnclave 上，无论是 GU-Enclave 还是 HU-Enclave，它们的性能几乎与基线相同（小于 5% 的开销）。本文推测这是因为 AMD SME（没有完整性保护）的内存加密性能比 SGX 更快（见第 5.5.6 节的评估）。

Lighttpd。该实验利用 Occlum 库操作系统，将 Lighttpd 服务器运行在 SGX 和 HyperEnclave（包括 GU-Enclave 和 HU-Enclave）上。对于客户端，实验使用 Apache HTTP 基准测试工具[219]，通过本地回环运行 100 个并发客户端，获取不同大小的网页并评估吞吐量。如图 5.11(c) 所示，HU-Enclave 表现出与预期一样的最佳性能（基线的 81%～88%），因为本项实验的开销主要来自频繁的模式切换；GU-Enclave 达到了基线 69%～78% 的性能；而 SGX 仅达到基线 51%～63% 的性能。

Redis。Redis 是一个兼具内存密集和 I/O 密集的常见应用，可以评估综合场景下的性能。实验利用 Occlum 库操作系统在 SGX 和 HyperEnclave（包括 GU-Enclave 和 HU-Enclave）上运行 Redis 数据库服务器。与 SQLite 类似，实验将 Redis 配置为内存数据库，并使用 YCSB 工作负载 A。实验首先加载了 50,000 条记录（总共



(a) NBench　(b) SQLite

(c) Occlum 运行 Lighttpd　(d) Occlum 运行 Redis

图 5.11　HyperEnclave 和 Intel SGX 上几个真实应用的性能

50 MB 的数据），然后客户端与服务器建立 20 个连接，不断增加请求的频率，并测量不同吞吐量下的平均延迟。如图 5.11(d) 所示，HU-Enclave 可以达到基线最大吞吐量的 89%，而 GU-Enclave 和 SGX 分别为基线的 72% 和 48%。

5.6　讨论

本节简要讨论 HyperEnclave 的局限性与未来工作。

内存加密。HyperEnclave 需要依赖内存加密来抵御物理攻击（第 5.3.3 节）。如果平台不支持硬件内存加密，HyperEnclave 可以考虑使用软件方法[220]，但也会带

来更大的性能开销。

移植 HyperEnclave 到其他平台。HyperEnclave 依赖硬件虚拟化扩展（主要是二级地址转换机制）提供隔离保证，以及 TPM 建立可信根和生成随机数。这两项需求在大多数服务器场景中可用。目前，ARM 服务器一般都采用 ARMv8 架构，已支持硬件虚拟化[221]，而 RISC-V 在 2021 年的最新规范中也引入了关于虚拟化的 H 扩展[222]。ARM 和 RISC-V 平台上也有对 TPM 的支持[223]。

然而，由于不同架构的虚拟化机制不同，将 HyperEnclave 移植到其他平台需要大量的工程工作。以 ARMv8 架构为例，HyperEnclave 将使用 EL2 和 EL1 特权级来分别运行 RustMonitor 和主操作系统，而 EL0（或 EL1 以支持 P-Enclave）将会作为安全模式来运行飞地。不过由于 ARM 不存在客户机模式（如 x86 下的非根模式）等类似的概念，因此 GU-Enclave 和 HU-Enclave 将不作区分，都运行在 EL0，这就无法发挥 HU-Enclave 更快模式切换的优势（第 5.3.5 节）。此外，飞地开发套件也要做相应的移植，因为其中包含大量平台相关的汇编代码用于飞地模式切换。

基于进程的 TEE 的局限性。HyperEnclave 采用类似 Intel SGX 的进程级 TEE 模型，虽然具有 TCB 小、生态好等优势，但也有其固有的局限性。例如，基于进程的 TEE 需要依赖库操作系统（例如 Occlum）才能兼容已有应用，在进行 I/O 操作时需要多次模式切换与数据拷贝而带来显著开销等。而基于虚拟化的 TEE 通过在飞地中运行一个完整的通用操作系统，虽然能很好地解决这些问题，但也会使 TCB 变得非常庞大。未来，HyperEnclave 可以考虑同时支持这两种 TEE 模型（可通过扩展 P-Enclave 来支持虚拟机级飞地），并提供统一的编程接口，允许用户灵活进行模式选择，在性能与安全方面作出更好的权衡。

5.7　本章小结

为防止管理程序对微虚拟机内敏感数据的随意访问，本章提出了 HyperEnclave，一个为微虚拟机场景设计的可信执行环境。HyperEnclave 具有很少的硬件依赖，仅依赖如今广泛可用的硬件虚拟化与可信平台模块，利用一个轻量级的微虚拟机管理程序来实现可信执行环境的基本功能，从而方便移植到多种平台。HyperEnclave 为微虚拟机提供了灵活的隔离模式，来满足不同场景下应用对安全与性能的需求。HyperEnclave 通过基于进程的、与 SGX 兼容的编程模型与接口，支持在非 Intel 平台上无修改运行已有的 SGX 应用。实验结果表明，HyperEnclave 具有较小的开销，性能与安全性与现有的硬件方法相当。

第 6 章　总结与展望

6.1　本文工作总结

本文引入了微虚拟机的概念，由于其具有轻量专用、功能解耦、灵活管控的特点，在性能、灵活性、安全性等方面具有诸多优势。本文对微虚拟机的研究主要涵盖了两部分：上层的只支持单应用、小而专的微虚拟机操作系统，以及底层的微虚拟机管理程序。为解决微虚拟机在任务调度、针对应用的专用化、敏感数据保护等方面的不足，本文分别基于用户态微虚拟机优化通用操作系统的调度性能，基于特权态微虚拟机进行针对应用的专用化操作系统的设计，基于微虚拟机管理程序优化微虚拟机的安全防护性能，主要贡献总结如下：

（1）基于用户态微虚拟机的任务调度性能优化。

为了方便又高效地支持应用程序自定义的任务调度策略，需要将内核的线程管理、调度、网络等功能都卸载到用户态。为此，本文设计了 Skyloft 库操作系统。Skyloft 利用 Intel 最新硬件上的用户态中断技术，在用户态实现了低开销的处理器间中断与时钟中断处理，从而为用户态调度器提供了微秒级的抢占支持，降低了多样化负载下的任务处理延迟。同时，Skyloft 利用一个内核模块，允许用户态调度器切换属于不同应用的线程，从而实现多应用的统一调度，提高了 CPU 利用率。Skyloft 也集成了基于内核旁路的用户态网络技术，使得网络栈能与调度器相互配合。实验结果表明，相比现有的可由应用自定义调度策略的系统，Skyloft 能将延迟敏感应用的最大吞吐量提高 19%。

（2）特权态微虚拟机的组件化操作系统设计与性能优化。

高性能的微虚拟机操作系统需要能够方便地针对应用程序的需求进行专用化定制。为此，本文提出了一种组件化的操作系统设计方法 ArceOS，通过将组件按照与操作系统设计理念的相关性划分，降低了耦合性并提升了可重用性。ArceOS 还为组件提供了灵活的定制功能，可以通过应用指定的配置文件，构建出 Unikernel、宏内核、虚拟机管理程序等多种形态的操作系统。此外，ArceOS 还基于 Rust 语言的特性设计了 API 快速路径，专门针对微虚拟机场景进行了性能优化，同时也具有足够的兼容性。实验结果表明，ArceOS 的组件化设计能够方便实现操作系统功能的定制。ArceOS 的 API 快速路径设计在小文件读写上比使用 POSIX 接口快了 50% 。ArceOS 同时兼容已有 C 应用程序，与 Linux 相比，使 Redis 的延迟降低了 33%。ArceOS 组件化操作系统目前已开源，并被多家单位用于科研或二次开发。

