面向 AI、HPC、BD 等应用的存储优化设计综述

张正立[1)]

[1)](华中科技大学计算机科学与技术学院，武汉 中国 430074)

摘 要 近年来由计算体系架构创新设计带来的计算系统功能性能指标提升为计算系统设计与应用注入了新的活力.几十年来, 围绕高速、高效、灵活、安全等目标体系架构不断向前发展与演进, 特别是近年 来随着人工智能、大数据、云计算、高性能计算, 以及物联网等技术的快速发展与广泛应用, 对计算系统提出了越来越高的要求, 催生着新颖的计算体系架构不断涌现, 其内涵与外延得到了极大的丰富.首先分析了体系架构演变和优化，从以处理器为重心、以存储器为重心, 以及以总线为重心到现在逐渐发展成为软件定义互连结构为重心。面向高性能计算、深度学习、云存储三个方面，分别分析了三篇文章针对高性能应用的存储系统优化设计。

关键词 计算体系架构、文件系统、cache

Abstract In recent years, the improvement in functional performance indicators of computing systems brought about by the innovative design of computing architecture has injected new vitality into the design and application of computing systems. For decades, system architectures have continued to develop and develop around the goals of high speed, efficiency, flexibility, and security. Evolution, especially in recent years with the rapid development and widespread application of technologies such as artificial intelligence, big data, cloud computing, high-performance computing, and the Internet of Things, has put forward increasingly higher requirements for computing systems, giving rise to novel computing System architecture continues to emerge, and its connotation and extension have been greatly enriched. First, the evolution and optimization of the system architecture are analyzed, from focusing on the processor, focusing on the memory, and focusing on the bus to now gradually developing into a software-defined interaction The connection structure is the center of gravity.

Facing the three aspects of high-performance computing, deep learning, and cloud storage, three articles analyzed the storage system optimization design for high-performance applications.

Key words Computing architecture, file system, cache

1 引言

随着摩尔定律与迪纳德缩放 (Dennard scaling) 定律逐步逼近物理极限, 一方面晶体管集成密度的提升越来越困难, 另一方面晶体管集成密度的提升带来了功耗墙问题, 依靠制程工艺进步提升计算系统性能与效能的途径已经难以为继. 因此, 除了一些机构继续在提升芯片制程工艺方面深耕外, 不论学术界还是产业界, 越来越多的研究人员将目光关注点锁定在计算体系架构革新这一领域中, 而近年来由计算体系架构创新设计带来的计算系统功能性能指标提升为计算系统设计与应用注入了新的活力. 体系架构是指计算、存储以及互连等一组部件的组织形式与使用方法, 是人工复杂系统研究的核心范畴, 不仅决定着计算系统的功能与性能, 还决定着计算系统的效能与安全. 国内外研究者们在自动化计算技术发展之初即注意到体系架构对计算系统的影响, 因此人们对体系架构的研究与探索从未停滞. 几十年来, 围绕高速、高效、灵活、安全等目标体系架构不断向前发展与演进, 特别是近年来随着人工智能、大数据、云计算、高性能计算,以及物联网等技术的快速发展与广泛应用, 对计算系统提出了越来越高的要求,催生着新颖的计

算体系架构不断涌现，其内涵与外延得到了极大的丰富.

2 体系架构演变和优化

组件呈现形式及组件互连关系构成了体系架构的核心部分. 虽然概括地说组件主要包括存储部件、计算核心、互连结构，以及外围的输入与输出设备等，但是在计算设备发展的不同历史时期，系统设计与运行的中心是有所变化的，大致可分为以处理器为重心、以存储器为重心,、以总线为重心以及软件定义架构 4 个阶段.

2.1 以处理器为重心

处理器是计算系统的核心计算部件. 在处理器发展初期，其计算性能有限，而存储器容量较小，两者的发展相对均衡，加上计算系统本身对信息存储要求较低，而且人们的关注重点更多地在于提升处 理器计算效率，因此，计算系统体系架构的设计是以处理器为重心的. 1971 年，由英特尔 (Intel) 公司 研发的世界上首款微处理器诞生，运算器与控制器合二为一，处理器的发展进入了快车道，数据位宽 从最早的 4 位逐步发展到 32 位乃至 64 位，时钟频率从最早的 4.77 MHz 逐步发展到几百 MHz 乃至 几 GHz. 从 20 世纪 80 年代开始，存储器与处理器之间从性能、发展速度等方面就开始出现了较大的 差距. 但是在差距还不足够大的情况下，人们虽然采用了一系列的措施弥补两者之间的差距，如设计更大的片上 Cache、更宽更快的片外存储带宽等，但是此时研究的着力点仍然在处理器性能提升方面. 在这个阶段内，计算系统仍然以处理器为重心进行设计开发. 以处理器为重心的体系架构如图 1 所示，是一种非常典型的冯诺依曼体系架构，包括单个中央处理器 (CPU)、存储器，以及输入/输出设备等，数据的传输与处理均要通过处理器完成.

