上海交通大学硕士学位论文

融合动态内存管理的服务器无感知计算存储优化架构研究

姓　　名：金凌啸

学　　号：123033910038

导　　师：管海兵、马汝辉

院　　系：计算机学院

学科/专业：计算机科学与技术

申请学位：工学硕士

2026 年 1 月 17 日

A Dissertation Submitted to

Shanghai Jiao Tong University for the Degree of Master

RESEARCH ON SERVERLESS COMPUTING STORAGE OPTIMIZATION ARCHITECTURE INTEGRATING DYNAMIC MEMORY MANAGEMENT

Author: Lingxiao Jin

Supervisor: HaiBing Guan and Ruhui Ma

School of Computer Science

Shanghai Jiao Tong University

Shanghai, P.R. China

January 17th, 2026

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全知晓本声明的法律后果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

上海交通大学

学位论文使用授权书

本人同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于：

□**公开论文**

□**内部论文**，保密□1 年/□2 年/□3 年，过保密期后适用本授权书。

□**秘密论文**，保密____年（不超过 10 年），过保密期后适用本授权书。

□**机密论文**，保密____年（不超过 20 年），过保密期后适用本授权书。

（请在以上方框内选择打“✓”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

随着云计算技术的飞速发展，服务器无感知计算作为一种新兴的云原生范式，凭借其按需付费、弹性伸缩和零运维等优势，已在工业界和学术界获得了广泛应用。然而，随着应用场景的丰富，现有的服务器无感知架构在处理 I/O 密集型任务时面临着严峻的性能挑战。主流服务器无感知平台通常采用内存与 CPU 算力线性耦合的资源分配策略，用户为了获得足够的计算能力往往被迫超额配置内存，导致大量已分配的内存资源在实际运行中处于闲置状态，造成了严重的资源浪费。这种资源供给与实际需求的不匹配，不仅增加了用户的成本，也降低了云平台的整体资源利用效率。因此，如何在不改变现有编程模型的前提下，挖掘并利用这些闲置的内存资源来加速文件 I/O，同时实现集群层面的负载均衡，成为了当前服务器无感知计算研究中亟待解决的关键问题。

针对上述挑战，本文提出了一种融合动态内存管理与集群负载均衡的服务器无感知计算存储优化架构——Ephemera。该架构旨在将计算节点中未被充分利用的内存转化为高性能的临时存储介质，通过分层协同的设计实现对存储性能的极致优化。本文的设计需要满足以下三个设计目标，以满足复杂服务器无感知计算工作流的需求。首先，本文旨在实现透明的内存 I/O 集成，其挑战在于无缝集成此功能，而无需用户修改其现有代码库。这一复杂过程涉及拦截 I/O API 并将基于磁盘的文件访问转换为基于内存的操作，同时保持与各种函数执行环境的向后兼容性。其次，本文专注于异构任务资源协同，由于不同实例的内存需求各异，这带来了挑战。在异构实例（如 CPU 密集型或 I/O 密集型任务）之间实现高效的内存共享，对于优化资源利用率和提高整体性能至关重要。最后，本文强调协调的集群工作负载编排，这解决了在集群级别维持内存文件系统有效性的挑战。平衡集群节点间的工作负载分布对于防止资源争用和通过避免将相同类型的工作负载分配给单个节点来确保内存文件系统的最佳性能至关重要。

具体工作包括以下三个方面：

- **在函数运行时层面，本文设计并实现了透明化的运行时守护进程**。该组件驻留在每个函数实例内部，利用系统调用拦截技术接管用户代码中的文件操作请求。它在用户态构建了一个轻量级的内存文件系统，通过内存映射技术将原本指向磁盘的读写操作无缝重定向至内存中，实现了数据的零拷贝访问。

- **在单机节点层面，本文提出了基于租户隔离的动态内存共享机制**。针对同一节点上混合部署的异构任务，本文设计了租户管理器来协调资源分配。该组件引入了“收割池”与“分配池”的概念，将处于内存富余状态的 CPU 密集型任务与急需内存支持的 I/O 密集型任务进行配对，将闲置内存动态“收割”并划拨给 I/O 任务使用，实现节点内部资源的细粒度互补与协同。
- **在集群调度层面，本文构建了基于资源画像的全局负载均衡控制器**。为了在更大范围内优化资源配置，本文引入了具备特征感知能力的集群调度器。该组件在函数部署阶段通过试运行采集其内存使用峰值与 I/O 访问特征，构建精准的任务资源画像。在任务调度阶段，控制器基于各工作节点的实时内存水位与已分配配额，采用最小化供需差值的策略将任务分发至最合适的节点。

本文实现了 `Ephemera` 的原型并进行了广泛的实验评估。实验结果表明，与传统的文件系统相比，`Ephemera` 在 I/O 密集型任务上取得了显著的性能提升。在端到端延迟方面，系统平均降低了 50% 的文件操作耗时，在开启内存共享机制的理想场景下，延迟降幅最高可达 95.73%。此外，实验还证实了该架构在多实例并发场景下能够显著提升集群的内存资源利用率，且引入的运行时开销极低，证明了其在实际生产环境中的可行性与应用价值。

关键词：服务器无感知计算，文件系统，资源收割，资源调度

Abstract

With the rapid development of cloud computing, serverless computing has emerged as a new cloud-native paradigm and has gained wide adoption in both industry and academia due to its pay-as-you-go pricing model, elastic scalability, and zero-operation overhead. However, as application scenarios become increasingly diverse, existing serverless architectures face severe performance challenges when handling I/O-intensive workloads. Mainstream serverless platforms typically adopt a resource allocation strategy that linearly couples memory capacity with CPU compute power. To obtain sufficient computational capability, tenants often have to over-provision memory, which leaves a large fraction of the allocated memory idle during execution and results in significant resource waste. This mismatch between resource provisioning and actual demand not only increases user costs but also reduces overall resource utilization across cloud platforms. Therefore, without changing the existing programming model, exploiting and utilizing these idle memory resources to accelerate file I/O while achieving cluster-level load balancing becomes a critical problem in current serverless computing research.

To address these challenges, this paper proposes `Ephemera`, a serverless storage optimization architecture that integrates dynamic memory management with cluster load balancing. This architecture aims to transform underutilized memory on compute nodes into high-performance temporary storage and to achieve aggressive storage performance optimization through a layered and cooperative design. To meet the demands of complex serverless computing workflows, our design satisfies the following three design goals. First, this paper aims for transparent memory I/O integration. The challenge involves seamlessly integrating this functionality without requiring users to modify their existing codebases. This complex process entails intercepting I/O APIs and converting disk-based file access into memory-based operations while maintaining backward compatibility with various function execution environments. Second, this paper focuses on heterogeneous task resource synergy, which presents challenges due to the varying memory requirements of different instances. Achieving efficient memory sharing among heterogeneous instances (such as CPU-intensive versus I/O-intensive tasks) optimizes resource utilization and improves overall performance. Fi-

nally, this paper emphasizes harmonized cluster workload orchestration, which addresses the challenge of maintaining in-memory file system effectiveness at the cluster level. Balancing workload distribution across cluster nodes prevents resource contention and ensures optimal in-memory file system performance by avoiding the allocation of homogeneous workloads to a single node.

Here are the main contributions:

- **At the function runtime level, this paper designs and implements a transparent Runtime Daemon.** This component resides inside each function instance and takes over file operation requests from user code through system call interception. It constructs a lightweight in-memory file system in user space and seamlessly redirects read and write operations originally targeting disks to memory via memory-mapped I/O, enabling zero-copy data access.
- **At the single-node level, this paper proposes a tenant-isolated dynamic memory sharing mechanism.** To handle heterogeneous workloads co-located on the same node, this paper designs a Tenant Manager to coordinate resource allocation. This component introduces the concepts of a "harvesting pool" and an "allocation pool," pairs CPU-intensive tasks with surplus memory and I/O-intensive tasks that urgently require memory, dynamically harvests idle memory, and reallocates it to I/O workloads. This design enables fine-grained complementarity and cooperation among resources within a node.
- **At the cluster scheduling level, this paper builds a global load balancing Cluster Controller based on resource profiling.** To optimize resource allocation at a larger scale, this paper introduces a Cluster Controller with feature-awareness. During function deployment, the Controller collects peak memory usage and I/O access characteristics through trial executions and constructs accurate resource profiles for each workload. During scheduling, the Controller considers real-time memory watermarks and allocated quotas of each worker node and dispatches tasks to the most suitable nodes by minimizing the supply–demand gap.

This paper implements a prototype of `Ephemera`and conducts extensive evaluations. The results show that, compared with traditional file systems, `Ephemera`achieves substantial performance improvements for I/O-intensive workloads. In terms of end-to-end latency,

the system reduces average file operation time by 50%, and in ideal scenarios with memory sharing enabled, latency reductions reach up to 95.73%. Moreover, the experiments demonstrate that this architecture significantly improves cluster-level memory utilization under multi-instance concurrency, while introducing negligible runtime overhead, which confirms its practicality and value in real-world production environments.

Key words: Serverless Computing, File System, Resource Harvesting, Resource Scheduling

目 录

插 图

表 格

算 法

第 1 章 绪论

1.1 研究背景及意义

云计算作为信息技术领域的根本性范式转变，通过按需提供可配置的计算资源共享池，彻底重塑了软硬件服务的交付与消费模式。纵观其演进历程，云架构呈现出资源抽象粒度不断细化、管理责任逐步向服务商转移的显著趋势。在云计算的初级阶段，以虚拟机（Virtual Machine, VM）为核心载体的基础设施即服务（Infrastructure as a Service, IaaS）占据主导地位[1]。IaaS 通过硬件虚拟化技术实现了物理资源的池化与隔离，成功将企业的资本支出转化为运营支出。然而，尽管 IaaS 屏蔽了底层硬件的复杂性，租户仍需承担繁重的操作系统维护、中间件配置以及资源预置工作，难以完全聚焦于业务逻辑本身。在此之上，云服务提供商进一步向上抽象，形成平台即服务（Platform as a Service, PaaS），为应用开发提供运行时环境、数据库和中间件等托管平台[2]。尽管 PaaS 简化了部署流程，但其受限于运行时环境的刚性约束以及缺乏真正的“缩容到零（Scale-To-Zero）”能力，导致用户在无负载期间仍需承担闲置资源成本。与此同时，软件即服务（Software as a Service, SaaS）通过浏览器或轻量客户端向终端用户直接交付完整应用，用户无需关心底层平台和基础设施[3]。为了进一步降低运维复杂度并提升开发敏捷性，以 Docker 为代表的容器化技术推动计算范式向微服务架构演进[4]。容器虽然实现了应用运行环境的标准化与轻量级隔离，但开发者在一定程度上仍需关注集群容量规划与底层节点管理。在此背景下，为了实现极致的资源利用率与“零运维”愿景，服务器无感知计算（Serverless Computing）应运而生[5-7] 。作为云计算演进的高级形态，它将应用逻辑与基础设施彻底解耦，标志着计算模式从粗粒度的资源租赁向细粒度的函数执行转变。

自从 AWS Lambda①发布以来，服务器无感知计算已经从一个新颖的云计算概念转变为学术界和工业界广泛认可的云计算范式。服务器无感知计算并不意味着没有服务器；相反，它将云计算抽象为函数即服务 (Function as a Service, FaaS) 和后端即服务 (Backend as a Service, BaaS)[8]。云租户只需将其业务逻辑编码为函数并将其组织成应用程序。相比之下，云服务提供商负责资源管理、负载均衡和其他后端服务。与现有的计算范式[9] 相比，服务器无感知计算使云租户免于底层基础设施管理的负担，同时仍能获得高可靠的服务。服务器无感知计算的最新进展不断解决各种瓶颈，包

① https://aws.amazon.com/lambda/

括冷启动[10-12]、资源管理[5,13-14]、异构硬件支持[15-17] 和工作流优化[18-21]。尽管服务器无感知计算最初是为短生命周期的事件驱动型任务设计的，但服务器无感知计算服务支持的函数类型已变得更加多样化[22]，包括大数据分析、机器学习模型训练和推理。与短生命周期的作业相比，这些更复杂的作业类型涉及通用的计算操作，并且需要与持久存储相关的 I/O 操作。例如，MapReduce[23] 需要存储 map 阶段的中间结果以供 reduce 阶段使用。在机器学习模型的训练过程中[24]，需要保存性能最佳的模型，而在模型推理的初始阶段[25]，则需要将预训练模型从磁盘加载到内存中。作业类型的多样化对服务器无感知计算平台的文件系统管理解决方案提出了新的要求。

在当前云计算环境下，服务器无感知计算因其无需开发者管理服务器资源、按需分配资源等优势而日益受到关注。然而，随着应用场景的日益多样化，服务器无感知计算在实际应用中面临着诸多挑战，尤其是在处理 I/O 密集型工作负载时。这些挑战主要包括以下几个方面：首先，内存资源的低效利用是一个显著问题。虽然云平台为服务器无感知计算实例分配了内存资源，但在实际应用中，内存的使用率往往未能充分发挥其潜力，导致资源浪费。例如，许多服务器无感知计算函数在执行过程中并未完全利用分配的内存，造成大量内存闲置，降低了整体资源的利用效率。此外，不同类型的任务对内存的需求差异较大，如何在多样化的工作负载下实现内存资源的动态调配和高效利用，仍然是一个亟待解决的问题。其次，I/O 操作的性能瓶颈限制了 I/O 密集型任务的执行效率。传统的存储系统在高频繁读写操作下表现不佳，难以满足服务器无感知计算架构下对高吞吐量和低延迟的需求。这不仅影响了任务的执行速度，也制约了整体系统的可扩展性与响应能力。尤其是在数据密集型应用，如大数据分析、机器学习模型训练和推理等场景中，I/O 性能的瓶颈更加明显，严重影响了应用的整体性能和用户体验。

解决上述挑战具有重要的现实意义，不仅能够提升现有服务器无感知计算系统的性能和效率，还将在多个层面上带来广泛的影响和积极的推动作用。第一，显著提升资源利用效率，增强服务器无感知计算的服务供给能力。通过有效解决由内存过度配置引发的资源闲置问题，能够大幅提高整体内存资源的利用率。一方面，对于用户而言，这意味着能够最大化已分配资源的价值，在不增加成本的基础上挖掘潜在的计算性能；另一方面，对于云服务提供商而言，系统整体性能的提升将显著增强平台的市场竞争力，从而吸引更多用户并促进生态发展。第二，突破 I/O 性能瓶颈，增强系统响应能力。优化 I/O 密集型任务的执行效率，能够显著缩短在线数据分析、实时机器学习推理等延迟敏感型应用的响应时间。高效的 I/O 处理能力将直接提升用户

体验，增强服务器无感知计算在高性能计算领域的竞争力。第三，提升平台的灵活性与可扩展性，推动技术生态发展。构建智能化的资源调度与高效的临时存储管理机制，将使服务器无感知计算平台能够更好地适应大规模数据处理和复杂工作流的需求。这不仅增强了系统在高负载下的稳定性，也为服务器无感知计算在大数据、人工智能及物联网等前沿领域的深入应用奠定了坚实基础。

本工作旨在探索和开发一种多层级的系统解决方案，利用过度配置的内存资源来优化函数执行期间的 I/O 效率。该框架能够在服务器无感知计算环境下实现透明的内存 I/O 集成，而无需用户修改其现有代码库。这一复杂过程涉及拦截 I/O API 并将基于磁盘的文件访问转换为基于内存的操作，同时保持与各种函数执行环境的向后兼容性。其次，本文专注于异构任务资源协同，在异构实例之间实现高效的内存共享，优化资源利用率和提高整体性能。最后，本文强调协调的集群工作负载编排。平衡集群节点间的工作负载分布对于防止资源争用和通过避免将相同类型的工作负载分配给单个节点来确保内存文件系统的最佳性能至关重要。

1.2 国内外研究现状

为了全面阐述本文的研究背景与技术路线，本节将从四个关键维度对相关领域的现有工作进行综述。首先，第1.2.1节重点介绍了面向服务器无感知计算环境的文件系统研究，分析了现有方案在处理 I/O 密集型任务时的优势与不足。其次，第1.2.2节探讨了分层文件系统的设计原理，特别是利用异构存储介质进行性能优化的策略，这为本文利用内存作为缓存层提供了理论依据。然后，第1.2.3节梳理了资源收割技术在云计算中的应用，阐述了如何挖掘和利用系统中的闲置资源。最后，第1.2.4节概述了集群任务调度与工作负载均衡研究，从传统集群、容器编排到服务器无感知计算场景的演进入手，分析现有工作在多维资源调度与负载分配方面的主要思路与局限。通过对这些领域的深入分析，旨在明确现有技术的局限性，从而引出本文的研究动机。

1.2.1 面向服务器无感知计算的文件系统优化

为了支持通用任务，服务器无感知计算平台通常需要为函数提供基于文件的存储抽象，以兼容大量依赖 POSIX 接口的现有应用和库。围绕这一需求，已有研究主要聚焦于如何在服务器无感知计算环境中提供高性能、可扩展且具备持久性的存储服务，同时兼顾多租户隔离、安全性和运维复杂度等因素。

Merenstein 等人[26] 设计了 F3，这是一个面向服务器无感知计算场景、具有位置

感知数据调度能力的可堆叠文件系统。F3 能够智能地区分临时数据和需要高持久性的数据，并将临时数据透明地定向到节点本地磁盘，从而减少跨网络访问远程存储带来的额外开销。通过这种冷热数据分离以及本地优先的策略，F3 在减轻远程存储负载、降低访问延迟方面取得了良好效果。然而，该系统的优化重点主要集中在数据放置与路由策略，即在远程存储与本地磁盘之间做出合适选择，对于节点内部更高速的内存资源如何参与数据路径、如何被系统性地组织与利用，尚未给出深入讨论。Schleier-Smith 等人[27] 提出了 FaaSFS，这是一种专为服务器无感知计算设计的共享文件系统。FaaSFS 采用乐观并发控制机制处理 POSIX 调用，将文件系统访问封装为事务，并利用本地缓存状态保证在出现冲突时能够回滚并恢复一致性，从而在高并发服务器无感知计算环境下提供较强的一致性保证。这种设计使得多个函数实例能够共享统一的命名空间和文件视图，便于开发者迁移传统应用。但与此同时，复杂的事务机制以及与远程元数据服务的频繁交互也可能引入额外的延迟和运行时开销，在 I/O 密集或对尾延迟敏感的工作负载下，其性能成本不容忽视。在运行时优化方面，RunD[28] 提出了一种轻量级安全容器运行时，通过结合 virtio-fs 和 virtio-blk 来优化 rootfs 的访问路径。具体而言，RunD 使用 virtio-fs 支持 rootfs 的只读部分，使主机与来宾之间能够高效共享页面缓存，从而减少冷启动时的镜像加载开销；同时使用 virtio-blk 支持 rootfs 的可写部分，以获得更高的 I/O 吞吐性能并保证数据隔离。这一分离只读与可写路径的设计，在提升函数启动速度和运行期间的 I/O 性能方面表现出良好效果，但其优化重心仍然围绕函数镜像及 rootfs，而非函数执行过程中动态产生的中间数据和临时文件。Hattori 等人[29] 引入了 Sentinel 运行时，通过对指定文件系统进行只读挂载，结合页面缓存共享与复用机制，有效减轻了冷启动延迟和内存占用。Sentinel 的基本思路是将函数依赖的只读数据和二进制文件尽可能地在多次调用之间复用，从而摊薄启动成本、减少重复加载造成的内存浪费。然而，Sentinel 的优化对象主要是只读工作负载，对于大量存在写入、更新以及生成临时中间结果的写密集型任务，其支持能力相对有限，缺乏面向写路径和短生命周期数据的专门优化手段。

总体来看，上述工作在为服务器无感知计算提供持久化存储、共享文件系统语义以及优化冷启动等方面取得了重要进展，极大丰富了服务器无感知计算平台的存储基础设施。不过，这些研究的关注点多集中在分布式存储架构和运行时隔离机制的优化上，对于计算节点内部尚未被充分利用的资源潜力关注较少。特别是服务器无感知计算函数中普遍存在的闲置内存资源，尚未被系统性地转化为提升 I/O 性能的加速层；节点内部也通常缺乏在多容器或多函数实例之间进行细粒度、受控的内存空间共

享机制。

因此，如何在保持现有编程接口和存储语义（例如继续支持 POSIX 接口、尽量不修改函数代码）的前提下，挖掘并整合节点内部的闲置内存资源，将其以透明或半透明的方式集成到服务器无感知计算平台的 I/O 路径中，为函数的中间数据和临时文件提供高性能的内存级存储支持，是当前研究中尚未充分探索但具有重要潜力的方向。

1.2.2 分层文件系统

为了利用无服务期计算中被浪费的内存，考虑引入分层存储管理（Hierarchical Storage Management，HSM），将磁盘结合内存，提供更高效的文件操作。分层存储管理系统可以追溯到几十年前，当时磁盘和磁带是唯一的大规模存储技术。已经有过多种面向块存储介质的商业 HSM 解决方案。IBM Tivoli Storage Manager 是其中一个成熟的 HSM 系统，它能够将很少使用或已经足够老化的文件透明地迁移到更低成本的媒体上。EMC DiskXtender 是另一个 HSM 系统，它能够自动将不活动数据从昂贵的层迁移到更低成本的媒体上。AutoTiering[30]是另一个基于块的存储管理系统的例子。它使用采样机制来估算在其他层上运行虚拟机的 IOPS，并根据 IOPS 测量和迁移成本计算它们的性能分数，并相应地排序所有可能的移动。一旦达到一个阈值，就会启动一个实时迁移。随着非易失存储器（Non-Volatile Memory，NVM）的出现，分层存储管理的形式受到了更多研究者的关注，研究者考虑将 NVM 与 DRAM 或磁盘进行结合。Narayanan 等人提出的 Strata[31]是一个多层的用户空间文件系统，利用 NVM 作为高性能层，SSD/HDD 作为低性能层。它利用 NVM 的字节寻址特性将日志合并，并将其迁移到较低层。在 Strata 中，文件数据只能在 NVM 中分配，只能从更快的层迁移到更慢的层。Strata 的配置粒度是页面，这增加了记账开销并浪费了文件访问的本地信息。Dulloor 等人提出的 X-Mem[32]依赖于离线分析机制，用于将内容分配到 DRAM 或 NVM 中。X-Mem 分析器跟踪每个内存访问并追踪它们，以找到每个数据结构的最佳存储匹配。但是 X-Mem 要求用户对源代码进行几个修改。Agarwal 等人提出了 Thermostat[33]，一种用于管理两层内存的大页的方法，它会将冷页面透明地迁移到 NVM 作为缓慢的内存，而将热页面迁移到 DRAM 作为快速内存。这种方法的缺点是对于那些主存储器大部分区域的温度均匀的应用程序性能会下降。Zheng 等人提出的 Ziggurat[34]是一个跨越 NVM 和磁盘的分层文件系统，通过准确和轻量级的预测器来管理数据的放置，以将传入的文件写入定向到最适合的层。在后台，Ziggurat 估计文件数据的“温度”，并将冷文件数据从 NVM 迁移到磁盘上。为了充分利用磁

盘带宽，Ziggurat 将数据块合并成大型的连续写入。Ziggurat 弥合了基于磁盘和基于 NVM 的存储之间的差距，为应用程序提供高性能和大容量。这些工作都巧妙地结合了各个存储介质的特性，如内存的高性能和磁盘的大容量，在此基础上还探索如何更合理地将内容分配到对应的存储介质中。

虽然分层文件系统在传统数据中心和高性能计算领域已经相当成熟，但将其直接应用于服务器无感知计算环境仍面临诸多挑战。服务器无感知计算具有高度的动态性、短暂的生命周期以及严格的资源限制，这与传统分层系统所处的相对稳定的物理环境截然不同。现有的分层策略往往依赖于复杂的预测模型或持久化的元数据管理，难以适应服务器无感知计算函数的快速启动和销毁特性。因此，需要探索一种轻量级、适应性强的分层存储机制，能够在容器级别动态地利用瞬时内存资源，构建高效的临时缓存层，以满足服务器无感知计算应用对高性能 I/O 的需求。

1.2.3　资源收割技术的应用

服务器无感知计算中固有的资源预分配机制往往导致资源在实际使用期间的利用率不足，例如函数为应对负载波动而预留的 vCPU 和内存在大部分时间处于闲置状态。围绕这一问题，资源收割技术逐渐成为提升资源利用率和降低整体运行成本的重要研究方向[35-37]。其基本思想是在不显著影响前台服务质量的前提下，安全、动态地回收过度配置或暂时闲置的资源，并将其用于其他计算任务或系统优化。

Yu 等人[38] 提出了 Libra，这是一个面向云服务提供商的端到端解决方案，用于在服务器无感知计算平台中进行资源收割和再利用。Libra 利用机器学习模型预测函数调用的资源需求和执行时间，据此在函数生命周期内识别出可被安全回收的资源窗口，并在这些窗口中进行资源收割。在保证服务等级协议（Service Level Agreement, SLA）的前提下，Libra 将收集到的空闲资源用于加速大规模、输入各异的服务器无感知计算函数调用，从而在平台整体层面提高资源利用率与任务吞吐。然而，这种预测驱动的方案在很大程度上依赖模型对工作负载行为的准确刻画，一旦预测偏差较大，可能影响收割决策的安全性和收益。Zhang 等人[39] 从公有云实际工作负载出发，对 Microsoft Azure 上的服务器无感知计算任务特征进行了系统分析，并围绕 Harvest VM（一类价格较低、但可能随时被抢占或性能波动的虚拟机形态）设计了专门的服务器无感知计算负载均衡器。该系统能够识别 Harvest VM 中潜在的驱逐信号及资源变化，动态调整不同 VM 间的任务调度策略，使部分服务器无感知计算请求可以在 Harvest VM 上以更低成本运行，同时尽量降低被抢占带来的服务质量影响。这一工作展示了将可抢占资源纳入服务器无感知计算平台执行底层的可行性，为云环境中

弹性、低成本的资源收割提供了实践经验。Freyr+[40] 则从资源管理器视角出发，提出了一种专为服务器无感知计算平台设计的新型资源管理框架。Freyr+ 对每个函数实例的资源利用率进行细粒度、实时监控，从过度配置的函数中动态收集空闲资源，并将这些资源再分配给资源不足的函数，以此实现集群范围内的资源再平衡。与静态配额或单次部署时的资源规划不同，Freyr+ 更强调在函数运行期持续调整和再配置，通过闭环反馈提升整体资源利用效率和用户感知性能。除了在服务器无感知计算系统内部进行资源回收外，Liu 等人[41] 设计了 SMore，将服务器无感知计算函数与现有云工作负载进行并置部署，从云节点上回收可用的 GPU 资源，为服务器无感知计算环境下的深度学习推理任务提供支持。SMore 的核心思想是在不额外增加专用 GPU 集群的情况下，利用现有云节点上闲置的加速器资源，为函数执行提供硬件加速能力，从而在保证成本可控的前提下提升推理性能和吞吐。

总体而言，现有资源收割技术在服务器无感知计算或更广泛的云计算环境中，已经在 CPU、GPU 等计算资源的再分配方面取得了显著进展，主要目标是通过提升计算资源利用率来增加系统吞吐量并降低成本。然而，针对内存资源的系统性收割与再利用，尤其是将收割的内存显式用于缓解 I/O 瓶颈、构建高性能 I/O 加速层的工作仍然相对有限。在 I/O 密集型服务器无感知计算工作负载中，如果能够在单租户或单节点范围内，以受控方式共享和复用收割得到的内存资源，并将其直接转化为提升文件访问性能的缓存或内存文件系统，将有望显著降低数据访问延迟、减轻远程存储压力。因此，如何围绕内存这一关键资源构建面向 I/O 加速的收割与再利用机制，并与服务器无感知计算平台现有的调度和隔离机制协同工作，是一个具有重要研究价值且尚未被充分探索的方向。