（3）面向可信执行环境的微虚拟机灵活隔离与性能优化。

为防止管理程序对微虚拟机内敏感数据的随意访问，需要利用可信执行环境技术保护微虚拟机。然而，现有的可信执行环境存在通用性不足、性能开销大、隔离模式固定等问题。为此，本文提出了一种可信执行环境的设计方法 HyperEnclave，仅依赖如今广泛可用的硬件虚拟化扩展与可信平台模块，而无需特定的安全硬件，方便在多种平台上实现，并以较小的开销解决了在没有安全硬件支持下的一些安全问题。HyperEnclave 利用轻量级的微虚拟机管理程序，提供灵活的隔离模式，能够同时支持用户态与特权态的微虚拟机，来满足应用在不同场景下对性能与安全的需求。HyperEnclave 为不同形态的微虚拟机提供了统一的、与现有 TEE 技术兼容的编程模型与接口，能够支持已有应用的无修改运行。实验结果表明，HyperEnclave 引入的开销很小（如 SQLite 的开销小于 5%），与现有硬件 TEE 方法的性能与安全性相当。HyperEnclave 灵活的隔离模式也在一些场景下取得了更好的性能。HyperEnclave 目前已开源，并在蚂蚁集团落地商用。

6.2　未来工作展望

在微虚拟机的性能、安全、开发成本、兼容性上，仍有不少值得深入研究的方向。未来工作可从以下几方面展开：

（1）更多操作系统功能到用户态微虚拟机的卸载。

本文在第 3 章探索了操作系统的调度功能到用户态微虚拟机的卸载。对于其他性能关键、实现方式多样的功能，也可以通过到用户态微虚拟机的卸载，方便应用程序进行灵活定制来优化性能。例如第 3.6 节提到的对用户态外设中断的支持，可避免以轮询的方式实现用户态设备驱动。最近也有对应用感知的内核同步机制的研究[82]，但其基于 eBPF 的方法在灵活性与通用性上有所不足。此外，软硬件协同也是提高这种卸载性能的重要方式。未来可能需要引入一些新的硬件特性，允许在用户态执行更多特权操作，如页表管理、异常处理等，并提供足够的安全保证。

（2）特权态微虚拟机与通用操作系统的协同工作。

为让特权态微虚拟机操作系统支持更多应用程序，难免会涉及众多系统调用的实现，这带来了巨大的开发工作量。可以考虑一种特权态微虚拟机与一个通用操作系统协同的新架构，在特权态微虚拟机中实现性能关键、高度定制的功能，以提供性能保证；而性能非关键的、复杂繁琐的功能仍交给同级的通用操作系统去处理，以提供兼容性保证。例如目前已有一些架构，通过让一个实时操作系统与 Linux 协同来同时满足高实时性与高兼容性[224]。与用户态微虚拟机不同，在这种架构下，该通用操作系统处于与微虚拟机相同或更低的特权级，并不管控微虚拟

机的运行，因此需要在其中维护一个代理进程，用于代替微虚拟机来调用通用操作系统的功能。此外，微虚拟机管理程序也需要提供一套微虚拟机与通用操作系统间的高效通信机制。

（3）微虚拟机安全加固。

微虚拟机将应用程序与操作系统运行在同一地址空间同一特权级，虽然提高了性能，但损失了一定的安全性。为此，可以考虑在微虚拟机内引入轻量级的隔离机制（如第 2.3.2 节中介绍的）。由此引发的另一个问题是对隔离粒度的选择，细粒度的隔离能提供更强的安全性，但也会带来更大的性能损失。FlexOS[115] 为微虚拟机提供了多种隔离方式以及隔离粒度的配置，允许用户在安全与性能之间进行更多的权衡。本文第 4 章提出的组件化设计方法，也为实现微虚拟机内的细粒度隔离提供了基础。在第 4.6 节也讨论了基于内核组件形成微内核（利用页表隔离组件）的一些思路与挑战。不过，在组件间增加隔离会破坏微虚拟机单地址空间的特点，无法利用第 4 章的 API 快速路径优化，如何在细粒度隔离的同时还能保证性能也是一个重要挑战。

（4）基于组件化设计的系统软件统一构建。

本文第 4 章提出了组件化的操作系统设计方法，一个自然的想法是，能够利用这种方法统一从一系列组件中构建出本文第 3 章的库操作系统以及第 5 章的微虚拟机管理程序。这不仅可以极大地减少开发代价，还扩充了组件库，方便构建新的系统软件。由于目前这两个系统未采用组件化的设计，需要先从组件与系统的相关性考虑，将它们进行功能解耦来形成组件，并充分考虑组件间的依赖关系与可重用性。此外，基于组件化的重新构建是否会影响系统的性能，也值得进一步探究。

参考文献

[1] Belay A, Prekas G, Klimovic A, et al. IX: A protected dataplane operating system for high throughput and low latency[C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Berkeley, CA, USA: USENIX Association, 2014: 49-65.

[2] Synopsys, Inc. The Linux kernel open source project on open hub: Languages page[EB/OL]. 2024[2024-03-01]. https://www.openhub.net/p/linux/analyses/latest/languages_summary.

[3] SecurityScorecard. Linux: Vulnerability Statistics[EB/OL]. 2024[2024-03-01]. https://www.cvedetails.com/vendor/33/Linux.html.

[4] Lipp M, Schwarz M, Gruss D, et al. Meltdown: Reading kernel memory from user space[C]//27th USENIX Security Symposium (USENIX Security 18). Berkeley, CA, USA: USENIX Association, 2018: 973-990.

[5] The kernel development community. Page Table Isolation (PTI)[R/OL]. 2023[2024-02-18]. https://www.kernel.org/doc/html/next/x86/pti.html.

[6] Corbet J. KAISER: hiding the kernel from user space[R/OL]. (2017-11-15)[2024-03-18]. https://lwn.net/Articles/738975/.

[7] Soares L, Stumm M. FlexSC: Flexible system call scheduling with Exception-Less system calls[C]//9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). Berkeley, CA, USA: USENIX Association, 2010.

[8] Han S, Marshall S, Chun B G, et al. MegaPipe: A new programming interface for scalable network I/O[C]//10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). Berkeley, CA, USA: USENIX Association, 2012: 135-148.

[9] Axboe J. Efficient IO with io_uring[R/OL]. (2019-10-05). https://kernel.dk/io_uring.pdf.

[10] Narayanan V, Balasubramanian A, Jacobsen C, et al. LXDs: Towards isolation of kernel subsystems[C]//2019 USENIX Annual Technical Conference (USENIX ATC 19). Berkeley, CA, USA: USENIX Association, 2019: 269-284.

[11] Narayanan V, Huang Y, Tan G, et al. Lightweight kernel isolation with virtualization and VM functions[C]//VEE '20: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2020: 157–171.

[12] Huang Y, Narayanan V, Detweiler D, et al. KSplit: Automating device driver isolation[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Berkeley, CA, USA: USENIX Association, 2022: 613-631.

[13] The Linux Foundation. Real-Time Linux[EB/OL]. (2024-01-19)[2024-03-01]. https://wiki.linuxfoundation.org/realtime/start.