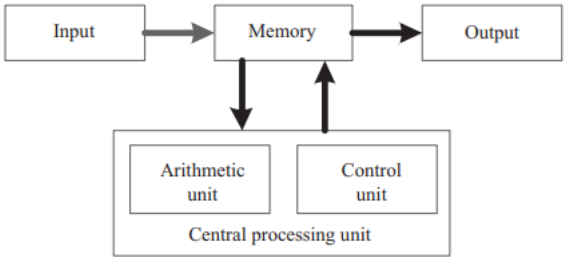


图 1

2.2 以存储器为重心

根据 Amdahl 定律，具有高性价比特性的计算系统其带宽应当是平衡的. 但是在过去几十年中，处理器基本遵照摩尔定律的预测快速发展，即每过 18 个月芯片集成晶体管数目翻一番，而存储器每年的速度则仅为 7%. 究其原因，两者设计目标有明显的差别：存储器是以容量最大化、成本最小化 为设计目标，而处理器则以性能最大化为设计目标，因此工业界也划分为了两个明显的阵营，从设计方法、设计目标，以及生产工艺等方面有着明显的差别. 自 20 世纪 80 年代开始，存储器与处理器之间的发展速度差异仍然在不断加剧. 当数据访存速度难以满足计算需求时，计算系统的发展面临着越 来越严重的 “存储墙” 问题. 虽然人们采取了多种手段提升数据访存速度，包括提升带宽、增加缓存、 设计层次化存储结构，以及改变数据存取方式等，以求缓解算存比不平衡的问题，但是至今尚无根本的 解决办法. 因此，体系架构设计重心逐渐由处理器转移到存储器，提出了近存储计算、存内计算等概念，以存储器为重心的体系架构逐渐成了主流. 如图 2 所示，以存储器为重心的体系架构一般由多个处理器组成，处理器围绕存储器便于就近完 成数据访存操作. 另外为了提升数据访存速度，设计了由寄存器、高速缓存、主存储器与外部存储器等 多级存储结构 .

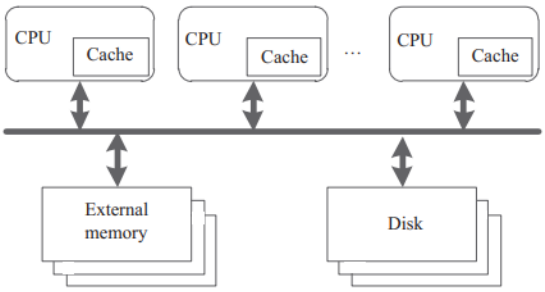


图 2

2.3 以互联为重心

随着数据密集程度不断提升, 对计算系统的性能需求快速增长, 单计算系统已经无法满足大数据量的应用需求, 往往需要多套计算设备组合在一起形成庞大的计算系统. 特别是随着超级计算机、云计算, 以及分布式计算等发展, 需要借助总线结构将多个处理机与存储系统结合起来, 通过控制系统的 调度管理解决不同处理系统与存储系统之间大规模并发需求, 完成大型复杂计算任务的协同计算. 因此, 在更高的层面上以总线为重心的体系结构应运而生. 总线结构是实现计算系统数据与指令汇聚与 分发的设计重心, 先后经历了共享总线与交换总线两个阶段. 以总线为重心的体系架构如图 3 所示. 另一方面, 随着集成电路的发展, 20 世纪 90 年代中期出现了将多个具有特定功能的集成电路组 合在单芯片上的系统或产品, 即片上系统 (SoC), 已经成为集成电路产业未来的重要发展方向. 随着片 上处理核心的数量增多、规模增大, 片内处理核心及其与片外其他部件之间的高效低耗信息传输与交 互对于提高片上系统性能具有重要意义. 因此, 以互联为重心的计算体系架构也下沉到单个芯片内部, 片上网络 (network on chip, NoC) 近年来已经成为了研究的热点, 主要包括体系结构、互连拓扑、路由方法、流控机制, 以及容错机制等关键技术.

