高频数据访问场景下的单体架构性能优化

刘中一[1,2],李梦箫[1,2],阎松柏[1,2]
(1. 中国民航信息网络股份有限公司,北京 101318;
2. 民航旅客服务智能化应用技术重点实验室,北京 101318)

摘　要:单体架构以其数据与程序皆在本地的独特部署方式,获得了其他架构难以比拟的高效与稳定,因此在云时代仍有一席之地,非常适用于面对海量数据的高频数据访问场景,但也存在着单机资源容量难以应对数据增长、服务启动时因数据加载等资源消耗大导致性能抖动、服务或数据更新后因数据变冷带来的交易时间骤升等一系列问题。该文的优化方案中通过使用定址文件映射共享内存技术降低单机内存容量需求、引入中间控制层完成数据和应用解耦、利用写时复制技术实现资源加载成本最小化、利用硬件内存管理单元与页表加热来加速内存访问效率等多项技术,使得上述问题得到有效缓解。实际验证中,单体架构在能够支持更高数据容量的同时,兼顾了数据的快速加载,维持了交易响应时间的稳定性,并且仍然可以保持单体架构的性能优势。这种改良的设计使单体架构的适用范围得到进一步扩大,为有海量数据和高频数据访问需求的实时交易系统提供了一种通用的可选方案。

关键词:单体架构;文件映射;共享内存;高频数据访问;写时复制
中图分类号:TP391　　**文献标识码**:A　　**文章编号**:1673-629X(2024)12-0009-07
doi:10.20165/j.cnki.ISSN1673-629X.2024.0247

Performance Optimizations of Monolithic Architecture in High-frequency Data Access System

LIU Zhong-yi[1,2], LI Meng-xiao[1,2], YAN Song-bai[1,2]
(1. TravelSky Technology Limited, Beijing 101318, China;
2. Key Laboratory of Intelligent Passenger Service of Civil Aviation, Beijing 101318, China)

Abstract: With its unique way of deploying data and programs locally, the monolithic architecture achieves efficiency and stability that other architectures cannot match. Therefore, it still has a place in the cloud era and is quite suitable for high-frequency data access scenarios facing massive data. However, there are also a series of problems such as the resource capacity of a single machine being difficult to cope with data growth, performance jitters due to high resource consumption such as data loading when starting a service, and a sudden increase in transaction time due to cold data after service or data updates. In this paper, optimization solutions are proposed to effectively alleviate these problems. These solutions include reducing the memory capacity requirements of a single machine by using memory mapping with specified address and shared memory technology, decoupling data and applications through the introduction of an intermediate control layer, minimizing resource loading costs through copy-on-write technology, and accelerating memory access efficiency through hardware memory management units and page table heating. In practical validation, the monolithic architecture not only supports higher data capacity but also maintains stable transaction response times while ensuring rapid data loading. Furthermore, it still retains the performance advantages of the monolithic architecture. This improved design expands the applicability of the monolithic architecture, providing a universal optional solution for real-time transaction systems with massive data and high-frequency data access requirements.

Key words: monolithic architecture; memory mapping; shared memory; high-frequency data access; copy-on-write

0　引　言

互联网与大数据技术的蓬勃发展,给许多依赖高频数据访问的服务后台处理系统带来了挑战,此场景具有高度复杂的业务逻辑与极高的数据读取频率,又

收稿日期:2024-04-07　　**修回日期**:2024-08-13
基金项目:国家自然科学基金(U2033203)
作者简介:刘中一(1987-),男,硕士,高级工程师,研究方向为民航客票运价系统、密集计算系统;通讯作者:李梦箫(1990-),男,研究方向为民航客票运价系统。

有严苛的性能要求与巨量的并发压力。以民航某高性能高频数据访问计算系统为例,一次交易平均需要访问数据 10^3 ~ 10^4 次以上,要求在 1 ~ 2 秒之内返回处理结果,且系统整体需要支撑 10^4 ~ 10^5 以上的业务峰值并发访问量。当前流行的分布式/微服务架构,很难满足此场景极致的性能要求[1]。在数据访问方面,如若缺乏精巧复杂的本地缓存机制,10^3 ~ 10^4 次跨网络数据访问将造成系统响应时间过长,同时集群节点堆叠而成的 10^6 ~ 10^8 次数据访问并发,也将给数据库带来难以承受的访问压力。由此,尽量减少程序访问数据的损耗成本与资源占用,以达到提升速度分散压力的目的,是此类应用架构选型的关键点,单体架构是其中一种可行的选择。