1.2.4 集群任务调度与工作负载均衡

在大规模分布式集群与云计算平台中，任务调度与工作负载均衡始终是资源管理的核心问题之一。随着数据中心规模与应用形态的快速演化，如何在保证系统吞吐与任务响应时间的前提下，实现对 CPU、内存、存储与网络等多维资源的高效调度，已成为学术界与工业界长期关注的研究重点。现有研究大体可以分为传统集群与数据中心调度、云计算与容器编排环境中的负载均衡以及服务器无感知计算场景下的在线调度与负载均衡等几个方向。

在传统集群与数据中心环境中，典型代表是 Google 的 Borg[42] 与 Omega[43]，以及业界公开的 Mesos[44] 和 YARN[45] 等系统。Borg 采用基于约束的集中式调度，对作业的资源需求（CPU、内存、端口等）进行统一建模，在逻辑上由一个全局调度器

在集群范围内做出决策，从而在大规模集群中实现较高的资源利用率和任务吞吐[42]。Omega[43] 则提出共享状态调度架构，多个并发调度器在共享的集群状态上独立决策，并通过乐观并发控制机制在提交更新时检测并解决冲突，以提高可扩展性和调度吞吐。Mesos[44] 采用双层调度架构，底层 Mesos master 通过资源提供机制向上层框架（如 Hadoop、Spark）提供一组资源，由各框架在所获得的资源份额内自行进行任务调度和管理，从而支持多种计算框架在同一集群上的共存与资源共享。YARN[45] 将全局资源管理与应用级调度解耦，通过 ResourceManager 与每个应用对应的 ApplicationMaster 的分层架构，对大规模集群资源进行统一管理和应用内细粒度调度。在此类集群管理系统及相关工作中，负载均衡与多租户资源共享通常依赖基于队列长度、CPU/内存利用率等指标的启发式策略，以及面向多资源公平性的分配模型等方法[46]。这些传统调度系统在 CPU 与内存等资源的静态或粗粒度平衡方面效果显著，但对 I/O 密集型应用、临时数据密集访问以及细粒度内存利用的建模与优化相对不足。

在云计算和容器编排场景下，Kubernetes 逐渐成为事实上的容器编排标准，其内置调度器采用“过滤（Filtering）+ 打分（Scoring）”的两阶段流程：首先根据资源请求、节点亲和性/反亲和性、污点与容忍度等硬性约束筛选候选节点，随后基于多种优先级函数（如资源利用均衡、拓扑感知、亲和性偏好等）对候选节点进行加权打分与排序，最终选择得分最高的节点[47]。在负载均衡方面，Kubernetes 一方面通过 Service、Ingress 等抽象实现对无状态服务的流量层负载分发，另一方面在调度决策中兼顾 CPU 和内存的分配均衡，以避免节点过载。同时，配合 HPA/VPA 等自动扩缩容机制，系统能够根据历史利用率或自定义指标动态调整副本数，实现弹性伸缩。围绕容器云环境的调度优化，Al-Dhuraibi 等[48] 综述了多种基于预测的弹性伸缩与调度方法，例如利用时间序列分析或机器学习预测短期负载波动，从而提前触发容器迁移或资源重分配。Mao 等[49] 则提出 DeepRM，尝试利用深度强化学习解决多资源装箱问题，旨在复杂状态空间下直接优化平均作业减速比等服务级别目标。然而，上述研究大多将任务视为资源消耗的“黑盒”，主要在较粗粒度上对 CPU 和内存容量进行建模，对任务内部的 I/O 模式、临时文件读写行为关注不足，也鲜少显式刻画“内存—存储”耦合对系统性能的潜在影响。周凯[50]为解决容器集群调度中因忽视多样化需求而导致的资源负载失衡问题，提出了一种基于高斯混合模型（Gaussian Mixture Model, GMM）的定制化调度策略 CS-GMM ，该策略通过对容器的资源与属性需求进行精准聚类并分配独立权重，在负载均衡度和调度成功率等方面显著优于原生 Kubernetes 调度器。

随着服务器无感知计算平台（如 AWS Lambda、Azure Functions 等）的兴起，资源管理进一步细化到函数粒度，需在更短的生命周期和更高的突发性下实现弹性伸缩。与传统长期运行服务相比，服务器无感知计算任务具有无状态、执行时间短且调用链路复杂的特征，使得传统面向 VM 或容器的调度策略难以直接套用。在服务器无感知计算调度优化方面，研究者重点关注如何通过调度降低冷启动延迟与数据传输开销。金鑫等人[51]提出了一种面向服务器无感知计算场景的可定制函数调度算法 FuncSched，该算法创新性地结合了函数的时间执行特征与空间资源消耗特征，通过多维度的优先级计算有效缓解了队头阻塞问题，显著降低了函数平均完成时间。张信民等人[52]针对服务器无感知计算中统一 CPU 调度策略难以满足多样化需求及短任务延迟高的问题，提出并实现了一种调度隔离机制以及综合考虑函数执行、启动与等待时间的可定制调度策略 FaaSchedule，能显著降低短任务的平均完成时间。SAND[53] 提出了一种基于亲和性的调度策略，将同一应用的多个函数实例分配至同一节点，利用本地消息总线替代远程存储，从而显著降低交互延迟。针对函数工作流的调度，FaasFlow[54] 设计了以数据为中心的调度器，通过自适应地将数据依赖型函数调度到同一工作节点，利用共享内存实现零拷贝的数据传递，避免了不必要的网络 I/O。此外，面对极度突发的负载，Nightcore[55] 引入了细粒度的调度机制，它不仅优先将内部函数调用调度至本地节点以降低网络延迟，还实时计算并动态调整函数的最佳并发度，从而有效避免了突发负载下的资源过载和队头阻塞问题。Atoll[56] 提出了一种分层的半全局调度架构，并采用了截止期感知的 “最短剩余松弛时间优先” 策略，通过将集群划分为由独立调度器管理的 Worker Pool 并根据请求的剩余可用时间进行优先级排序，从而在保证扩展性的同时显著降低了请求截止期错失率。总体而言，现有服务器无感知计算调度研究主要聚焦于降低启动延迟、优化函数链的数据传输以及应对突发流量的路由。然而，针对 I/O 密集的服务器无感知计算工作负载，现有系统对内存内部结构与实际利用率的建模仍较为粗糙，往往未能充分利用空闲内存来进行中间数据缓存与本地化访问优化。

综上所述，国内外在集群任务调度与工作负载均衡方面已经取得了大量成果：从早期批处理集群的集中式与两级调度，到云原生环境下的容器编排，再到服务器无感知计算场景下面向短生命周期突发任务的在线调度与弹性伸缩，主流研究大多围绕提高 CPU 与内存利用率、提升系统吞吐与降低响应时间展开，并逐步引入多资源公平共享等机制。然而，从服务器无感知计算尤其是 I/O 密集型工作负载的角度来看，现有工作仍存在不足。首先，多数调度与负载均衡策略采用静态或粗粒度的内存模

型，将内存主要视为容量与计费维度，难以及时识别和利用由于过度配置而产生的大量可回收空闲内存。其次，节点本地内存与 I/O 性能之间的耦合关系尚未得到充分利用，难以在集群范围内实现对内存冗余与 I/O 热点的协同优化。针对这些不足，后续研究将重点围绕：如何构建细粒度的内存与 I/O 使用画像，将峰值内存需求、平均占用以及中间数据规模纳入调度决策；如何在集群层面实现 I/O 密集与 CPU 密集函数的协同部署，使内存富余任务为 I/O 密集任务提供更多本地缓存空间。

1.3　文章研究思路

本文的研究思路如图 1.1所示。本文的研究工作首先从深入分析服务器无感知计算环境下的资源利用特征入手。通过对主流商业云平台（如 AWS Lambda 和阿里云函数计算①）的广泛调研与实测，揭示了当前服务器无感知计算模型中普遍存在的内存资源过度配置现象，以及 I/O 密集型任务在传统存储架构下面临的性能瓶颈。基于这一观察，本文提出利用闲置内存资源来加速文件 I/O 的核心思路，旨在将计算节点中未被充分利用的内存转化为高性能的临时存储介质，从而在不增加额外成本的前

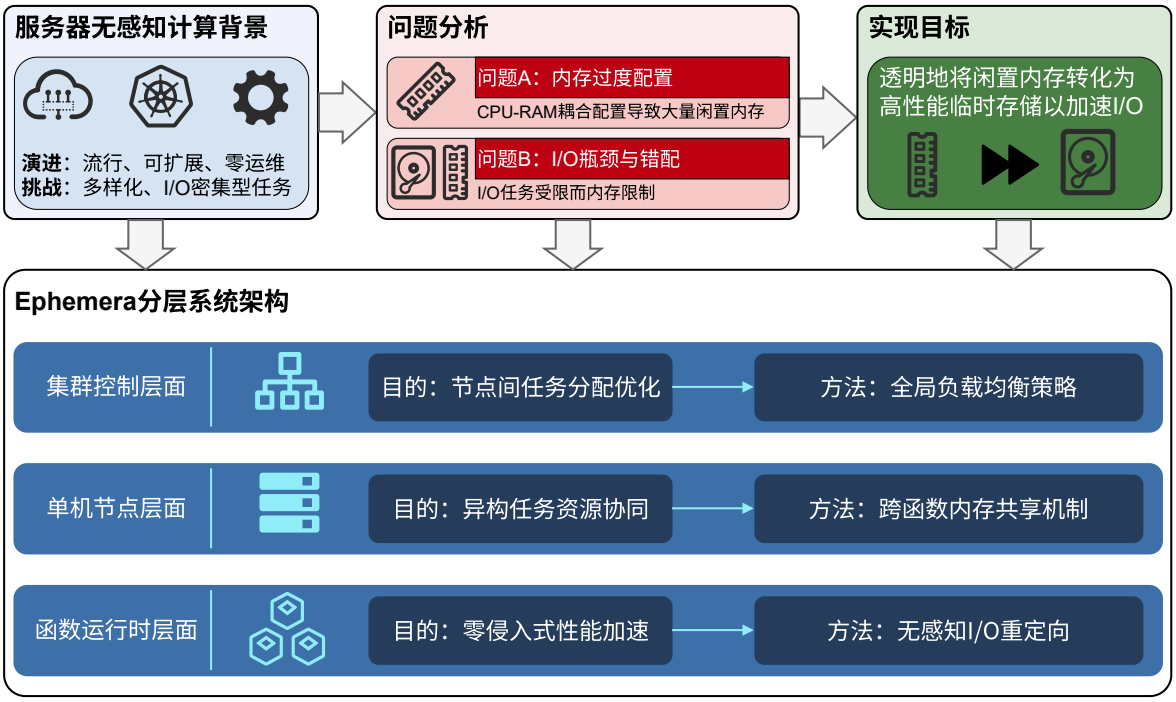


图 1.1　融合动态内存管理的存储优化研究思路图

Figure 1.1　Research Roadmap of Storage Optimization with Dynamic Memory Management

① https://www.aliyun.com/product/fc

提下提升系统整体性能。

在此基础上，本文设计并提出了一种融合动态内存管理与集群负载均衡的服务器无感知计算存储优化架构——Ephemera。该架构采用了分层设计的理念，自底向上分别构建了函数级、节点级和集群级的三层管理机制。在函数级，通过透明拦截技术实现对用户代码无感知的 I/O 重定向；在节点级，设计了跨函数的内存共享机制以支持异构任务间的资源协同；在集群级，引入了全局负载均衡策略以优化节点间的任务分配。这一设计思路旨在解决内存资源隔离、数据安全共享以及大规模集群调度等关键技术挑战。

最后，针对构建的系统原型开展了全面的实验验证。通过选取具有代表性的基准测试集和真实应用场景，从端到端延迟、I/O 吞吐量、内存利用率等多个维度对系统进行了量化评估。同时，将本系统与现有的主流文件系统解决方案进行了对比分析，验证了所提架构在提升 I/O 性能和优化资源利用率方面的有效性与优越性，从而形成从理论分析、系统设计到工程实现与验证的完整研究闭环。

1.4 文章结构

本文共分为六章，各章节的具体内容安排如下：

第一章为绪论。本章首先阐述了服务器无感知计算作为新兴云计算范式的兴起背景及其在处理多样化工作负载时面临的挑战，特别是 I/O 性能瓶颈与资源利用率低下的问题。随后，本章对面向服务器无感知计算的文件系统、分层存储架构、云环境下的资源收割技术以及集群任务调度进行了全面的文献综述，分析了国内外研究现状及现有方案的局限性。最后，概括了本文的研究思路、主要贡献以及论文的组织结构。

第二章介绍了技术背景与研究动机。本章详细描述了服务器无感知计算中的文件系统现状，并通过在 AWS Lambda 和阿里云函数计算平台上的实际测试数据，深入剖析了内存资源过度配置的成因及其普遍性。同时，通过对比实验验证了基于内存的文件操作在延迟和带宽方面的显著优势，从而确立了利用闲置内存加速 I/O 的研究动机，为后续的系统设计提供了坚实的数据支撑。

第三章详细阐述了 Ephemera 系统的设计与架构。本章首先提出了系统的总体设计目标，即透明性、高效性和负载均衡。接着，深入探讨了系统的三个核心组件：驻留在函数实例中的运行时守护进程，负责拦截 I/O 请求并管理本地内存缓存；运行在节点层面的租户管理器，负责同一租户下不同函数实例间的内存共享与安全隔离；

以及位于控制平面的集群控制器，负责全局的节点选择与工作负载编排。本章重点论述了这些组件如何协同工作以实现高效的存储优化。

第四章描述了系统的具体实现细节。本章重点介绍了关键技术模块的工程实现，包括如何利用库打桩技术在用户态透明地拦截文件系统调用，如何基于共享内存机制构建高性能的数据交换通道，以及各组件之间通信协议的具体实现。此外，还介绍了系统原型的开发环境、依赖库及代码结构，展示了从设计理念到可运行系统的转化过程。

第五章展示了系统评估与分析。本章首先介绍了实验环境的搭建、基准测试工具的选择以及评估指标的定义。随后，通过一系列对比实验，详细分析了 Ephemera 在不同文件大小以及不同类型工作负载下的性能表现。实验结果重点展示了系统在降低文件操作延迟、提升 I/O 吞吐量以及提高集群内存利用率方面的实际效果，并深入讨论了性能提升的来源及系统的开销。

第六章为总结与展望。本章对全文的研究工作进行了系统性的总结，回顾了本文提出的核心观点和创新成果。同时，客观分析了当前系统设计与实现中存在的局限性，并据此提出了未来进一步优化和扩展的研究方向。

第 2 章 技术背景与研究动机

本章围绕“在服务器无感知计算平台中利用闲置内存提升 I/O 性能”这一核心思路，依次介绍相关技术背景与关键问题。首先，第2.1节从函数执行环境视角梳理服务器无感知计算中的文件系统形态与临时存储语义，明确 I/O 支持在函数生命周期中的作用。随后，第2.2节通过对 AWS Lambda 与阿里云函数计算的实验结果，量化并分析当前平台中普遍存在的内存过度分配与利用不充分现象。在此基础上，第2.3节进一步讨论如何以 `tmpfs` 与 `mmap` 等机制将内存转化为高性能 I/O 层，为后续系统设计提供可行性依据。最后，第2.4节对本章内容进行总结，凝练本文的研究动机与核心目标。

2.1 服务器无感知计算中的文件系统

服务器无感知计算代表了云原生应用的一种新兴范式。在该范式下，租户使用 Python、Go 或 JavaScript 等高级语言编写封装了特定应用逻辑的函数[6]，并由云平台负责后续的部署与调度。这种范式采用“按需付费”的计费模型，租户仅需为函数执行期间实际消耗的计算时间和资源付费。相较于传统的云计算服务（如 PaaS 或 IaaS），服务器无感知计算所提供的 FaaS 模式具有更高的弹性，能够更灵活地应对负载波动。尽管服务器无感知计算在架构设计上强调“无状态”模型，但在实际生产环境中，数据处理任务往往涉及中间状态的存储与交换。因此，为服务器无感知计算函数提供高效、可靠且符合其生命周期特征的文件系统支持显得尤为关键。

在服务器无感知计算环境中，函数实例通常托管于轻量级的微虚拟机（MicroVM，如 Firecracker[57]）或容器之中。从函数执行环境的视角审视，其文件系统呈现出典型的分层结构特征。底层通常为只读层，包含了用户的代码包、依赖库以及运行时环境。为了确保执行环境的安全性与启动速度，该层在函数生命周期内保持不可变。上层则为可写层，通常被限制在特定的临时目录（如 `/tmp`）下，这是函数实例唯一被允许进行本地写操作的区域。对于开发者而言，这种分层结构是透明的。开发者可以使用标准的 POSIX 兼容接口（如 Python 的 `open()`、`os.path` 或 Node.js 的 `fs` 模块）访问文件系统，而无需依赖特定的 SDK 或 API，从而降低了迁移与开发成本。

文件系统的生命周期与函数实例的生命周期紧密绑定。当函数被触发（冷启动）时，云平台会初始化执行环境并挂载文件系统，此时 `/tmp` 目录通常被映射为宿主机

上的一个临时目录。在执行阶段，函数可从外部存储（如 S3）拉取数据，利用 `/tmp` 创建临时文件以进行解压、预处理或模型加载，并最终将处理结果持久化至外部存储。如果函数实例被复用（温启动），`/tmp` 中的数据可能会被保留，开发者可以利用这一特性进行本地缓存以加速后续调用。然而，一旦实例被销毁（回收），`/tmp` 内的所有数据将随之丢失。因此，这种存储介质被定义为“临时存储（Ephemeral Storage）”。

现有的研究工作[26-29]已致力于服务器无感知计算文件系统的设计与优化，旨在为 I/O 密集型函数提供更好的支持。当前工业界主流解决方案是提供可配置的磁盘空间，这在 AWS Lambda 和阿里云函数计算等商业服务器无感知计算平台以及开源服务器无感知计算平台中都很常见。例如，AWS Lambda 提供了 512 MB 的免费临时存储空间，并支持最高达 10 GB 的扩容配置，用户通过读写 `/tmp` 目录即可处理 I/O 密集型任务；阿里云函数计算亦提供了类似的文件系统支持方案。综上所述，提升文件 I/O 性能对于服务器无感知计算至关重要。高效的 I/O 处理能力不仅能显著提升 I/O 密集型负载的执行效率，还能有效缩短端到端延迟，进而降低服务器无感知计算应用的总体拥有成本。

2.2　服务器无感知计算平台内存利用不充分问题

尽管服务器无感知计算通过“按需付费（Pay-as-you-go）”模式承诺降低运营成本，但在当前的计费机制下，租户实际是为**分配**的资源而非实际**使用**的资源付费。由于主流云平台普遍采用内存与 CPU 算力线性耦合的资源分配策略（即增加内存配置通常意味着获得更多的 CPU 时间片），开发者为了追求更低的执行延迟，往往不得不超额配置内存。这种资源供给与实际需求的不匹配，直接导致了内存资源的严重利用不足，意味着租户正在为大量处于闲置状态的内存付费，这在很大程度上违背了服务器无感知计算成本效益最大化的初衷。

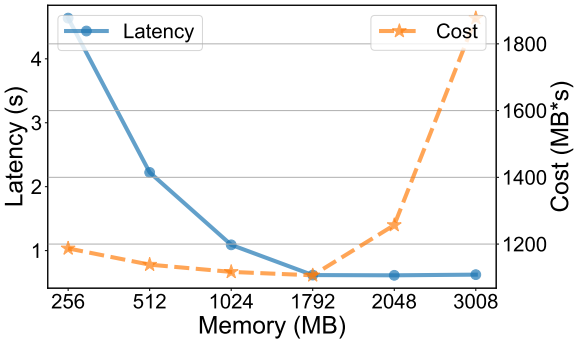
为了量化评估这一现象，本研究从 FunctionBench[58] 基准测试集中选取了三个具有代表性的函数负载，并在 AWS Lambda 和阿里云函数计算平台上，针对多种资源配置方案进行了深入的实验验证。

所选用的函数涵盖了不同的计算特征与资源需求：

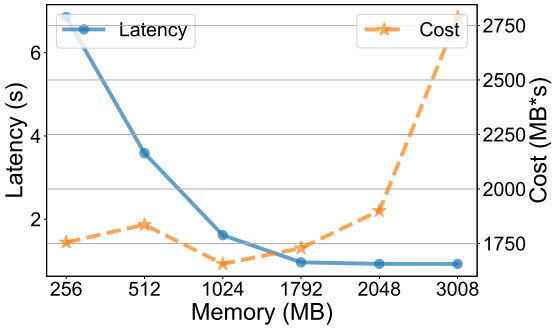
- ***Gzip Compression***：利用 `gzip` 库评估文件压缩性能，属于典型的 I/O 密集型任务；
- ***Pyaes***：基于 `pyaes` 库测量 AES 加密与解密性能，属于 CPU 密集型任务；
- ***Linpack***：使用 `numpy` 求解线性方程组，同样属于 CPU 密集型任务。

经测定，上述三个函数在正常执行过程中所需的最小内存峰值分别为 77 MB、40 MB 和 93 MB。这些基准数据将作为后续分析资源分配是否存在冗余的重要参考依据。

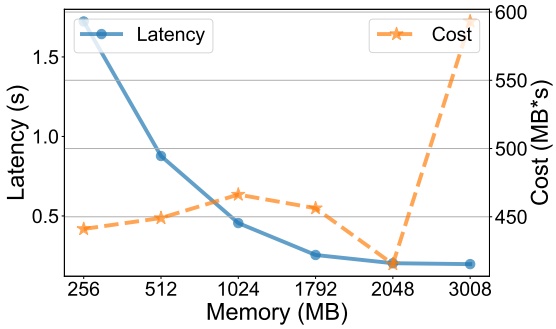
本文首先在 AWS Lambda 上进行测试。AWS Lambda 提供以内存为中心的资源配置，允许开发人员设置内存配额，vCPU 按内存分配的比例进行分配。本文对 256 MB 到 3008 MB 的内存配置进行了采样，其中在内存设置为 1792 MB 时可以获得一个完整的 vCPU。图 2.1 展示了 AWS Lambda 上不同资源配置下这三个应用程序的执行时间和成本（成本按照执行时间与内存大小的乘积计费模型计算）。从执行时间的角度来看，由于 AWS Lambda 上的 CPU 能力与分配的内存成正比，执行时间整体上随内存增加而减小，但在达到 1792 MB 之后，进一步增大内存带来的性能收益明显下降。从成本角度来看，这三个应用程序分别在 1792 MB、1024 MB 和 2048 MB 配



(a) Gzip 压缩
(a) Gzip Compression

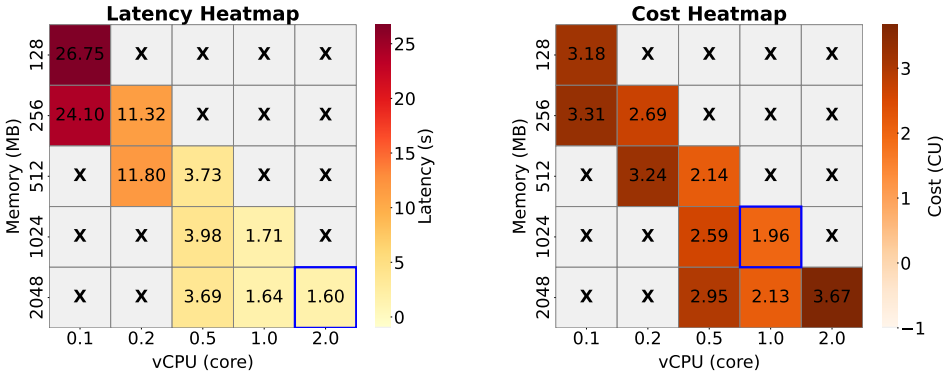


(b) Pyaes 加密
(b) Pyaes

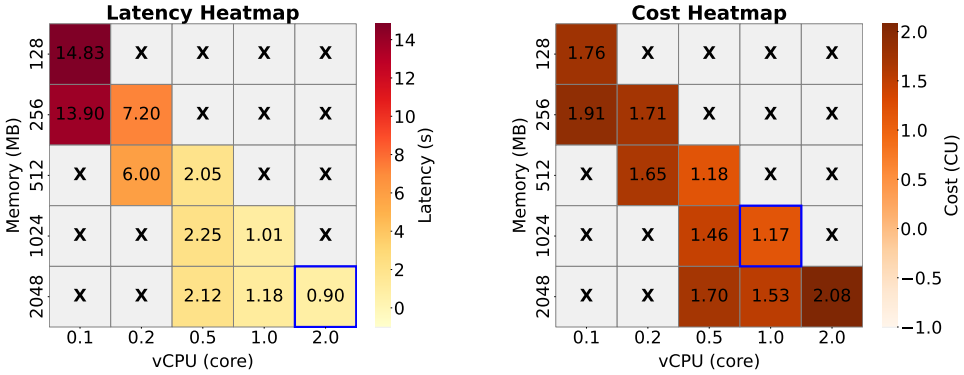


(c) Linpack 基准
(c) Linpack

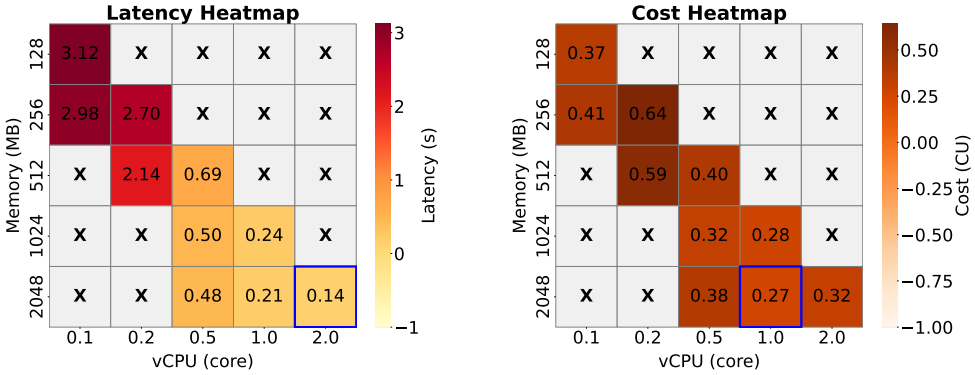
图 2.1 AWS Lambda 测试图
Figure 2.1 Evaluation on AWS Lambda



(a) Gzip 压缩
(a) Gzip Compression



(b) Pyaes 加密
(b) Pyaes



(c) Linpack 基准
(c) Linpack

图 2.2　阿里云函数计算测试图
Figure 2.2　Evaluation on Aliyun Function Compute

置下成本最低。综合延迟与成本两个维度可以发现，无论用户的优化目标是延迟最优还是成本最优，平台给函数分配的内存都显著高于应用程序运行所需的最小内存，从而导致大量内存长期处于闲置状态，内存资源利用率不足。