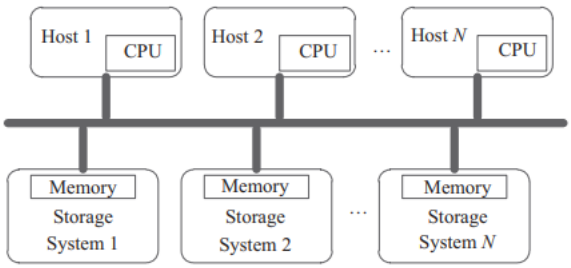


图 3

2.4 软件定义架构

计算架构重心的变化可以看出, 在不同的时期, 受应用需求和技术水平的影响, 虽然体系架构的设计有不同的着重点, 但是其总体目标旨在克服计算系统的短板问题, 追求系统的平衡性. 而体系架构重心从计算与存储部件变化为部件之间的连接关系也说明了当前体系架构在朝着平面化与去中心化方向发展, 即所有计算部件与存储部件地位相同, 而如何将这些部件有效连接并充分利用起来是体系架构设计的关键问题.

当前, 在计算系统朝着多种功能一体化、大众服务个性化、新业务高效部署方向 发展, 特别是在云计算与边缘计算的快速崛起的情况下, 对计算系统灵活性、高效性、开放性, 以及可 扩展性提出了越来越高的要求. 受此影响, 计算系统正在由以总线为重心向以软件定义互连结构为重心进一步演进. 新一代软件定义体系架构以模块化、标准化的软件定义节点 (包括计算节点与存储节点) 为基础, 以软件定义互连结构实现对软件定义节点之间的层次化组织, 能够根据应用需求改变计算结构, 最优化匹配应用计算需求, 在体系架构层面实现系统性能、效能、灵活性, 以及可靠性的综合平衡与同步提升, 是未来体系架构重要的发展方向.

3 针对高性能应用的存储系统优化设计

3.1 HadaFS：面向 HPC 的文件系统优化设计

HadaFS 发表于顶级会议 FAST 2023，由无锡国家超级计算中心、清华大学、山东大学、中国工程院的学者为我们分享了他们在尖端超级计算机和高性能计算领域的最新的成果，提出了一种名为 HadaFS 的新型 Burst Buffer 文件系统，实现了可扩展性和性能的优势与数据共享和部署成本的优势的良好结合。

高性能计算（HPC）正在经历计算规模和数据爆发式增长的时代。为了满足 HPC 应用不断增长的 I/O 需求或突发流量 I/O 性能需求，研究人员提出 Burst Buffer（BB）技术，通过 SSD 等新型存储介质构建数据加速层，作为前端计算和后端存储之间的缓冲区，为 HPC 应用提供高速 I/O 服务，提高了系统的性能。

取决于 SSD 阵列的部署位置 ，BB 可以分为两种类型：

1）本地 BB，即 SSD 作为本地磁盘部署在每个计算节点上，专门为单个计算节点服务；

2）共享 BB，即 SSD 部署在计算节点可以访问的专用节点上 （例如 I/O 转发节点），以支持共享数据访问。

本地 BB 具有良好的可扩展性和性能优势，系统性能可以随着计算节点的数量线性增长。但本地 BB 数据共享不友好，要么以静态数据迁移方式运

行，要么需要应用程序通过计算节点迁移数据，迁移效率低下，造成计算资源浪费。本地 BB 还会造成严重的资源浪费，因为 HPC 应用程序之间 I/O 负载的差异巨大，数据密集型应用程序相对较少。未来随着超级计算机规模的迅速扩大，本地 BB 的部署成本将急剧上升。

共享 BB 天然具有数据共享和部署成本的优势，但难以为数十万规模的客户端提供高效的数据访问处理性能。如何统一本地 BB 和共享 BB 的优势，满足多样化的应用需求，降低 BB 的建设成本，支持大规模的 BB 数据管理和迁移，是亟待解决的问题。BB 虽然具有高性能的优势，但具有容量小的缺点，所以 BB 必须与 GFS（如 Lustre 等全局文件系统）协同工作才能满足容量要求。

HadaFS 相当于是堆叠在磁盘阵列或存储服务器的全局文件系统上的一个分布式文件系统，HadaFS 的整体架构如图 4 所示，包括 HadaFS 客户端、HadaFS 服务器、数据管理工具 Hadash。