单体架构是将系统进行业务处理用到的业务表现层、业务逻辑层、数据访问层等都打包成一个应用,连同业务数据一起整合后,部署在单台业务服务器上[2]。单体架构的出现时间很早,并随着分布式架构、微服务架构和云计算的发展,作为反例被不断诟病[3-4]。但随着计算机硬件设备的不断更新换代,特别是大容量 SSD(Solid State Drive,固态硬盘)、PM(Persistent Memory,持久化内存)/SCM(Storage Class Memory,存储级内存)等 NVM(Non-Volatile Memory,非易失性存储器)设备的面世、成本的降低及应用的扩大[5],单体架构性能优越、扩展简便的优势便再次凸显,重新成为特定场景下后台应用的可选架构之一。然而,单体架构由于其本身应用与数据耦合的特点,天然存在一些短板,主要体现在以下三点:

(1)从资源使用角度看,数据与应用耦合且共占共享服务器资源,二者的变化会互相感知并互相影响,极端情况下可能因兼容性问题造成服务中断。

(2)从数据更新角度看,在大数据量或缓存情况下,本地应用需要花费较长时间进行数据读取、构建与加载,此过程会抢占同机交易进程的硬件资源,可能会造成服务性能变差。

(3)从应用维护角度看,应用本身在正常更新、故障重启等情况下,也需要重新进行数据预读预热,此过程耗费时间、影响交易,容易造成波动。

因此,在选择单体架构时,必须通过设计改进扬长避短,尽量发挥其优势,妥善规避或解决其劣势。在此领域国内外的研究成果较少,基本都集中在结合业务加快数据预热、提高缓存命中率[6-7]等方面。该文设计了一种单体架构的通用改良方案,利用多种优化手段在单台业务服务器内解决了应用与数据之间的过度耦合,实现了应用与数据高效更新与平稳切换,成果已在多个高性能密集计算系统中得到验证,具有较高的实践意义与推广价值。

1 关键技术

方案设计整体以突出单体架构性能优势、降低高耦合性缺陷与性能波动为目标。在数据访问环节,采用了基于文件映射的共享内存技术,访问速度方面具有显著优势,并依托 Linux 系统的内存缓存机制充分削减了真实数据访问量,并能够支持更高的数据总量与访问并发;在进程控制环节,引入了中间控制层,将管理进程与工作进程分开,在加速进程切换的同时,也为服务的管理、更新及扩容提供了便利;在数据加载环节,通过 Linux 系统的 COW(Copy-on-Write,写时复制)技术大幅减少数据加载中的资源消耗,通过 MMU(Memory Management Unit,内存管理单元)硬件预热页表提升内存访问效率,以实现应用与数据的平稳切换。

1.1 采用定址文件映射突破单体架构容量限制

目前主流的单台服务器通常已是多 CPU 多计算核心的配置,部署多个应用进程服务可以充分利用服务器的 CPU 资源。内存访问是目前除 CPU 寄存器与高速缓存外最快的数据访问方式,而共享内存可以实现一份数据在多个进程间的共享,兼具了内存访问的高速度和多进程共享存储区的便利性,是一种综合性价比高的数据存储与访问方式。

共享内存在 Linux 平台的实现方式有两种,一种通过 System V 标准接口中的 shm 系统调用族实现[8],一种通过系统调用 mmap 映射文件机制实现[9](Windows 平台的 file mapping 原理与此相近)。

shm 机制可以实现多服务进程之间的内存共享,也支持复杂对象访问,但其限制为需要提前定义共享内存大小及申请内存资源,并需将所有数据加载到内存中。一旦要进一步支持数据多版本存储或服务快速切换场景,则需要占用不少于数据量两倍的内存,对计算服务器的硬件需求较高。除此之外 shm 数据是非持久化的,管理共享内存的进程异常退出会有数据丢失的风险。