接着，本文在阿里云函数计算上进行测试。与 AWS Lambda 不同，阿里云函数计算解耦了内存容量和 CPU 的分配，但是 vCPU 大小与内存大小（GB）的比例必须在 1:1 到 1:4 之间。本文对 128 MB 到 2048 MB 的内存配置和 0.1 到 2 的 vCPU 配置进行了采样。图 2.2 展示了阿里云函数计算上不同资源配置下这三个应用程序的执行时间和成本。由于云实例中受限的 vCPU 与内存比例约束，部分配置不可用，并在图中用 x 表示。产生最小延迟和最低成本的配置用蓝色框标出。从成本角度来看，这三个应用程序分别在 1024MB（1.0 vCPU）、1024 MB（1.0 vCPU）和 2048 MB（1.0 vCPU）配置下成本最低，内存配比均显著高于各个应用程序的最小内存峰值。因此，即便在具备更细粒度 vCPU 配置能力的情况下，用户在追求延迟或成本最优时，最终选择的配置仍然会导致函数获得远高于实际需求的内存容量，存在明显的内存过度配置。同时，云提供商为同一 vCPU 配置提供多种内存选项的做法，在缺乏精确内存需求预测的前提下，往往会促使租户采用“向上取整”或“以性能冗余换取心智负担减轻”的策略，从而进一步放大内存浪费。

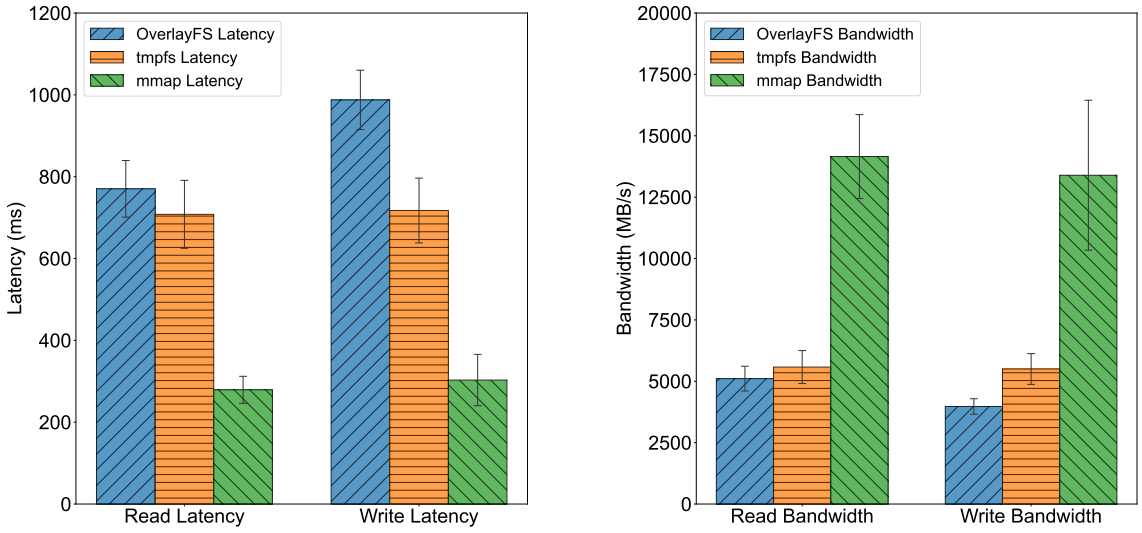
从根本原因上看，当前主流服务器无感知计算平台在资源分配与计费模型上仍以内存为核心维度，将内存配额既作为性能上限的控制手段，又作为计费的主要基础。一方面，平台缺乏对函数真实运行时内存行为（例如峰值占用、平均占用及其随请求规模变化）的精细建模与反馈机制，用户难以依据历史运行数据对资源进行精准配置，只能依靠经验或保守估计预留充足的安全边界；另一方面，平台在调度与负载均衡阶段通常只将内存视为容量约束，而很少考虑内存使用的时间特性以及不同函数之间的互补性，导致节点级别存在大量无法被复用的瞬时空闲内存。

这些现象共同表明，服务器无感知计算平台上存在系统性内存利用不充分的问题。大量被过度分配但未被实际使用的内存，既不能有效提升函数性能，也未被平台重新组织起来用于改善其他函数的运行环境，例如缓解远程存储访问带来的 I/O 瓶颈或为短时中间数据提供低延迟存储空间。因此，如何在不改变现有计费与编程模型的前提下，更准确地刻画函数的内存需求、识别和收集节点上的冗余内存，并将其转化为可被其他任务利用的有效资源，是服务器无感知计算平台进一步提升资源利用率和性能的关键研究动机之一。

2.3　内存加速 I/O

为了评估内存对文件操作性能的影响，本文评估了 mmap、tmpfs 和 OverlayFS[①] 对随机磁盘 I/O 性能的影响。OverlayFS 用作基准，它重叠多个目录层以形成统一的文件系统视图。与其他传统文件系统一样，OverlayFS 利用内核的页面缓存机制在内存中缓冲文件数据，通过减少磁盘 I/O 操作来提高读/写性能。mmap 技术将文件内容映射到内存地址空间，允许应用程序像直接进行内存操作一样访问文件数据。另一方面，tmpfs 作为一种内存文件系统，直接将文件数据存储在内存中。

实验使用 Docker 容器作为实验环境，评估随机磁盘 I/O 操作的性能，包括延迟和带宽。结果如图 2.3所示。由于内存文件系统的特性，tmpfs 在随机 I/O 操作期间表现出低延迟和高带宽的优势。与 OverlayFS 相比，tmpfs 在读取性能上提高了 8%，在写入性能上显著提高了 28%。与传统文件操作不同，mmap 利用内存映射将文件内容直接映射到用户空间，允许应用程序读写数据而无需在用户空间和内核空间之间进行复制。因此，mmap 减少了内核和用户空间之间数据移动的开销，显著提高了 I/O 操作的性能。与 OverlayFS 相比，mmap 在读取性能上提高了 63%，在写入性能上提高了 69%。因此，虽然传统文件系统已经受益于页面缓存，但通过基于内存的解决方



(a) 延迟
(a) Latency

(b) 带宽
(b) Bandwidth

图 2.3　基于内存的文件系统对比图
Figure 2.3　Comparison of Memory-Based File Systems

① https://docs.kernel.org/filesystems/overlayfs.html

案更积极地利用内存可以进一步提高文件系统性能。无论是通过内存文件系统直接存储文件数据，还是通过内存映射技术间接利用内存资源，都可以有效降低文件操作延迟并提高数据处理速度。

基于上述实验结果，特别是图 2.3中 `tmpfs` 与 `mmap` 相较于 OverlayFS 的显著性能优势，本文发现了一个关键的系统优化机遇：利用服务器无感知计算平台中普遍存在的闲置内存来构建高性能的 I/O 层。从数据访问特征来看，服务器无感知计算函数在执行过程中产生的大量中间结果、临时文件或解压数据集，通常具有生命周期短、访问频繁且容量中等的特点。相比于通过网络访问远程对象存储或读写本地持久化磁盘，将这类数据通过内存文件系统或内存映射技术驻留在被过度分配的内存中，可以在不改变代码语义的前提下，获得极高的 I/O 带宽与低延迟收益。这意味着，如果云平台能够在资源管理层面进行架构创新，将原本处于闲置状态的多余内存显式地转化为内存文件系统或高速缓冲区，则可以将被动的资源浪费转变为主动的平台级 I/O 加速。因此，如何在不破坏现有编程接口与计费模型的基础上，透明地利用这些冗余内存资源来突破 I/O 瓶颈，是本论文后续设计与实现的核心研究目标。

2.4 本章小结

本章首先介绍了服务器无感知计算环境下的分层文件系统架构及其与函数生命周期的高度耦合机制，重点阐述了临时存储在中间数据处理中的关键作用。在此基础上，分析了当前主流平台的 I/O 支持方案，指出提升文件 I/O 性能是优化函数执行效率与降低应用成本的核心要素。随后，通过在 AWS Lambda 和阿里云函数计算平台上的实际测试，量化了服务器无感知计算函数执行期间普遍存在的内存资源过度配置现象。实验数据表明，无论是以内存为中心的资源分配策略还是解耦的资源分配策略，都难以避免内存资源的闲置浪费。最后，本章通过对比实验验证了基于内存的文件操作在降低延迟和提升带宽方面的显著优势。这些发现共同构成了本文的研究动机：即利用服务器无感知计算节点中已分配但未使用的闲置内存资源，构建高效的内存文件系统，以解决 I/O 密集型任务的性能瓶颈问题。

第 3 章 Ephemera 系统设计

基于上一章对服务器无感知计算平台中“内存过度分配导致闲置”现象及“利用内存加速 I/O”潜力的分析，本章给出 Ephemera 的系统设计。该设计在不改变函数编程与计费模型的前提下，将节点上可收割的冗余内存组织为临时存储加速层，并通过容器、节点与集群三个层面的机制协同落地。首先，第3.1节给出系统架构与端到端工作流程，明确集群级控制器、节点级租户管理器与函数级守护进程三类组件的分工与协同关系。随后，第3.2节深入介绍容器层面的运行时守护进程设计，说明其如何通过系统调用拦截、内存文件系统与内存适配机制在函数侧透明加速 I/O。接着，第3.3节阐述节点层面的租户管理器如何在租户隔离边界内进行内存收割与共享，并通过容器池与保障机制维持稳定性。进一步地，第3.4节介绍集群层面的控制器如何结合特征分析与在线调度在全局范围内平衡内存供需。最后，第3.5节对本章设计要点进行归纳，为后续实现与评估章节建立统一的系统视角。

3.1 系统架构及工作流程概述

本章首先对系统的整体设计思路进行概述，并介绍各关键组件之间的协同方式。Ephemera 的核心目标是在不改变函数编程模型的前提下，充分利用每个函数运行时已经分配但在大部分时间处于闲置状态的内存空间，从而提升服务器无感知计算平台中临时存储访问的效率，降低 I/O 密集型工作负载的响应延迟。

如图 3.1所示，Ephemera 自上而下由三个层次的组件构成，分别为集群级控制器、节点级租户管理器和函数级守护进程。这三类组件分别负责全局调度决策、节点内资源协调以及单个函数实例的本地 I/O 加速，共同构成了一个跨层次的内存收割与利用框架。

集群级控制器常驻于服务器无感知计算平台的控制平面，负责处理函数的部署请求和调用请求，是系统的全局决策中枢。在函数部署阶段，控制器内部集成的性能分析模块会对新函数进行试运行，分析其执行特征，包括内存使用量以及文件输入输出行为等，并将分析结果持久化保存，以便在后续调度决策中使用。在函数实际被调用时，控制器内部的调度模块根据各工作节点当前的负载状况以及先前记录的函数特征，将调用请求分配到合适的节点上，在尽量满足服务质量要求的同时实现跨节点的负载均衡和资源利用最大化。

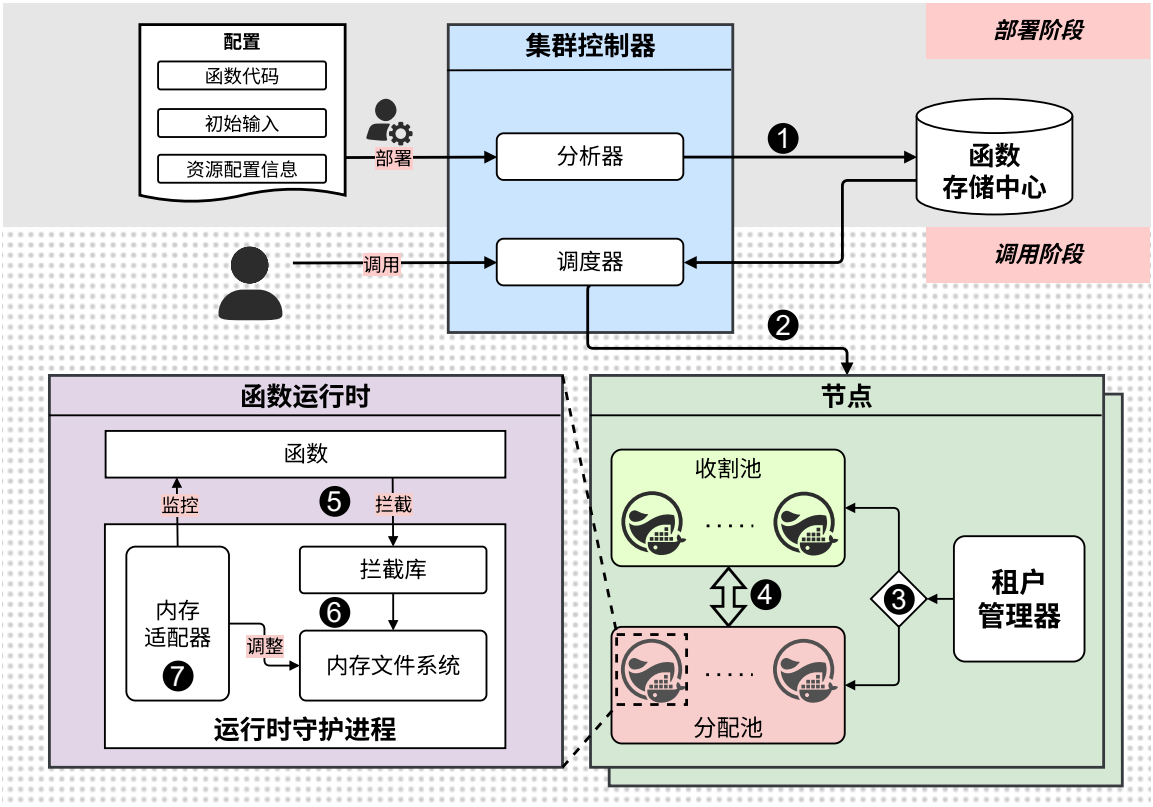


图 3.1　Ephemera 的整体架构与工作流程图
Figure 3.1　Overall Architecture and Workflow of Ephemera

节点级租户管理器部署在每个工作节点上，并以租户为粒度进行划分，用于实现和控制同一租户内部的内存共享机制。节点及租户管理器在每个节点上维护一个内存资源池，并根据函数实例的运行情况，将节点上的执行实例划分为两类：一类是处于资源相对富余状态、可以贡献多余内存的“收割池”实例，另一类是资源紧张、需要额外内存支持的“分配池”实例。通过在同一租户范围内进行这样的分组和协调，租户管理器可以在不打破租户隔离边界的前提下，在多个实例之间动态转移内存资源，从而提高单节点内存资源的整体利用效率。

函数级守护进程随每个函数运行时一同部署，直接嵌入函数执行环境中。该守护进程负责在本地运行时内部构建和维护一个基于内存的文件系统，并根据函数执行过程中的实际资源使用情况，动态调整该文件系统的大小。函数级守护进程通过监测运行时内存占用变化，识别出当前函数在保证正常执行所需内存之外的可用空闲内存，将其组织为可供文件 I/O 使用的高速存储空间，以此在本地提供一个面向临时数据和中间结果的高性能存储层。

图 3.1同时展示了 Ephemera 的整体工作流程，主要可分为函数部署阶段和函数

执行阶段两部分。

在函数部署阶段，当控制平面接收到租户上传的函数代码或镜像，以及与之对应的资源配置（包括处理器和内存的配额限制）与输入数据时，集群级控制器中的性能分析模块会基于这些输入数据对函数进行一次或多次试运行。在试运行过程中，该模块持续记录函数的关键执行特征，例如内存使用高峰、文件访问大小等，并将这些分析结果作为元数据存储在控制平面，为后续函数调度和资源分配提供依据。

在函数执行阶段，当有新的调用请求到达时，集群级控制器中的调度模块会被触发。调度模块综合考虑当前各工作节点的负载水平（例如正在运行的实例数量、内存占用状况等）以及该函数在部署阶段得到的执行特征分析结果，将本次调用请求分配到最合适的节点上，以尽量平衡不同节点之间的内存压力和 I/O 压力。当某一节点被选中并启动对应的函数执行实例后，该节点上的租户管理器立即接管该实例，并根据其实时内存使用情况与租户整体的资源状况，将实例归入收割池或分配池。通过对实例进行这样的分组管理，租户管理器可以在同一租户内部动态协调各实例之间的内存分配，使多余的内存资源能够被需要的实例及时利用。

在每个具体的函数运行时环境中，函数级守护进程负责处理与文件访问相关的具体操作，并协调内存的本地使用。守护进程内部集成了一个系统调用拦截组件，用于在函数运行期间截获常见的文件相关系统调用，将原本指向底层文件系统的访问请求转向本地的内存文件系统。随后，这些被截获的文件访问请求由守护进程管理的内存文件系统进行处理，针对函数产生的中间数据和临时文件提供低延迟的本地读写服务。同时，守护进程中的内存适配组件持续监测函数执行过程中的内存利用率变化，在保证函数本身稳定运行的前提下，按照系统策略动态调整内存文件系统可用的空间大小，使得内存资源在不同时间段内能够在计算需求和 I/O 加速需求之间进行灵活权衡与分配。

3.2 容器层面运行时守护进程

在函数即服务平台中，平台通常为每一次函数调用请求分配一个独立的函数运行时环境。本论文在每个函数运行时内部嵌入一个守护进程，用于在函数执行期间统一管理文件输入输出请求，并协调运行时内存资源的使用情况。

如图 3.2所示，运行时守护进程的整体架构由三个主要模块构成：用于拦截文件系统相关操作的拦截库、用于在本地内存中提供文件语义的内存文件系统，以及用于监控内存使用并动态调整内存文件系统规模的内存适配器。其中，拦截库负责对接函

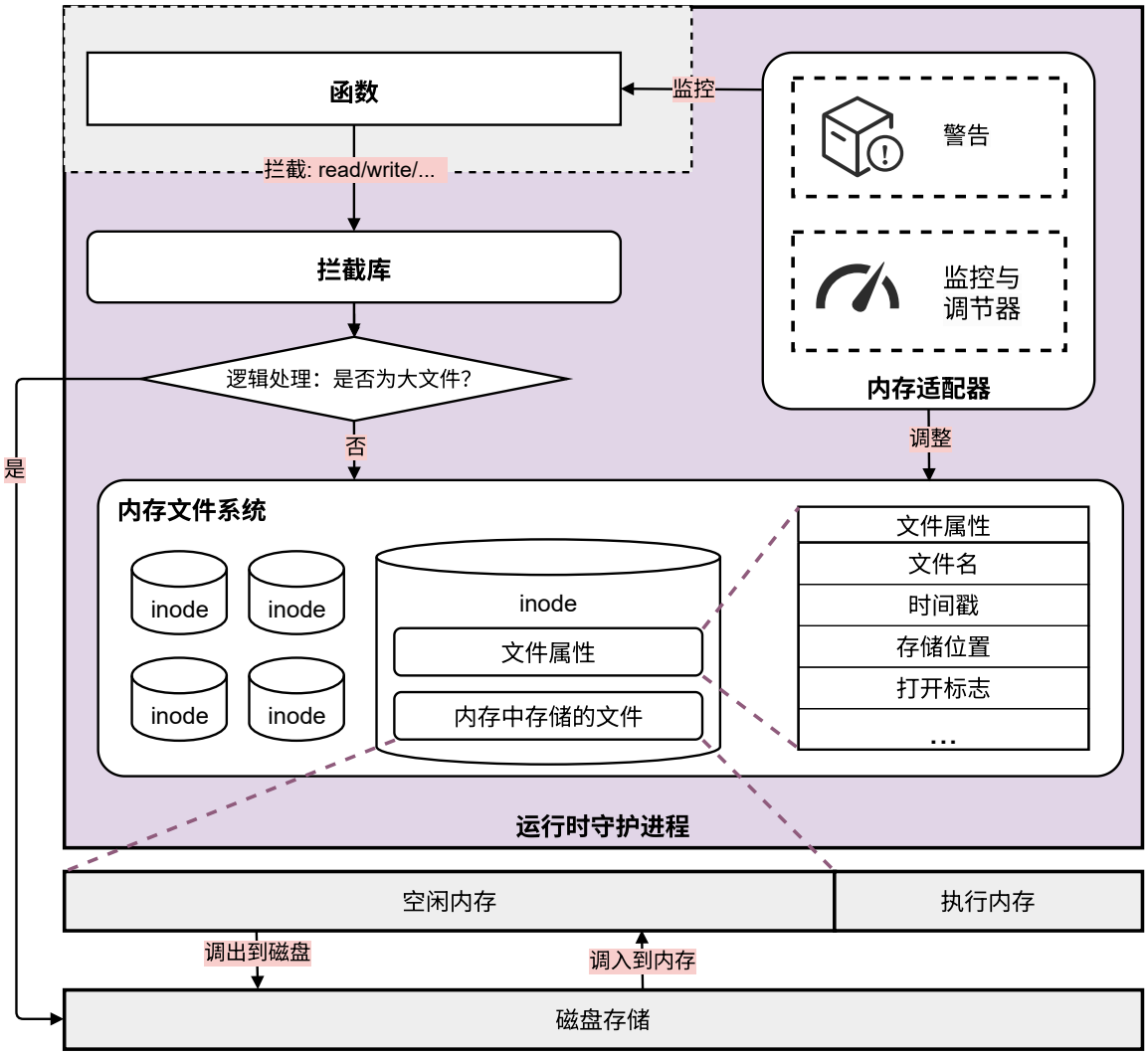


图 3.2　运行时守护进程概览图
Figure 3.2　Runtime Daemon Overview

数代码发出的文件访问调用，内存文件系统负责以文件的方式组织和管理驻留在内存中的数据，而内存适配器则负责在函数执行内存需求与 I/O 加速内存需求之间进行动态权衡。

3.2.1　拦截库

拦截库的作用是识别并拦截与文件系统相关的操作调用，将原本直接提交给底层操作系统的文件访问请求，转换为可由运行时守护进程处理的内部请求。当服务器无感知计算函数首次在容器中执行时，运行时环境会完成与拦截库的注册，使得后续在该运行时内产生的相关系统调用能够被统一拦截和重定向。

表 3.1 被拦截的文件系统相关接口
Table 3.1 Intercepted File-System-Related APIs

接口名称	基本作用
open	打开一个已存在的文件，或在需要时创建新文件，并返回相应的文件描述符，用于后续读写操作。
close	关闭给定的文件描述符，释放与该描述符相关联的系统资源。
read	从文件描述符当前偏移位置开始读取指定字节数的数据，并将读取结果写入用户提供的缓冲区。
write	将用户缓冲区中的指定字节数数据写入到文件描述符所对应的文件中，并相应更新文件偏移量。
lseek	调整文件描述符当前的读写偏移量，用于支持随机访问、重新定位读写位置或在文件末尾进行追加写入。
mkdir	在给定路径位置创建新的目录项，并在文件系统中建立相应的目录结构。

在实际工作过程中，拦截库首先根据系统调用号和参数类型对收到的调用进行分类判断，确认该调用是否属于文件相关操作。对于被识别为文件访问的调用，拦截库会解析出其关键参数（例如路径、访问模式、偏移量和数据缓冲区等），并按照统一格式转交给内存文件系统进行进一步处理。表 3.1列出了当前被拦截的主要文件系统相关接口。

通过对上述关键接口的统一拦截与转发，运行时守护进程得以在不改变函数源代码和编程接口的前提下，接管函数对临时文件和中间数据的访问路径，从而为后续的内存加速与资源管理提供基础。

3.2.2 内存文件系统

为充分利用函数实例中原本处于闲置状态的内存资源，本论文在每个函数运行时内部设计并维护了一个轻量级的内存文件系统。该内存文件系统为函数提供与常规文件系统兼容的接口，同时将数据优先存放在本地内存中，用于加速临时文件和中间结果的访问。

在整体设计上，内存文件系统首先记录每个容器的内存限制。在保证函数自身正常执行所需内存的前提下，系统将剩余可用内存划分出一部分作为文件数据存储空间，以此构建一个位于函数运行时内部的高速存储层。当拦截库将文件访问请求转交给内存文件系统时，后者根据文件的路径和当前存储状态决定是直接在内存中完成操作，还是退回到底层持久化存储执行。

类似于传统文件系统，内存文件系统为每个文件或目录维护一个索引节点结构，用于记录相关的元数据信息。索引节点中保存了文件名、访问标志、当前读写偏移量、并发访问控制信息、最近访问时间戳、文件大小，以及文件当前所处的存储位置（在内存中或在磁盘上）等内容。通过索引节点中记录的访问标志，内存文件系统可以约束不同实例或不同线程对同一文件的读写权限，防止出现非法覆盖或访问冲突。

当函数通过打开接口访问某个文件时，内存文件系统会根据当前内存使用情况和文件大小，决定是否将该文件的数据载入内存存储区域。对于体积较小、访问频繁或延迟敏感的文件，更倾向于将其内容缓存于内存文件系统中，以减少对底层磁盘或远程存储的访问。对于体积较大或访问不频繁的文件，则可以只在内存中维护其元数据，将数据本体仍存放在磁盘上，通过原生文件系统接口进行访问，以避免占用过多内存空间。

根据索引节点中记录的存储位置信息，内存文件系统的核心逻辑会选择不同的处理路径。当文件被标记为驻留在内存中时，读写请求直接作用于对应的内存数据结构，并在完成后同步更新索引节点中记录的偏移量、文件大小和时间戳等信息。当文件被标记为驻留在磁盘上时，系统则将相应操作交回底层文件系统处理，同时保持索引节点中的元数据与底层状态一致。最终，内存文件系统按照统一的返回格式将处理结果返回给函数运行时，使得对上层应用而言，底层实现细节完全透明。

3.2.3 内存适配器

由于函数即服务平台对每个函数运行时均施加了严格的内存限制，如果为内存文件系统分配过多内存，可能会挤占函数代码和数据本身的运行空间，甚至引发内存不足故障。为此，本论文在运行时守护进程中设计了内存适配器模块，用于在函数执行过程中动态调整内存分配策略，并与节点上的租户管理器协同完成内存扩缩。

内存适配器主要承担三方面职责。第一，持续监控函数本身以及内存文件系统的总体内存使用情况，捕获内存占用的变化趋势和突发行为。第二，当监测到可用内存接近阈值或出现明显紧张时，内存适配器会与所在节点的租户管理器进行通信，请求额外的可共享内存，或在必要时归还一部分此前获得的内存配额，以缓解节点整体的内存压力。第三，当实例可用的内存上限被重新确定后，内存适配器需要据此触发文件数据在内存与磁盘之间的迁移与替换，保证在新的内存配额下，内存文件系统仍能维持基本的读写服务能力。

为防止引入内存文件系统后导致系统整体发生内存溢出，内存适配器设计了一种基于访问状态的文件数据置换机制。在该机制下，内存适配器通过查询索引节点

中记录的访问时间戳与文件状态，评估各文件的近期访问活跃度。当可用内存不足时，优先选择长时间未被访问、且处于关闭状态的文件进行回收，将其数据从内存文件系统中移除，并根据需要同步到磁盘存储中，从而为新的内存文件分配腾出空间。通过将已关闭文件的时间戳设置为较低值，可以自然实现对这类文件的优先淘汰。

需要注意的是，当文件正在被函数代码读写时，如何在尽可能不引入额外锁开销的前提下保证应用层操作与底层内存管理的一致性，是内存适配器设计中的关键问题之一。如果在文件活跃访问期间频繁地对其数据进行主动迁移和回收，容易导致应用视角下的数据与实际存储状态不一致，进而引入复杂的同步和恢复逻辑。为此，本论文在设计阶段对内存调度的触发时机与粒度进行了严格约束：适配器避免对当前活跃文件进行频繁的主动置换，而是优先选择空闲状态的文件进行淘汰。仅在确需对热点数据进行迁移时，才通过互斥锁机制保障操作的原子性与安全性。