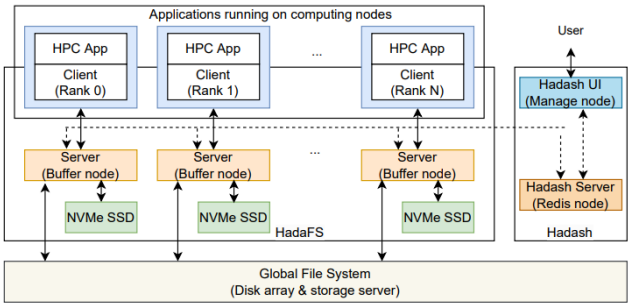


图 4

HadaFS 遵循绕过内核的思路，直接将客户端挂载到应用程序中使用，避免引入内核的 I/O 请求 stage-in 和 stage-out 开销。为了更好地给应用程序提高全局文件视图，HadaFS 提出了名为 LTA 的新方法，每个 HadaFS 客户端只连接一台 HadaFS 服务器（桥接服务器），桥接服务器负责处理客户端产生的所有 I/O 请求，并将数据写入底层文件。当客户端需要访问另一台服务器上的数据时，必须通过桥接服务器进行转发。因此，服务器是一个全连接结构。

在 HPC 中计算节点通常负责读写数据，很少执行目录树访问。为了提高可扩展性和性能，HadaFS 放弃了目录树的思想，采用了全路径索引方法。数据存储在生成该文件的 HadaFS 客户端对应的桥接服务器上，文件元数据以 key-value 方式存储。

HadaFS 采用了宽松的一致性语义，依赖于基本文件系统（ext4）的缓存机制来提高性能，其一致性语义主要依赖于元数据同步，不支持在客户端和服务端缓存数据。为此，HadaFS 针对不同的应用场景提出了三种元数据同步策略。HadaFS 没有使用分布式锁机制，因此 HadaFS 本身很难保证数据的一致性，只有在第三种元数据同步策略下才支持原子写。为了保证数据的一致性，用户至少要了解应用程序的文件共享模式，可以通过 Darshan、Beacon 等获得，自行保证数据一致性。

众所周知，超级计算机上同时运行着很多作业。这些作业往往会争夺共享资源，从而导致 I/O 干扰。将客户端动态映射到服务器也有助于提高应用程序性能。得益于 HadaFS 灵活的设计，用户可以动态制定 HadaFS 客户端到 HadaFS 服务器的连接关系，可以有效帮助隔离不同应用的 BB 资源，解决作业间的 I/O 干扰，缺点是对运维人员的要求略高。

总体而言，文章提出了一种名为 HadaFS 的新型 Burst Buffer 文件系统，基于共享 BB 架构为计算节点提供了本地 BB 式的访问，结合了本地 BB 的可扩展性和性能的优势与共享 BB 的数据共享和部署成本的优势。HadaFS 提出的 LTA 架构通过桥接服务器处理计算节点的 I/O 请求，实现了与节点本地 BB 相当的可扩展性，并提供新的接口以减少单个服务器上大量连接带来的干扰。HadaFS 提出了三种元数据同步策略，以解决传统文件系统复杂的元数据管理与 HPC 应用程序的各种一致性语义需求之间的不匹配问题。此外，HadaFS 内部集成了名为 Hadash 的数据管理工具，可以为用户提供全局的数据视图和高效的数据迁移。最后，HadaFS 已经部署在 SNS 上（超过 100000 个计算节点）并支持数百个应用程序，可以为多种超大规模应用提供稳定、高性能的 I/O 服务。

3.2 盘古 2.0：云存储系统优化设计

盘古（Pangu）是阿里云存储系统，为上层的所有应用提供基础服务。

这篇论文讲述了盘古自 2016 年以后的不断更新发展，盘古从追求容量的 1.0 时代向追求高性能与低时延的 2.0 时代转型。这篇论文更侧重于工程实践，揭示了很多实践中出现的问题，并应用了大量的前沿技术，为读者提供了一个完整的存储系统的 design choices。

主要技术要点包括：1. 用户态操作系统，减少

内核切换开销，减少数据拷贝。2. Append-only 文件系统，充分利用局部性，可回放的特性。3. Hardware & Software co-design，部分功能硬件卸载。

2015 年以前，盘古 1.0 以 HDD 硬盘为主要存储介质，并使用传统 Ethernet/IP 网络进行分布式互联。近年来，计算机硬件不断更迭，更高速的 NVMe SSD 发展迅速，RDMA 网络技术日趋成熟。在更高性能的业务的要求下，盘古应用新兴的硬件和技术推出了 2.0 版本。