[14] Baumann A, Barham P, Dagand P E, et al. The multikernel: a new OS architecture for scalable multicore systems[C]//SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2009: 29–44.

[15] Liedtke J. On micro-kernel construction[C]//SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1995: 237–250.

[16] Klein G, Elphinstone K, Heiser G, et al. seL4: formal verification of an OS kernel[C]//SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2009: 207–220.

[17] Google. Zircon[EB/OL]. (2024-02-02)[2024-03-01]. https://fuchsia.dev/fuchsia-src/concepts/kernel.

[18] Hunt G C, Larus J R. Singularity: rethinking the software stack[J]. SIGOPS Oper. Syst. Rev., 2007, 41(2): 37–49.

[19] Madhavapeddy A, Mortier R, Rotsos C, et al. Unikernels: Library operating systems for the cloud[C]//ASPLOS '13: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2013: 461–472.

[20] Cutler C, Kaashoek M F, Morris R T. The benefits and costs of writing a POSIX kernel in a high-level language[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Berkeley, CA, USA: USENIX Association, 2018: 89-105.

[21] Redox Developers. Redox - Your Next(Gen) OS[EB/OL]. 2023[2024-02-07]. https://redox-os.org.

[22] Boos K, Liyanage N, Ijaz R, et al. Theseus: an experiment in operating system structure and state management[C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). Berkeley, CA, USA: USENIX Association, 2020: 1-19.

[23] Narayanan V, Huang T, Detweiler D, et al. RedLeaf: Isolation and communication in a safe operating system[C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). Berkeley, CA, USA: USENIX Association, 2020: 21-39.

[24] Engler D R, Kaashoek M F, O'Toole J. Exokernel: An operating system architecture for application-level resource management[C]//SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1995: 251–266.

[25] Belay A, Bittau A, Mashtizadeh A, et al. Dune: Safe user-level access to privileged CPU features[C]//10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). Berkeley, CA, USA: USENIX Association, 2012: 335-348.

[26] Martins J, Ahmed M, Raiciu C, et al. ClickOS and the art of network function virtualization [C]//11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Berkeley, CA, USA: USENIX Association, 2014: 459-473.

[27] Kivity A, Laor D, Costa G, et al. OSv—Optimizing the operating system for virtual machines [C]//2014 USENIX Annual Technical Conference (USENIX ATC 14). Berkeley, CA, USA: USENIX Association, 2014: 61-72.

[28] Kuenzer S, Bădoiu V A, Lefeuvre H, et al. Unikraft: Fast, specialized unikernels the easy way [C]//EuroSys '21: Proceedings of the Sixteenth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2021: 376–394.

[29] Manco F, Lupu C, Schmidt F, et al. My VM is lighter (and safer) than your container[C]//SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2017: 218–233.

[30] Jeong E, Wood S, Jamshed M, et al. mTCP: a highly scalable user-level TCP stack for multicore systems[C]//11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Berkeley, CA, USA: USENIX Association, 2014: 489-502.

[31] DPDK Project. DPDK: Data Plane Development Kit[EB/OL]. 2023[2024-02-15]. https://www.dpdk.org.

[32] Zhang I, Raybuck A, Patel P, et al. The Demikernel datapath OS architecture for microsecond-scale datacenter systems[C]//SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2021: 195–211.

[33] Baumann A, Peinado M, Hunt G. Shielding applications from an untrusted cloud with Haven [C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Berkeley, CA, USA: USENIX Association, 2014: 267-283.

[34] Arnautov S, Trach B, Gregor F, et al. SCONE: Secure linux containers with intel SGX[C]//12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Berkeley, CA, USA: USENIX Association, 2016: 689-703.

[35] Tsai C, Porter D E, Vij M. Graphene-SGX: A practical library OS for unmodified applications on SGX[C]//2017 USENIX Annual Technical Conference (USENIX ATC 17). Berkeley, CA, USA: USENIX Association, 2017: 645-658.

[36] Shen Y, Tian H, Chen Y, et al. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX[C]//ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2020: 955–970.

[37] TrustedFirmware.org. OP-TEE[EB/OL]. 2024[2024-03-18]. https://optee.readthedocs.io/en/latest/general/about.html.

[38] Marinos I, Watson R N, Handley M. Network stack specialization for performance[C]//SIGCOMM '14: Proceedings of the 2014 ACM Conference on SIGCOMM. New York, NY, USA: Association for Computing Machinery, 2014: 175–186.

[39] Peter S, Li J, Zhang I, et al. Arrakis: The operating system is the control plane[C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Berkeley, CA, USA: USENIX Association, 2014: 1-16.

[40] Yang Z, Harris J R, Walker B, et al. SPDK: A development kit to build high performance storage applications[C]//2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). 2017: 154-161.

[41] Kwon Y, Fingler H, Hunt T, et al. Strata: A cross media file system[C]//SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2017: 460–477.

[42] Kadekodi R, Lee S K, Kashyap S, et al. SplitFS: reducing software overhead in file systems for persistent memory[C]//SOSP '19: Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2019: 494–508.

[43] Ren Y, Min C, Kannan S. CrossFS: A cross-layered Direct-Access file system[C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). Berkeley, CA, USA: USENIX Association, 2020: 137-154.

[44] Prekas G, Kogias M, Bugnion E. ZygOS: Achieving low tail latency for microsecond-scale networked tasks[C]//SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2017: 325–341.

[45] Kaffes K, Chong T, Humphries J T, et al. Shinjuku: Preemptive scheduling for μsecond-scale tail latency[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Berkeley, CA, USA: USENIX Association, 2019: 345-360.

[46] Heo T. sched: Implement BPF extensible scheduler class[R/OL]. (2023-11-10)[2024-02-15]. https://lwn.net/Articles/951156/.

[47] Ford B, Back G, Benson G, et al. The Flux OSKit: a substrate for kernel and language research[C]//SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1997: 38–51.

[48] Schatzberg D, Cadden J, Dong H, et al. EbbRT: A framework for building Per-Application library operating systems[C]//12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Berkeley, CA, USA: USENIX Association, 2016: 671-688.

[49] Olivier P, Chiba D, Lankes S, et al. A binary-compatible unikernel[C]//VEE 2019: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2019: 59–73.

[50] Kuo H, Williams D, Koller R, et al. A Linux in unikernel clothing[C]//EuroSys '20: Proceedings of the Fifteenth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2020.

[51] McKeen F, Alexandrovich I, Berenzon A, et al. Innovative instructions and software model for isolated execution[C]//HASP '13: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. New York, NY, USA: Association for Computing Machinery, 2013.

[52] Kaplan D. AMD x86 memory encryption technologies[C]//25th USENIX Security Symposium (USENIX Security 16). Berkeley, CA, USA: USENIX Association, 2016.

[53] Intel. Intel® Trust Domain Extensions (Intel® TDX)[R/OL]. 2023. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html.

[54] Alves T, Felton D. TrustZone: Integrated hardware and software security[J]. Technology In-Depth, 2004, 3(4): 18-24.

[55] McCune J M, Li Y, Qu N, et al. TrustVisor: Efficient TCB reduction and attestation[C]//SP '10: Proceedings of the 2010 IEEE Symposium on Security and Privacy. USA: IEEE Computer Society, 2010: 143–158.

[56] Hofmann O S, Kim S, Dunn A M, et al. InkTag: secure applications on an untrusted operating system[C]//ASPLOS '13: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2013: 265–278.