3.3 节点层面的租户管理器

本节介绍节点层面的租户管理器的整体设计与工作机制。在多租户的服务器无感知计算平台中，同一物理节点上通常同时驻留来自多个租户的大量函数实例。如何在保证租户间隔离与安全的前提下，提高节点内内存资源的利用率，是系统设计中的关键问题之一。

为此，本论文在每个物理节点上，为每个租户单独部署一个租户管理器，用于统一管理该租户在该节点上的所有函数容器以及与之相关的内存配额。租户管理器在容器之间构建起租户内的内存共享与调节机制，通过对容器内存使用模式的分类管理，在不跨越租户边界的前提下实现节点内的内存共享，缓解局部内存紧张，同时降低整体资源浪费。

图 3.3展示了租户管理器在节点侧的总体结构。在每个节点中，针对每一位租户，系统均实例化一个独立的租户管理器。在该管理器内部，本论文设计了一种面向租户级别的内存共享机制。为了支持高效的共享决策，管理器引入了容器池的概念，根据运行中函数实例在内存与文件访问上的使用模式对其进行分类管理，从而在不同容器之间实现更有针对性的内存共享与调度。此外，为避免过度共享导致的内存紧张甚至内存不足故障，租户管理器内部还配套实现了一套保障机制，用于在极端情况下主动回收和重整内存配额，维持系统的稳定运行。

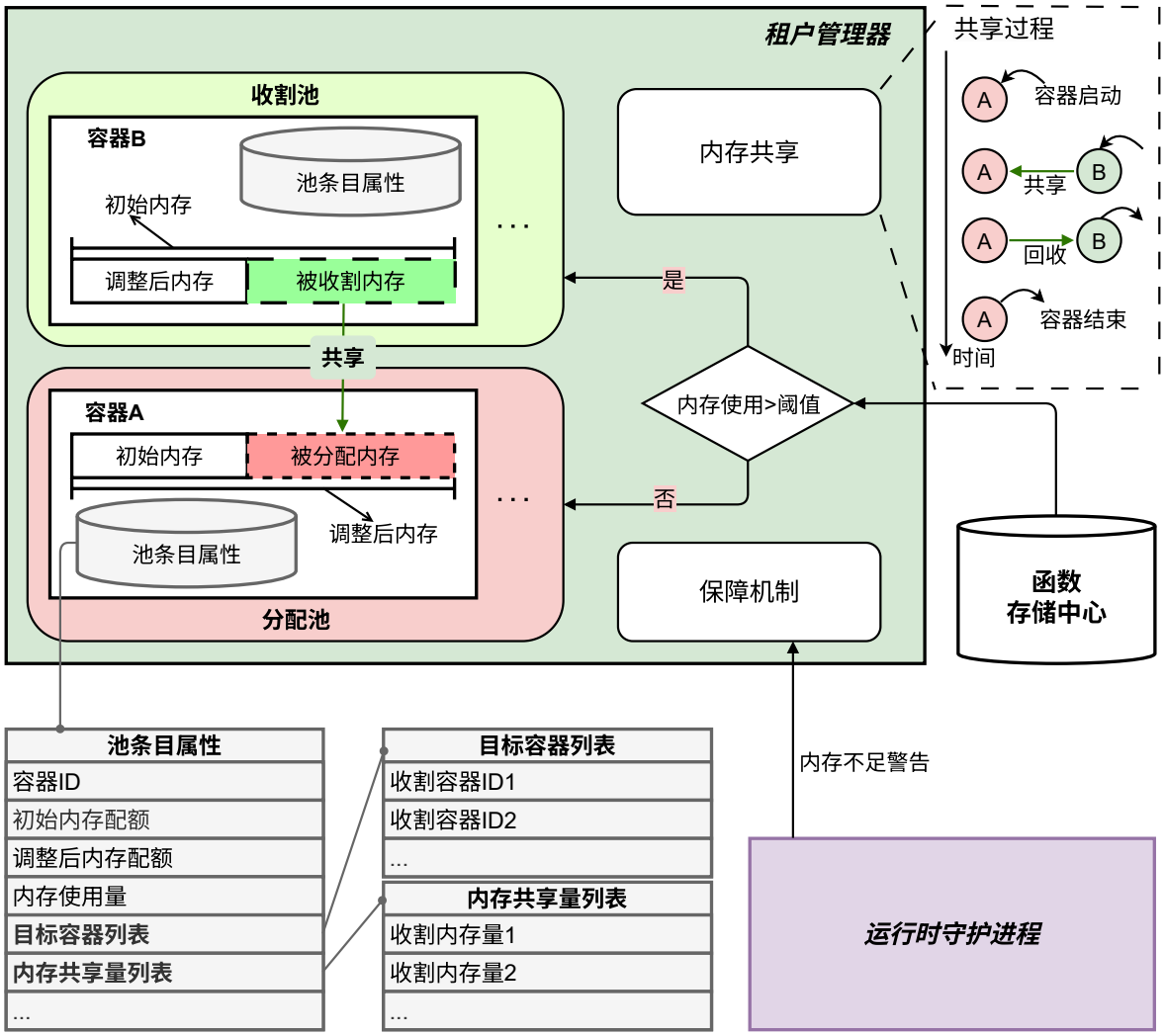


图 3.3　租户管理器概览图
Figure 3.3　Tenant Manager Overview

3.3.1　内存共享机制

在设计内存复用与共享机制时，一个核心问题是应当选择何种粒度进行内存共享。从细粒度到粗粒度，现有研究已经探索了多种不同的共享单元，例如以单个函数为单位的共享，或以内含若干函数的单个工作流为单位的共享。

单函数级别的内存共享机制[59]允许同一函数的多个实例之间共享内存，这种机制在一定场景下可以降低单函数内的重复内存预留。然而，由于同一函数的多个实例通常具有相似的内存需求和相近的执行行为，它们在时间上往往同时出现内存富余或同时出现内存吃紧的情况，能够彼此互助的机会有限，导致整体的共享效率并不理想。

单工作流级别的共享机制[60-61]则将共享范围扩展到由多个函数组成的工作流之内，允许同一工作流中不同函数实例之间共享内存资源。这里的工作流通常由租户通过平台提供的编排服务显式定义，用于描述函数的调用顺序或依赖关系图，以实现复杂应用任务的自动化执行。这类机制可以一定程度上利用不同函数之间在时间上的错峰行为，但由于工作流内部往往存在严格的执行顺序约束，许多函数实例在时间上并不能并行执行，因而在实际运行时，能够同时参与共享的实例集合依然有限，对工作流之外函数的内存利用也难以覆盖。

与上述较细粒度的共享方式相比，本论文采用了以单个租户为整体单位的内存共享机制。具体来说，本论文允许同一租户名下在某个节点上的全部函数实例，在遵循安全与隔离约束的前提下，共同参与该租户配额范围内的内存共享与调节。在服务器无感知计算平台中，租户通常对应一个用户账户或组织账户，该账户在平台上部署并拥有一组函数，这些函数从逻辑上都归属于该租户。以租户为单位进行共享，一方面扩大了可参与共享的对象集合，使得不同函数、不同应用之间可以在时间上互补利用内存资源；另一方面也更符合平台在计费和资源配额上的边界划分，有利于在共享的同时保持租户间的资源隔离。

在一个物理节点上，系统会为每个租户分配一个独立的租户管理器实例。每个租户管理器仅负责本租户在该节点上的函数容器和内存资源。这种设计天然地防止了跨租户的直接共享，从而避免了因数据访问边界不清而带来的潜在安全隐患与策略冲突，也使得不同租户之间的资源分配与治理策略可以彼此独立地演进。

在租户内部，内存共享遵循一定的优先级策略。当需要收集冗余内存时，租户管理器优先从当前节点上内存富余程度较高的容器中收割可用内存。这样可以减少在大量容器之间频繁移动小块内存带来的开销，使共享决策更加集中、高效。相反，当系统需要为某些容器额外分配内存时，则优先满足那些额外需求量较小的容器请求，使更多容器能够以较为充足的内存配额顺利完成执行，从整体上提升租户内任务的完成率与体验。

通过上述原则，租户管理器在一个租户内部构建出动态的资源池化效果：一部分在当前时刻对内存需求较低或处于空闲状态的容器，向节点内资源紧张的容器暂时让出部分内存；而随着任务进展和容器生命周期的变化，这些共享关系又可以被重新评估和调整，从而在整个执行周期内更好地平衡节点资源使用。

3.3.2 容器池

容器池是租户管理器内部进行内存分配与共享决策的核心数据结构，用于对同一租户在单个节点上的所有容器进行统一记录和动态分类管理。容器池的主要职责，一方面是追踪各容器的内存使用情况与访问模式，另一方面是根据这些观测结果将容器划分到不同的功能子池中，以此驱动后续的共享与调度操作。

容器池根据容器在执行过程中的文件访问特征和内存使用情况，将其划分为三类：当某个容器在内存文件系统中产生的文件数据体量较大、且其整体内存使用接近甚至超过原有内存限制的阈值时，该容器会被归入需要额外内存支持的分配子池；当容器在执行期间的文件规模和内存占用均明显低于其配置的内存上限时，该容器被划分至可被收割的子池，其未被使用的富余内存可在后续共享过程中被调拨给其他容器使用；其余既不明显紧张、也不明显富余的容器，则被视为在当前阶段可以自给自足的普通容器，不参与内存资源的主动共享。

在实现上，容器池中为每个容器维护一个条目，该条目记录了与该容器相关的关键属性以及与其他容器之间内存共享关系的必要信息。对于某个容器 C，其在容器池中的条目可抽象表示为如下属性元组：

$$\mathrm{Entry}(C) = \left(C_{ID}, T_{lim}, M_{\text{peak-native}}, F_{\text{max-size-runtime}}, T_{\text{current-lim}}, L_{\text{peers}}, L_{\text{deltas}}\right) \tag{3.1}$$

其中：

- C_{ID}：容器的唯一标识符，用于在租户管理器内部区分不同容器。
- T_{lim}：调度器最初为该容器分配的内存上限，是该容器在未参与任何内存共享之前的基准配额。
- $M_{\text{peak-native}}$：在该容器生命周期内，函数本身（不包括内存文件系统）的原生内存使用峰值。该指标反映了函数业务逻辑对内存的固有需求上界。
- $F_{\text{max-size-runtime}}$：该容器内的文件系统在运行期间曾处理过的最大单个文件大小，反映了 I/O 加速带来的额外内存压力上限。
- $T_{\text{current-lim}}$：容器当前生效的内存限制，可由租户管理器结合共享决策在运行时动态调整；初始化时满足 $T_{\text{current-lim}} = T_{lim}$。
- $L_{\text{peers}} = [P_1, P_2, \ldots, P_k]$：与容器 C 存在内存共享关系的 k 个对等容器标识符的有序列表。
- $L_{\text{deltas}} = [\Delta M_1, \Delta M_2, \ldots, \Delta M_k]$：与 L_{peers} 一一对应的内存数量列表，其中 ΔM_i 表示与对等容器 P_i 之间涉及的共享内存量。

根据容器 C 所属子池的不同，L_{peers} 和 L_{deltas} 的含义也有所差异：

- 当容器 C 被归类到可被收割的子池时，L_{peers} 列出了从该容器获得内存的所有对等容器。对每个 P_i，L_{deltas} 中对应的 ΔM_i 表示从 C 中被收割并分配给 P_i 的内存量。此时，容器 C 的当前内存上限由初始配额减去被收割总量计算得到：

$$T_{\text{current-lim}} = T_{lim} - \sum_{i=1}^{k} \Delta M_i \tag{3.2}$$

- 当容器 C 被归类到需要额外支持的分配子池时，L_{peers} 列出了向该容器提供内存的所有对等容器。对每个 P_i，L_{deltas} 中对应的 ΔM_i 表示从 P_i 收到并分配给 C 的内存量。在这种情况下，容器 C 的当前内存上限由初始配额加上额外获得的总量计算得到：

$$T_{\text{current-lim}} = T_{lim} + \sum_{i=1}^{k} \Delta M_i \tag{3.3}$$

当 $k = 0$ 时，说明该容器当前没有与其他容器发生内存共享关系，其内存限制保持为初始值，即 $T_{\text{current-lim}} = T_{lim}$。

在函数请求到达并触发某一容器实例启动时，租户管理器会首先查询该容器在容器池中的条目及其所属子池，以决定相应的内存处理策略。如果该容器属于可被收割的子池，则它在后续运行过程中更有可能成为其他容器的内存供给方；如果该容器属于需要额外支持的分配子池，则租户管理器会尝试为其从其他容器处获取一定额度的额外内存。对于处于自给自足状态的容器，则按照常规方式启动，无需额外的共享干预。

在函数执行结束或生命周期发生重要变化之后，租户管理器会重新评估相关容器的状态，并据此更新容器池条目。对于分配子池中的容器，在其执行完成或内存需求显著下降时，需要将此前获取的多余内存归还给相应的可被收割容器，并同步更新各自条目中的共享关系与当前配额。对于可被收割子池中的容器，则在归还完共享内存后恢复至原始配额；而对自给自足的容器，则执行常规的终止操作。随着容器数量的增加，租户管理器在每次共享决策时需要评估的对象数量也会增加，从而导致一定的决策开销。但当系统检测到某一时刻已经为目标容器分配或回收了足够的内存时，租户管理器会中止对更多候选容器的遍历，以降低额外的管理开销。

例如，考虑一个简单场景：某租户在同一节点上先后启动了两个容器，容器 A 内存紧张而容器 B 内存富余。当容器 A 首先到达并执行时，由于尚无合适的共享对象，租户管理器将其暂时归为需要额外支持的分配子池，但仍以初始内存配额启动执行。

随后，容器 *B* 到达并被识别为内存富余，因此被划入可被收割子池。此时，租户管理器根据容器 *A* 的内存需求，从容器 *B* 中收割一定数量的内存，并将该部分内存转移给容器 *A*。同时，在容器池中更新两个容器的条目：在容器 *A* 的条目中记录容器 *B* 为其内存来源之一，并记录从 *B* 处获得的内存量；在容器 *B* 的条目中则记录其向容器 *A* 输出的共享内存量。在调整完成后，两个容器便分别以更新后的当前内存限制继续执行。一旦容器 *B* 较早结束执行或其执行阶段发生变化，系统会回收其向容器 *A* 提供的那部分内存，并同步更新相关条目，使得后续的内存共享决策能够基于最新状态做出判断。

3.3.3　保障机制

在实际运行环境中，函数输入规模、节点整体负载以及背景干扰等多种因素都会对内存使用情况产生影响，使得容器在运行时的实际内存需求可能偏离调度时的预估。在采用内存共享机制的情况下，如果没有额外的控制手段，这种偏差可能在极端情况下演变为严重的内存不足故障，进而导致函数执行失败甚至影响节点稳定。

为了避免此类情况，本论文在租户管理器内部设计了一套保障机制，用于在内存共享行为可能引发风险时进行主动干预。当容器侧的运行时守护进程检测到内存使用即将触及安全阈值，或出现明显的内存紧张迹象时，会向所在节点上的租户管理器发送告警信息，指示当前容器面临潜在的内存不足风险。租户管理器在收到告警后，会立即启动对应租户范围内的内存重整流程：一方面，尝试从容器池中已分配出去的共享内存中进行强制回收，优先解除对当前告警容器的外部依赖；另一方面，尽量减少在此阶段为其他容器发起新的共享或扩大共享规模的操作，以稳定整体内存使用。

对于已经发生严重内存紧张的容器，租户管理器会将其条目标记为“不可收割”状态。对于被标记的容器，在后续新的容器尝试发起内存共享请求时，租户管理器将不再考虑从这些容器中收割内存。通过这一策略，可以避免因多次从同一高风险容器中回收内存而导致进一步的内存不足甚至进程异常终止。在风险解除后，系统可以在更长时间尺度上重新评估相关容器的实际内存使用轨迹，并在必要时更新其初始内存限制或调度策略，以更好地适应其真实负载特征。

总体而言，保障机制在内存共享体系中扮演了“安全阀”的角色：在正常情况下，租户管理器通过容器池和内存共享策略提升节点内存利用率；一旦检测到潜在风险，保障机制则优先维护函数执行的正确性和节点的总体稳定性，在性能与安全之间取得平衡。

3.4 集群层面的集群控制器

在多节点的服务器无感知计算平台中，如何在全局视角下调度函数请求、平衡各节点的内存压力，并为节点侧的租户管理器创造有利的资源环境，是系统设计的关键问题之一。为此，本论文在集群层面引入了集群控制器组件，对租户在整个集群范围内的函数调用进行统一管理和调度决策。

系统为每一位租户分配一个唯一对应的集群控制器实例，用于集中管理该租户在全局范围内的函数配置与运行信息。在部署阶段，控制器内部的特征分析模块对函数的资源使用特征进行离线分析和建模，为后续调度提供基础数据。在调用阶段，控制器内部的调度模块结合函数特征与各节点的实时负载情况，将新的函数请求分发到最合适的节点，使得内存资源在集群范围内得到更为均衡和高效的利用。

3.4.1 特征分析模块

特征分析模块负责在函数部署或更新时，对函数的资源使用行为进行系统化的离线评估和记录。通过分析该模块收集的结果，系统能够为每个函数建立较为准确的内存需求与文件访问特征画像，从而为后续的集群调度和节点内存共享决策提供依据。

首先，特征分析模块需要确定每个函数在典型运行场景下的内存限制、实际内存使用量以及文件访问规模。在服务器无感知计算平台中，函数的资源利用率往往与输入数据规模密切相关。为避免仅依据单一输入导致的低估风险，特征分析模块基于用户提供的多组代表性输入数据，对目标函数进行多次并行执行。在多次执行过程中，系统持续监控两个关键指标：一是函数运行期间的实际内存使用峰值，二是函数访问的磁盘文件大小。

在全部测试运行完成后，特征分析模块从观测到的多种执行情形中选择资源消耗最高的一组作为保留结果。这一结果被视为该函数在实际运行过程中的资源消耗上限估计，用于后续的工作负载均衡以及节点侧租户管理器划分容器池时的参考基准。通过保留观测到的最大资源消耗情形，可以在一定程度上抵御运行时输入规模波动带来的不可预见性，从而降低因资源配置不足导致任务失败或频繁触发内存保护机制的风险。

具体而言，特征分析模块为每个函数记录以下两类信息：一方面是函数本身的内存使用量上界，另一方面是运行过程中访问的文件数据的最大体量。在系统中，函数产生的中间文件和临时数据可以存放于内存文件系统以加速访问。因此，当对函数整

算法 3.1 工作负载均衡分配

Input: **任务**：内存限制 T_{lim}，最大内存使用量 T_{mem}；**节点**：节点内存限制 N_{lim}，内存分配量 N_{alloc}，以及内存需求量 N_{req}

Output: 最优节点 *optimalNode*

```
Function SelectOptimalNode(task, nodes):
    candidates ← {} // 初始化候选节点集合
    for node in nodes do
        // 检查节点剩余容量是否满足任务限制
        if node.N_lim ≤ node.N_alloc + task.T_lim then
            candidates.add(node) // 加入候选名单
        end
    end
    if candidates is not empty then
        optimalNode ← null
        optimalScore ← −∞
        for node in candidates do
            // 计算得分：衡量内存需求与分配之间的偏差
            score ← −|node.N_req + task.T_mem − node.N_alloc − task.T_lim|
            if score > optimalScore then
                optimalScore ← score // 更新最高分
                optimalNode ← node // 记录最优节点
            end
        end
        optimalNode.allocate(task) // 执行分配
        return optimalNode
    else
        return null // 无可用节点
    end
End Function
```

体资源需求进行评估时，需要同时考虑函数的内存使用峰值以及在运行期间访问的最大文件数据规模。两者的和可以视为该函数在最极端情况下可能使用的内存需求上限，为调度模块和节点侧租户管理器提供统一的参考值。

在函数成功完成特征分析之后，集群控制器会将这些分析结果与函数标识关联存储。在后续的每一次函数调用到来时，调度模块可以直接查询对应函数的资源画像，而无需再次进行离线分析，从而降低在线决策的时间开销。

3.4.2 调度模块

调度模块负责在函数调用到达时，基于特征分析模块提供的函数资源画像和各个节点的实时负载情况，将请求分发到合适的节点上运行，伪代码如算法 3.1所示。

调度模块的目标是：在不突破节点内存限制的前提下，使各节点上内存分配与实际需求之间的差距尽可能接近，从而在集群范围内实现较均衡的内存供需关系。

当某个函数调用请求到达时，调度模块首先在控制器内部检索该函数先前由特征分析模块生成的资源信息，其中包括：函数的内存配置上限 T_{lim}（即平台为该函数初始分配的内存额度），以及根据特征分析获得的最大实际内存使用量 T_{mem}。T_{mem} 反映了函数在典型及极端输入条件下的实际内存和文件访问总需求上界。

随后，调度模块需要评估当前集群中各节点的负载状态。对于每一个节点，调度模块维护并实时更新以下几个关键指标：

- 节点的内存总限制 N_{lim}，即该节点可用于承载服务器无感知计算函数的可用内存总量；
- 节点的总内存分配量 N_{alloc}，表示当前已经分配给该节点上所有任务的内存额度之和，即所有正在运行任务的 T_{lim} 之和；
- 节点的总内存需求量 N_{req}，表示基于特征分析结果估算出的节点上所有任务的实际内存需求总和，即这些任务对应的 T_{mem} 之和。

在掌握上述信息后，调度模块首先筛除那些无法安全容纳新任务的节点。具体而言，对于某个候选节点，如果在将当前任务的内存上限 T_{lim} 加入该节点的总分配量之后，会导致 $N_{alloc} + T_{lim}$ 超过节点总内存限制 N_{lim}，则该节点被视为无法容纳本次函数调用，将不会进入候选集合（行 3-行 7）。通过这种过滤，可以确保任何被选中的节点在形式上不会突破其可用物理内存上限。

对于剩余的候选节点，调度模块需要在其中进一步选择最合适的目标节点。为此，本论文采用了“最小化内存供需差值”的策略，即在每一个候选节点上，调度模块预估在接纳新任务之后，该节点的总内存分配量与总内存需求量之间的绝对差值：

$$\left|(N_{req} + T_{mem}) - (N_{alloc} + T_{lim})\right| \tag{3.4}$$

其中，$N_{req} + T_{mem}$ 表示在新任务加入后该节点的总需求估计，$N_{alloc} + T_{lim}$ 表示该节点分配出去的总内存额度（行 12）。调度模块从所有候选节点中选取上述差值最小的节点作为调度目标（行 13-行 16）。这一策略的设计动机在于：当节点的内存分配量和实际需求量接近时，既避免了大面积的内存闲置，又降低了整体内存不足的风险，有助于在全局范围内实现资源利用的均衡。

整体来看，这种调度方法在满足节点内存安全约束的前提下，尽可能缩小各节点内的过度分配与真实需求之间的距离，从而提高集群中每一单位内存的有效利用率。与此同时，调度模块在实际实现中仅需要对当前活跃节点进行一次线性扫描即可完

成一次调度决策，因此其时间复杂度随节点数量线性增长。在节点规模扩展时，调度决策的开销也相应线性扩展，能够较好地满足大规模集群环境对在线调度延迟的要求。

通过集群层面的特征分析与调度模块协同工作，系统在函数部署阶段建立起可靠的资源画像，在函数调用阶段则基于这些画像与节点负载状态进行精细化调度。在集群控制器完成节点选择后，对应租户在目标节点上的租户管理器即可在更准确的预估基础上开展节点内的内存共享和再分配，从而形成“集群级粗粒度调度 + 节点级细粒度共享”的协同管理体系。

3.5　本章小结

本章围绕系统的整体设计，从不同层次对关键组件及其协同工作方式进行了系统性的介绍。

首先，在系统架构及工作流程概述部分，从全局视角给出了系统的整体结构与关键数据路径，说明了如何在现有服务器无感知计算平台的基础上引入基于内存的文件系统，加速函数的 I/O 操作，同时通过节点侧租户管理器和集群控制器协同工作，实现从单节点到全局范围的资源管理与调度。

随后，在容器层面运行时守护进程部分，本章介绍了部署在每个函数运行环境旁路的守护进程机制。该守护进程负责在不修改用户函数代码的前提下，对文件系统调用进行透明拦截与重定向，将频繁访问的中间数据尽可能地落在内存文件系统中，并在此过程中尽量减少与用户态运行时的耦合，从而保证系统的通用性和可移植性。

接着，在节点层面的租户管理器部分，本章重点阐述了在多租户共享同一物理节点的前提下，如何通过统一的租户管理器实现对节点内内存资源的集中监控与动态调配。租户管理器一方面根据各个函数的资源画像与实际运行状态划分和调整容器池中可用于收割的内存空间，另一方面在节点整体内存压力升高时触发保护机制，及时回收可释放的缓存空间，避免影响正常函数执行，从而在性能与稳定性之间取得折中。

最后，在集群层面的集群控制器部分，本章介绍了面向多节点环境的全局控制与调度策略。通过为每一位租户配置专属的集群控制器，系统在部署阶段利用特征分析模块对函数进行离线资源画像构建，在调用阶段由调度模块结合函数资源需求与各节点当前的内存负载状态，完成对函数请求的节点级分配。这种基于函数特征的调度方式在不突破节点内存限制的前提下，尽量缩小各节点内存分配与实际需求之间的

差距，实现集群范围内更为均衡的资源利用。

综上，本章从容器运行时、节点管理到集群控制三个层面，构建了一个分层协同的系统设计：集群控制器负责全局视角下的函数调度与资源画像管理，节点侧租户管理器承担本地内存共享与保护机制，容器侧运行时守护进程则具体落实 I/O 路径上的内存加速。各层组件共同作用，为后续实现高效的内存感知 I/O 加速和调度优化奠定了完整的系统基础。

第 4 章 Ephemera 系统实现

本论文通过约 3000 行代码实现了 Ephemera 原型系统。具体而言，运行时守护进程采用 C 语言编写，以便更灵活地控制内存管理；租户管理器和集群控制器则使用 Python 实现，利用其简单的语法与丰富的库生态加速开发。此外，系统运行过程中使用 Docker 作为应用程序的隔离沙箱，以便在可控环境中复现与评估系统行为。

在上述实现基础上，本章从工程落地角度展开，依次给出各关键组件的实现细节。首先，第4.1节介绍函数调用路径的实现方式，说明代理进程与启动器进程如何解耦调度与执行。随后，第4.2节阐述运行时守护进程的实现，包括文件操作拦截、基于 mmap 的内存文件系统与内存监控机制。接着，第4.3节给出节点侧租户管理器的实现，重点说明事件驱动的状态管理、容器池划分与内存借还/回收流程。最后，第4.4节介绍集群控制器的实现，包含特征分析模块与调度模块的数据结构与决策逻辑。

4.1 函数调用实现

本系统的函数调用机制借鉴了开源服务器无感知计算平台 OpenWhisk[①]的典型架构设计，采用代理进程与启动器进程相互分离的模式，以此实现请求调度与函数执行的解耦，如图 4.1所示。系统中设计了一个常驻的代理进程，作为所有函数调用请求进入容器的统一入口。代理进程的核心职责包括：接收来自上层管理组件发出的函数调用请求，解析其中携带的函数标识、参数与资源限制信息，并将解析后的请求转交给具体的执行单元，即启动器进程。同时，代理进程还负责跟踪每次调用的执行状态，在调用结束后将结果返回给上层。

启动器进程则专注于函数执行本身。对于每一次被代理进程转发的调用请求，启动器进程会在隔离的运行环境中加载用户定义的函数代码，完成必要的初始化操作，并在函数执行期间维护标准输入和标准输出的数据通道。启动器进程还负责在执行过程中捕获运行时错误与异常情况，将错误信息和退出状态一并反馈给代理进程，以便上层组件进行重试、告警或统计分析等后续处理。