如今，盘古的设计目标有以下三点：

1. 低时延——百微秒级的 IO 平均时延，毫秒级的 P999 尾时延。

2. 高吞吐。

3. 对上层服务（如 EBS、OSS、数据库）提供统一的底层抽象。

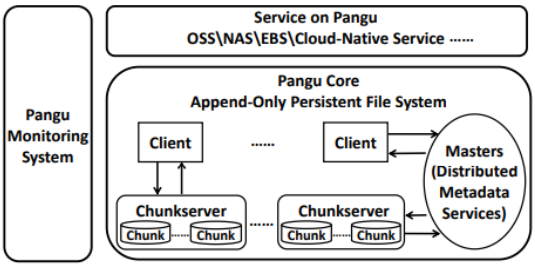


图 5

盘古的总体架构如图 5 所示，本篇论文介绍了其中 Pangu Core 的设计。

盘古系统由三部分组成，Client、Chunkserver 和 Master，其中 Client 和 Chunkserver 是论文的重点。

- **Pangu Client:** 上层应用服务通过 SDK 向 Client 提出对盘古系统的任务请求，Client 随后负责与 Master 和 Chunkserver 通信来完成任务请求。
- **Chunkserver:** 文件被抽象为 stream，stream 由许多个 chunk 组成，这些 chunk 会分散存储在多个 Chunkserver 上。
- **Master:** 管理 metadata，提供 namespace 服务和 stream 服务。namespace 服务主要包括对文件目录树和命名空间的维护；stream 服务维护对 chunk 的索引等。采用基于 Raft 的一致性协议。

可以看到这里用的存算分离架构，Client 是系统的计算节点，Chunkserver 是存储节点。

当上层应用向 Client 提出任务请求时，Client 需要首先向 Master 通信查找相关 metadata，从而得知数据在哪几个 Chunkserver 上。

Client 设计

Client 获得 metadata 后，与 Chunkserver 进行通信，进行数据读取/更新。这里为满足 SLA，即满足上层服务的低时延要求，提出了如下设计。

数据需要在多个 Chunkserver 上进行备份，由 Client 分别进行写入，对某一台 Chunkserver 写入成功时，会收到 Success 的回复。实际往往采用三份备份。其中采用了较为激进的设计思路来降低时延，使用冗余换时间。

Chunkserver 设计

Chunkserver 是直接存储数据的节点。其上运行了一个用户态操作系统 USSOS（User-Space Storage Operating System)），重新设计了 OS 的基本功能。用户态 OS 通过 Kernel Bypass 直接让用户态程序管理硬件，避免了内核态/用户态切换开销，同时可以采用更针对性的定制化方案，对 Chunkserver 这种需要处理大量数据的 data-intensive 的应用来说比较合适。实现方案使用 DPDK + SPDK + RDMA。DPDK（Data Plane Development Kit）和 SPDK（Storage Performance Development Kit）分别是针对网络和存储的用户态程序开发工具包，实现用户态程序直接接管硬件，让数据通路避免通过内核繁杂的逻辑，通过定制逻辑可以减少上下文切换和数据拷贝，提高性能。RDMA（Remote Direct Memory Access）则是将网卡（NIC）与 DMA 技术融合在了一起，RDMA 网卡可直接将本地内存数据通过网络传输到远端机器内存中，不需要两边的 CPU 处理网络协议栈，更不需要内核参与，实现了 kernel bypass（内核旁路） + hardware offload（硬件卸载）。

内存管理采用了

1. **Run-to-completion model**: 即一个线程完成一波数据的全部处理逻辑，可以避免线程切换开销带来的时延、避免跨线程通信和同步，各自用自己的内存，数据局部性好，易开发。
2. **Huge-page memory（内存大页）**：普通 OS

通常使用 4KB 的页面大小，这样对于大量数据处理来说，会涉及很多的页面，带来很多的缺页，引入 IO，效率低。因此采用了 huge-page，使用 2MB 甚至 1GB 的页面大小。

3. **Zero-copy（零拷贝）**：在存储协议栈和网络协议栈之间使用共享内存，再利用 RDMA 网卡对内存数据的直接读取与写入，实现数据通路全程零拷贝。

盘古实现了一个用户态文件系统 USSFS（User-Space Storage File System），其中提供了一个文件抽象 FlatLogFile。这个文件系统是 Append-only 的，即修改数据不会原地替换，而是通过不断添加日志的方式进行修改。Append-only 的顺序读写方式对 SSD 很友好，减少了随机读写，且不会重复擦写同一块区域，有利于提高硬盘寿命。日志积累到一定程度后，使用一个垃圾回收机制将结果整理，保存检查点。如果发生崩溃，也可以通过日志进行回放恢复。

以上完成了一个基本成形的盘古系统，但随着实际的运行，网络、内存、CPU 成为了新的瓶颈。例如如果一个服务器装载了 12 个 SSD，可提供 36 GB/s 的读速度和 9.6 GB/s 的写速度。这样的速度对网络、内存乃至 CPU 的性能都提出了挑战。在文章中对网络，将无损网络改为有损网络，并对流量进行了优化，如 EC 编码，LZ4 压缩算法，动态带宽；对内存更倾向于使用多块较小容量的内存以充分利用内存通道的带宽，且充分利用 NUMA 架构，使用 RDMA 传输背景流量，网卡直接与 CPU LLC Cache 通信，跳过内存；对 CPU，使用混合 RPC 序列化算法、超线程（Hyper Thread）管理、数据压缩的逻辑卸载到 FPGA 硬件等策略进行优化。

