关于 ARM 架构 CPU 处理网络请求的研究

姓名：谢宝玛　　学号：1120233506

摘要

随着云计算、物联网（IoT）及边缘计算的爆发式增长，高效的网络请求处理能力已成为衡量计算架构性能的核心指标。ARM 架构凭借其卓越的功耗比（PUE）和日益增强的单核性能，正从传统的移动终端领域迅速向数据中心服务器及高性能嵌入式设备扩展。

本研究旨在深入探讨 ARM 处理器在处理典型网络请求时的微架构性能表现。本文分析了 ARM 架构在指令集效率、中断处理机制及内存一致性模型对网络 I/O 吞吐量和延迟的影响。同时，本文重点对比了 ARM 与传统 x86 架构在多核扩展性与每瓦性能（Performance per Watt）方面的差异。

关键词：　ARM 架构；网络请求处理；性能对比；边缘计算；能效优化

1 ARM 架构概述

1.1 ARM 处理器的基本特点

ARM 架构作为目前全球应用最广泛的指令集架构之一，其核心哲学在于平衡。

1. 精简指令集 (RISC)：　与 x86 的复杂指令集 (CISC) 不同，ARM 采用固定长度的指令格式。这种设计简化了指令译码逻辑，使得处理器在处理网络包解析等频繁、简单的重复性操作时，拥有更高的流水线执行效率。
2. 低功耗与高能效：　ARM 并非单纯追求极致的单核频率，而是通过优化每瓦性能 ($Performance / Watt$) 实现高效能。在需要 24/7 不间断运行的网络服务 (如路由器、网关）中，这一特性显著降低了运维成本和散热压力。
3. 多核与异构处理能力：　ARM 能够轻松实现数十甚至上百个核心的集成。通过 big.LITTLE (大小核) 或 DynamIQ 架构，处理器可以根据网络负载动态分配任务：轻量级的背景心跳包由低功耗小核处理，而重负载的流量加解密则交给高性能大核。

1.2 ARM 架构在网络处理中的优势

ARM 不仅仅是一个通用的计算核心，其针对现代网络需求集成了多种硬件级加速手段：

1. 高并发处理能力： 凭借较短的流水线和高效的上下文切换（Context Switch）机制，ARM 核心在处理成千上万个并发 TCP 连接（C10K 甚至 C10M 问题）时，往往表现出比传统架构更低的尾延迟（Tail Latency）。
2. 硬件支持的网络加速：

（1）NEON 指令集： 作为单指令流多数据流 (SIMD) 扩展，NEON 可用于加速网络协议栈中的校验和计算、多媒体数据封包等操作。

（2）Crypto 扩展： ARMv8 及后续架构提供了专门的硬件加解密指令，能够显著提升 HTTPS/TLS 握手及数据传输过程中 AES、SHA 等算法的处理速度。

1.3 常见 ARM 处理器型号及应用场景

为了覆盖从微型传感器到巨型数据中心的全部需求，ARM 家族形成了三大主要系列：

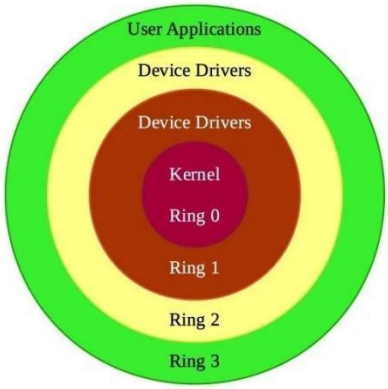
系列名称	核心定位	网络处理典型应用场景
Cortex-A 系列	高性能应用处理器	智能手机、工业网关、高性能 Wi-Fi 6 路由器。
Cortex-M 系列	低功耗微控制器	IoT 传感器节点、智能家居连接模块（处理简单的 MQTT/CoAP 请求）。
Neoverse 系列	基础设施级服务器核心	云计算服务器（如 AWS Graviton）、5G 基站、大规模 CDN 节点。