[57] Chen X, Garfinkel T, Lewis E C, et al. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems[C]//ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2008: 2–13.

[58] Intel. User interrupts[R/OL]//Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. 2023: 7.1-7.8. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[59] Intel. Introduction to virtual machine extensions[R/OL]//Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3. 2023: 24.1-24.4. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[60] Trusted Computing Group. TPM 2.0 Library[R/OL]. 2019. https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[61] Amazon Web Services, Inc. FreeRTOS[EB/OL]. 2024[2024-03-17]. https://www.freertos.org.

[62] Porter D E, Boyd-Wickizer S, Howell J, et al. Rethinking the library OS from the top down[C]//ASPLOS XVI: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2011: 291–304.

[63] Purdila O, Grijincu L A, Tapus N. LKL: The Linux kernel library[C]//9th RoEduNet IEEE International Conference. 2010: 328-333.

[64] Google. gVisor: The container security platform[EB/OL]. 2024[2024-03-17]. https://gvisor.dev.

[65] Tsai C, Arora K S, Bandi N, et al. Cooperation and security isolation of library OSes for multi-process applications[C]//EuroSys '14: Proceedings of the Ninth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2014.

[66] Wahbe R, Lucco S, Anderson T E, et al. Efficient software-based fault isolation[C]//SOSP '93: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1993: 203–216.

[67] Inria. OCaml[EB/OL]. 2024[2024-03-03]. https://ocaml.org.

[68] Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization[C]//SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2003: 164–177.

[69] Kantee A. Flexible operating system internals: The design and implementation of the anykernel and rump kernels[D]. Finland: Aalto University, 2012.

[70] Maeda T, Yonezawa A. Kernel Mode Linux: Toward an operating system protected by a type theory[C]//Saraswat V A. Advances in Computing Science – ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003: 3-17.

[71] Raza A, Unger T, Boyd M, et al. Unikernel Linux (UKL)[C]//EuroSys '23: Proceedings of the Eighteenth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2023: 590–605.

[72] Corporation M. Introduction to receive side scaling[EB/OL]. (2023-12-16)[2024-02-15]. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[73] Intel. PCI-SIG SR-IOV Primer: An introduction to SR-IOV technology[M]. 2011.

[74] Iyer R, Unal M, Kogias M, et al. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling[C]//SOSP '23: Proceedings of the 29th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2023: 466–481.

[75] Ousterhout A, Fried J, Behrens J, et al. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Berkeley, CA, USA: USENIX Association, 2019: 361-378.

[76] Fried J, Ruan Z, Ousterhout A, et al. Caladan: Mitigating interference at microsecond timescales [C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). Berkeley, CA, USA: USENIX Association, 2020: 281-297.

[77] Kuznetsov D, Morrison A. Privbox: Faster system calls through sandboxed privileged execution [C]//2022 USENIX Annual Technical Conference (USENIX ATC 22). Berkeley, CA, USA: USENIX Association, 2022.

[78] Zhou Z, Bi Y, Wan J, et al. Userspace bypass: Accelerating syscall-intensive applications [C]//17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). Berkeley, CA, USA: USENIX Association, 2023: 33-49.

[79] Fleming M. A thorough introduction to eBPF[R/OL]. (2017-12-02)[2024-02-18]. https://lwn.net/Articles/740157/.

[80] Høiland-Jørgensen T, Brouer J D, Borkmann D, et al. The eXpress data path: fast programmable packet processing in the operating system kernel[C]//CoNEXT '18: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies. New York, NY, USA: Association for Computing Machinery, 2018: 54–66.

[81] Zhong Y, Li H, Wu Y J, et al. XRP: In-Kernel storage functions with eBPF[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Berkeley, CA, USA: USENIX Association, 2022: 375-393.

[82] Park S, Zhou D, Qian Y, et al. Application-Informed kernel synchronization primitives[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Berkeley, CA, USA: USENIX Association, 2022: 667-682.

[83] Humphries J T, Natu N, Chaugule A, et al. ghOSt: Fast & flexible user-space delegation of Linux scheduling[C]//SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2021: 588–604.

[84] Kaffes K, Humphries J T, Mazières D, et al. Syrup: User-defined scheduling across the stack [C]//SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2021: 605–620.

[85] Tsirkin M S, Huck C. Virtual I/O Device (VIRTIO) Version 1.2[R/OL]. OASIS Open, 2022 [2024-03-17]. https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html.

[86] Rosen R. Resource management: Linux kernel Namespaces and cgroups[R/OL]. Haifux, 2013 [2024-03-17]. http://www.haifux.org/lectures/299/netLec7.pdf.

[87] Docker, Inc. Docker: Accelerated container application development[EB/OL]. 2024[2024-03-17]. https://www.docker.com.

[88] Linux Containers. Linux Containers[EB/OL]. 2024[2024-03-17]. https://linuxcontainers.org/.

[89] Boyd-Wickizer S, Chen H, Chen R, et al. Corey: An operating system for many cores[C]//8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08). Berkeley, CA, USA: USENIX Association, 2008.

[90] Clements A T, Kaashoek M F, Zeldovich N, et al. The scalable commutativity rule: designing scalable software for multicore processors[C]//SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2013: 1–17.

[91] Bhat S S, Eqbal R, Clements A T, et al. Scaling a file system to many cores using an operation log[C]//SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2017: 69–86.

[92] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity principles, implementations, and applications[J]. ACM Trans. Inf. Syst. Secur., 2009, 13(1).

[93] The kernel development community. Kernel Address Sanitizer (KASAN)[R/OL]. 2023[2024-03-03]. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[94] Bershad B N, Savage S, Pardyak P, et al. Extensibility safety and performance in the SPIN operating system[C]//SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1995: 267–283.

[95] Cardelli L, Donahue J, Jordan M, et al. The Modula–3 type system[C]//POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: Association for Computing Machinery, 1989: 202–212.

[96] Google. The Go Programming Language[EB/OL]. 2024[2024-02-17]. https://go.dev.

[97] Klabnik S, Nichols C. The Rust Programming Language[EB/OL]. (2023-02-09)[2024-02-29]. https://doc.rust-lang.org/book/.

[98] Levy A, Campbell B, Ghena B, et al. Multiprogramming a 64kb computer safely and efficiently [C]//SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2017: 234–251.

[99] Corbet J. Next steps for Rust in the kernel[R/OL]. (2022-09-19)[2024-03-18]. https://lwn.net/Articles/908347/.

[100] Litton J, Vahldiek-Oberwagner A, Elnikety E, et al. Light-Weight contexts: An OS abstraction for safety and performance[C]//12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Berkeley, CA, USA: USENIX Association, 2016: 49-64.

[101] Jing Y, Huang P. Operating system support for safe and efficient auxiliary execution[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Berkeley, CA, USA: USENIX Association, 2022: 633-648.

[102] Intel. VMX support for address translation[R/OL]//Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3. 2023: 29.1-29.26. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[103] Nikolaev R, Back G. VirtuOS: an operating system with kernel virtualization[C]//SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2013: 116–132.

[104] Intel. VM functions[R/OL]//Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3. 2023: 26.17-26.18. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[105] Mi Z, Li D, Yang Z, et al. Skybridge: Fast and secure inter-process communication for microkernels[C]//EuroSys '19: Proceedings of the Fourteenth EuroSys Conference 2019. New York, NY, USA: Association for Computing Machinery, 2019.