为解决 shm 机制的弊端,该文使用基于文件映射的共享内存数据库[10],将复杂的业务对象直接转换成文件方式映射到固定地址的共享内存中(见图 1),这种内存数据库具有极高的数据存取效率和支持复杂对象存取[11],其主要优点在于:

(1)指定固定地址进行 mmap,可以将数据按照业务或技术维度拆分为多个文件,做到单机多进程并行加载甚至多机分布式数据构建处理而不会引发地址空间冲突,同集群的多机可以同样的固定地址对所有文件进行 mmap 映射,即可完成远程整体数据的装载还原,实现“一处构建,多处使用”。另外,基于 NVMM(Non-Volatile Main Memory,非易失性主内存)的文

件系统支持新的 DAX(Direct Access,直接访问)mmap,可表现更高的写入性能[12-13],从而具备更快的构建性能。

(2)mmap 文件映射不需要提前把所有的数据都存放在内存中,可以充分使用 Linux 系统的 Cache 机制,对于区分实时/历史数据及冷/热数据明显的应用来说,可以节省大量内存空间,优化内存空间之间的数据交换[14],共享内存访问的局部性理论及访存依赖相关研究也证明使用共享内存可以很好地支持并发程序,进而提升计算效率[15]。使用这项技术能够让单体架构使用远超过物理内存极限的数据总量,突破单体架构难以纵向扩容的限制,易于通过并发程序设计提高性能。

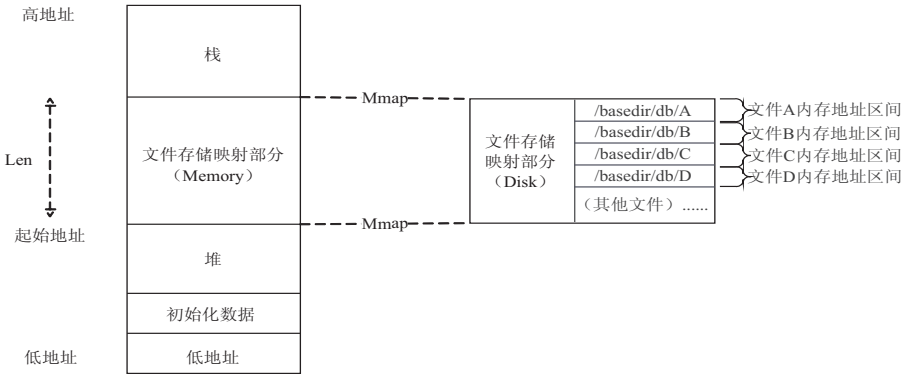

图 1　mmap 映射文件加载机制

1.2　引入中间控制进程降低数据加载资源消耗

对于单体架构而言,在多 CPU 的单台服务器上部署多个应用服务进程,可以充分利用服务器上的 CPU 资源,因此单体架构中一般需要进行进程或资源管理。以常见的交易中间件为例,其架构往往是双层,一层为管理进程,一层为交易进程,生成交易进程时与业务或资源无关,由交易进程自主完成交易或计算所需资源的初始化,典型例子如 Oracle Tuxedo,其进程结构如图 2 所示。

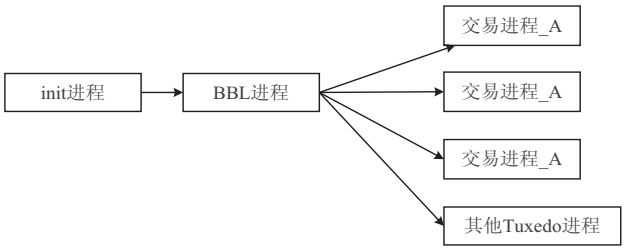

图 2　Tuxedo 进程结构

而在 1.1 节提到的场景中,本地大量数据加载即需要占用一定时间与大量硬件资源,在此前提下如若多个应用服务进程串行数据加载,则整体加载完成、服务预期就绪时间会变长;但如果换成并行加载的控制思路,则会在短时间内产生大量的资源挤兑,影响当前交易效率,出现交易延时或超时等问题。