2 网络请求处理机制

2.1 网络请求的基本流程

一个网络请求从物理网线到达应用程序，需要经历从硬件到内核再到用户空间的复杂传递过程：

CPU 内核态和用户态描述图：



Standard IA Protection Rings

1. 硬件接收： 数据包首先到达网卡（NIC），网卡负责在物理链路层解码数据，并进行初步的校验（如 CRC 校验）。为了减轻 CPU 负担，网卡通常使用 DMA（直接存储器访问） 将接收到的数据直接写入内存中的 环形缓冲区（Ring Buffer），避免数据在 CPU 和内存之间的多次拷贝。现代 ARM 处理器通过高速内存接口（如 LPDDR 或 DDR）以及大容量缓存，可以更高效地配合 DMA 传输。
2. 触发中断：当网卡接收到数据包后，会向 CPU 发送 硬件中断信号（IRQ）。ARM 处理器响应中断的延迟和中断控制器（GIC）设计密切相关，高性能 ARM 核心在处理网络中断时可以实现 低延迟中断响应。为了减少中断风暴，Linux 内核通常启用 中断合并（Interrupt Coalescing） 或 NAPI（New API）轮询机制，在高流量情况下减少频繁切换到内核态的开销。
   软中断和硬中断的区别：

特性	硬中断（HardIRQ）	软中断（SoftIRQ）
触发来源	硬件设备	硬中断挂起
优先级	高	较低
处理时间	必须尽快完成	可延迟处理
执行上下文	中断上下文，不允许阻塞	中断上下文或线程上下文
CPU 分配	通常固定分配给某个 CPU	支持多 CPU 并发处理

3. 内核处理

CPU 响应中断后进入内核态，开始处理网络协议栈。在 Linux 中，网络处理分为两个阶段：

上半部（Top Half）：快速响应中断，主要负责从环形缓冲区取出数据包的元信息，标

记缓冲区状态，尽量减少中断占用时间。

下半部 (Bottom Half / Softirq)：通过 Softirq 或 Tasklet 机制处理复杂的 TCP/IP 协议栈逻辑，包括数据校验、路由查找、分包重组、协议头解析等。ARM 处理器的多核特性可以通过 多队列（RSS/Receive Side Scaling） 将不同流量分配到不同核心，提高并发处理能力。

4. 交付应用： 内核完成协议处理后，将数据拷贝到对应 Socket 接收缓冲区，并唤醒等待的应用程序（如 Nginx、Node.js 或自研网络服务）。应用程序通过 系统调用（如 recv() 或 read()） 将数据从内核态读取到用户空间。为了进一步优化性能，ARM 平台上可以采用 零拷贝（Zero-copy）机制 或 用户态网络栈（DPDK/XDP），减少内核到用户空间的数据复制开销，从而显著降低延迟并提高吞吐量。

2.2 操作系统与网络栈的作用

在 ARM 平台上，Linux 内核针对高并发场景采用了多种优化模型：

1. NAPI (New API) 机制： 为了防止高频请求引发“中断风暴”，Linux 采用 中断与轮询（Polling）相结合 的 NAPI 模式。当请求密集时，内核关闭硬件中断，改用轮询方式批量处理数据包，这极大减少了 ARM 核心在上下文切换上的开销。
2. GIC (Generic Interrupt Controller)： ARM 特有的通用中断控制器，支持将特定的网络中断绑定到特定的 CPU 核心（亲和性设置 IRQ Affinity），避免核心间频繁的缓存失效（Cache Miss）。
3. RPS/RFS (Receive Packet Steering)： 在多核 ARM 服务器上，内核可以通过软件方式将网络负载均衡到各个核心，充分发挥 ARM 多核并行的优势。

2.3 网络请求处理的性能瓶颈

尽管 ARM 架构具有高效能，但在处理极端网络负载时，仍面临以下瓶颈：

瓶颈维度	影响因素	对 ARM 架构的挑战
CPU 处理能力	协议栈解析、SSL/TLS 加解密	在高频单核性能上，部分中低端 ARM 核心可能弱于高频 x86 核心。
内存带宽	数据在内核态与用户态之间的拷贝	大规模小包处理时，若内存控制器的带宽不足，会导致处理延迟增加。
中断负载	硬件中断的频繁触发	若中断分发不均，会导致“热点核心”过载，而其他核心处于闲置。
系统调用开销	read, write, epoll_wait	频繁的模式切换（用户态 $\leftrightarrow$ 内核态）会消耗大量的 CPU 时钟周期。