[106] Hedayati M, Gravani S, Johnson E, et al. Hodor: Intra-Process isolation for High-Throughput data plane libraries[C]//2019 USENIX Annual Technical Conference (USENIX ATC 19). Berkeley, CA, USA: USENIX Association, 2019: 489-504.

[107] Bhargava R, Serebrin B, Spadini F, et al. Accelerating two-dimensional page walks for virtualized systems[C]//ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2008: 26–35.

[108] Intel. Protection keys[R/OL]//Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. 2023: 4.36. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[109] Arm Limited or its affiliates. Domains, short-descriptor format only[R/OL]//ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. 2011: B3.1353. https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA-/Memory-access-control/Domains--Short-descriptor-format-only.

[110] Arm Limited or its affiliates. Armv8.5-A memory tagging extension white paper[R/OL]. 2019 [2024-03-03]. https://developer.arm.com/documentation/102925/latest/.

[111] Waterman A, Asanović K, Hauser J. Physical Memory Protection[R/OL]//The RISC-V Instruction Set Manual, Volume II: Privileged Architecture. 20211203 ed. RISC-V International, 2021: 56-61. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.

[112] Watson R N, Woodruff J, Neumann P G, et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization[C]//2015 IEEE Symposium on Security and Privacy. 2015: 20-37.

[113] Sung M, Olivier P, Lankes S, et al. Intra-unikernel isolation with intel memory protection keys [C]//VEE '20: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2020: 143–156.

[114] Li G, Du D, Xia Y. Iso-UniK: lightweight multi-process unikernel through memory protection keys[J]. Cybersecurity, 2020, 3(1): 11.

[115] Lefeuvre H, Bădoiu V A, Jung A, et al. FlexOS: Towards flexible OS isolation[C]//ASPLOS '22: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2022: 467–482.

[116] Hunt G D H, Pai R, Le M V, et al. Confidential computing for OpenPOWER[C]//EuroSys '21: Proceedings of the Sixteenth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2021: 294–310.

[117] Arm Limited or its affiliates. Arm Confidential Compute Architecture[R/OL]. 2024[2024-03-03]. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[118] Costan V, Devadas S. Intel SGX explained[J/OL]. IACR Cryptology ePrint Archive, 2016, 2016: 86. http://eprint.iacr.org/2016/086.

[119] Bahmani R, Brasser F, Dessouky G, et al. CURE: A security architecture with CUstomizable and Resilient Enclaves[C]//30th USENIX Security Symposium (USENIX Security 21). Berkeley, CA, USA: USENIX Association, 2021: 1073-1090.

[120] Feng E, Lu X, Du D, et al. Scalable memory protection in the PENGLAI enclave[C]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). Carlsbad, CA, USA: USENIX Association, 2021: 275-294.

[121] Liu Y, Zhou T, Chen K, et al. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation[C]//CCS '15: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2015: 1607–1619.

[122] Van't Hof A, Nieh J. BlackBox: A container security monitor for protecting containers on untrusted operating systems[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Berkeley, CA, USA: USENIX Association, 2022: 683-700.

[123] Amazon Web Services, Inc. AWS Nitro Enclaves User Guide[R/OL]. 2024[2024-03-03]. https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html.

[124] Microsoft. Configure credential guard[R/OL]. 2024[2024-03-03]. https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/configure.

[125] Microsoft. Guarded fabric and shielded VMs overview[R/OL]. 2024[2024-03-03]. https://learn.microsoft.com/en-us/windows-server/security/guarded-fabric-shielded-vm/guarded-fabric-and-shielded-vms.

[126] Lutas. A, Ticle. D, Cret. O. Hypervisor based memory introspection: Challenges, problems and limitations[C]//Proceedings of the 3rd International Conference on Information Systems Security and Privacy - ICISSP. SciTePress, 2017: 285-294.

[127] Xu Y, Cui W, Peinado M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems[C]//SP '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy. USA: IEEE Computer Society, 2015: 640–656.

[128] Wang W, Chen G, Pan X, et al. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX[C]//CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2017: 2421–2434.

[129] Ferraiuolo A, Baumann A, Hawblitzel C, et al. Komodo: Using verification to disentangle secure-enclave hardware from software[C]//SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2017: 287–305.

[130] Lee D, Kohlbrenner D, Shinde S, et al. Keystone: an open framework for architecting trusted execution environments[C]//EuroSys '20: Proceedings of the Fifteenth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2020.

[131] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of High-Coverage tests for complex systems programs[C]//8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08). Berkeley, CA, USA: USENIX Association, 2008.

[132] Zalewski M. American fuzzy lop[EB/OL]. 2013[2024-03-18]. https://lcamtuf.coredump.cx/afl/.

[133] Vyukov D. Coverage-guided kernel fuzzing with syzkaller[R/OL]. (2016-03-02)[2024-03-18]. https://lwn.net/Articles/677764/.

[134] Gu R, Shao Z, Chen H, et al. CertiKOS: An extensible architecture for building certified concurrent OS kernels[C]//12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Berkeley, CA, USA: USENIX Association, 2016: 653-669.

[135] Seemakhupt K, Stephens B E, Khan S, et al. A cloud-scale characterization of remote procedure calls[C]//SOSP '23: Proceedings of the 29th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2023: 498–514.

[136] Wierman A, Zwart B. Is tail-optimal scheduling possible?[J]. Operations Research, 2012, 60 (5): 1249-1257.

[137] Demoulin H M, Fried J, Pedisich I, et al. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with Perséphone[C]//SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2021: 621–637.

[138] Ma T, Chen S, Wu Y, et al. Efficient scheduler live update for Linux kernel with modularization [C]//ASPLOS 2023: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. New York, NY, USA: Association for Computing Machinery, 2023: 194–207.

[139] Marsh B D, Scott M L, LeBlanc T J, et al. First-class user-level threads[C]//SOSP '91: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1991: 110–121.

[140] Karsten M, Barghi S. User-level threading: Have your cake and eat it too[J]. Proc. ACM Meas. Anal. Comput. Syst., 2020, 4(1).

[141] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload analysis of a large-scale key-value store [C]//SIGMETRICS '12: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems. New York, NY, USA: Association for Computing Machinery, 2012: 53–64.

[142] Boucher S, Kalia A, Andersen D G, et al. Lightweight preemptible functions[C]//2020 USENIX Annual Technical Conference (USENIX ATC 20). Berkeley, CA, USA: USENIX Association, 2020: 465-477.

[143] Shiina S, Iwasaki S, Taura K, et al. Lightweight preemptive user-level threads[C]//PPoPP '21: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: Association for Computing Machinery, 2021: 374–388.

[144] Mason C. schbench[EB/OL]. 2023[2024-03-20]. https://kernel.googlesource.com/pub/scm/linux/kernel/git/mason/schbench/+/refs/tags/v1.0.

[145] Dormando. memcached - a distributed memory object caching system[EB/OL]. 2018[2024-02-15]. https://memcached.org.

[146] McClure S, Ousterhout A, Shenker S, et al. Efficient scheduling policies for Microsecond-Scale tasks[C]//19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). Berkeley, CA, USA: USENIX Association, 2022: 1-18.

[147] The kernel development community. CFS Scheduler[R/OL]. 2024[2024-02-15]. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[148] Stoica I, Abdel-Wahab H. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation[R]. USA: Old Dominion University, 1995.