在进程间通信方面，系统各组件之间采用基于管道的通信机制，以保证数据传输过程中的低延迟与较小的系统开销。代理进程与上层管理组件之间通过一对命名管道保持长期连接：输入方向的命名管道用于接收函数调用请求以及动态内存限制更

① https://github.com/apache/openwhisk

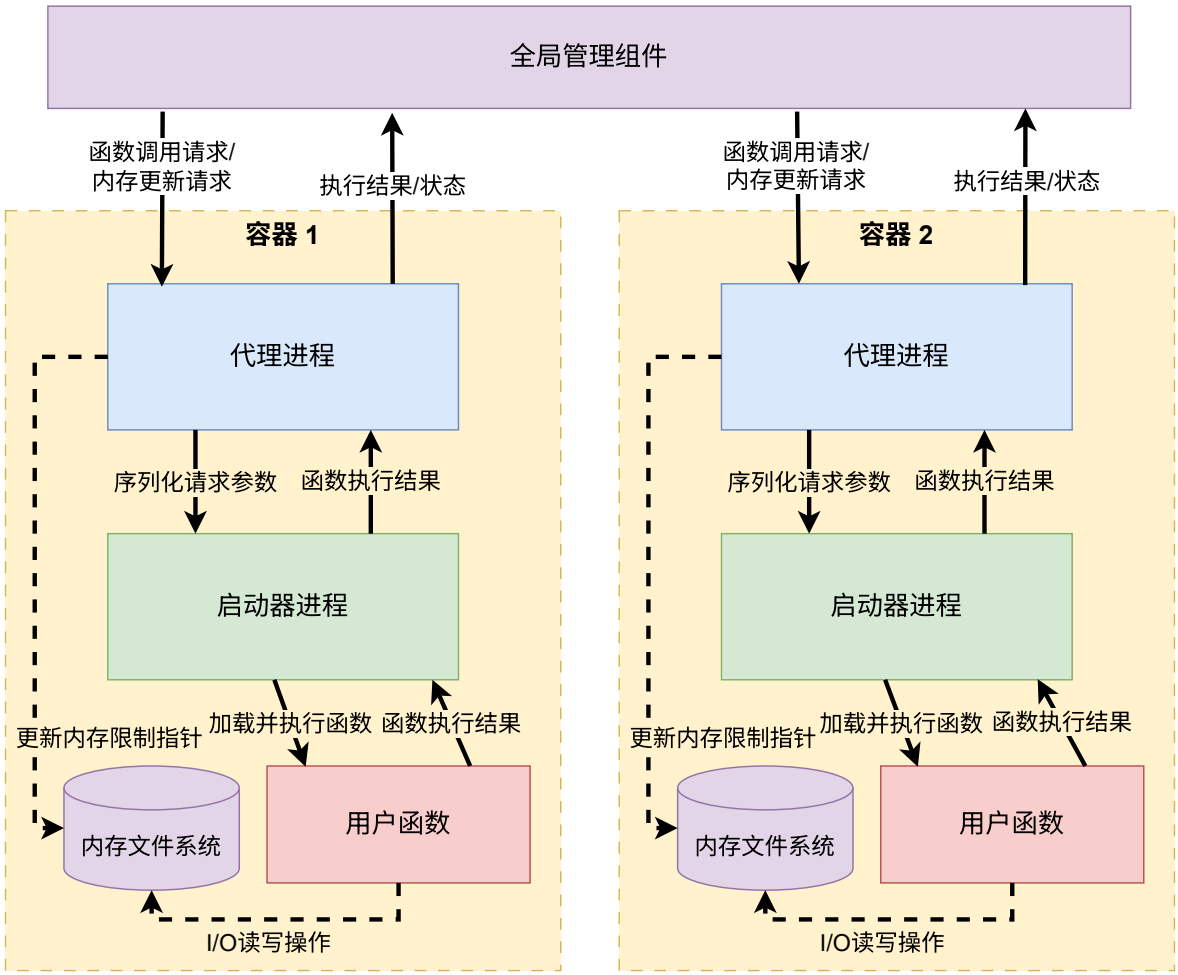


图 4.1　函数调用实现图

Figure 4.1　Implementation of Function Invocation

新指令，输出方向的命名管道用于回传函数执行结果、错误信息以及状态码。通过这种方式，管理组件无需直接感知具体的函数运行环境，只需面向代理进程进行统一交互，从而实现接口的简化与模块的解耦。

当代理进程需要启动或复用启动器进程时，会为该启动器创建一对专用的匿名管道，并在进程创建过程中将启动器的标准输入与标准输出重定向到这对管道上。之后，代理进程会将序列化后的请求参数写入启动器的标准输入，启动器在函数执行完成后再通过标准输出将结果返回给代理进程。由于整个过程基于内核提供的管道原语实现，数据在内核空间中以流的形式传递，无需额外的网络收发，大幅降低了单次调用的额外开销。

除了负责请求转发，代理进程还承担了部分动态资源管理功能。当代理进程通过命名管道接收到上层发来的内存限制更新指令时，会直接修改映射在内存文件系统

中的内存限制指针变量。该指针由代理进程与内存文件系统组件共同访问，用于指示当前容器在内存收割框架下可用于 I/O 加速的内存上限。通过这种基于共享内存的设计，系统无需频繁访问磁盘或通过额外的进程间消息进行同步，即可在较短时间内完成内存额度的调整，从而保证内存限制策略的实时生效。

为了进一步提升资源利用率并降低系统在空闲阶段的能耗，本系统在各个管道上采用阻塞式输入输出策略。当命名管道或匿名管道中没有待处理的数据时，代理进程和启动器进程会自动进入阻塞睡眠状态，由操作系统在有新数据到达时唤醒相关进程。与持续轮询管道状态的忙轮询方式相比，这种阻塞式模型能够在无请求或请求稀疏的阶段显著降低处理器占用率，避免无效的上下文切换和空转计算资源浪费。配合前述的解耦架构与共享内存限制指针设计，该通信与调用机制为后续的内存收割与 I/O 加速机制提供了稳定而高效的运行基础。

4.2 运行时守护进程实现

运行时守护进程的核心目标是在不修改用户函数源代码的前提下，透明地拦截其文件访问请求，并将这些请求尽可能重定向到内存中的文件表示，从而减少系统调用开销与磁盘访问延迟，提升 I/O 性能。为此，本论文将运行时守护进程实现为一个动态链接库 `runtime_daemon.so`，由专门的启动器组件在函数启动时加载并初始化。用户函数本身被编译为共享库 `main.so`，在启动器进程中通过 `dlopen` 和 `dlsym` 进行动态加载调用。由于用户函数与运行时守护进程运行在同一进程空间内，且符号解析由启动器进程事先完成，因此用户函数内部对标准文件操作接口的调用，可以被无缝地重定向到本论文实现的拦截逻辑上。

在底层访问路径上，本论文利用 `mmap` 和 `munmap` 系统调用来实现对内存中文件内容的直接读写。`mmap` 能够将磁盘上的文件映射到进程的地址空间，使得对文件的访问可以通过普通的内存读写完成，而无需每次都触发内核态的 `read` 或 `write` 系统调用。`munmap` 则用于在文件不再需要驻留内存时撤销映射，释放对应的虚拟地址空间和物理页。在运行时守护进程的实现中，文件的元数据和调度状态由用户态维护，而真正的数据读写通过内存映射区域完成，由此构成了一个轻量级的内存文件系统。

4.2.1 指令拦截实现

指令拦截的实现基于函数覆盖与动态链接技术的结合。首先，启动器进程用于加载用户函数和运行时守护进程；其次，启动器进程中定义了一组与标准 C 库同名的函数，例如 open、read、write、lseek、mkdir 和 close 等。当用户函数 main.so 被 dlopen 到启动器进程中时，其内部对这些 API 的调用将首先解析到启动器进程中的同名实现，而不是 glibc 提供的版本。

这些包装函数的行为本身十分简单，它们并不直接执行系统调用，而是将控制权直接地转交给运行时守护进程中导出的 exec_open、exec_read、exec_write、exec_lseek、exec_mkdir、exec_close 等函数。启动器进程在启动阶段调用 dlopen 加载运行时守护进程的动态链接库 runtime_daemon.so，并用 dlsym 将上述函数指针解析并保存下来。此后，任何来自用户函数的文件操作请求，都会先进入启动器进程中的包装层，再经由函数指针调用运行时守护进程内部逻辑，从而完成统一的拦截与调度决策。

为了在不同实验模式下灵活切换，启动器进程还通过编译期宏和运行期接口组合使用来控制拦截行为。通过 ONLY_IN_DISK 和 ONLY_IN_MEMORY 宏可以在编译时选择完全使用传统磁盘文件系统、使用不带调度的内存文件系统，或者启用带调度的混合模式。在运行时，启动器进程首次请求时强制所有新建文件仅驻留磁盘，以便特征分析模块能够观察到真实的磁盘访问模式；在后续请求中将文件的默认放置位置切换为内存，使得 I/O 请求优先命中内存文件系统。在有调度的模式下，启动器进程还会在后续请求中启动额外的监控线程，通过运行时守护进程的监控入口函数对内存使用与文件访问行为进行持续观察和调整。

4.2.2 内存监控实现

为了保证内存文件系统在提供加速的同时不突破预设的内存上限，运行时守护进程内部集成了基于 /proc 文件系统的内存监控机制。Linux 内核为每个进程在 /proc/[pid] 目录下维护了一组虚拟文件，其中 statm 文件用于描述进程的内存使用情况。假设被监控的进程号为 pid，读取 /proc/[pid]/statm 即可获得七个以空格分隔的字段：size、resident、shared、text、lib、data 和 dt。各个字段的内容如表 4.1所示。所有字段的单位均为页，页大小可以通过 sysconf(_SC_PAGESIZE) 获取。

在本系统中，本论文使用第二个字段 resident 作为进程物理内存占用的指标。

表 4.1 Linux /proc/[pid]/statm 各字段含义
Table 4.1 Fields of Linux /proc/[pid]/statm

字段名	含义与说明
size	进程的虚拟内存大小，即当前映射在该进程虚拟地址空间中的总页数，包含尚未实际使用的映射区域。
resident	进程的常驻内存大小，表示当前实际驻留在物理内存中的页数，包含代码段、数据段、堆、栈以及通过 mmap 映射的文件页。
shared	与其他进程共享的物理页数，典型包括只读代码段和共享库等，被多个进程同时引用。
text	进程自身代码段（text segment）的页数，主要由只读指令和常量数据组成，不包括共享库中的代码。
lib	进程使用的共享库代码段页数，包括动态链接库等，由多个进程共享。
data	进程私有数据段和堆栈区域的页数，包括全局/静态变量、堆分配对象以及栈帧等，不含共享库数据。
dt	与动态链接相关的额外页数，一般包括动态链接器在装载和重定位阶段使用的辅助数据。

由于 resident 计入了所有实际映射到物理内存的页，包括内存文件系统通过 mmap 建立的文件映射区域，因此它能够较为准确地反映当前内存文件系统对物理内存的压力。运行时守护进程内部的内存适配器线程会周期性地读取 /proc/[pid]/statm，将解析得到的 resident 值与预设的内存限制进行比较；当检测到常驻内存占用接近或超过阈值时，调度模块会在一组候选文件中挑选合适的对象，将其从内存中调出并仅保留在磁盘上，以此控制整体内存占用不超过预算。这样，内存文件系统不仅能够为热点数据提供低延迟访问，还能在多次函数调用和多租户场景下维持稳定的内存边界。

4.2.3 内存文件系统实现

为了将拦截到的文件操作重定向到内存，本论文在运行时守护进程内部实现了一套轻量级内存文件系统，其核心由 inode 表、文件元数据结构和映射管理逻辑构成。内存文件系统仅对特定目录（实现中为 /tmp）下的文件生效，以保证兼容性和隔离性。系统启动时，运行时守护进程通过 init_root_inode 创建根 inode，并将其路径固定为 /tmp。后续每次拦截到 exec_open 请求时，首先会在全局 inode 数组中查找目标路径；如果还不存在对应条目，则在父目录已存在的前提下创建新的 inode，

并为文件分配一个 file_info_t 结构，用于记录与该文件相关的所有运行时信息。

在内存文件系统中，一个 inode 主要存储文件或目录的路径名和权限信息，并通过指针关联到文件元数据。如果 inode 表示的是目录，其 file 指针为空；如果是普通文件，则 file 指向一个 file_info_t 结构。该结构中包含真实内核文件描述符 disk_fd，用于在必要时访问底层磁盘文件；包含 flags、offset 和 file_size 等字段，用于维护打开模式、当前读写位置以及逻辑上的文件长度；同时还包含一个 place 标志指示文件当前驻留位置（内存还是磁盘）、一个指向映射内存区域的指针 in_memroy_content 及其长度 mmap_len，以及同步并发访问的互斥锁和记录调度决策所需的时间戳。在逻辑层面，内存文件系统通过维护这一全局 inode 表和文件元数据数组，实现了对所有被拦截文件的集中管理。

当文件被判定应当驻留内存时，exec_open_memory 会负责将其内容映射到进程地址空间中。随后通过底层封装的 __open 调用打开实际的磁盘文件，并根据是否为首次打开、是否携带 O_TRUNC 标志以及文件名中是否带有持久化前缀，对 mmap_len 进行设定：对于需要保留原始内容的持久文件，直接以当前文件大小为映射长度；对于临时文件，则预先通过 ftruncate 将文件扩展到一个固定的最大容量，从而减少后续扩容带来的系统调用开销。完成长度设定后，运行时守护进程调用 mmap 将该文件映射为一个可读写的共享内存区域，并将返回的指针保存在 in_memroy_content 字段中。此后读操作，通过在 exec_read_memory 中按照 offset 和 file_size 计算可读长度，然后在用户给定缓冲区和映射内存之间执行 memcpy 即可完成。写操作则在 exec_write_memory 中根据当前偏移和写入长度直接覆盖映射区域，并更新逻辑文件大小与偏移位置。与传统基于系统调用的 I/O 不同，这一过程完全在用户态通过内存拷贝完成，只在必要时由内核负责脏页回写，大大降低了频繁调用 read/write 带来的开销。

当某个文件不再需要驻留内存，或者系统内存压力过大时，调度模块可以通过 scheduler_file_out 等接口将文件从内存中“调出”。该过程会调用 munmap 解除对文件的映射，并将文件的 place 标记为磁盘模式，此时后续对该文件的访问会切换到 exec_open_disk、exec_read_disk 和 exec_write_disk 等路径上。在最终回收 inode 时，运行时守护进程还会通过 ftruncate 将磁盘文件截断到逻辑长度 file_size，以去除之前为映射预分配但未实际使用的尾部空间，保证磁盘上文件内容与逻辑视图保持一致。

为了在全局层面协调多个文件之间的内存占用，运行时守护进程维护了若干辅

助函数用于统计当前内存文件系统的总体规模和选择调度对象。例如，通过遍历 inode 表计算所有文件逻辑大小总和，可以估计当前内存中文件数据的体量；通过时间戳，可以在关闭文件和打开文件之间优先选择合适的调出候选，从而既保证热点数据尽可能留在内存，又避免同一文件在短时间内反复调入调出。结合前文基于 `/proc/[pid]/statm` 的内存监控机制，内存文件系统在提供高性能 I/O 的同时，能够动态适配不同的内存限制配置，在多次函数调用和多租户负载下保持良好的资源利用率和稳定性。

4.3 租户管理器实现

在实现层面，节点层的租户管理器采用“集中控制线程 + 事件队列 + 容器资源池状态”的总体结构。在每台物理节点上，系统启动一个专门的租户管理线程作为全局控制器，该线程从进程内的消息队列中持续读取事件，这些事件对应不同函数容器的生命周期阶段（初始化、函数调用开始、函数调用结束以及安全保护触发等）。围绕这一控制线程，系统为每个函数容器维护一个抽象的资源池项结构，用于记录该容器的初始内存配额、当前内存上限、基于历史特征分析得到的最大内存与文件占用，以及与其他容器之间的内存借还关系。整体流程如图 4.2所示。通过这种实现方式，租户管理器可以在函数调用粒度上动态调整各容器的内存上限，实现节点内租户函数实例的细粒度内存共享。

在数据结构设计方面，租户管理器为每个容器构建了一个 `PoolItem` 结构，用于统一描述容器的资源状态。具体而言，`PoolItem` 中记录了容器名称、对应的 Docker 进程句柄、初始内存限制 M_{init}、当前内存限制 M_{now}，以及通过特征分析得到的函数最大内存使用量 $U_{\max}$ 和最大文件访问需求 $F_{\max}$。为了预留安全裕量，系统对特征分析结果按系数 1.1 放大，即在内部计算中采用 $1.1U_{\max}$ 与 $1.1F_{\max}$。此外，`PoolItem` 还维护了两张映射表：一张记录当前容器向哪些其他容器出借了多少内存，另一张记录当前容器从哪些容器借入了多少内存。通过这些状态变量，租户管理器可以在任意时刻恢复每个容器的逻辑内存上限以及完整的借还拓扑，为后续的内存回收与安全保护操作提供基础。

为便于调度决策，租户管理器将所有容器划分为两个逻辑池：收割池与分配池。当某个容器首次启动并返回特征分析信息时，管理线程会触发 `add_to_pool` 逻辑，根据“初始内存限制”与“估计的内存 + 文件需求”之间的关系将其归类。如果 $1.1U_{\max}+1.1F_{\max}$ 显著小于 M_{init}（例如小于 $0.9M_{\text{init}}$），则该容器被视为“内存富余”，加入收割

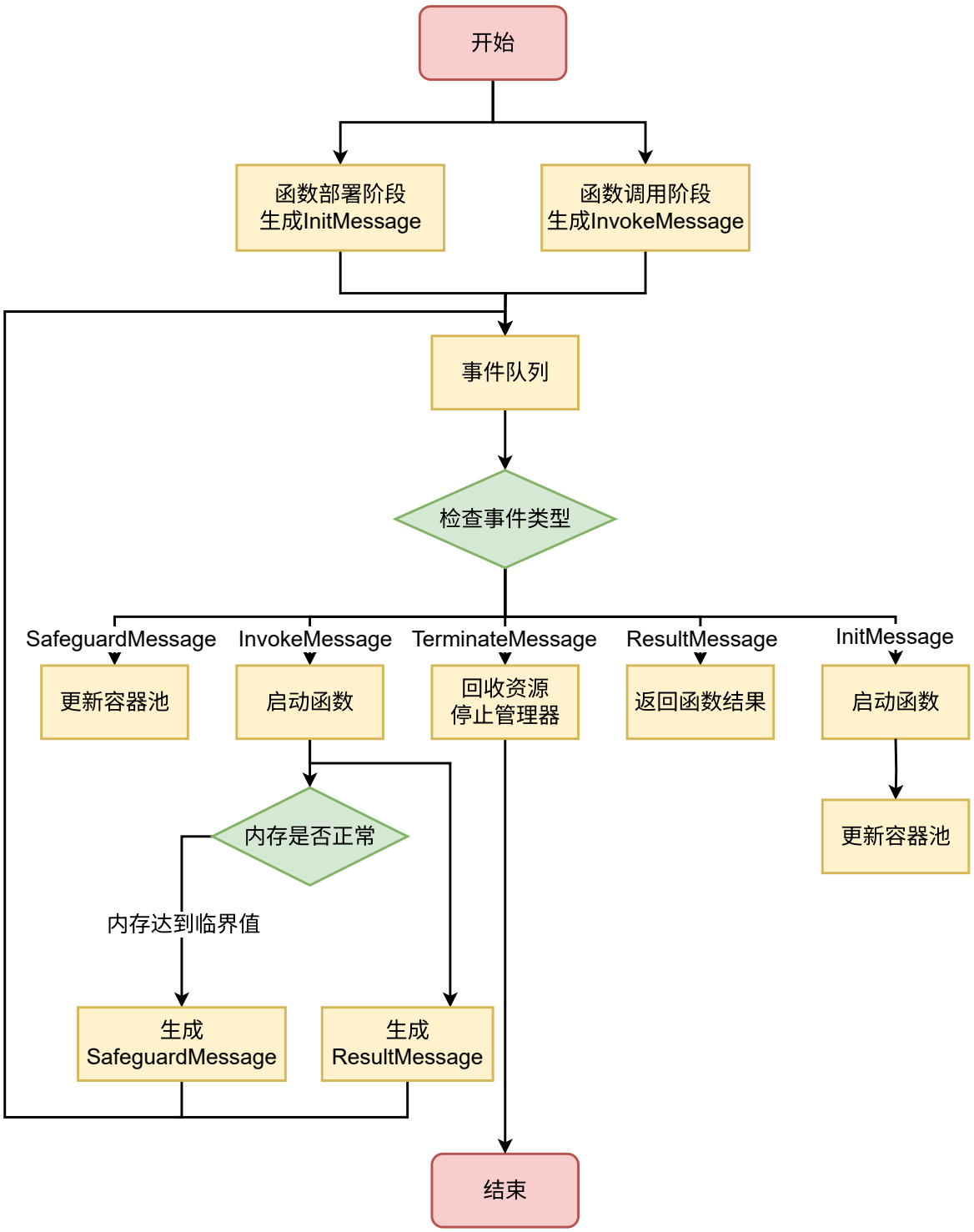


图 4.2　租户管理器流程图
Figure 4.2　Tenant Manager Flowchart

池，后续可以向其他容器出借剩余内存；如果 $1.1U_{\max} + 1.1F_{\max}$ 大于 M_{init}，则该容器被视为“内存短缺”，加入分配池，在函数调用时需要从其他容器借入额外内存；处于两者之间的容器被视为自给自足，不加入任何一个池，也不参与内存共享。为了支持对照实验，系统实现了一个布尔配置开关，当禁用内存共享机制时，`add_to_pool` 退化为仅构建 `PoolItem` 而不对其进行池划分，从而将所有容器视作独立、静态配额管理的实例。

在事件处理逻辑上，租户管理器采用一个长生命周期的管理线程，对全局事件队列进行循环处理。队列中的事件类型主要包括五类：容器初始化事件（InitMessage）、函数调用开始事件（InvokeMessage）、函数调用结束事件（ResultMessage），安全保护事件（SafeguardMessage）以及结束事件（TerminateMessage）。对于初始化事件，管理线程会结合特征分析信息构造对应的 `PoolItem`，并将其加入合适的资源池；对于调用开始事件，管理线程根据目标容器所属的资源池类型，决定是否需要触发内存再分配；对于调用结束事件，管理线程负责将该容器及其关联容器的临时借出或借入内存恢复为初始状态；而安全保护事件则用于在接近内存告警的极端情况下强制触发一次恢复操作，以快速消除潜在的资源风险；结束事件用于停止整个管理器，并回收资源。基于这种事件驱动的实现，内存共享逻辑的触发时机与容器生命周期自然对齐，从而简化了并发环境下的状态管理。

在函数调用调度层面，系统为每个容器维护一个独立的本地请求队列，以保证同一容器内函数实例的串行执行。实现中，租户管理器在一个全局字典中为每个容器名称关联一个队列，队首元素表示当前正在执行的请求，后续元素表示排队等待的请求。当某个函数调用开始事件到达时，如果对应容器的队列当前为空，则直接将该消息入队，并立即触发一次调用流程；否则，仅将其追加到队列尾部，由前一条请求结束时再继续触发。相应地，当管理线程收到某容器的函数调用结束事件时，会首先从该容器队列弹出当前请求；如果队列仍非空，则直接启动下一条调用且不执行内存归还操作；只有当队列完全清空时，才会触发该容器的内存恢复逻辑。通过这种按容器划分的请求队列，系统保证了内存借还的边界与函数调用的开始和结束严格对齐，避免了在同一容器内部多个请求并发导致的资源状态混乱。

在内存共享决策方面，租户管理器在处理函数调用开始事件时会根据容器当前的角色（位于收割池、分配池或自给自足）执行不同逻辑。若当前容器属于收割池，说明其自身在特征分析结果中长期具有较大内存富余，系统会遍历分配池中所有正在运行的短缺容器，计算它们的缺口 $D = 1.1U_{\max} + 1.1F_{\max} - M_{\text{now}}$ 以及当前收割容器

的可用富余 $S = M_{\text{now}} - 1.1U_{\max} - 1.1F_{\max}$。当某个短缺容器的 D 满足 $0 < D \le S$ 时，租户管理器会通过 `memory_re_allocation` 过程，将数量为 D 的内存从收割容器的当前限制中扣除，并增加到短缺容器的当前限制中，同时在双方的映射表中记录这笔借还关系。若当前容器属于分配池，则说明其在历史上难以仅凭初始配额满足需求，此时管理线程会计算该容器的总缺口，并依次扫描所有正在运行的收割容器，按照其富余内存量逐步借入，直至缺口被填平或收割容器的富余资源全部耗尽。对于既不在收割池也不在分配池的自给自足容器，系统直接发起调用请求，不参与任何内存迁移。

在内存配额的实际调整上，租户管理器通过与 Docker 运行时的协同，实现对容器 cgroup 内存限制的动态修改。每当发生一次内存再分配时，系统不仅会更新收割方与分配方在逻辑上的 M_{now}，还会通过 Docker 命令将新的上限同步到内核层。具体而言，管理线程会调用诸如

```
docker update --memory <target>m <container>
```

的命令，修改目标容器的内存上限；同时，还会向容器内部运行的代理进程发送一条携带新内存限制参数的更新消息，使容器内部应用能够感知并适配这一变化。这种“外部 cgroup 更新 + 内部代理通知”的双层更新方式保证了内存共享机制在操作系统层与应用层的一致性，从而减少了因信息不一致导致的异常行为。

为了保证内存借还关系的可回滚性，系统在实现中设计了一套对称的恢复机制。当某次函数调用结束或安全保护事件触发时，租户管理器会根据容器当前的角色调用 `retrieve_in_pool` 逻辑，将先前的借还关系成对地撤销。对于处于收割池的容器，它在之前可能向多个分配池容器出借了内存，此时系统会依次从这些容器中收回对应数量的内存，并更新它们的当前内存上限；对于处于分配池的容器，则会将自己从外部借入的所有额外额度归还给对应的收割容器。恢复过程中，管理线程会同步更新所有相关容器的 Docker 配额，并从双方的映射表中删除对应记录。最后，通过 `reset_pool_item` 过程，将本容器的当前内存限制重置为初始值，清空其所有借还关系标记，并将运行状态标识重置为空闲。该机制保证了每一次函数调用结束后，相关容器都能回到干净的基线状态，从而支持在后续调用中重新、安全地进行新的内存共享决策。

此外，为了在极端情况下保护系统稳定性，租户管理器实现了基于安全保护事件的紧急恢复通道。当监控组件检测到某个容器接近内存崩溃边界时，可以向全局管理线程发送安全保护事件，请求立即对其内存状态进行“清零式”恢复。管理线程在接

收到该事件后，会调用与函数调用结束时相同的恢复逻辑，将该容器及其相关容器之间的所有借还关系撤销，使其回到初始配额，避免因长时间的出借或借入导致资源压力在少数容器上持续累积。结合前述按容器队列串行执行的约束，这一保护机制能够在复杂多租户场景下为节点层面的内存共享提供额外的安全保障。