3 ARM 处理器优化网络请求的技术

ARM 架构凭借其高能效比和丰富的专用指令集，在现代边缘计算、移动端和云原生服务器（如 Graviton、Ampere）中展现了卓越的网络处理能力。

3.1 硬件层优化

硬件层的优化主要通过专用电路和并行计算能力，减轻通用 CPU 核心的运算压力。

3.1.1 NEON SIMD 加速数据包处理

NEON 技术是 ARM Cortex™-A 系列处理器的 128 位 SIMD（单指令，多数据）架构扩展，旨在为消费性多媒体应用程序提供灵活、强大的加速功能，从而显著改善用户体验。它具有 32 个寄存器，64 位宽（双倍视图为 16 个寄存器，128 位宽。）

目前主流的 iPhone 手机和大部分 android 手机都支持 ARM NEON 加速，因此在编写移动端算法时，可利用 NEON 技术进行算法加速，以长度为 4 的寄存器大小为例，相应的提速倍数约是原始的 4 倍。

本文主要介绍 float32x4_t 相关的结构及函数，float32x4_t 可以理解为 vector (4),同理 typexN_t 即为 vector(N)。在 NEON 编程中，对单个数据的操作可以扩展为对寄存器，也即同一类型元素矢量的操作，因此大大减少了操作次数。这里以一个小例子来解释如何利用 NEON 内置函数来加速实现统计一个数组内的元素之和。以 C++代码为例：
原始算法代码如下：

```
#include <iostream>
using namespace std;

float sum_array(float *arr, int len)
{
if(NULL == arr || len < 1)
{
cout<<"input error\n";
return 0;
}
float sum(0.0);
for(int i=0; i<len; ++i)
{
```

```
sum += *arr++;
}
return sum;
}
```

对于长度为 N 的数组，上述算法的时间复杂度时 O(N)。
采用 NEON 函数进行加速：

```
#include <iostream>
#include <arm_neon.h> //需包含的头文件
using namespace std;

float sum_array(float *arr, int len)
{
if(NULL == arr || len < 1)
{
cout<<"input error\n";
return 0;
}

int dim4 = len >> 2; //  数组长度除 4 整数
int left4 = len & 3; //  数组长度除 4 余数
float32x4_t sum_vec = vdupq_n_f32(0.0);//定义用于暂存累加结果的寄存器且初始化为 0
for (; dim4>0; dim4--, arr+=4) //每次同时访问 4 个数组元素
{
float32x4_t data_vec = vld1q_f32(arr); //依次取 4 个元素存入寄存器 vec
sum_vec = vaddq_f32(sum_vec, data_vec);//ri = ai + bi  计算两组寄存器对应元素之和并存放到相应结果
}
float sum = vgetq_lane_f32(sum_vec, 0)+vgetq_lane_f32(sum_vec, 1)+vgetq_lane_f32(sum_vec,
2)+vgetq_lane_f32(sum_vec, 3);//将累加结果寄存器中的所有元素相加得到最终累加值
for (; left4>0; left4--, arr++)
sum += (*arr) ; //对于剩下的少于 4 的数字，依次计算累加即可
return sum;
}
```

上述算法的时间复杂度时 O(N/4)
从上面的例子看出，使用 NEON 函数很简单，只需要将依次处理，变为批处理（如上面的每次处理 4 个）。
上面用到的函数有：
float32x4_t vdupq_n_f32 (float32_t value)
将 value 复制 4 分存到返回的寄存器中
float32x4_t vld1q_f32 (float32_t const * ptr)
从数组中依次 Load4 个元素存到寄存器中
相应的 有 void vst1q_f32 (float32_t * ptr, float32x4_t val)

将寄存器中的值写入数组中
float32x4_t vaddq_f32 (float32x4_t a, float32x4_t b)
返回两个寄存器对应元素之和 r = a+b
相应的 有 float32x4_t vsubq_f32 (float32x4_t a, float32x4_t b)
返回两个寄存器对应元素之差 r = a-b
float32_t vgetq_lane_f32 (float32x4_t v, const int lane)
返回寄存器某一 lane 的值