[149] Corbet J. An EEVDF CPU scheduler for Linux[R/OL]. (2023-03-09)[2024-02-15]. https://lwn.net/Articles/925371/.

[150] Clements A. Proposal: Non-cooperative goroutine preemption[R/OL]. (2019-01-18)[2024-02-15]. https://github.com/golang/proposal/blob/master/design/24543-non-cooperative-preemption.md.

[151] Li Y, Lazarev N, Koufaty D, et al. LibPreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling[C]//2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA). 2024: 922-936.

[152] Miller S, Kumar A, Vakharia T, et al. Enoki: High velocity Linux kernel scheduler development [C]//EuroSys '24: Proceedings of the Nineteenth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2024: 962–980.

[153] Blumofe R D, Joerg C F, Kuszmaul B C, et al. Cilk: an efficient multithreaded runtime system [C]//PPOPP '95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: Association for Computing Machinery, 1995: 207–216.

[154] Von Behren R, Condit J, Zhou F, et al. Capriccio: scalable threads for internet services[C]//SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2003: 268–281.

[155] Wheeler K B, Murphy R C, Thain D. Qthreads: An API for programming with millions of lightweight threads[C]//2008 IEEE International Symposium on Parallel and Distributed Processing. 2008: 1-8.

[156] Nakashima J, Taura K. MassiveThreads: A thread library for high productivity languages[M]// Agha G, Igarashi A, Kobayashi N, et al. Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday. Berlin, Heidelberg: Springer, 2014: 222-238.

[157] Rust Team. async/.await Primer[R/OL]//Asynchronous Programming in Rust. 2023[2024-02-03]. https://rust-lang.github.io/async-book/01_getting_started/04_async_await_primer.html.

[158] Teabe B, Nitu V, Tchana A, et al. The lock holder and the lock waiter pre-emption problems: nip them in the bud using informed spinlocks (I-Spinlock)[C]//EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2017: 286–297.

[159] Corbet J. Deferred scheduling for user-space critical sections[R/OL]. (2023-10-27)[2024-03-18]. https://lwn.net/Articles/948870/.

[160] Iorgulescu C, Azimi R, Kwon Y, et al. PerfIso: Performance isolation for commercial Latency-Sensitive services[C]//2018 USENIX Annual Technical Conference (USENIX ATC 18). Berkeley, CA, USA: USENIX Association, 2018: 519-532.

[161] Qin H, Li Q, Speiser J, et al. Arachne: Core-Aware thread management[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Berkeley, CA, USA: USENIX Association, 2018: 145-160.

[162] Mehta S. x86 user interrupts support[R/OL]. (2021-09-13)[2024-02-15]. https://lwn.net/Articles/869140/.

[163] Blumofe R D, Leiserson C E. Scheduling multithreaded computations by work stealing[J]. J. ACM, 1999, 46(5): 720–748.

[164] Fleming M. A survey of scheduler benchmarks[R/OL]. (2017-06-14)[2024-03-20]. https://lwn.net/Articles/725238/.

[165] Meta Platforms, Inc. RocksDB - A persistent key-value store[EB/OL]. 2022[2024-02-15]. https://rocksdb.org.

[166] Lankes S, Pickartz S, Breitbart J. HermitCore: A unikernel for extreme scale computing[C]// ROSS '16: Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers. New York, NY, USA: Association for Computing Machinery, 2016.

[167] Kuenzer S, Ivanov A, Manco F, et al. Unikernels everywhere: The case for elastic CDNs [C]//VEE '17: Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2017: 15–29.

[168] Lankes S, Breitbart J, Pickartz S. Exploring Rust for unikernel development[C]//PLOS '19: Proceedings of the 10th Workshop on Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2019: 8–15.

[169] IEEE and The Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017[S/OL]. 2018th ed. IEEE, 2018. https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html.

[170] Palix N, Thomas G, Saha S, et al. Faults in linux: ten years later[C]//ASPLOS XVI: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2011: 305–318.

[171] Rust Team. The Rust community's crate registry[EB/OL]. 2024[2024-02-05]. https://crates.io.

[172] Masmano M, Ripoll I, Crespo A, et al. TLSF: a new dynamic memory allocator for real-time systems[C]//Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004. 2004: 79-88.

[173] Bonwick J. The slab allocator: An Object-Caching kernel[C]//USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference). Berkeley, CA, USA: USENIX Association, 1994.

[174] Peterson J L, Norman T A. Buddy systems[J]. Commun. ACM, 1977, 20(6): 421–431.

[175] Rust Team. Foreign Function Interface[R/OL]//The Rustonomicon. 2024[2024-02-03]. https://doc.rust-lang.org/nomicon/ffi.html.

[176] rust-unofficial. Awesome Rust: A curated list of Rust code and resources[EB/OL]. (2024-03-16)[2024-03-16]. https://github.com/rust-unofficial/awesome-rust.

[177] Zhang Y, Crowcroft J, Li D, et al. KylinX: A dynamic library operating system for simplified and efficient cloud virtualization[C]//2018 USENIX Annual Technical Conference (USENIX ATC 18). Berkeley, CA, USA: USENIX Association, 2018: 173-186.

[178] Wikipedia contributors. Dynamic dispatch — Wikipedia, the free encyclopedia[EB/OL]. 2024[2024-02-07]. https://en.wikipedia.org/w/index.php?title=Dynamic_dispatch&oldid=1196627019.

[179] smoltcp-rs. a smol tcp/ip stack[EB/OL]. 2024[2024-02-07]. https://github.com/smoltcp-rs/smoltcp.

[180] Free Software Foundation, Inc. lwIP - A Lightweight TCP/IP stack - Summary[EB/OL]. 2024[2024-02-07]. https://savannah.nongnu.org/projects/lwip/.

[181] Harabień R. rust-fatfs: A FAT filesystem library implemented in Rust[EB/OL]. 2023[2024-03-18]. https://github.com/rafalh/rust-fatfs.

[182] Felker R. musl libc[EB/OL]. 2024[2024-03-10]. https://musl.libc.org.

[183] Snell Q, Mikler A, Gustafson J. NetPIPE: A network protocol independent performance evaluator[EB/OL]. 2001[2024-03-20]. https://netpipe.cs.ksu.edu.

[184] Ellmann S. ixy.rs: Rust rewrite of the ixy network driver[EB/OL]. 2021[2024-03-13]. https://github.com/ixy-languages/ixy.rs.

[185] The kernel development community. VFIO - "Virtual Function I/O"[R/OL]. 2024[2024-03-13]. https://docs.kernel.org/driver-api/vfio.html.

[186] Redis Ltd. Redis[EB/OL]. 2024[2024-01-27]. https://redis.io.

[187] Wang H, Wang P, Ding Y, et al. Towards memory safe enclave programming with Rust-SGX [C]//CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2019: 2333–2350.

[188] Intel. Enclave operation[R/OL]//Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4. 2023: 36.1-36.16. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[189] Asylo Authors. Asylo: An open and flexible framework for enclave applications[EB/OL]. (2021-06-09)[2024-01-27]. https://asylo.dev.

[190] Microsoft. Azure confidential computing[EB/OL]. 2024[2024-01-27]. https://azure.microsoft.com/en-us/solutions/confidential-compute.

[191] Menetrey J, Pasin M, Felber P, et al. Twine: An embedded trusted runtime for webassembly [C]//2021 IEEE 37th International Conference on Data Engineering (ICDE). Los Alamitos, CA, USA: IEEE Computer Society, 2021: 205-216.