为避免数据加载或其他资源初始化带来的高成本,该文引入一层中间控制层进程进行资源初始化,之后再借助 Linux 系统调用 fork 完成大量交易进程的复制派生。fork 是 Linux 上的一种创建子进程的方式,它在创建子进程的时候采用了 Copy-on-Write 机制[16],子进程的代码段、数据段、堆栈都是指向父进程的物理空间,当子进程中有更改相应段的行为发生时才会再为子进程相应的段去分配实际的物理空间,著名的内存数据库 Redis 也是利用 fork 快速创建子进程来进行持久化落盘[17]。除了内存资源外,父子进程还可以共享其他资源,如文件句柄(包含 Socket 套接字)等。

对于访问只读类型数据、拥有大量数据的后台系统,通过 fork 创建多个子进程不仅可以实现服务进程的快速创建,由 Copy-on-Write 技术创建的子进程还将共享父进程所有的内存数据资源,配合文件映射共享内存技术,使得多个子进程在生成时就获取了其父进程文件与内存之间的映射关系,即已经完成了数据初始化与加载处理,无需重复读取数据文件进行映射。

就 Linux 的进程管理而言,得益于父子进程之间的天然联系,二者之间可以使用 Linux 本身的进程管理和通信机制来完成控制逻辑,完成进程的创建、监管及退出等一系列操作。从这一角度来看,引入的中间控制进程是一个合适的管理进程和监控点,也是未来集群管理的切入点与扩展点。

1.3　利用内存管理单元提高共享内存访问效率

Linux 操作系统通过页表来完成虚拟地址到物理地址的映射关系。实际的进程使用的都是虚拟内存,由 MMU 根据页表最终完成虚拟地址到物理地址的连接。对于虚拟地址来说,刚完成 malloc 接口或 mmap 系统调用申请到的虚拟内存页的状态都是“已分配未映射”状态,并未和物理内存映射起来,如图 3 所示。随后的业务交易中使用到这个内存页的时候会产生缺页中断,接着进入内核态为其分配物理内存页面,填充

物理内存页面中的内容,最后在页表中建立映射关系,之后的内存访问均在用户态中进行。因此当刚完成 mmap 映射的数据加载之后,如果处于高并发的请求服务状态,可能短期内会产生大量的缺页中断(page fault)进而导致系统交易时间出现明显衰减,事实上,抛开需要磁盘 IO 参与的主缺页中断(major page fault),有研究证明就算是访问数据已在物理内存中而只是缺乏和虚拟内存之间映射的次缺页中断(minor page fault)也会消耗几千个 CPU 周期来处理[18]。利用 MMU 及优化页表访问也在被持续深度研究[19-20],比如提升内存页大小来降低内存碎片提供性能[21],使用了扁平化页表来提升内存访问效率[22]。

该文通过虚拟地址和既有硬件 MMU 来共享文件页表,加速定位共享文件数据页,提升数据访问效率,此特性也被利用在某些高效共享内存文件系统的设计中,能够支持虚拟机间的高效内存共享。因此,可以先在父进程中进行页表加热,提前完成热点/当前数据的缺页中断处理,之后 fork 创建子进程的时候,内核会通过 copy_process 将父进程的所有资源拷贝到子进程中,其中涵盖了父进程的虚拟地址空间的所有内容包括页表,最后内核会为子进程分配其独立的顶级页表起始地址,这样多个服务子进程的共享内存访问效率都得到了提升。

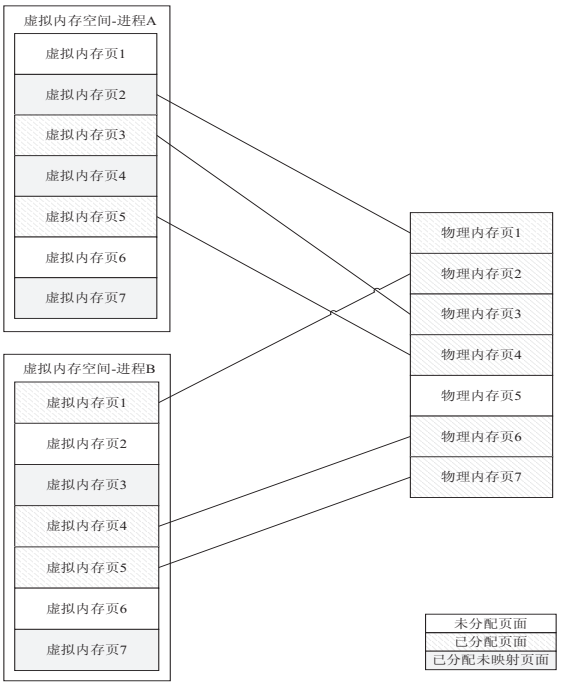

图 3 虚拟内存到物理内存映射机制(页表)