针对实现的性能评估需求，系统还构建了一套独立的时间测试版本模块，在保持核心调度逻辑不变的前提下，用模拟的容器接口和特分析数据替代真实 Docker 运行环境。在这一版本中，容器启动、资源更新和容器内部通信被抽象为简化的函数调用和消息注入，管理线程仍然按照“创建–调用–结束”的请求序列驱动收割池与分配池之间的内存再分配流程。通过这一实现，系统可以利用启用/禁用内存共享机制的配置开关，对比“动态内存共享模式”和“静态配额管理模式”两种方案的性能差异，为后续的实验评估和分析提供实现层面的支撑。

4.4 集群控制器实现

本节在前文设计的基础上，详细介绍集群控制器在系统中的具体实现方式。实现主要由两个部分构成：运行在函数实例本地的特征分析模块，以及运行在控制平面上的调度模块。前者负责在函数首次执行时采集其资源使用特征，为后续的内存文件系统调度和集群级调度提供输入；后者则在集群层面对函数请求进行多节点放置决策，以提升整体内存利用率并降低资源过载风险。

4.4.1 特征分析模块实现

特征分析模块部署在每个函数实例的本地运行环境中，形成功能计算与特征采集的一体化执行框架。整体上，它负责在不修改业务代码的前提下，记录函数在典型输入下的内存占用轨迹和文件访问模式，并将这些原始观测归纳为可供集群控制器使用的任务特征。

在控制流程上，特征分析模块通过一个长期运行的前端控制进程接收来自集群控制平面的指令（如函数创建、调用、内存上限更新和终止等），将其解析为统一的请求格式并转发到本地执行环境。对于首次调用，请求会触发一轮“分析模式”的执行：系统在给定的内存上限约束下运行目标函数，同时由监控线程持续跟踪进程的内存使用情况和文件访问行为，典型指标包括峰值内存占用、不同文件的访问规模等。

监控结束后，特征分析模块将多维度的观测结果整理为结构化的特征摘要，并通过本地控制通道返回给上层控制平面。上层组件据此构建任务级的资源需求画像，

例如估计函数在典型输入下的最大常规内存需求和最大文件工作集大小，并形成后续调度所需的任务描述。

在随后的常规调用阶段，特征分析模块不再重复完整的分析流程，而是将首次调用得到的特征视为该函数在一段时间内的稳定资源使用模式。每次调用之前，控制平面会根据最新的内存上限配置更新本地约束参数，函数运行时则在运行时守护模块的配合下，以“以内存文件系统为主”的模式执行，从而在满足给定资源约束的前提下复用前期分析得到的特征。

4.4.2 调度模块实现

在实现层面,控制平面为每个工作节点维护一条状态记录,使用结构体 `NodeState` 抽象表示：

- 节点标识符，用于标记节点信息；
- 当前活跃任务集合，用于支持任务结束后的增量更新；
- 三个与内存相关的聚合量：配置内存上限 N_{lim}、名义分配量 N_{alloc}（所有活跃任务配额 T_{lim} 之和）以及估算需求量 N_{req}（所有活跃任务需求上界 T_{mem} 之和）。

上述三个聚合量在任务调度和回收时以 $O(1)$ 方式增量更新，无需在每次决策时遍历节点上全部任务重新计算，从而保证调度器在节点规模增大时仍能维持近似线性的时间复杂度。

与之对应,每一次待调度的函数调用在控制平面内部被包装为 `TaskDescriptor` 结构，除函数标识和租户标识外，核心字段包括：平台为本次调用预留的配额 T_{lim}，以及由特征分析模块提供的需求上界 T_{mem}。这两个字段在进入调度器之前已经确定，调度逻辑本身不再修改它们，只负责基于当前 `NodeState` 集合选择目标节点。

调度过程可概括为如下两步。首先是容量约束过滤：调度器顺序扫描所有在线节点，对每个节点检查 $N_{alloc} + T_{lim} \le N_{lim}$ 是否成立。对不满足该条件的节点，直接从候选集合中剔除。

在得到候选节点集合后，调度模块进入评价与选择阶段。此时，对于每一个候选节点，调度器基于其当前 `NodeState` 快照计算新任务加入后的名义分配量与需求量：

$$N'_{alloc} = N_{alloc} + T_{lim}, \quad N'_{req} = N_{req} + T_{mem} \tag{4.1}$$

随后，以 $E = |N'_{req} - N'_{alloc}|$ 作为该节点在当前任务下的评价值，表示“估算需求”与“已分配配额”之间的偏差。调度器在单次扫描过程中维护当前最优节点及其

最小评价值 $E_{\min}$，当发现更小的 E 时进行更新；若出现评价值相同的节点，则按照“调度后剩余配额更大”这一次级规则进行打破平局，即优先选择 $N_{lim} - N'_{alloc}$ 更大的节点，以为后续可能到来的大内存任务预留空间。

选定目标节点后，调度器会立即对其 `NodeState` 执行一次原子性的增量更新：将新任务加入活跃集合，将 N_{alloc} 增加 T_{lim}，将 N_{req} 增加 T_{mem}。对应地，当任务在目标节点执行完成并上报结束事件时，控制平面再次对同一 `NodeState` 做相反方向的减量更新，并在活跃集合中移除该任务。

为了便于实验评估，调度模块额外实现了一个“传统贪心”模式，作为对照组。在该模式下，调度器在候选节点集合上不再计算 E，而是直接选择调度后剩余配额 $N_{lim} - (N_{alloc} + T_{lim})$ 最大的节点。实现上，这一模式与特征感知模式共用同一套 `NodeState` 维护与容量过滤逻辑，只在评价函数计算处切换，便于在同一实验环境下对比两种策略的差异。

在性能测试版本中，系统通过构造离线任务轨迹输入文件，对调度模块进行批量回放测试。控制平面按顺序读取任务描述，调用上述调度过程为每个任务选择节点，并通过计时器记录总耗时和单次调度平均耗时；同时，测试框架也会统计在整个回放过程中各节点的 N_{alloc} 与 N_{req} 演化轨迹，为后续分析调度策略提供数据支撑。

4.5 本章小结

本章从实现角度对 `Ephemera` 的关键组件进行了系统性阐述。首先，在函数调用路径上，本论文实现了基于代理进程和启动器进程解耦的调用框架，通过命名管道与匿名管道组合构建低开销的进程间通信通道，并结合共享内存中的内存限制指针与阻塞式 I/O 策略，在保证数据通信正常的同时降低了空闲阶段的资源消耗。

其次，围绕单个函数运行时，本章给出了运行时守护进程的具体实现细节，包括基于 `dlopen`/`dlsym` 与函数覆盖技术的文件操作拦截机制、利用 `mmap` 构建的轻量级内存文件系统，以及依托 `/proc/[pid]/statm` 的内存监控与文件调度流程。通过全局 inode 表与文件元数据结构的配合，系统在用户态统一管理了驻留在内存与磁盘之间的文件放置，实现了对热点文件的快速访问与在内存压力下的按需回收。

随后，本章介绍了节点层租户管理器的实现：通过集中控制线程与事件队列，将容器级内存状态抽象为 `PoolItem` 结构，并基于特征分析得到的 $U_{\max}$ 与 $F_{\max}$ 将容器划分为收割池与分配池；在函数调用的开始与结束事件上，租户管理器通过动态修改 cgroup 内存限制和维护借还关系映射，实现了节点内多容器之间的细粒度内存共

享与对称恢复机制，并通过安全保护事件在极端情况下提供了紧急回退通道。

最后，本章给出了集群控制器在实现层面的落地方式。一方面，特征分析模块在函数首次执行时采集峰值内存和文件工作集等多维特征，并将其整理为控制平面可直接使用的任务描述；另一方面，调度模块围绕 `TaskDescriptor` 和 `NodeState` 两类核心数据结构，采用增量维护的方式高效计算各节点名义配额与估算需求的偏差，在容量约束过滤基础上完成特征感知调度决策，并实现了与传统贪心策略的对照实现与离线回放测试。通过上述实现，本章将设计章节中提出的跨层内存收割与特征感知调度方案具体化，为后续的性能评估与实验分析提供了可运行的系统基础。

第 5 章 实验与分析

本章通过自定义基准测试、标准测试集及真实工作负载，全面评估了 Ephemera 的性能及系统开销。第5.1节首先介绍了实验的硬件环境、软件配置以及评估指标。第5.2节通过自定义的自定义基准测试，分析了内存限制、文件大小和操作频率对系统性能的影响。第5.3节使用 FunctionBench 和真实工作负载进一步评估了系统的综合性能。第5.4节验证了内存共享机制在混合部署场景下的有效性，而第5.5节则对比了不同共享机制的优劣。第5.6节评估了调度器在多节点环境下的负载均衡能力。最后，第5.7节分析了系统各个组件的运行开销。

5.1 实验配置

实验环境 本节给出实验环境与实验平台的具体配置，用于在服务器无感知计算的场景下对本系统进行性能评估。表 5.1总结了底层计算基础设施的硬件和软件配置，包括处理器型号与主频、物理内存容量、磁盘类型与带宽、操作系统内核版本以及容器运行时配置等关键信息。实验中，每个函数实例均被分配到一个独立的 Docker 容器中运行，容器作为函数的隔离执行环境，用于模拟当前主流服务器无感知计算平台的函数运行方式。

在原型系统中，函数调用经由节点上的管理组件调度到具体容器内执行。所有实验均在同一集群环境中进行，以避免跨平台差异带来的干扰。统一的硬件与软件设置，为在典型场景下系统地考察本系统在延迟、带宽以及资源利用率方面的性能特征提供了可靠基础。

评估指标 为了全面刻画系统在不同工作负载下的表现，本次评估主要关注以下几类性能指标：应用程序执行延迟、I/O 带宽、I/O 强度以及内存利用率。

应用程序执行延迟反映了从函数开始执行到产生最终结果所经历的总时间，是衡量用户体验和系统吞吐能力的核心指标。I/O 带宽用于度量在给定时间内系统完成的数据读写量，用来反映内存加速机制在高负载情况下的服务能力。内存利用率则刻画了节点物理内存和可收割内存在实验期间的实际占用情况，用于分析内存收割机制对资源利用的提升效果。

I/O 强度用于描述应用程序执行过程中 I/O 操作占比的高低，即 I/O 相关操作相

表 5.1　实验测试平台配置
Table 5.1　Experimental Testbed Configuration

组件	规格
CPU 设备	Intel Xeon Gold 6248R
处理器基础频率	3.00 GHz
插槽数量	4
线程数量	192（96 个物理核心）
内存容量	512 GB
SSD 容量	11 TB
操作系统	Ubuntu 22.04 LTS
Docker 版本	24.0.7

对于总执行时间的重要程度。实际测量时，通过结合系统调用跟踪工具与时间测量工具进行评估：首先使用 `strace` 记录与 I/O 相关的系统调用占所有系统调用的比例，记为 p；然后使用 `/usr/bin/time` 分别统计程序在内核态花费的时间 t_k 与在用户态花费的时间 t_u。I/O 强度随即计算为：

$$Intensity_{I/O} = \frac{t_k \times p}{t_k + t_u} \tag{5.1}$$

该值越大，说明程序执行中与 I/O 相关的活动越占主导地位，更能体现底层文件系统与内存加速机制对整体性能的影响。

工作负载　本次评估使用的服务器无感知计算工作负载如表 5.2所示，涵盖了自定义基准测试函数以及代表性实际应用，力求在不同 I/O 特性与计算特性下对系统表现进行对比分析。

在自定义基准测试部分，本论文构造了一系列 I/O 密集型函数，用于考察不同内存限制、不同文件大小以及不同文件操作频率对内存文件系统性能的影响。这些微基准函数主要围绕顺序读写、随机读写以及多文件并发访问等典型场景展开，使得评估结果能够反映内存文件系统在多种 I/O 模式下的行为。

在实际应用评估部分，本论文选取了现有公开基准中的部分服务器无感知计算函数，并结合若干真实工作负载进行实验。一部分函数来自 FunctionBench[58]，涵盖了图像处理、数据分析、压缩解压等常见的服务器无感知计算应用场景；同时，本论文还引入了若干具有代表性的实际工作负载，以进一步展示系统在更贴近真实生产环境下的性能表现。

表 5.2 实验工作负载及其 I/O 强度
Table 5.2 Workloads and Their I/O Intensity

工作负载名称	I/O 强度	描述
微型基准测试	动态	在两个大小相同的文件上执行多次文件操作。
顺序磁盘 I/O	96.34%	对一个 32MB 文件执行多次顺序读写操作，突出连续数据访问场景。
随机磁盘 I/O	94.23%	对一个 32MB 文件执行多次随机读写操作，突出非顺序数据访问场景。
Gzip 压缩	动态	使用 gzip 对文件进行压缩与解压，涉及多次读写操作。
图像处理	64.17%	读取一张图像，将其转换为灰度图像，并将处理后的图像写回磁盘。
MapReduce 任务	87.84%	使用 2 个 Mapper 读取文件并统计单词出现次数，将结果写入磁盘，再由 1 个 Reducer 汇总 Mapper 输出。
机器学习推理	19.93%	使用 DNN 模型识别手写字符，主要关注模型推理过程。
视频处理	62.12%	读取一个视频文件，将其转换为灰度视频，并将处理后的视频写回磁盘。
大整数阶乘计算	0%	计算一个大整数的阶乘，不涉及文件操作。

为了评估内存共享机制的效果，本论文将自定义基准测试中构造的 I/O 密集型函数视为典型的 I/O 负载，并额外设计了以大整数阶乘计算为代表的计算密集型任务。通过在同一节点上并行部署 I/O 密集型任务与计算密集型任务，观察它们在不同内存限制和共享策略下的执行时间与资源占用情况，可以直观地分析内存收割机制和内存文件系统对多类型工作负载共存时的影响。最后，本论文同样采用这些 I/O 密集型和计算密集型任务作为组合工作负载，用于评估调度策略在多任务混合作业场景下的性能与公平性。

对比基线 为了定量评估本系统在 I/O 加速和资源利用方面的改进效果，本次实验选择了两种文件系统配置作为对比基线：基于联合文件系统 OverlayFS 的容器配置，以及基于 tmpfs 的容器配置。

OverlayFS 是容器运行时默认广泛采用的一种文件系统组织方式。其核心思想是将只读的基础镜像作为下层，将可写层作为上层，在运行时通过“叠加”的方式向容器暴露一个统一的文件视图。读取操作会优先访问可写层，如果未命中则回退到底

层镜像层；写入操作则仅发生在可写层，以保证基础镜像在运行期间保持只读状态。联合文件系统会对读取的数据进行缓存，以减少重复访问带来的开销，并在粗粒度上保证容器之间的镜像共享，从而降低存储占用。

tmpfs 是一种以内存作为主要介质的临时文件系统，其数据不会持久存储在磁盘上，而是在系统重启或卸载后被清空。当启动 Docker 容器实例时，可以通过 `--mount type=tmpfs` 选项将 tmpfs 挂载到特定目录，容器中的应用即可将该目录视为高速存储区域，用于保存中间结果和临时文件。tmpfs 的可用容量受分配给 Docker 实例的内存限制约束，当容器接近内存上限时，进一步的写入会受到限制。

在对比实验中，本论文分别使用基于 OverlayFS 的容器配置和挂载 tmpfs 的容器配置作为基线，测量在相同工作负载下的执行延迟、I/O 带宽以及内存利用情况。通过与本系统在相同环境下的实验结果进行对比，可以更清晰地分析内存文件系统与内存共享机制在提升 I/O 性能、改善节点内存利用率方面的具体收益。

5.2　自定义基准测试

本节针对典型的文件访问场景设计了一组自定义基准测试，用于在可控环境下系统地评估所提出机制在不同约束条件下的性能表现。本论文通过改变节点可用内存限制、实际文件使用大小以及文件操作次数等关键参数，使系统处于多种压力与资源约束组合下，对比分析不同配置对 I/O 延迟的影响。在实验过程中，运行时守护进程作为 Docker 运行时的一部分随平台启动，并在容器生命周期内与函数实例共同驻留与合作完成文件访问请求的拦截与重定向，从而真实反映系统在实际服务器无感知计算环境中的行为。

5.2.1　内存限制的影响

在这一小节中，本文评估不同内存限制条件下，`read` 与 `write` 系统调用的端到端延迟变化情况。实验中将每个容器的内存限制从 32 MB 逐步提升至 256 MB，并分别在 `Ephemera`、原生 OverlayFS 以及挂载 tmpfs 的环境中运行相同工作负载，以对比不同系统的延迟差异。图 5.1给出了在各类内存限制下的实验结果。

当内存限制为 128 MB 时，可完全容纳实验中涉及的全部文件，此时系统相较于 OverlayFS 的读写延迟最高降低了 54.73%。更为显著的是，当内存限制为 64 MB 时，虽然只能容纳部分文件，但系统依然实现了最高 92.67% 的延迟降低，表明在中等内存压力下，适当利用内存文件系统与调度策略可以显著提升 I/O 性能。在内存严重紧

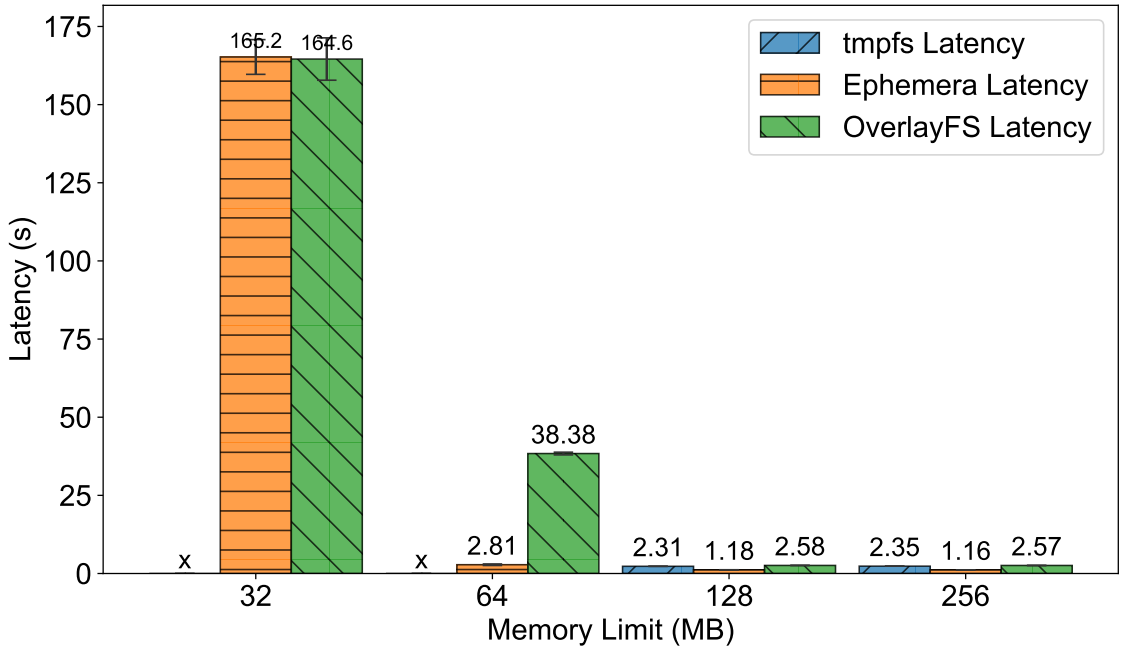


图 5.1 内存限制的影响对比图
Figure 5.1 Impact of Memory Limit

张、无法有效驻留文件数据时，系统的性能表现逐渐接近 OverlayFS。其原因在于，为避免频繁在内存与磁盘之间迁移大文件带来的额外开销，调度机制会倾向于不再将文件放入内存文件系统，而是退化为直接依赖底层文件系统进行读写，从而避免反复换入换出导致的性能抖动。

5.2.2 文件使用大小的影响

本小节关注实际文件使用大小对 `read` 与 `write` 系统调用延迟的影响。实验中将单次工作负载实际使用的文件大小从 32 MB 逐步增加至 160 MB，在每种文件大小配置下固定容器内存限制为 150 MB，并分别在 `Ephemera`、原生 OverlayFS 以及挂载 tmpfs 的环境中运行相同工作负载，以对比不同系统的延迟差异。图 5.2展示了不同文件大小条件下系统延迟的变化趋势。

当内存容量足以完整容纳所有参与操作的文件时，系统能够将数据长期保留在内存文件系统中，读写请求几乎完全命中内存，因此整体性能提升较为显著。在内存只能存放部分但非全部文件的情况下，系统通过调度策略优先将热点或高频访问数据保存在内存中，最大限度减少回退到底层文件系统的次数。在这一场景下，延迟降低最高达到 89.5%，说明在中等数据规模与有限内存资源下，合理利用内存对 I/O 性

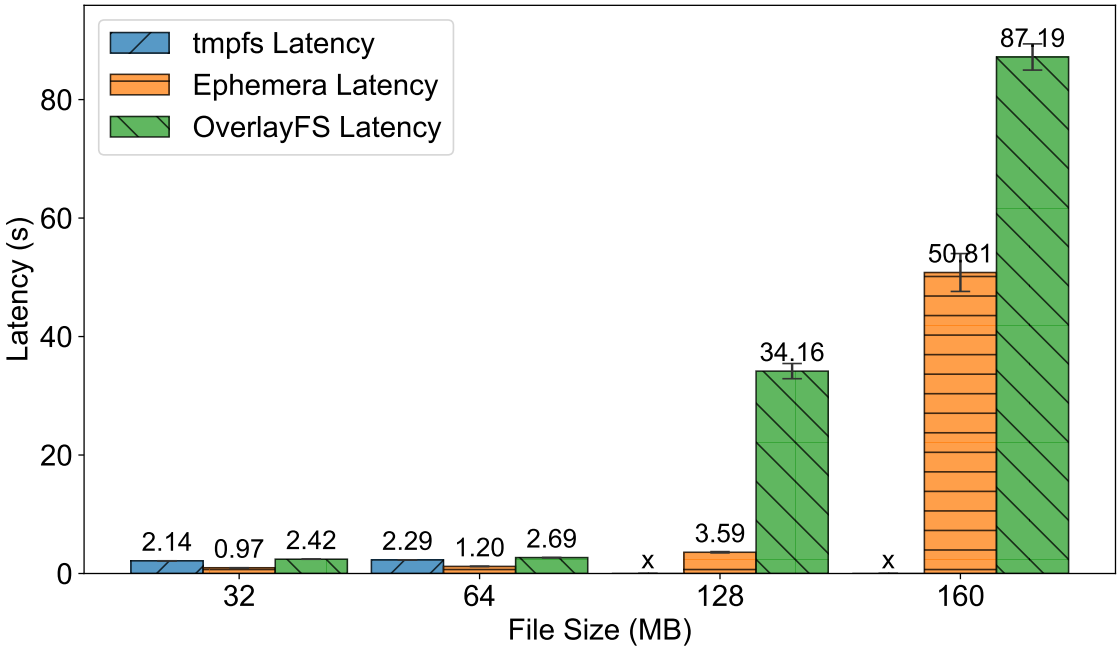


图 5.2　文件使用大小的影响对比图
Figure 5.2　Impact of File Usage Size

能具有显著放大效应。即使在极端情况下，内存只能容纳单个文件的大部分内容，仍然需要一定比例的调度和回退操作，与 OverlayFS 相比，系统依然实现了 41.72% 的延迟降低，体现出内存加速机制在多种容量配置下均具有稳定收益。

5.2.3　文件操作次数的影响

本小节从操作频率角度出发，分析文件操作次数对 `read` 与 `write` 系统调用延迟的影响。实验中将单次工作负载的文件操作次数从 500 逐步增加到 500000，并分别在 `Ephemera`、原生 OverlayFS 以及挂载 tmpfs 的环境中运行相同工作负载，以对比不同系统的延迟差异。图 5.3展示了随文件操作次数变化的延迟对比结果。

当文件操作次数较少时，内存加速的优势尚未完全体现，此时系统相对 OverlayFS 的延迟降低约为 28.03%。随着文件操作次数持续增加，内存命中率提高、底层磁盘访问次数显著减少，系统的性能优势被逐步放大，在操作次数较高的配置下，延迟降低最高可达 53.04%。这一现象反映出建立内存映射与元数据管理本身存在一定的固定开销，但在高频读写场景中，这部分开销能够被大量低延迟的内存访问所摊薄，从而体现出明显的整体性能收益。

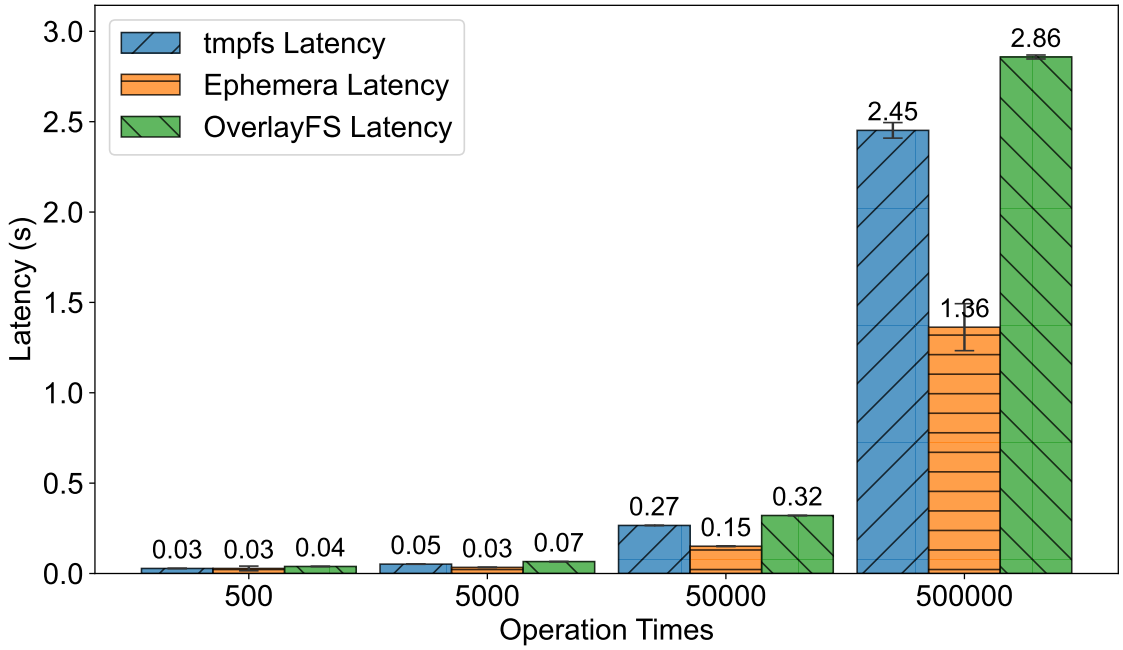


图 5.3 文件操作次数的影响对比图
Figure 5.3 Impact of File Operation Times

5.2.4 可扩展性分析

在图 5.1所示的内存限制实验中，可以观察到当内存限制设置为 32 MB 和 64 MB 时，挂载 tmpfs 的 Docker 实例由于可用内存不足而触发内存耗尽（OOM），导致实验失败，对应位置在图中以符号 X 标记。类似地，在图 5.2的文件大小实验中，当单个文件大小增加至 128 MB 和 160 MB 时，基于 tmpfs 的方案无法在给定内存限制下完成执行。与此形成对比的是，`Ephemera` 能够根据当前内存限制和正在访问的文件大小，动态调整内存文件系统的容量与使用策略，从而在保证不触发 OOM 的前提下持续提供服务。这一能力使系统在面对不同规模的工作负载和多变的内存约束时，表现出更好的可扩展性和鲁棒性，为在真实服务器无感知计算平台中部署多租户 I/O 加速机制提供了可行基础。