总体而言，论文并没有创造什么独一无二的新技术，但其揭示了许多工程实践中会遇到的问题，并且将现有的前沿工程技术融会贯通，形成了一组可行的方案，值得学习。盘古为追求低时延，总体比较激进，大量使用空间换时间、超发、冗余的设计思路，因此带来了 CPU、内存和网络的瓶颈，又通过更为前卫的方式消除这些瓶颈。这其中必然存在许多可以 tradeoff 的部分，同时一些处理方案比较简单粗暴，针对细节技术有很多的可优化空间

3.3 Shade：面向深度学习 I/O 特征的 Cache 缓存优化设计

这篇文章分析了深度学习 I/O 特征，介绍了如何使用 CACHE 加速数据并行训练的远端访存。

文章一开始就阐明了深度学习训练（DLT）展现了比较独特的 IO 负载特征，这对存储系统的设计引入了一些新的挑战。一方面，DLT 在训练过程中，需要不断从 Remote Storage 获取数据样本，具备 I/O 密集型的特征。另一方面，DLT 广泛利用 GPUs 加速训练过程，所以同样具备计算密集型的特征。然而，指数级增长的样本数据集使得这些数据不可能完全存储在内存中，并且 GPUs 处理能力在持续提升，这导致滞后的 I/O 性能成为了整个分布式 DLT 系统中的性能瓶颈。文章提到，尽管目前的 DLT 框架通常使用随机抽样策略来平等地处理所有样本，但最近的研究表明，不是所有样本都同等重要，不同的数据样本对提高模型精度的贡献是不同的。因此，我们很容易能推断到，可以通过利用那些贡献更高的数据样本的局部性，来优化整个 DLT 系统里的 I/O 情况。因此，文章作者们设计和实现了 SHADE，这是一种新的 DLT 感知型的缓存系统。SHADE 通过细粒度的样本级的动态重要性检测机制，并通过一种新的 Rank 方式捕捉不同批次里数据样本的重要性，这为 DLT jobs 提供了更精确的缓存策略。

为了分析存储系统对分布式深度学习训练效率的影响，文章做了一个实验，研究当分布式深度学习 Job 使用本地 vs. 远程存储介质运行时的性能差异。即使所有其它训练配置保持相同，相比于更快的存储介质（例如 RAM，本地内存），远程存储介质也会显著加大训练时间（大约 2.5 倍）。更为夸张的，研究表明 I/O 甚至占据了总训练时间的 85-90%。而另一方面，文章也论证了，在日愈膨胀的数据集和抢占式的 VM 恢复方面，大型数据集必须放在持久化云存储中，而不是本地内存或者本地盘上。因此，如果能有效降低 I/O 的延迟，就可以显著提升 GPU 的训练效率。直观的解决方案就是缓存来降低 I/O 延迟。

作者尝试通过挖掘 DL 的训练方式，定位其 I/O 特征。传统上，基于 SGD 的深度学习训练对于训练样本的“重要性”毫不关注，因为它们只是在每个训练周期结束时随机排列顺序，从而平等对待所有的训练样本。然而，文章提到在最近的一些研究中，研究人员发现，在基于 SGD 的深度学习训练

中，一组特定的训练样本往往对模型质量产生很小或没有影响，因此可以忽略它们。另一方面，通过找到比其它样本更重要的训练样本集合，即对损失函数最大贡献的样本集合（这个过程被称为重要性采样），这些样本在几个 epochs 后会导致在反向传播中的隐藏层输出和目标标签之间出现更大的 loss。因此，通过优先使用相对更重要的训练样本来训练，可以明显降低整个训练系统的训练时间和减少测试错误。例如下图 6 是识别字母的训练任务，在整个训练过程中，更难学习的样本反而是更加重要的。比如左边三个字母“c e t”非常清晰，很快就能被神经网络所识别。然而右边两个字母“e t”却非常模糊潦草，相对来说需要更多轮的反馈和训练，所以显然更加重要。