2 方案实现

方案设计上,应用进程结构从传统的双层变为三层,即 Master/Controller/Worker 层。其中 Controller 为 Master 的子进程,Worker 又是 Controller 的子进程,整体进程结构如图 4 所示。

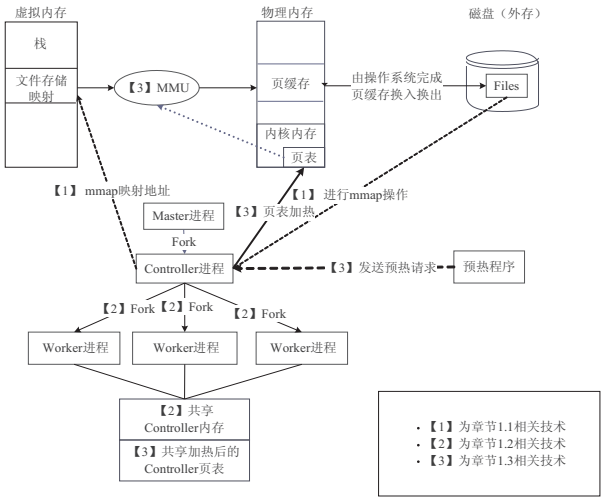

图 4 整体方案设计图

其中 Master 作为控制进程和守护进程创建 Controller 进程(即章节 1.2 描述的中间控制进程),利用章节 1.1 中的技术将磁盘上的数据加载到自己的虚拟地址空间;依据章节 1.1 中的内存访问局部性原理和章节 1.3 的技术原理,通过预热程序发送预热请求到 Controller 进程,可以确保热点数据被操作系统纳入页缓存,并顺势完成页表加热,进而利用 MMU 硬件实现内存访问加速。接下来 Controller 使用 fork 创建 Worker 进程,直接共享 Controller 的虚拟内存和页表,负责处理请求完成业务,Master/Controller/Worker 的主要接口如表 1 所示。

表 1 Master/Controller/Worker 接口列表

角色	接口	说明
Master	StartController	启动 Controller 子进程
	StopController	关闭 Controller 子进程
	Status	心跳访问,返回管理的 Controller 信息(包含监听端口、数据版本、应用版本等)
Controller	LoadData	进行数据映射和构建
	UnLoadData	关闭数据映射和析构
	BindListenPort	绑定请求处理的监听端口
	CloseListenPort	关闭请求处理的监听端口
	CreateWorkers	创建 Workers 子进程
	StopWorkers	停止 Workers 子进程
Worker	ProcessRequest	进行实际的业务数据请求

在系统环境中,最典型场景是数据批次更新完成后进行应用切换的场景,应用更新的场景与此类似:

(1)Master 获知新数据已就绪,开始切换流程。当前环境中已存在提供服务的 Controller 与 Worker。Master 进行 StartController 操作,派生出新 Controller,进行新数据的加载和数据预处理。

(2)新 Controller 根据 Master 进程的传入信息,使用 mmap 映射新的数据文件。

(3)完成数据文件映射后,Controller 绑定一个用于处理请求的服务端口。此服务端口的 socket 将在 fork 子进程之前完成,以便于后期子进程 Worker 自动继承业务端口,统一接受来自同端口的业务请求。

(4)Controller fork 出 Worker 子进程,Worker 自动继承了 Controller 的虚拟地址空间,此时 Worker 子进程自动集成 Controller 的虚拟地址空间,自动完成了数据文件的映射。

(5)当新的 Controller 对外提供的服务端口可以提供服务时,Master 执行 StopController 操作,将旧的 Controller 及其子进程 Workers 退出。