[192] Lind J, Priebe C, Muthukumaran D, et al. Glamdring: Automatic application partitioning for intel SGX[C]//2017 USENIX Annual Technical Conference (USENIX ATC 17). Berkeley, CA, USA: USENIX Association, 2017: 285-298.

[193] Tsai C, Son J, Jain B, et al. Civet: An efficient Java partitioning framework for hardware enclaves[C]//29th USENIX Security Symposium (USENIX Security 20). Berkeley, CA, USA: USENIX Association, 2020: 505-522.

[194] Intel. Intel SGX for Linux[EB/OL]. (2024-01-18)[2024-01-27]. https://github.com/intel/linux-sgx.

[195] Brasser F, Gens D, Jauernig P, et al. SANCTUARY: ARMing TrustZone with user-space enclaves[C]//26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society, 2019.

[196] Schwarz M, Weiser S, Gruss D. Practical enclave malware with Intel SGX[C]//Perdisci R, Maurice C, Giacinto G, et al. Detection of Intrusions and Malware, and Vulnerability Assessment. Cham: Springer International Publishing, 2019: 177-196.

[197] Kocher P, Horn J, Fogh A, et al. Spectre attacks: Exploiting speculative execution[C]//2019 IEEE Symposium on Security and Privacy (SP). 2019: 1-19.

[198] Azab A M, Ning P, Shah J, et al. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world[C]//CCS '14: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2014: 90–102.

[199] Yun M, Zhong L. Ginseng: Keeping secrets in registers when you distrust the operating system[C]//26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society, 2019.

[200] McKeen F, Alexandrovich I, Anati I, et al. Intel® Software Guard Extensions (Intel® SGX) support for dynamic memory management inside an enclave[C]//HASP '16: Proceedings of the Hardware and Architectural Support for Security and Privacy 2016. New York, NY, USA: Association for Computing Machinery, 2016.

[201] Intel. Intel® Architecture Memory Encryption Technologies Specification[R/OL]. 2021. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf.

[202] Shanahan M. Exception Handling in the Intel(R) SGX SDK[EB/OL]. (2020-06-09)[2024-01-27]. https://github.com/MWShan/linux-sgx/blob/master/docs/DesignDocs/IntelSGXExceptionHandling-Linux.md.

[203] Brasser F, Müller U, Dmitrienko A, et al. Software grand exposure: SGX cache attacks are practical[C]//11th USENIX Workshop on Offensive Technologies (WOOT 17). Berkeley, CA, USA: USENIX Association, 2017.

[204] Moghimi A, Irazoqui G, Eisenbarth T. Cachezoom: How SGX amplifies the power of cache attacks[C]//Fischer W, Homma N. Cryptographic Hardware and Embedded Systems – CHES 2017. Cham: Springer International Publishing, 2017: 69-90.

[205] Hähnel M, Cui W, Peinado M. High-Resolution side channels for untrusted operating systems [C]//2017 USENIX Annual Technical Conference (USENIX ATC 17). Berkeley, CA, USA: USENIX Association, 2017: 299-312.

[206] Van Bulck J, Piessens F, Strackx R. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic[C]//CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2018: 178–195.

[207] Huo T, Meng X, Wang W, et al. Bluethunder: A 2-level directional predictor based side-channel attack against SGX[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019, 2020(1): 321–347.

[208] Moghimi D, Van Bulck J, Heninger N, et al. CopyCat: Controlled Instruction-Level attacks on enclaves[C]//29th USENIX Security Symposium (USENIX Security 20). Berkeley, CA, USA: USENIX Association, 2020: 469-486.

[209] Information technology - Trusted platform module library - Part 1: Architecture[S/OL]. ISO/IEC 11889-1:2015, 2015. https://www.iso.org/standard/66510.html.

[210] Kivity A, Kamay Y, Laor D, et al. kvm: the Linux virtual machine monitor[C]//Proceedings of the Linux symposium: Vol. 1. Dttawa, Dntorio, Canada, 2007: 225-230.

[211] Van Bulck J, Weichbrodt N, Kapitza R, et al. Telling your secrets without page faults: Stealthy page Table-Based attacks on enclaved execution[C]//26th USENIX Security Symposium (USENIX Security 17). Berkeley, CA, USA: USENIX Association, 2017: 1041-1056.

[212] McVoy L, Staelin C. lmbench: Portable tools for performance analysis[C]//USENIX 1996 Annual Technical Conference (USENIX ATC 96). Berkeley, CA, USA: USENIX Association, 1996.

[213] Standard Performance Evaluation Corporation. SPEC CPU 2017[EB/OL]. (2022-12-06)[2024-01-27]. https://www.spec.org/cpu2017/.

[214] Mayer U F. Linux/Unix nbench[EB/OL]. (2017-01-02)[2024-01-27]. https://www.math.utah.edu/~mayer/linux/bmark.html.

[215] Lighttpd Developers. Lighttpd - fly light[EB/OL]. (2023-10-30)[2024-01-27]. https://www.lighttpd.net.

[216] SQLite Consortium. SQLite[EB/OL]. (2024-01-23)[2024-01-27]. https://www.sqlite.org.

[217] Raul. The nbench benchmark ported to SGX[EB/OL]. (2017-09-16)[2024-01-27]. https://github.com/utds3lab/sgx-nbench.

[218] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB[C]// SoCC '10: Proceedings of the 1st ACM Symposium on Cloud Computing. New York, NY, USA: Association for Computing Machinery, 2010: 143–154.

[219] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool[EB/OL]. 2023 [2024-01-27]. https://httpd.apache.org/docs/2.4/programs/ab.html.

[220] Zhao S, Zhang Q, Qin Y, et al. SecTEE: A software-based approach to secure enclave architecture using TEE[C]//CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2019: 1723–1740.

[221] Arm Limited or its affiliates. Arm Architecture Reference Manual for A-profile architecture [R/OL]. 2023. https://developer.arm.com/documentation/ddi0487/latest/.

[222] Waterman A, Asanović K, Hauser J. Hypervisor Extension, Version 1.0[R/OL]//The RISC-V Instruction Set Manual, Volume II: Privileged Architecture. 20211203 ed. RISC-V International, 2021: 99-136. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.

[223] Boubakri M, Chiatante F, Zouari B. Towards a firmware TPM on RISC-V[C]//2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). 2021: 647-650.

[224] Yang C F, Shinjo Y. Obtaining hard real-time performance and rich Linux features in a compounded real-time operating system by a partitioning hypervisor[C]//VEE '20: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2020: 59–72.