5.3 基准测试

本节基于 FunctionBench[58] 对系统进行系统化评估。FunctionBench 是一个面向服务器无感知计算平台的函数测试库，能够从 CPU、内存、磁盘与网络等多个维度评估平台性能。在众多函数中，本论文选取了三个与磁盘 I/O 性能紧密相关的基准，

用于突出系统在内存加速 I/O 方面的优势：压缩性能评估、顺序读写性能评估和随机读写性能评估。

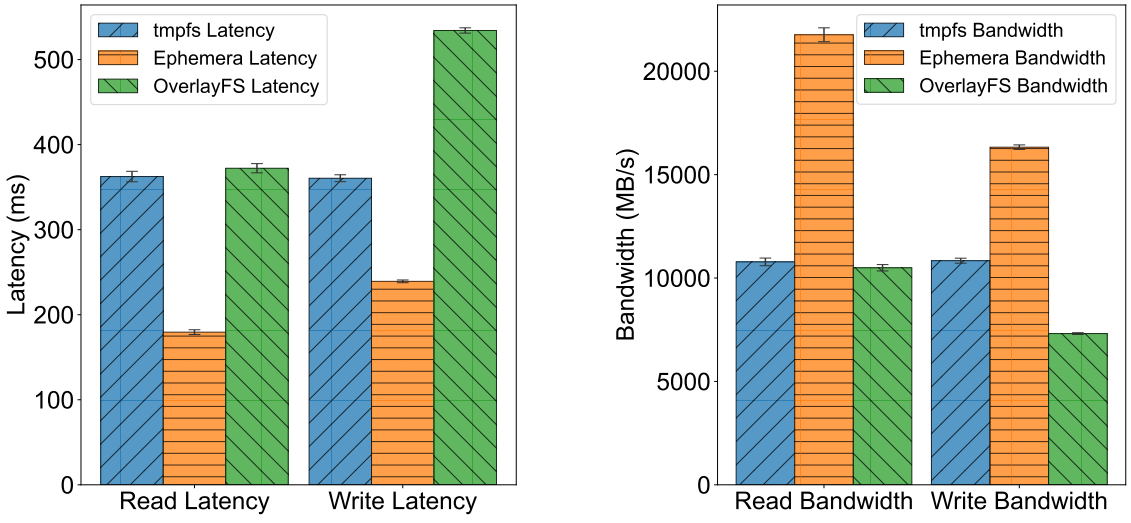
5.3.1　随机与顺序磁盘 I/O 性能

在随机磁盘 I/O 性能评估中，测试函数针对一个大小为 32 MB 的文件进行读写操作。函数首先向文件写入数据，然后通过 `lseek` 随机调整文件偏移量，重复上述写入过程 500000 次；随后，以相同大小从文件中读取数据，同样在每次读操作之间通过 `lseek` 随机调整偏移量，并重复读取过程 500000 次。这种方式模拟了典型的随机读写场景，对底层文件系统的寻址开销和缓存机制提出了较大挑战。

在顺序磁盘 I/O 性能评估中，文件以顺序方式被连续读写直至文件末尾，再通过 `lseek` 将偏移量重置到文件起始位置。与随机读写不同，顺序访问能够更好地利用底层磁盘和操作系统缓存的预取能力，是许多日志写入、顺序扫描类应用的常见访问模式。

为了避免内存限制成为基准测试的瓶颈，本论文在这一组实验中将单个函数容器的内存限制设置为 150 MB，使得所有参与评估的文件都可以完全驻留在内存之中，从而更加突出不同系统本身的差异。

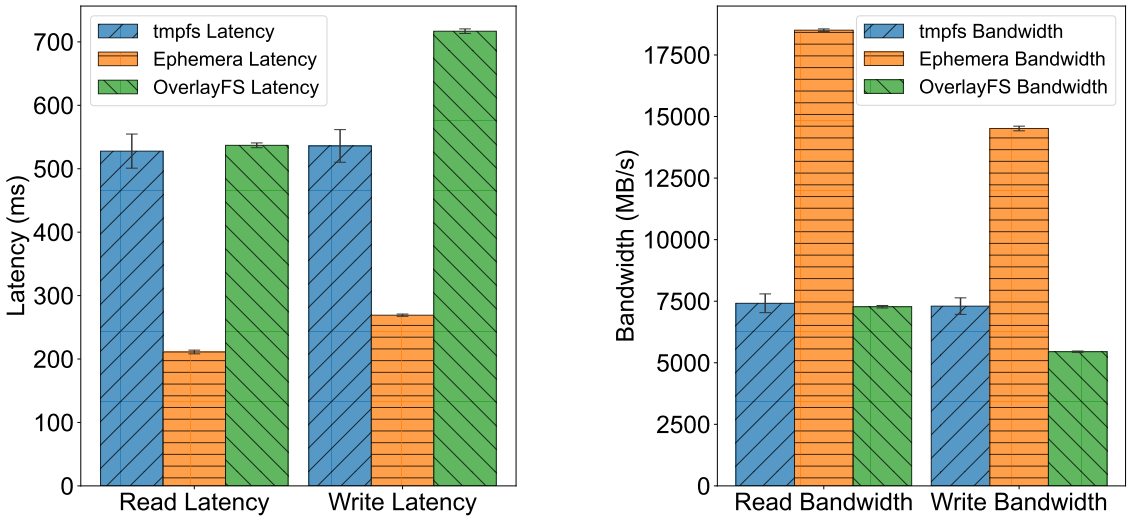
该基准主要考察两个方面：一是写入磁盘时的端到端延迟，二是读取磁盘时的端



(a) 延迟
(a) Latency

(b) 带宽
(b) Bandwidth

图 5.4　顺序磁盘 I/O 性能对比图
Figure 5.4　Sequential Disk I/O Performance



(a) 延迟
(a) Latency

(b) 带宽
(b) Bandwidth

图 5.5 随机磁盘 I/O 性能对比图
Figure 5.5 Random Disk I/O Performance

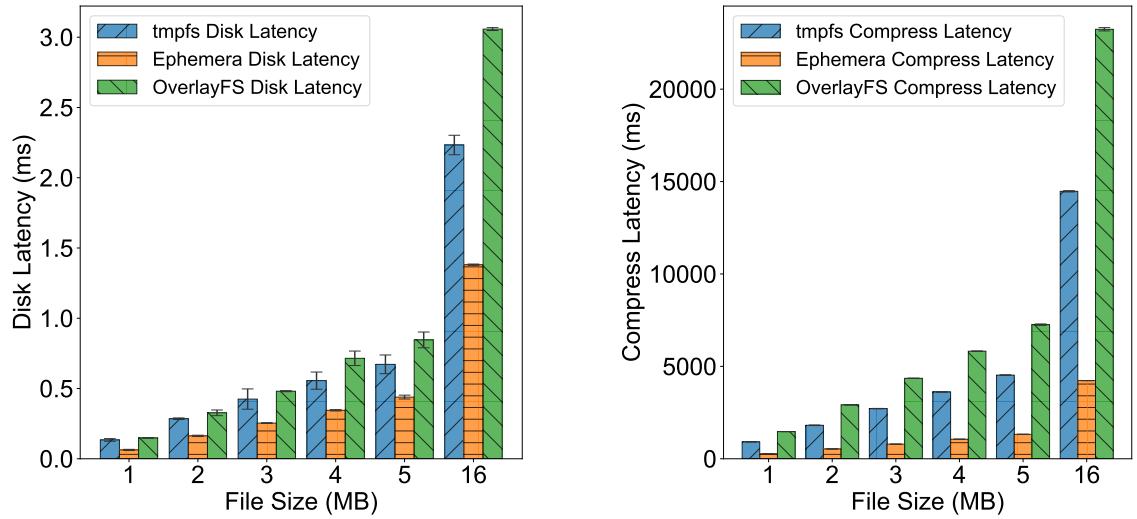
到端延迟。图 5.4与图 5.5的结果表明，在所有方案中，顺序读写操作的性能均优于随机读写操作。这是因为在顺序访问模式下，CPU 缓存与底层预取机制能够较高效地工作，即便数据已经被提升到内存层面，顺序访问仍然可以减少缓存未命中与地址转换带来的额外开销。

在顺序读写场景中，系统相较于基于 OverlayFS 的实现可获得约 55% 的延迟降低；而在随机读写场景中，这一延迟降低幅度进一步提升至约 62%。随着访问模式由顺序转向随机，传统基于磁盘或 OverlayFS 的实现更容易暴露出寻址和缓存方面的弱点，而内存文件系统受到的影响相对较小，因此相对性能提升更加明显。从结果可以看出，随机读写场景更有助于凸显基于内存的 I/O 加速优势。

5.3.2 压缩性能评估

在压缩性能基准中，测试函数首先将 1 MB 至 16 MB 不等大小的内容写入到文件中，随后多次读取该文件内容，使用压缩算法对数据进行压缩，并将压缩后的结果写入归档文件。这一过程会产生大量密集的文件读写操作，是典型的 I/O 与计算交织场景。

为了使所有测试文件都能完全驻留在内存当中，本论文将函数容器的内存限制设置为 64 MB，保证压缩过程不会因为内存不足而被外部因素干扰。该基准分别测



(a) 磁盘写入阶段延迟
(a) Disk Latency

(b) 压缩阶段延迟
(b) Compress Latency

图 5.6　压缩性能对比图
Figure 5.6　Compression Performance

量写入文件时的延迟（磁盘延迟）以及执行压缩阶段时的延迟（压缩延迟）。

从图 5.6可以观察到，相较于 OverlayFS，系统在磁盘延迟方面实现了大约 50% 的改善，而在压缩延迟方面的改善幅度更为显著，可达到约 81.8%。其根本原因在于压缩算法需要对同一批数据进行多次读取和处理，传统方案下频繁的文件 I/O 会引入大量系统调用与用户态、内核态之间的切换；利用内存文件系统和 mmap 等机制后，数据可以更高效地在内存中被反复访问，从而显著降低 I/O 开销。当 I/O 操作在总体运行时间中占比越高，压缩阶段的收益也就越突出。

5.3.3　真实工作负载

为了评估系统在更贴近实际应用场景中的表现，本论文进一步选取图像处理、MapReduce、机器学习推理和视频处理这四类工作负载进行对比实验。

图 5.7展示了 `Ephemera`、OverlayFS 与 tmpfs 在这四类真实工作负载上的端到端延迟对比结果。与 OverlayFS 相比，系统在图像处理、MapReduce、机器学习推理和视频处理四类任务上的延迟分别降低了 47.2%、92.0%、19.9% 和 44.3%。这种提升主要来源于内存文件系统对文件访问的加速作用：频繁的中间结果写入与读取不再经过慢速磁盘路径，而是直接在内存中完成，从而显著缩短了 I/O 等待时间。

与 tmpfs 相比，`Ephemera` 在上述四类负载上的延迟分别降低了 27.3%、88.5%、

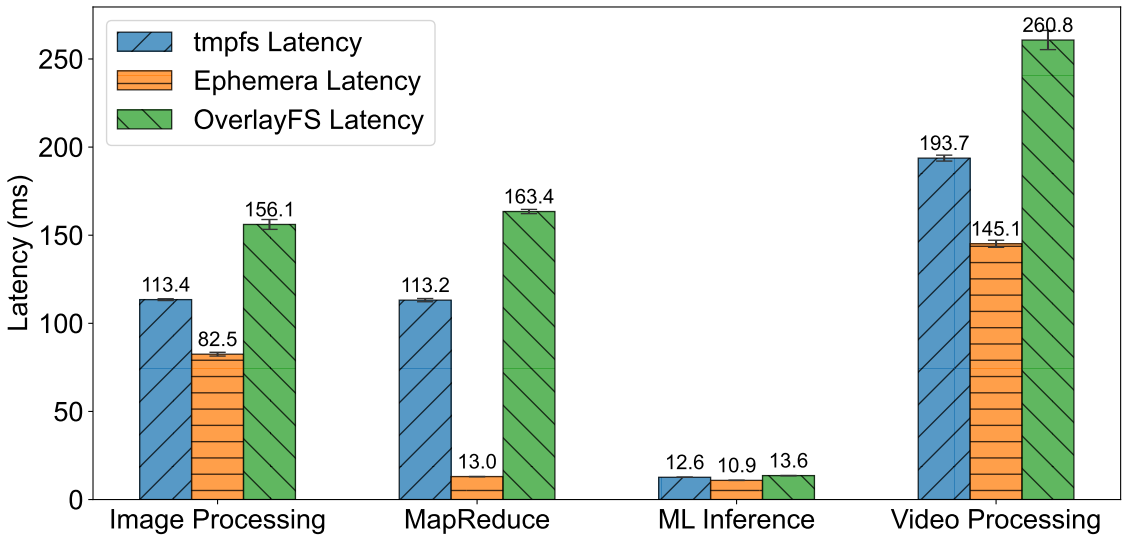


图 5.7 真实工作负载性能对比图
Figure 5.7 Workloads Performance

13.9% 和 25.1%。尽管 tmpfs 同样将数据存放在内存中，但系统利用 mmap 等机制减少了用户空间与内核空间之间的上下文切换次数，降低了系统调用开销，使得整体 I/O 路径更加轻量。在 MapReduce 和视频处理等 I/O 较为密集的负载中，这种优势尤为明显。

5.3.4 I/O 强度的影响

结合图 5.7与表 5.2可以观察到一个整体趋势：工作负载的 I/O 强度越高，系统带来的延迟降低幅度通常越显著。这一现象与系统的设计目标相一致：系统主要针对 I/O 路径进行优化，并未对纯计算过程进行专门加速，因此在文件操作占比较大的应用中更容易体现出明显的性能收益。

需要注意的是，I/O 强度与延迟降低之间并非简单的线性关系。例如，顺序磁盘 I/O 任务的 I/O 强度比随机磁盘 I/O 任务高约 2.11%，但从图 5.4与图 5.5的对比来看，顺序 I/O 场景下延迟降低约为 55%，低于随机 I/O 场景下约 62% 的降低幅度。这表明，除了 I/O 总量之外，具体的 I/O 访问模式（顺序还是随机）同样会对最终的性能收益产生重要影响。

5.4　内存共享机制验证

本节主要验证节点层面内存共享机制在实际运行中的有效性，以及其对不同类型函数任务的影响。

在系统中，租户管理器会根据任务的计算与内存特征，将容器划分到两类资源池中：一类是倾向于“出让”多余内存资源的收割池，另一类是需要额外内存资源以提升 I/O 性能的分配池。当函数实例启动时，租户管理器会根据其所属资源池动态调整容器的内存上限，从而在节点内部不同函数间实现内存共享与再分配。

为了考察这一机制的效果，本节选取了两个代表性任务：一个是为了追求更高计算能力而等比分配了较大内存的 CPU 密集型任务，即大整数阶乘计算；另一个是对内存容量敏感、需要更多内存空间的 I/O 密集型任务，该任务与第5.2节中所使用的 I/O 基准函数保持一致。CPU 密集型任务主要消耗 CPU 资源，对 I/O 与内存文件系统的依赖相对较弱；而 I/O 密集型任务则需要大量文件读写，对内存文件系统的可用空间高度敏感。

实验从三个维度进行对比：

1. 使用原始 OverlayFS 作为文件系统基线；
2. 使用内存文件系统但不启用内存共享机制；
3. 同时启用内存文件系统与节点层面的内存共享机制。

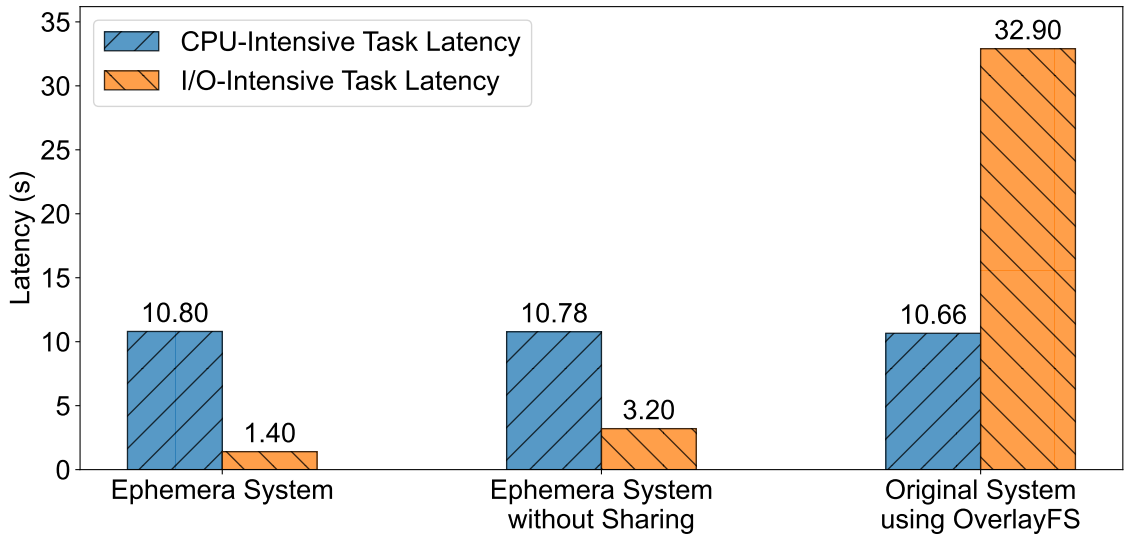


图 5.8　内存共享机制有效性对比图
Figure 5.8　Efficacy of Memory Sharing Mechanism

图 5.8展示了上述三种配置下 CPU 密集型任务与 I/O 密集型任务的端到端延迟。结果表明：在启用内存共享机制前，CPU 密集型任务的性能几乎不受内存文件系统与共享机制的影响；其主要瓶颈在于算术运算本身，因此无论是否通过内存共享调整容器内存上限，延迟变化都可以忽略不计。

对于 I/O 密集型任务，在没有内存共享的前提下，仅依靠内存文件系统，即可相较于基于 OverlayFS 的实现减少约 90.27% 的延迟。这是因为 I/O 路径被完全迁移到内存中，绕开了磁盘与 OverlayFS 带来的额外开销。

在此基础上进一步启用内存共享机制后，I/O 密集型任务可以从同一节点上 CPU 密集型任务所在的容器中收割更多可用内存，从而显著扩展内存文件系统的可用空间，减少因空间不足导致的回退与写回开销。实验结果显示，此时 I/O 密集型任务的延迟降低幅度进一步提升到约 95.73%，比单纯使用内存文件系统时又有明显改善。

综上可以看出，内存共享机制对 CPU 密集型任务的影响微乎其微，但对 I/O 密集型任务的性能具有显著的正向作用。系统通过在节点内部对多余内存进行收割和再分配，使得不同类型任务能够在不增加物理内存的前提下获得更适配的内存配置，充分发挥内存文件系统的 I/O 加速能力。

5.5 不同内存共享机制对比

在验证单租户内存共享机制有效性的基础上，本节进一步对比不同粒度的内存共享方案，包括单函数共享机制、单工作流共享机制和单租户共享机制，并与不启用共享机制的情况进行比较。目标是说明不同共享粒度在多函数、多工作流环境下实现内存再分配的能力存在显著差异。

首先，单函数共享机制以“函数”为划分单位，租户管理器管理的容器池中仅包含某一个函数的多个并发实例。在这一机制下，内存共享仅发生在同一函数的不同实例之间。本节的实验中，本论文同时运行两个 I/O 密集型任务实例和两个 CPU 密集型任务实例，分别属于不同的函数。由于单函数共享机制只在同一函数的实例之间进行内存调配，因此 CPU 密集型任务实例多余的内存无法被 I/O 密集型任务实例利用，I/O 密集型任务在这一机制下仍然面临内存不足的问题。

其次，单工作流共享机制以“工作流”为粒度组织内存共享。在实验设计中，本论文构造了一个由 CPU 密集型任务和 I/O 密集型任务串联组成的工作流：首先执行 CPU 密集型任务，然后执行 I/O 密集型任务。在这种设置下，租户管理器管理的容器池中包含来自同一工作流的不同函数实例。理论上，工作流内部不同函数之间有机会

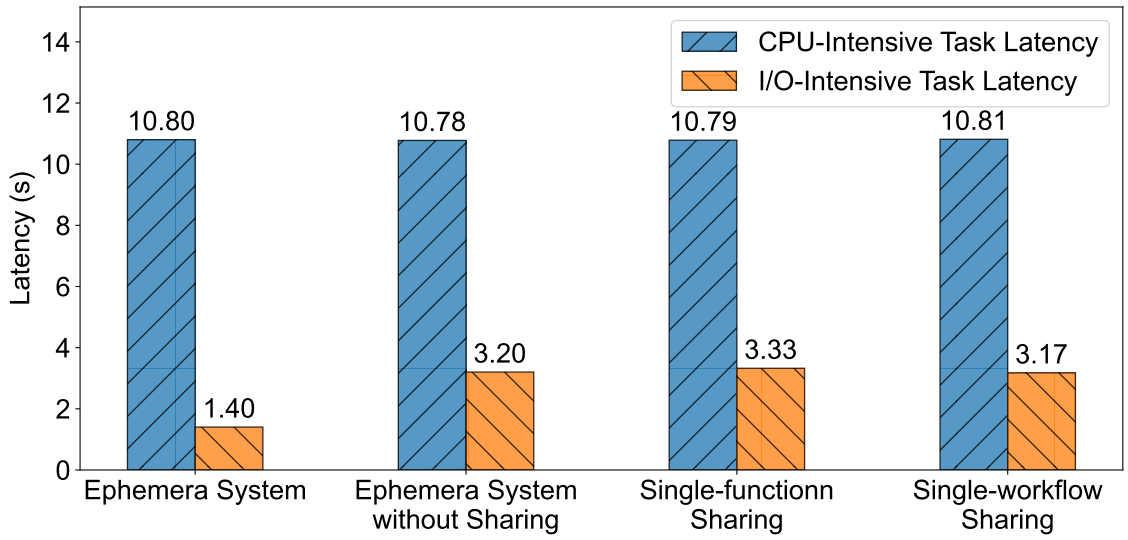


图 5.9　不同共享机制对比图
Figure 5.9　Sharing Mechanisms Comparison

共享内存资源。然而，由于实验中的工作流是严格顺序执行：当 I/O 密集型任务开始运行并加入分配池时，用于收割内存的 CPU 密集型任务已经完成并退出；因此在这一时间点上，收割池中不再存在可以出让内存的活跃实例，导致 I/O 密集型任务实际上无法从 CPU 密集型任务处获得额外内存，内存共享在时间维度上没有真正发生。

最后，单租户共享机制以“租户”为粒度，将同一租户下的所有函数实例统一纳入同一个容器池。无论是 CPU 密集型任务还是 I/O 密集型任务，只要同属一个租户且在同一节点上并发运行，就可以通过租户管理器在收割池与分配池之间共享内存。在这一机制下，多余的内存不再局限于同函数或同工作流内部流动，而是可以在租户内部更大范围内调配，从而显著提高内存利用率。

图 5.9展示了在无共享机制、单函数共享机制、单工作流共享机制和单租户共享机制四种场景下，CPU 密集型任务与 I/O 密集型任务的端到端延迟。从图中可以得出以下几点结论：

- CPU 密集型任务的延迟在四种场景中几乎保持不变。这是因为其瓶颈主要集中在算术运算和 CPU 计算本身，对内存文件系统容量和内存共享机制的依赖非常有限，因此即便参与收割池以出让部分内存，也不会对整体运行时间造成明显影响。
- I/O 密集型任务的延迟仅在单租户共享机制场景下表现出明显改善，与不启用任何共享机制的情形相比具有显著的性能提升。在单函数共享机制下，CPU 密

集型任务与 I/O 密集型任务属于不同函数，二者之间没有内存共享通道，I/O 密集型任务难以获取额外内存，因此性能与无共享机制接近。在单工作流共享机制下，尽管 CPU 密集型任务和 I/O 密集型任务处于同一工作流，但由于执行顺序的限制，它们在时间上并不并发运行，当 I/O 密集型任务需要内存时，CPU 密集型任务已结束，导致无法发生有效的内存转移。

- 单租户共享机制则突破了函数和工作流的边界，只要同一租户下存在并发运行的 CPU 密集型与 I/O 密集型任务，租户管理器就能在运行时动态调整各自的内存上限，将 CPU 密集型任务中未被充分利用的内存收割出来，分配给更需要内存的 I/O 密集型任务。正是这种更灵活的共享粒度，使得 I/O 密集型任务在单租户共享机制下的延迟显著低于其他三种场景。

综合以上分析可以看出，从单函数、单工作流到单租户，内存共享粒度逐步扩大，对不同类型任务之间的内存再分配能力也逐步增强。其中，单租户共享机制在兼顾隔离性与灵活性的前提下，能够最大程度地挖掘节点上闲置内存的潜力，为 I/O 密集型工作负载提供更稳定、持续的性能提升。

5.6 调度有效性验证

集群控制器中的调度器会综合考虑各节点当前的工作负载情况、待调度任务的内存需求以及任务的内存配置等信息，为每一个到达的请求选择合适的节点，从而在全局范围内获得尽可能优的整体性能。为了进行对比，本论文选取一种简单的基线策略：始终优先选择当前可用内存最多的节点作为任务的目标节点，即“内存最多优先”的工作负载均衡算法，将其作为基准方案。在此基础上，对比本系统所采用的调度策略与基线策略在工作负载分布与任务响应时间方面的差异。

本节采用固定到达率的任务流来模拟真实服务器无感知计算环境中的请求模式。具体地，本论文设定每秒到达六个任务的请求速率，持续一段时间，将这些请求交由调度器处理，并记录各节点上的工作负载分布及每个任务的端到端响应延迟。任务集合由同一租户下的四个 I/O 密集型任务和两个 CPU 密集型任务构成，部署在两个逻辑节点上，每个节点的总内存均为 1024 MB。在具体实现中，本论文在一台物理机器上分别启动两个独立的租户管理器实例，用以模拟两个节点的行为。

表 5.3展示了基线策略与本系统调度策略下各节点内存利用情况及不同类型任务的性能表现。与仅按可用内存排序的基线算法相比，本系统的工作负载均衡方法能够更加有效地平衡内存的分配和使用，在不损害 CPU 密集型任务性能的前提下，显著

表 5.3　节点工作负载状态
Table 5.3　Node Workload Condition

	节点 1			节点 2		
	需求内存	分配内存	使用内存	需求内存	分配内存	使用内存
Ephemera	778 MB	750 MB	748 MB	262 MB	450 MB	262 MB
基准方案	1032 MB	600 MB	600 MB	8 MB	600 MB	8 MB

提升 I/O 密集型任务的执行性能。

在基线配置中，由于调度器始终偏向当前可用内存最多的节点，缺少文件操作量层面的调度语义，导致节点一几乎完全被 I/O 密集型任务占据。然而节点一为每个 I/O 密集型任务提供的内存仍然不足，使得所有 I/O 密集型任务都面临内存紧张的问题，内存文件系统难以获得充足空间，性能受限明显。与此同时，节点二几乎全部由 CPU 密集型任务填满。CPU 密集型任务本身对内存需求不高，结果是在节点二上出现大量未被实际利用的空闲内存，造成严重的资源浪费。

与之相比，本系统的调度策略在任务调度时不仅考虑节点当前可用内存总量，还会参考任务类型以及任务在内存文件系统上的潜在加速收益，使得 CPU 密集型任务与 I/O 密集型任务更均衡地分布在两个节点上。在这种情况下，每个 I/O 密集型任务都有机会从同节点内的 CPU 密集型任务处“收割”额外的可用内存，从而扩大内存文件系统的空间，提高 I/O 性能。实验结果表明，采用本系统调度策略后，节点一上的内存浪费被压缩到极低水平，仅约 2 MB。造成这部分浪费的主要原因在于，虽然 CPU 密集型任务本身只消耗大约 4 MB 内存，但 Docker 对容器内存设置存在最小值限制，实际可分配的最小内存约为 6 MB，因此不可避免地出现 2 MB 左右的剩余空间。