其关键设计主要在于中间控制进程 Controller 的引入,在中间控制层进行 mmap 映射及热点/当前数据页表加热,可以让 Controller 接下来生成的所有 Worker 子进程数据处于已加热状态,因而其切换后的交易不会出现性能下降。另外,在双层结构中,数据装载过程由位于第二层的交易进程直接进行,一旦发生数据版本不符或格式有误的异常情况,会造成交易进程无法服务甚至服务崩溃,而三层结构下会由全新生成的 Controller 进行数据装载,即使失败也不会影响由已有的 Controller 和 Worker 进程支撑的交易。通过使用多套 Master 层支持多应用版本,多 Controller 层支持多套数据版本,既解耦了应用和数据的过度耦合,避免出现服务中断,又能够支持一致性切换、灰度上线等各种灵活的服务策略。这种设计也可以在一定程度上降低 Controller 及 Worker 在代码上存在内存泄漏或句柄泄漏问题的严重性。

3　效果验证

3.1　采用定址文件映射技术的优化效果

根据文献[10]中有关于 mmap 内存映射相较于其他常用数据库的读取效率比对,结果证明在读取普通的 key-value 结构时,mmap 相较于嵌入式的数据库 berkeley DB 有超过 10 倍的性能优势,相较于 Redis 的读取响应时间更是提升了上百倍。而当读取复杂对象的时候,这个性能优势会更加明显。在需要极高数据读取频率的业务场景中,可以节约大量数据访问的开销。另外,由于 mmap 的特性,无需将所有数据都加载到内存中即可完成内存映射,因此具备极高的加载和更新效率。以某高性能高频数据访问与密集计算系统为例,加载多套数据库,每次加载完成后向其发送来自生产系统一个月的交易请求约 280 万个,最终其加载数据服务的情况和每次交易进程的 RSS(Resident Set Size,常驻内存大小)情况见表 2。验证环境为:Intel Xeon(R) CPU E5-2667 16 核,768 GB 内存,Red Hat 7.3 操作系统,加载完成共启动 80 个服务进程。

表 2　数据更新中应用启动耗时

数据版本	文件数量	文件体积/GB	启动时间/s	交易进程 RSS 内存大小/GB
20231210_002502	82 355	5 167	15.8	127
20231210_014819	82 364	5 168	15.7	128
20231210_020548	82 356	5 168	15.3	128
20231210_033108	82 535	5 236	18.3	131

该实验结果证明了定址文件映射技术不但数据加载效率高,还可装载远大于物理内存的数据,如使用 shm 机制的共享内存方案,能够加载的数据量无法超过实验机物理内存大小,即 768 G;而本例中加载的文件体积最高为 5 236 G,已达到物理机内存的 6.8 倍;实际上,对于大部分总体数据量很大的业务系统,其热点数据比例一般都不高,在本例中统计完成整月请求后交易进程的常驻内存大小,其平均值只占到总数据量的约 2.5%。

3.2　使用中间控制进程完成数据装载的优化效果

在使用 mmap 映射数据的场景下,由于 mmap 是内核态的系统调用,三层进程结构本身只需要二层完成一次数据装载,然后 fork 出三层进程即可,因此其数据装载的系统调用次数和两层结构相比为 1:N(N 为服务进程数量)。实验验证环境为:Intel Xeon(R) CPU E7-8891 80 核,12 TB 内存,Red Hat 7.3 操作系统,加载完成共启动 80 个服务进程,即 N 为 80;实验模拟了两次数据更新,分别在传统二层结构及优化后的三层结构中进行压力测试,获取响应时间曲线并观察性能抖动情况,采样时间为两秒一次,结果如图 5 所示。

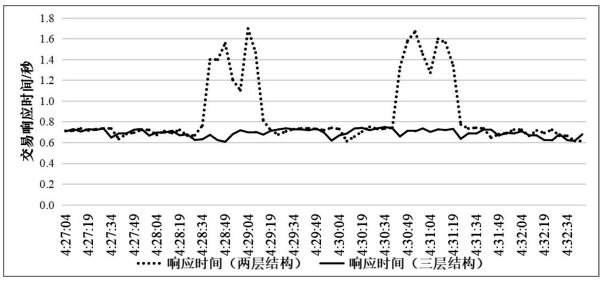

图 5　数据装载过程性能稳定性对比

通过 NMON 工具分别统计双层和三层结构下数据装载过程中的 CPU 使用率情况,采样间隔为每秒一次,其结果如图 6 所示。

从图 5 和图 6 中可以观察到的是,两层结构相对于三层结构而言,其数据装载过程会消耗大量的 CPU 资源在内核态的系统调用上,这也会实时地反映在交易的响应时间上,而三层结构受益于“mmap+fork”机