致　谢

衷心感谢导师陈渝老师对我的指导与帮助。在我读本科期间，陈老师就引领我走进了操作系统的奇妙世界，让我对计算机的底层原理产生了浓厚的兴趣，从此踏上了这条充满挑战与未知的道路。在陈老师的鼓舞下，我写了一遍又一遍的操作系统，每次都会收获全新的认识与满满的成就感，让我的代码能力与系统设计能力得到了巨大的提升，为日后的科研打下了扎实的基础。陈老师治学严谨，思维敏锐，工作勤奋，以身作则，也是我们学习的榜样。在陈老师的悉心指导与栽培下，我才得以从迷茫中找到方向，从困境中发现出路，从失败中重新振作，最终顺利完成多项研究成果。

特别感谢陈康老师对我的指导与帮助。陈康老师对我的科研工作提出了许多宝贵的意见与建议，每次学术交流都令我深受启发。陈康老师丰富的科研经验以及对问题的深刻理解，让我少走了不少弯路。另外，也要感谢向勇老师。与陈渝老师一样，向老师也是指引我走上操作系统道路的启蒙恩师。向老师与陈老师开设的操作系统课程与训练营，培养了一大批像我一样对操作系统充满热情的学生。

感谢田凯夫、尤予阳同学在 Skyloft 项目上对我的鼎力相助。感谢晏巨广、胡柯洋、闭浩扬、郑友捷、汪乐平、杨金博、萧络元、陈乐、中国科学院大学的齐呈祥、北京大学的袁世平等同学，泉城实验室的石磊老师，以及项目组其他成员对 ArceOS 项目的重大贡献。感谢蚂蚁集团的闫守孟、刘双、中科院信工所的王文浩老师对 HyperEnclave 项目的大力支持与帮助，也感谢他们为我提供了这个施展才能的机会与平台。

感谢实验室的甄艳洁学姐、魏超学长、戴臻旸、张译仁、陈张萌、刘丰源、许善朴、朱懿、柴小虎等同学，已毕业的沈游人学长、王润基等同学，与大家共同学习、讨论、娱乐、成长的时光令我终生难忘。

最后，感谢我的父母、家人与其他朋友，您们的支持和鼓励是我最坚实的后盾，让我在追求梦想的道路上永不孤单。

声 明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名：贾越凯 日 期：2024、5、26

个人简历、在学期间完成的相关学术成果

个人简历

1996 年 10 月 19 日出生于浙江省新昌县。

2015 年 8 月进入清华大学计算机科学与技术系计算机科学与技术专业，2019 年 7 月本科毕业并获得工学学士学位。

2019 年 8 月免试进入清华大学计算机科学与技术系攻读博士至今。

在学期间完成的相关学术成果

学术论文：

[1] **Yuekai Jia**, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, Zhengyu He. HyperEnclave: An Open and Cross-platform Trusted Execution Environment[C]//2022 USENIX Annual Technical Conference (USENIX ATC 22). Berkeley, CA, USA: USENIX Association, 2022: 437–454. (TH-CPL A 类会议)

[2] Zhenyang Dai, Shuang Liu, Vilhelm Sjöberg, Xupeng Li, Yu Chen, Wenhao Wang, **Yuekai Jia**, Sean Noble Anderson, Laila Elbeheiry, Shubham Sondhi, Yu Zhang, Zhaozhong Ni, Shoumeng Yan, Ronghui Gu, Zhengyu He. Verifying Rust Implementation of Page Tables in a Software Enclave Hypervisor[C]//ASPLOS '24: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. New York, NY, USA: Association for Computing Machinery, 2024: 1218–1232. (TH-CPL A 类会议)

[3] **Yuekai Jia**, Kaifu Tian, Yuyang You, Kang Chen, Yu Chen. Skyloft: A General High-Efficient Scheduling Framework in User Space. (SOSP '24 审稿中)

指导教师评语

随着对云计算等领域的高性能和高可靠需求，微虚拟机技术为通用操作系统的性能与安全优化提供了全新的视角，得到了越来越多的关注。该技术通过赋予应用程序部分操作系统功能，从而绕开原有操作系统的低效环节，实现性能、灵活性和安全性的显著提升。尽管如此，现有微虚拟机技术在多个关键领域仍面临挑战，包括任务调度优化、应用专用化系统构建、以及敏感数据的保护等方面。针对这些挑战，论文“微虚拟机性能优化关键技术研究”展开了深入研究，并提出了一系列创新解决方案。以下是对这些研究内容的扩展和改进：

（1）基于用户态中断机制的任务调度性能优化。为了提升通用操作系统上的任务调度性能，本文设计并实现了 Skyloft 库操作系统。Skyloft 利用 Intel 最新硬件支持的用户态中断技术，为用户态调度器提供了微秒级的抢占能力，有效优化了在多样化负载情况下的尾延迟问题。Skyloft 的设计理念在于允许不同应用的线程在同一用户态调度器下进行切换，实现多应用的统一调度，从而提升 CPU 的利用率。通过一系列实验分析，相比现有系统，Skyloft 在最大吞吐量上提升了 19%，在尾延迟上降低了 33%。

（2）应用专用的组件化操作系统设计。为了便于为每个应用定制微虚拟机操作系统，以获得最佳性能，论文提出了一种创新的组件化操作系统设计方法——ArceOS。该方法根据操作系统设计理念的相关性进行组件划分，旨在降低系统组件间的耦合性，提高组件的可重用性。ArceOS 利用 Rust 语言的安全性和并发性特性，设计了 API 快速路径，特别针对微虚拟机场景进行了性能优化，同时保证了与现有系统的兼容性。实验结果表明，采用 ArceOS 的组件化设计，可以方便地构建出针对特定应用优化的操作系统，其 API 快速路径设计使得小文件读写性能提升了 50%。此外，ArceOS 还能够兼容现有的 C 应用程序，在与 Linux 系统相比的情况下，能够使 Redis 数据库的延迟降低 33%。

（3）面向机密计算的敏感数据灵活保护机制。为了解决现有方法在保护微虚拟机内敏感数据时面临的性能与灵活性不高的问题，论文提出了一种新的可信执行环境设计方法——HyperEnclave。HyperEnclave 通过一个轻量级的微虚拟机管理程序，实现了对微虚拟机的灵活隔离，以满足不同场景下对性能和安全的需求。该方法仅依赖硬件虚拟化技术和可信平台模块，无需额外的安全硬件支持。实验结果表明，HyperEnclave 可配置不同安全/性能需求的灵活保护方案，引入的性能开销很小，在 SQLite 数据库上的开销小于 5%，其性能与安全性与现有的 Intel SGX

硬件方法相当。

本文的研究工作不仅在理论上提出了创新的解决方案，而且在实际应用中也展现出了显著的性能提升和安全保障，为微虚拟机技术的发展和应用提供了有力的技术支撑。论文结构严谨，逻辑清晰，论证充分，语言表达明确，具有较好的创新性，说明了该生掌握了深入的操作系统内核、安全管理、性能优化等方面的知识体系，并具有很强的工程实践能力。论文达到了博士学位论文的相应要求。

答辩委员会决议书

论文研究了微虚拟机的系统设计实现与性能优化，选题具有重要的理论意义和应用价值。

论文的主要工作和创新性成果包括：

1. 设计并实现了 Skyloft 库操作系统，通过抢占式用户态调度机制和用户态多应用切换机制，支持微秒级抢占，优化了多样化负载下的尾延迟和吞吐量。

2. 设计并实现了基于 Rust 语言和 Unikernel 架构的 ArceOS 操作系统，通过单向依赖内核组件化机制和 API 快速路径优化方法，在兼容 Linux 应用的场景下，提升了网络 I/O 性能。

3. 设计并实现了可信计算环境 HyperEnclave，仅基于硬件虚拟化与可信平台模块，提供了平台无关的安全隔高机制，与现有 SGX 硬件方法的性能与安全性相当。

论文工作表明作者掌握了本学科坚实宽广的基础理论和系统深入的专门知识，独立从事科研工作的能力强。论文结构合理，阐述清晰，写作规范，达到了工学博士论文的要求，是一篇优秀的博士学位论文。

贾越凯答辩过程中表达清楚，回答问题正确。经答辩委员会无记名投票，一致同意通过贾越凯的博士论文答辩，并建议授予工学博士学位。