从端到端延迟角度看，本系统的工作负载均衡方法显著改善了整体性能。与基准配置相比，所有任务的平均响应延迟降低了约 44.92%。需要注意的是，当前调度策略采用贪心式决策，对未来请求的到达模式缺乏预见能力。在某些时刻，这种“就近最优”的策略可能会错过为后续 I/O 密集型任务预留更多内存的机会，从而使得部分 I/O 密集型任务未能获得理论上可能的最大内存分配，潜在的性能优化空间尚未完全挖掘。尽管如此，在无全局未来信息的现实约束下，这种贪心策略仍然在复杂多变的负载条件下取得了较好的性能与实现复杂度之间的平衡。

5.7 系统开销测试

本节从运行时守护进程、租户管理器以及集群控制器三个方面，对系统的运行时开销和额外内存消耗进行定量分析，评估这些组件在实际部署环境中的成本。

5.7.1 运行时守护进程的开销

运行时守护进程的主要开销来源于对容器内存使用情况与内存文件系统状态的持续监控。当任务为 CPU 密集型时，其 I/O 访问较少，对内存文件系统依赖不强，此时引入 `Ephemera` 带来的额外收益有限，却需要承担一定的监控成本。为了量化这一影响，本论文选取一个 CPU 密集型的大整数阶乘计算任务作为测试对象，对比启用与不启用 `Ephemera` 时任务的执行延迟。

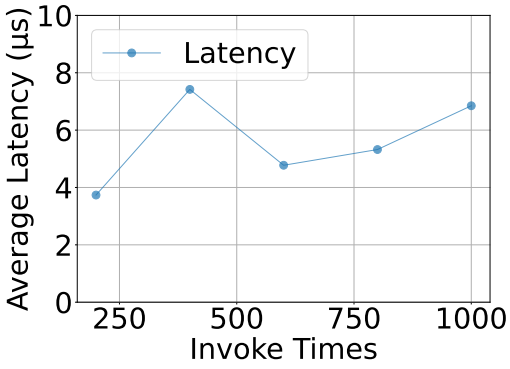
如图 5.8所示，当启用 `Ephemera` 之后，CPU 密集型任务的执行时间从 10.66 秒略微增加到 10.80 秒，对应的性能下降约为 1.29%。这一增幅主要来自运行时守护进程周期性检查内存使用状况、维护元数据结构等背景操作。整体来看，该开销在绝对值和相对比例上都相当有限，表明在 CPU 密集型场景下，即使无法直接从内存文件系统中获益，引入 `Ephemera` 也不会对性能造成明显影响。

5.7.2 租户管理器的开销

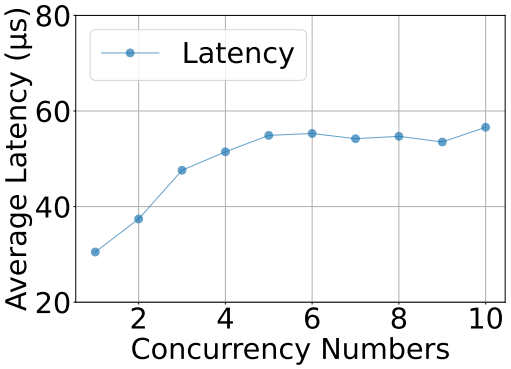
租户管理器的开销主要来自两部分：一是按需统计容器池中各容器的内存使用情况，用于判断是否具备分配额外内存或回收多余内存的条件；二是根据决策结果计算需要进行调整的容器数量，并与运行时守护进程协同完成具体的内存上限调节操作。

本论文将租户管理器的开销定义为：从其接收到调度器发来的函数调用请求，到将该请求转发至选定容器之间的平均延迟。为了考察请求到达频率对开销的影响，本论文在单个节点上同时部署一个 CPU 密集型阶乘计算任务和一个 I/O 密集型随机读写任务，并控制不同的请求速率进行实验。如图 5.10a所示，当请求在一秒内均匀到达，且请求数量从 200 增加到 1000 时，租户管理器的平均处理延迟几乎没有显著变化，始终维持在约 6 微秒左右。与函数的实际执行时间相比，该级别的延迟可以认为可以忽略不计。

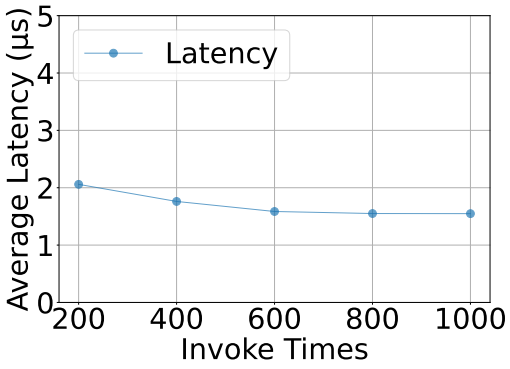
此外，本论文还进一步研究了并发实例数量对租户管理器开销的影响。图 5.10b显示，当并发实例数从 1 增加到 10 时，平均延迟先上升后趋于稳定，整体始终低于 60 微秒。这一现象的主要原因是：随着并发数增加，容器池中活跃的容器实例数量增多，租户管理器在做出一次内存调节决策时，需要遍历和计算的对象数量也随之增



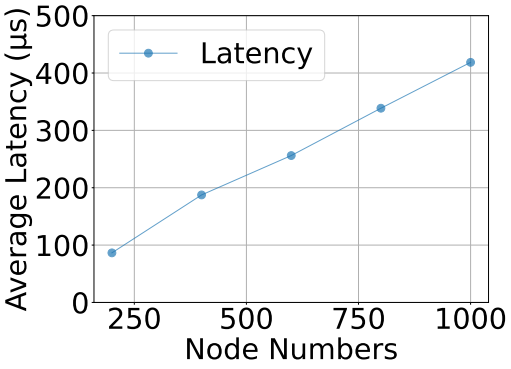
(a) 调用次数对租户管理器的影响
(a) Impact of Invoke Times on Tenant Manager



(b) 并发数对租户管理器的影响
(b) Impact of Concurrency on Tenant Manager



(c) 调用次数对集群控制器的影响
(c) Impact of Invoke Times on Cluster Controller



(d) 节点数对集群控制器的影响
(d) Impact of Node Number on Cluster Controller

图 5.10　`Ephemera` 的开销图
Figure 5.10　Overhead of `Ephemera`

加，从而在初始阶段导致平均延迟上升。但当并发实例进一步增多后，容器已经分配或收割到足够的内存，进入相对稳定状态，租户管理器不再频繁调整内存配置，额外计算次数减少，因此整体延迟趋于平稳。

5.7.3　集群控制器的开销

集群控制器的开销主要来自调度器对各节点工作负载状态的收集与分析。本论文将集群控制器的开销定义为：从调度器接收到上层发来的函数调用请求，到将该请求分配至具体目标节点之间的平均延迟。通过调节节点数量和请求频率，可以观察开销在不同规模和负载强度下的变化。

图 5.10c展示了在节点数固定为 2、请求在 1 秒内均匀到达的情况下，请求量从

200 增加到 1000 时调度器的平均延迟。实验结果表明，随着请求数量增多，平均延迟始终维持在约 2 微秒左右，并未出现明显的增长趋势，说明调度器在请求压力增加的情况下依然能够保持稳定、轻量的处理开销。

图 5.10d则反映了节点规模对调度延迟的影响。在保持请求数量为 200 的前提下，随着节点数从 200 持续扩展到 1000，平均延迟逐步增加，当节点数达到 1000 时，平均调度延迟约为 418 微秒。与典型服务器无感知计算函数的执行时间（通常为几十毫秒乃至数百毫秒）相比，这一数量级的延迟仍然可以认为是可忽略的。从趋势上看，调度开销随节点数近似线性增长，这是因为调度器需要检查每个节点当前的负载状况与可用资源，节点越多，遍历和比较所需的时间也越长。

5.7.4 内存开销分析

考虑到 Ephemera 本身也会消耗一定内存资源，本节最后对各组件的内存占用情况进行简要分析。

运行时守护进程的内存开销主要来自其维护的文件元数据结构，例如文件到内存页的映射关系、打开文件描述符状态等。在实验设置下，这部分开销始终保持在 1 MB 以内，相比节点总内存而言可以忽略不计。

租户管理器需要维护容器池中所有容器的状态信息，包括每个容器的当前内存上限、实际使用量、所属租户及资源池类别等元数据。这些信息以轻量级结构存储在内存中，实验测得的内存占用同样不足 1 MB，对整体系统并无显著影响。

集群控制器则需要暂时缓存每个节点的摘要信息，例如节点可用内存、活跃容器数量、历史负载统计等。在本实验规模下，这部分信息的内存占用也低于 1 MB。因此，从整体上看，Ephemera 在运行时引入的额外内存消耗相当有限，不会对租户正常使用的内存额度造成可感知的压缩，也不会成为系统扩展到更多节点和更多租户时的瓶颈。

5.8 本章小结

本章对 Ephemera 的性能进行了全面的实验评估。首先，自定义基准测试结果表明，Ephemera 在不同内存限制、文件大小和操作频率下均表现出优于 OverlayFS 的 I/O 性能。特别是在内存受限场景下，延迟降低最高可达 92.67%，并且相比 tmpfs 具有更好的可扩展性，有效避免了内存溢出问题。其次，基于 FunctionBench 和真实工作负载的测试进一步证实了系统在实际应用场景中的有效性。在 MapReduce 等高

I/O 强度任务中，延迟降低达到了 92.0%；在压缩任务中，压缩延迟改善了约 81.8%。此外，本论文验证了内存共享机制能够有效利用空闲内存资源。实验显示，开启共享机制后，I/O 密集型任务的延迟降低进一步提升至 95.73%，而对 CPU 密集型任务的性能影响仅为 1.29%。调度策略的评估显示，本论文的负载均衡算法能够合理分配节点资源，使任务平均延迟降低了 44.92%，同时将内存浪费控制在极低水平。最后，系统开销测试表明，`Ephemera` 引入的额外延迟和内存消耗均在可忽略范围内，证明了系统的高效性和实用性。

第 6 章 结论与展望

6.1 全文总结

本文围绕服务器无感知计算在处理 I/O 密集型任务时面临的性能瓶颈与资源浪费问题，提出并设计了存储优化框架 Ephemera。该框架通过整合函数级透明 I/O 重定向、节点级动态内存共享与集群级工作负载均衡，系统性地解决了当前服务器无感知计算平台中内存利用率低、文件访问延迟高以及资源分布不均衡等关键问题，为提升服务器无感知计算工作负载的整体性能提供了一种可扩展、低侵入的优化方案。

首先，本文通过对 AWS Lambda 与阿里云函数计算等主流平台的实测分析，指出了服务器无感知计算中普遍存在的资源供给与实际负载不匹配问题，即内存过度配置。进一步的实验对比表明，内存级文件访问在延迟和带宽方面明显优于传统磁盘路径，这一认识为利用闲置内存构建临时高速存储层奠定了理论与工程基础。

其次，针对上述观察，本文提出的 Ephemera 框架采用了分层式架构设计：在函数级，通过透明拦截机制将用户代码中的文件 I/O 自动重定向至内存文件系统，不需要对用户程序做任何修改；在节点级，通过共享内存池支持多实例间的高效数据协同，在保证隔离性的前提下显著提升内存资源利用率；在集群级，通过分析运行时 I/O 特征和节点状态，动态调整任务调度策略，减少热点节点拥塞，提升整体集群的负载均衡性。通过三层协同，Ephemera 将节点内部闲置内存转化为可用资源，并构建了一条高效、低延迟的数据访问路径。

最后，本文实现了 Ephemera 的完整原型系统，并从延迟、带宽、可扩展性、共享效率与系统开销等多个维度开展了系统评估。实验结果表明，与传统文件系统相比，Ephemera 能够将文件操作延迟平均降低 50%，最高可达到 95.73%，并在多实例场景中有效提升节点与集群层面的内存利用率。同时，系统的运行开销较低，不会对原有函数执行造成明显负担，体现了框架的可部署性与实用价值。

6.2 研究展望

内存文件系统的操作拦截能力 作为原型系统，当前的内存文件系统已实现了核心 I/O 操作的支持，涵盖文件的创建、读取及写入。为了实现 Ephemera 的实际部署，未来的工作将致力于扩展指令拦截的覆盖范围，以满足更多样化的应用需求。此外，

运行时守护进程作为函数实例的封装器（Wrapper），目前提供了针对 C 语言 I/O 操作的拦截接口。该架构设计具有良好的可扩展性，为未来适配多种编程语言奠定了基础。

针对输入敏感任务的分析　在服务器无感知计算领域，现有工作[38,62-63] 致力于根据输入数据预测函数的实际执行行为与资源消耗。此类方法与本文提出的方案在设计维度上是正交且互补的。通过集成上述预测技术，系统能够依据实时输入数据动态预估内存占用与文件 I/O 需求，从而为服务器无感知计算函数提供更加细粒度且高效的内存共享机制。

内存共享机制的扩展　当前租户管理器的内存共享机制局限于单节点范围，这在一定程度上制约了系统的可扩展性与灵活性。为了突破这一瓶颈，未来的研究将致力于探索分布式架构设计，重点利用远程直接内存访问（Remote Direct Memory Access, RDMA）技术实现低延迟的跨节点内存访问[64-66]。集成 RDMA 技术将赋予 `Ephemera` 在集群环境中构建全局内存池的能力，从而显著提升资源管理的效率以及系统的整体吞吐量与扩展性。

参考文献

[1] ARMBRUST M, FOX A, GRIFFITH R, et al. A View of Cloud Computing[J]. Communications of the ACM, 2010, 53(4): 50-58.

[2] FOSTER I, ZHAO Y, RAICU I, et al. Cloud Computing and Grid Computing 360-Degree Compared[C]//Grid Computing Environments Workshop. 2008: 1-10.

[3] 温金凤, 陈震鹏, 柳熠, 等. 服务器无感知平台性能度量研究[J]. 软件学报, 2025, 36(05): 1974-2005.

[4] 吴逸文, 张洋, 王涛, 等. 从 Docker 容器看容器技术的发展: 一种系统文献综述的视角[J]. 软件学报, 2023, 34(12): 5527-5551.

[5] MAMPAGE A, KARUNASEKERA S, BUYYA R. A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions[J]. ACM Computing Surveys, 2022, 54(11s): 1-36.

[6] LI Z, GUO L, CHENG J, et al. The Serverless Computing Survey: A Technical Primer for Design Architecture[J]. ACM Computing Surveys, 2022, 54(10s): 1-34.

[7] SHAFIEI H, KHONSARI A, MOUSAVI P. Serverless Computing: A Survey of Opportunities, Challenges, and Applications[J]. ACM Computing Surveys, 2022, 54(11s): 1-32.

[8] JONAS E, SCHLEIER-SMITH J, SREEKANTI V, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing[J]. arXiv preprint arXiv:1902.03383, 2019.

[9] MADNI S H H, LATIFF M S A, COULIBALY Y, et al. Resource Scheduling for Infrastructure as a Service (IaaS) in Cloud Computing[J]. Journal of Network and Computer Applications, 2016, 68(C): 173-200.

[10] LIU X, WEN J, CHEN Z, et al. FaaSLight: General Application-level Cold-start Latency Optimization for Function-as-a-Service in Serverless Computing[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(5): 1-29.

[11] PAN S, ZHAO H, CAI Z, et al. Sustainable Serverless Computing with Cold-Start Optimization and Automatic Workflow Resource Scheduling[J]. IEEE Transactions on Sustainable Computing, 2024, 9(3): 329-340.

[12] GOLEC M, WALIA G K, KUMAR M, et al. Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions[J]. ACM Computing Surveys, 2024, 57(3): 1-36.

[13] ZHAO H, PAN S, CAI Z, et al. faaShark: An End-to-End Network Traffic Analysis System Atop Serverless Computing[J]. IEEE Transactions on Network Science and Engineering, 2024, 11(3): 2473-2484.

[14] ASLANPOUR M S, TOOSI A N, CHEEMA M A, et al. Faashouse: Sustainable Serverless Edge Computing Through Energy-Aware Resource Scheduling[J]. IEEE Transactions on Services Com-

puting, 2024, 17(4): 1533-1547.

[15] DU D, LIU Q, JIANG X, et al. Serverless Computing on Heterogeneous Computers[C]//Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 797-813.

[16] CAI Z, CHEN Z, MA R, et al. SMSS: Stateful Model Serving in Metaverse With Serverless Computing and GPU Sharing[J]. IEEE Journal on Selected Areas in Communications, 2024, 42(3): 799-811.

[17] BHASI V M, SHARMA A, MOHANTY S, et al. Paldia: Enabling SLO-Compliant and Cost-Effective Serverless Computing on Heterogeneous Hardware[C]//IEEE International Parallel and Distributed Processing Symposium. 2024: 100-113.

[18] LI Z, XU C, CHEN Q, et al. DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration[C]//Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4. 2024: 57-72.

[19] WANG W, WU Q, ZHANG Z, et al. A Probabilistic Modeling and Evolutionary Optimization Approach for Serverless Workflow Configuration[J]. Software: Practice and Experience, 2024, 54(9): 1697-1713.

[20] RAZA A, AKHTAR N, ISAHAGIAN V, et al. Configuration and Placement of Serverless Applications Using Statistical Learning[J]. IEEE Transactions on Network and Service Management, 2023, 20(2): 1065-1077.

[21] JIN L, CAI Z, CHEN Z, et al. AARC: Automated Affinity-Aware Resource Configuration for Serverless Workflows[C]//Proceedings of the Annual ACM/IEEE Design Automation Conference. 2025: 1-7.

[22] 杨光, 刘杰, 曲慕子, 等. 服务器无感知计算系统性能优化技术研究综述[J]. 软件学报, 2025, 36(1): 47.

[23] COPIK M, BÖHRINGER R, CALOTOIU A, et al. FMI: Fast and Cheap Message Passing for Serverless Functions[C]//Proceedings of the ACM International Conference on Supercomputing. 2023: 373-385.

[24] KIENER M, CHADHA M, GERNDT M. Towards Demystifying Intra-Function Parallelism in Serverless Computing[C]//Proceedings of the International Workshop on Serverless Computing. 2021: 42-49.

[25] YANG Y, ZHAO L, LI Y, et al. INFless: a Native Serverless System for Low-Latency, High-Throughput Inference[C]//Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 768-781.

[26] MERENSTEIN A, TARASOV V, ANWAR A, et al. F3: Serving Files Efficiently in Serverless Computing[C]//Proceedings of the ACM International Conference on Systems and Storage. 2023: 8-21.

[27] SCHLEIER-SMITH J, HOLZ L, PEMBERTON N, et al. A FaaS File System for Serverless Com-

puting[J]. arXiv preprint arXiv:2009.09845, 2020.

[28] LI Z, CHENG J, CHEN Q, et al. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing[C]//USENIX Annual Technical Conference. 2022: 53-68.

[29] HATTORI J, KATO S. Sentinel: a fast and memory-efficient serverless architecture for lightweight applications[C]//Proceedings of the International Workshop on Serverless Computing. 2022: 13-18.

[30] YANG Z, HOSEINZADEH M, ANDREWS A, et al. AutoTiering: Automatic Data Placement Manager in Multi-tier All-flash Datacenter[C]//IEEE International Performance Computing and Communications Conference. 2017: 1-8.

[31] NARAYANAN D, HODSON O. Whole-system Persistence[C]//Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. 2012: 401-410.

[32] DULLOOR S R, ROY A, ZHAO Z, et al. Data Tiering in Heterogeneous Memory Systems[C]// Proceedings of the European Conference on Computer Systems. 2016: 1-16.

[33] AGARWAL N, WENISCH T F. Thermostat: Application-transparent Page Management for Two-tiered Main Memory[C]//Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. 2017: 631-644.

[34] ZHENG S, HOSEINZADEH M, SWANSON S. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks[C]//USENIX Conference on File and Storage Technologies. 2019: 207-219.

[35] YU H, WANG H, LI J, et al. Accelerating Serverless Computing by Harvesting Idle Resources[C]//Proceedings of the ACM Web Conference. 2022: 1741-1751.

[36] WANG Y, ARYA K, KOGIAS M, et al. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud[C]//Proceedings of the European Conference on Computer Systems. 2021: 1-16.

[37] FUERST A, NOVAKOVIĆ S, GOIRI Í, et al. Memory-Harvesting VMs in Cloud Platforms[C]//Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 583-594.

[38] YU H, FONTENOT C, WANG H, et al. Libra: Harvesting Idle Resources Safely and Timely in Serverless Clusters[C]//Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing. 2023: 181-194.

[39] ZHANG Y, GOIRI Í, CHAUDHRY G I, et al. Faster and Cheaper Serverless Computing on Harvested Resources[C]//Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles. 2021: 724-739.

[40] YU H, WANG H, LI J, et al. Freyr$^+$: Harvesting Idle Resources in Serverless Computing via Deep Reinforcement Learning[J]. IEEE Transactions on Parallel and Distributed Systems, 2024, 35(11): 2254-2269.

[41] LIU J, CAI Z, LIU Y, et al. SMore: Enhancing GPU Utilization in Deep Learning Clusters by Serverless-Based Co-Location Scheduling[J]. IEEE Transactions on Parallel and Distributed Systems, 2025, 36(5): 903-917.

[42] VERMA A, PEDROSA L, KORUPOLU M, et al. Large-Scale Cluster Management at Google with Borg[C]//Proceedings of the European Conference on Computer Systems. 2015: 1-17.

[43] SCHWARZKOPF M, KONWINSKI A, ABD-EL-MALEK M, et al. Omega: Flexible, Scalable Schedulers for Large Compute Clusters[C]//Proceedings of the ACM European Conference on Computer Systems. 2013: 351-364.

[44] HINDMAN B, KONWINSKI A, ZAHARIA M, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center[C]//Proceedings of the USENIX Conference on Networked Systems Design and Implementation. 2011: 295-308.

[45] VAVILAPALLI V K, MURTHY A C, DOUGLAS C, et al. Apache Hadoop YARN: Yet Another Resource Negotiator[C]//Proceedings of the Annual Symposium on Cloud Computing. 2013: 1-16.

[46] GHODSI A, ZAHARIA M, HINDMAN B, et al. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types[C]//Proceedings of the USENIX Conference on Networked Systems Design and Implementation. 2011: 323-336.

[47] BURNS B, GRANT B, OPPENHEIMER D, et al. Borg, Omega, and Kubernetes[J]. Communications of the ACM, 2016, 59(5): 50-57.

[48] AL-DHURAIBI Y, PARAISO F, DJARALLAH N, et al. Elasticity in Cloud Computing: State of the Art and Research Challenges[J]. IEEE Transactions on Services Computing, 2018, 11(2): 430-447.

[49] MAO H, ALIZADEH M, MENACHE I, et al. Resource Management with Deep Reinforcement Learning[C]//Proceedings of the ACM Workshop on Hot Topics in Networks. 2016: 50-56.

[50] 周凯, 王凯, 朱宇航, 等. 基于 GMM 的容器定制化调度策略[J]. 计算机科学, 2025, 52(6): 346-354.

[51] 金鑫, 吴秉阳, 刘方岳, 等. 服务器无感知计算场景下基于时空特征的函数调度[J]. 计算机研究与发展, 2023, 60(9): 2000-2014.

[52] 张信民, 李星儒, 樊浩, 等. 面向服务器无感知计算的可定制函数调度[J]. 中国科学: 信息科学, 2025, 55(3): 481-499.

[53] AKKUS I E, CHEN R, RIMAC I, et al. SAND: Towards High-Performance Serverless Computing[C]//USENIX Annual Technical Conference. 2018: 923-935.

[54] LI Z, LIU Y, GUO L, et al. FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service[C]//Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 782-796.

[55] JIA Z, WITCHEL E. Nightcore: Efficient and Scalable Serverless Computing for Latency-

Sensitive, Interactive Microservices[C]//Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021: 152-166.

[56] SINGHVI A, BALASUBRAMANIAN A, HOUCK K, et al. Atoll: A Scalable Low-Latency Serverless Platform[C]//Proceedings of the ACM Symposium on Cloud Computing. 2021: 138-152.

[57] AGACHE A, BROOKER M, IORDACHE A, et al. Firecracker: Lightweight Virtualization for Serverless Applications[C]//USENIX Symposium on Networked Systems Design and Implementation. 2020: 419-434.

[58] KIM J, LEE K. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service[C]//IEEE International Conference on Cloud Computing. 2019: 502-504.

[59] STOJKOVIC J, XU T, FRANKE H, et al. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency[C]//Proceedings of the Annual International Symposium on Computer Architecture. 2023: 1-15.

[60] MAHGOUB A, YI E B, SHANKAR K, et al. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs[C]//USENIX Symposium on Operating Systems Design and Implementation. 2022: 303-320.

[61] MAHGOUB A, YI E B, SHANKAR K, et al. WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2022, 6(2): 1-28.

[62] BHASI V M, GUNASEKARAN J R, SHARMA A, et al. Cypress: Input Size-Sensitive Container Provisioning and Request Scheduling for Serverless Platforms[C]//Proceedings of the Symposium on Cloud Computing. 2022: 257-272.

[63] MOGHIMI A, HATTORI J, LI A, et al. Parrotfish: Parametric Regression for Optimizing Serverless Functions[C]//Proceedings of the ACM Symposium on Cloud Computing. 2023: 177-192.

[64] BAI W, ABDEEN S S, AGRAWAL A, et al. Empowering Azure Storage with RDMA[C]//USENIX Symposium on Networked Systems Design and Implementation. 2023: 49-67.

[65] LU F, WEI X, HUANG Z, et al. Serialization/Deserialization-free State Transfer in Serverless Workflows[C]//Proceedings of the European Conference on Computer Systems. 2024: 132-147.

[66] HUANG J, ZHANG M, MA T, et al. TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes[C]//Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles. 2024: 421-437.

致 谢

硕士的两年半匆匆过去，我要向许多人致以最真挚的感谢。

首先，我要感谢我的导师管海兵和指导老师马汝辉为我提供了良好的学术环境和实验设备，让我能够顺利完成我的研究工作。

其次，我要感谢我的家人。无论是在学业的压力下，还是生活的琐碎中，他们始终是我最坚实的后盾。这份无条件的信任与关爱，给予了我勇往直前的力量。同时也要感谢我的朋友们，那些欢笑与倾听，让这段漫长的科研之路不再孤单。

此外，我还要感谢实验室的各位同门。科研从来不是孤军奋战，感谢大家在组会中提出的真知灼见，这些建议不仅完善了我的实验方案，也拓宽了我的研究视野。实验室里互助互励的氛围，是我科研路上最重要的动力源泉。

最后，向所有曾给予我温暖与帮助的人致以诚挚的敬意。这段旅程虽然告一段落，但你们给予我的力量将伴随我开启新的人生篇章。

学术论文和科研成果目录

学术论文

[1] JIN L, CAI Z, CHEN Z, et al. AARC: Automated Affinity-aware Resource Configuration for Serverless Workflows[C]//Proceedings of the Annual ACM/IEEE Design Automation Conference. 2025: 1-7.

[2] JIN L, CAI Z, WANG H, et al. Ephemera: Accelerating I/O-Intensive Serverless Workloads with a Harvested In-Memory File System[J]. ACM Transactions on Architecture and Code Optimization, 2025, 22(3): 1-24.