制,大大减少了 mmap 系统调用次数,其数据装载的稳定性显著优于两层结构。

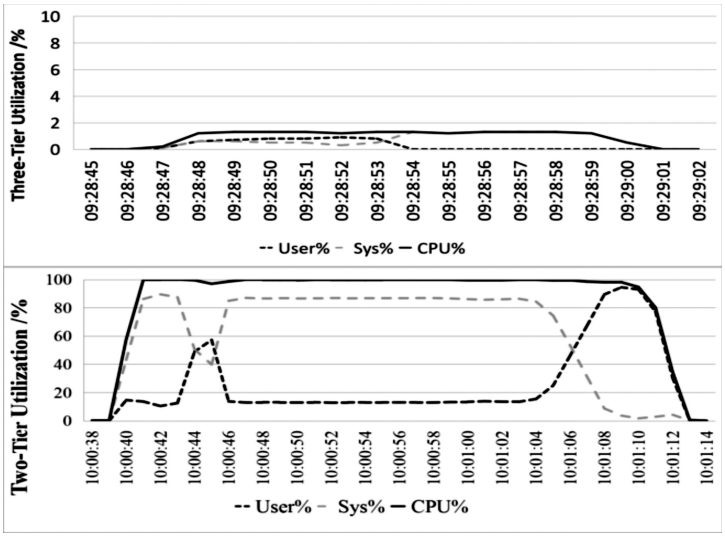

图 6 数据装载过程 CPU 状态对比

3.3 通过硬件 MMU 提高内存访问效率的优化效果

对于 1.3 中提到的在二层进程进行页表加热来利用 MMU 加速内存访问效率,避免大量缺页中断导致的内核态阻塞场景。下面的实验分别测试了二层进程进行加热再 fork 三层与二层不加热直接 fork 三层的场景下的响应时间对比,页表加热方式是数据装载后先预访问部分数据以触发页表完成映射,测试共使用 200 个请求以 80 并发去调用服务,避免过高的交易量导致后续请求全部命中已映射页表从而稀释响应时间差异,实验测试了三轮取平均值,其结果如表 3 所示。

表 3 页表加热对响应时间的影响

测试轮次	二层预加热响应时间/s	二层不加热响应时间/s	相差/%
1	0.582	0.819	28.94
2	0.570	0.815	30.12
3	0.568	0.846	32.92

结果可见页表映射是否提前建立,对交易响应时间的影响平均能达到 30% 左右,根据 1.3 中的原理可知,每个进程都有独立的页表,利用三层结构可以通过 fork 来拷贝页表的特性,通过简单的一次二层加热,让所有三层服务进程都可以提前拥有预先建立好的页表,进一步让数据更新和版本切换过渡的更加平滑。即使三层的服务进程发生异常退出,二层进程作为共享内存当前状态的一个快照,也可以保证新 fork 的三层服务进程仍然具备和之前的三层服务进程相同的数据一致性及内存热度。

4 结束语

综上所述,使用 mmap 文件映射共享内存的技术方案可以在很短的时间内完成大量的内存数据映射,比传统内存数据库技术支持更高数据量下的高性能计算。通过使用写时复制和 fork 的特性,也使得大量的工作进程可以在短时间内启动完毕,降低了新服务加载的成本。另外数据库同步或更新的过程可以完成内存缓存预热,并借由硬件 MMU 提速内存访问效率,使得三层服务进程在后续实际交易处理中遇到缺页异常的概率大大降低,令数据装载和流量切换过程中的性能表现的更加稳定。

这种通过多项技术方案改良的单体架构设计,提高了单体架构能够支撑的本地数据量,提升了数据加载和服务就绪速度,稳定了数据切换性能抖动,进一步拓宽了单体架构的能力上限和应用范围。

参考文献:

[1] AL-DEBAGY O,MARTINEK P. A comparative review of microservices and monolithic architectures[C]//2018 IEEE 18th international symposium on computational intelligence and informatics (CINTI). Budapest:IEEE,2018:149-154.