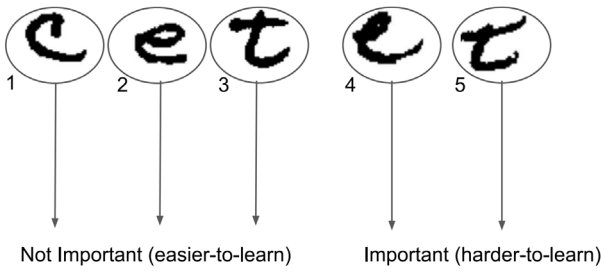


图 6

进一步的，作者通过多个 benchmark 的分析（如下图 7 所示），发现的确有一部分数据在一次 epoch 或者多次 epoch 中被访问的频率明显更高。因此，他们尝试从这种重要性差异的角度出发，设计更加有效的机制和策略，提高 DL 训练的可缓存性。文章将单个数据样本的 inter-batch 和 intra-batch 重要性相结合，以检测最重要的样本并将其放置在内存池化缓存中。此外，文章开发了一种新颖的基于排名的重要性技术，它基于样本对提高模型整体准确率的贡献对批次内的训练样本进行排名。基于排名的重要性进一步有助于增加在后续时期识别（预测）最重要样本的概率。利用这种技术，文章进一步设计了一种基于优先级的采样策略，确保在一个时期内多次访问重要样本以更多地训练难以学习的样本，以提高准确率提高率。因此，这种缓存解决方案将更重要的样本保留在缓存中，并避免随机淘汰，从而提高缓存命中率和训练吞吐量。

如上文所述，SHADE 的目标就是利用重要性抽样来提高 DLT 的 I/O 工作负载的缓存效率。想达到这个目标，需要解决以下的三个问题：1. 简单的重要性采样策略分配了每个 mini-batch 的得分，但是这种方式过于粗糙，以至于并不准确。也就是说，所有 mini-batch 里的样本数据默认情况下被分配同样的重要性得分，从而使每个样本的重要性估计并不准确，影响了缓存效率。因此，SHADE 希望能够精确评估每个样本数据在 mini-batch 中所具有的相对重要性；2. 即使正确识别了重要的样本，过度地向 DL 模型提供重复的样本可能会使模型训练出现偏差。因此在尝试增加样本击中率的同时，必须确保不会影响模型的准确性；3. 重要性得分在动态发生变化，可能很快就会过时。在后续 mini-batch 中，同一个样本对模型的贡献可能与之前的 mini-batch 是不同的。因此，获取最新的重要性得分信息是必要的，以做出明智的缓存决策。简单来说，SHADE 需要设计细粒度的动态重要性追踪机制，并且不能影响模型的准确性。

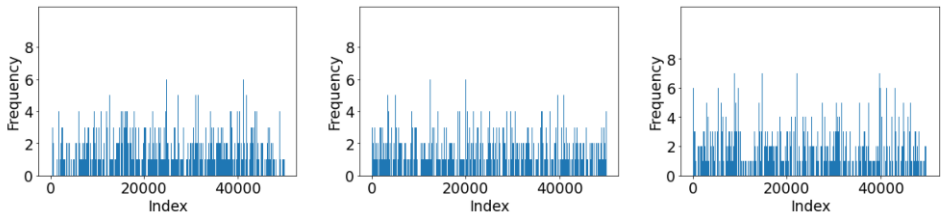
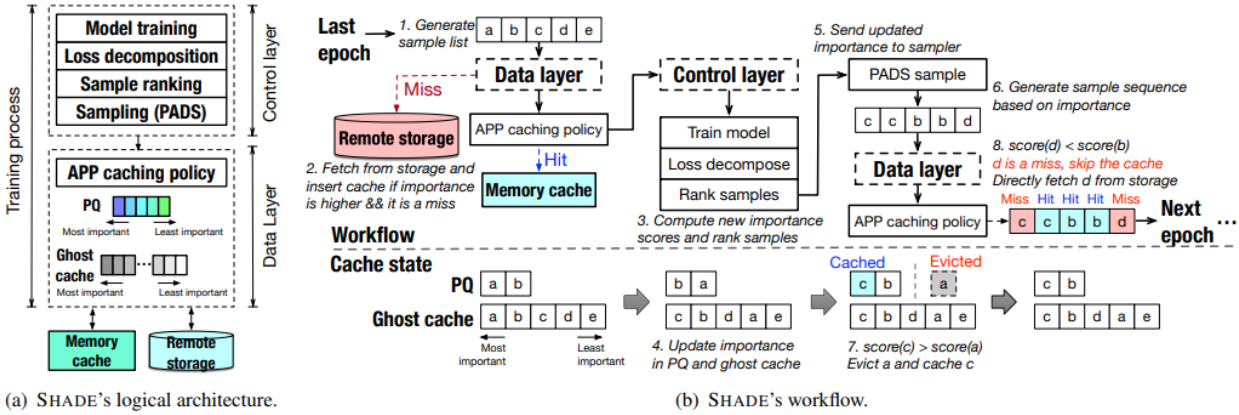


图 7

SHADE 是由两个部分组成的：控制层和数据层，控制层向数据层提供训练所需要的数据。在第一次迭代中，数据层数据层从远程存储中获取样本，并用要访问的样本填充缓存。在训练过程中，控制层找到与样本相关的重要性（loss 分解+排序），并更新优先队列（PQ）和跟踪数据层中样本重要性的 ghost 缓存。基于更新的重要性，控制层的采样器准备了一个带有相关重复信息的采样列表。当数据层接收到采样列表时，它会检查是否有利于缓存一个新的 item，而不是驱逐先前缓存的样本。假设正在访问的样本比 min_sample（当前缓存中重要性最低的样本）更重要。在这种情况下，min_sample

被驱逐，并且当前样本使用新的自适应优先级感知预测（APP）缓存策略进行缓存。这个过程在整个 DL 训练过程中重复进行。由于 SHADE 将最重要的样本保留在分布式缓存中，并反复使用这些难以学习的样本进行训练，因此可以确保提高准确率和良好的缓存命中率。图 8 显示了 SHADE 的架构以及其中的组件和相互作用。



(a) SHADE's logical architecture. (b) SHADE's workflow.

图 8

总体而言，为了使用 CACHE 加速数据并行训练的远端访存，使用了基于优先级的自适应采样、基于优先级的自适应预测缓存等策略，均衡 CACHE 命中率和采样准确性，提高了缓存命中，同时提高采样样本重要性加速了训练。

4 总结

新一代软件定义体系架构以模块化、标准化的软件定义节点 (包括计算节点与存储节点) 为基础，以软件定义互连结构实现对软件定义节点之间的层次化组织，能够根据应用需求改变计算结构，最优化匹配应用计算需求，在体系架构层面实现系统性能、效能、灵活性，以及可靠性的综合平衡与同步提升，是未来体系架构重要的发展方向.

参 考 文 献

[1]. Khan R I S, Yazdani A H, Fu Y, et al. {SHADE}: Enable Fundamental Cacheability for Distributed Deep Learning Training[C]//21st USENIX Conference on File and Storage Technologies (FAST 23). 2023: 135-152.

[2]. Li Q, Xiang Q, Wang Y, et al. More than capacity: performance-oriented evolution of Pangu in Alibaba[C]//21st USENIX Conference on File and Storage Technologies (FAST 23). 2023: 331-346.

[3]. He X, Yang B, Gao J, et al. {HadaFS}: A File System Bridging the Local and Shared Burst Buffer for Exascale Supercomputers[C]//21st USENIX Conference on File and Storage Technologies (FAST 23). 2023: 215-230.

[4]. 高彦钊,邬江兴,刘勤让,等.Review and thoughts on the development of computing architecture[J].SCIENTIA SINICA Informationis, 2022, 52(3):377-.DOI:10.1360/SSI-2021-0163.

[5]. Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. TOS, 17(4):1–32, 2021.

[6]. Yong Liu, Xin Liu, Fang Li, Haohuan Fu, Yuling Yang, Jiawei Song, Pengpeng Zhao, Zhen Wang, Dajia Peng, Huarong Chen, et al. Closing the" quantum supremacy" gap: achieving real-time simulation of a random quantum circuit using a new sunway supercomputer. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–12, 2021.

附录 Distributed Transactions at Scale in Amazon DynamoDB**论文汇报记录**.

问题 1 分布式事务和一般的事务区别在哪里？

单机事务通常在单个计算机或数据库上执行，所有的操作都在同一地理位置。因此，单机事务没有涉及到跨越网络或多个物理位置的操作。单机事务通常通过数据库系统提供的 ACID（原子性、一致性、隔离性、持久性）属性来保障事务的一致性。单机事务面对故障时，通常可以通过事务回滚等方式来进行恢复。由于涉及的范围较小，通常具有较低的通信开销和较好的性能。

分布式事务涉及到多个独立的计算机、数据库或者服务，这些分布在不同的地理位置。操作可能涉及到在不同地方执行的事务分支。分布式事务需要额外的机制来保障在不同节点上的操作的一致性，通常使用两阶段提交协议（2PC）等分布式事务协议。分布式事务需要考虑到网络故障、节点故障等更多的故障场景，需要使用一些复杂的机制来处理分布式环境下的故障情况。由于需要跨越多个节点，通信开销较大，可能面临性能上的挑战。

总体而言，分布式事务需要解决更多的问题，如一致性、通信开销、故障处理等，而一般事务在单机环境下相对更为简单。选择使用分布式事务需要权衡系统的需求和复杂性。