[2] MOSLEH M,DALILI K,HEYDARI B. Distributed or monolithic? A computational architecture decision framework[J]. IEEE Systems Journal,2016,12(1):125-136.

[3] NITIN R,ANAND R. Evaluating the performance of monolithic and microservices architectures in an edge computing environment[J]. International Journal of Fog Computing,2022,5(1):1-18.

[4] VILLAMIZAR M,GARCES O,OCHOA L,et al. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures[C]//2016 16th IEEE/ACM international symposium on cluster,cloud and grid computing (CCGrid). Cartagena:IEEE,2016:179-182.

[5] BLINOWSKI G,OJDOWSKA A,PRZYBYŁEK A. Monolithic vs. microservice architecture:a performance and scal-

ability evaluation[J]. IEEE Access,2022,10(1):20357-20374.

[6] ALSHAREF A,JAIN P,ARORA M,et al. Cache memory:an analysis on performance issues[C]//2021 8th international conference on computing for sustainable global development (INDIACom). New Delhi:IEEE,2021:184-188.

[7] GU R,LI C,DAI H,et al. Improving in-memory file system reading performance by fine-grained user-space cache mechanisms[J]. Journal of Systems Architecture,2021,115(1):101994.

[8] DUTT H. IPC through shared memory[M]//Interprocess communication with macOS:apple IPC methods. California:Apress Berkeley,2021:11-35.

[9] PAPAGIANNIS A,XANTHAKIS G,SALOUSTROS G,et al. Optimizing memory-mapped {I/O} for fast storage devices[C]//2020 USENIX annual technical conference. Boston:USENIX ATC,2020:813-827.

[10] 黄向平,彭明田,杨永凯. 基于内存映射文件的复杂对象快速读取方法[J]. 计算机技术与发展,2020,30(3):82-87.

[11] 梁海峰,杨　毅,刘中一. 面向复杂对象的高性能内存映射数据库 MMDB[J]. 计算机工程与设计,2023,44(3):937-944.

[12] LI S,LI D,WU D,et al. NVMFS-IOzone:performance evaluation for the new NVMM-based file systems[C]//Proceedings of the 13th ACM international systems and storage conference. Haifa:ACM,2020:87-97.

[13] IWATA S. An investigation of performance problems with msync() system calls on filesystem DAX[C]//Proceedings of the 14th ACM international conference on systems and storage. Haifa:ACM,2021:1.

[14] GU Y,FAN M,ZHAO Y,et al. The upgrade of data processing and storage system for EAST NBI[J]. Fusion Engineering and Design,2021,173(1):112849.

[15] 蒋炎岩. 并发程序共享内存访问依赖研究[D]. 南京:南京大学,2017.

[16] HA M,KIM S H. CCoW:optimizing copy-on-write considering the spatial locality in workloads[J]. Electronics,2022,11(3):461.

[17] PARK J,LEE Y,YEOM H Y,et al. Memory efficient fork-based checkpointing mechanism for in-memory database systems[C]//Proceedings of the 35th annual ACM symposium on applied computing. Brno:ACM,2020:420-427.

[18] TIRUMALASETTY C,CHOU C C,REDDY N,et al. Reducing minor page fault overheads through enhanced page walker[J]. ACM Transactions on Architecture and Code Optimization,2022,19(4):1-26.

[19] HAZARIKA A,PODDAR S,RAHAMAN H. Survey on memory management techniques in heterogeneous computing systems[J]. IET Computers & Digital Techniques,2020,14(2):47-60.

[20] HUR J Y. Contiguity representation in page table for memory management units[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems,2019,27(1):147-158.

[21] PARK C H,CHA S,KIM B,et al. Perforated page:supporting fragmented memory allocation for large pages[C]//2020 ACM/IEEE 47th annual international symposium on computer architecture (ISCA). Barcelona:IEEE,2020:913-925.

[22] KOKOLIS A,SKARLATOS D,TORRELLAS J. Pageseer:using page walks to trigger page swaps in hybrid memory systems[C]//2019 IEEE international symposium on high performance computer architecture (HPCA). Washington:IEEE,2019:596-608.