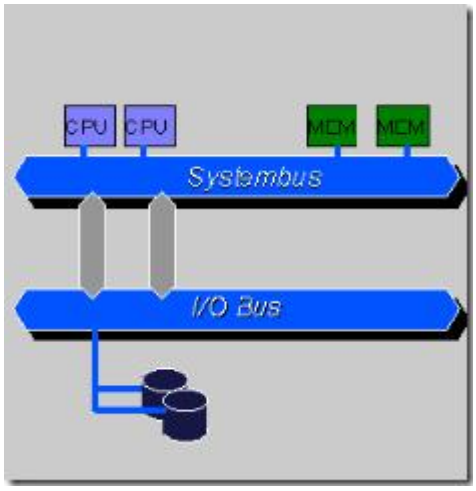
3.1.2 CRC 与加密硬件加速

ARMv8 架构引入了专门的 CRC32 指令和 加密扩展指令集 （AES, SHA）。这些硬件加速器允许处理器在不消耗额外 CPU 周期的情况下，快速完成网络协议中的数据完整性校验和 SSL/TLS 加密握手。

3.1.3 多核负载分配 （SMP, NUMA）

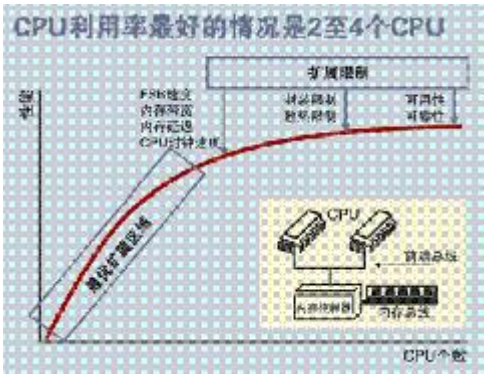
3.1.3.1 SMP(Symmetric Multi-Processor)

SMP (Symmetric Multi Processing),对称多处理系统内有许多紧耦合多处理器，在这样的系统中，所有的 CPU 共享全部资源，如总线，内存和 I/O 系统等，操作系统或管理数据库的复本只有一个，这种系统有一个最大的特点就是共享所有资源。多个 CPU 之间没有区别，平等地访问内存、外设、一个操作系统。操作系统管理着一个队列，每个处理器依次处理队列中的进程。如果两个处理器同时请求访问一个资源（例如同一段内存地址），由硬件、软件的锁机制去解决资源争用问题。



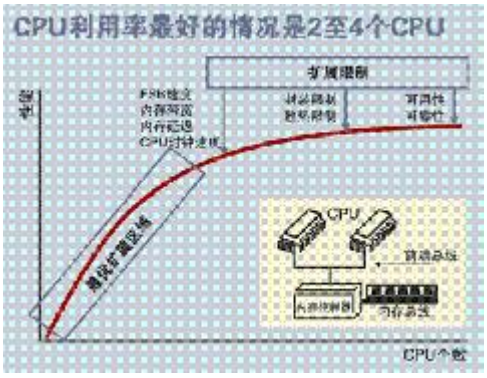
所谓对称多处理器结构，是指服务器中多个 CPU 对称工作，无主次或从属关系。各 CPU 共享相同的物理内存，每个 CPU 访问内存中的任何地址所需时间是相同的，因此 SMP 也被称为一致存储器访问结构 (UMA ： Uniform Memory Access) 。对 SMP 服务器进行扩展的方式包括增加内存、使用更快的 CPU 、增加 CPU 、扩充 I/O(槽口数与总线数) 以及添加更多的外部设备 (通常是磁盘存储) 。
SMP 服务器的主要特征是共享，系统中所有资源 (CPU 、内存、 I/O 等) 都是共享的。

也正是由于这种特征，导致了 SMP 服务器的主要问题，那就是它的扩展能力非常有限。对于 SMP 服务器而言，每一个共享的环节都可能造成 SMP 服务器扩展时的瓶颈，而最受限制的则是内存。由于每个 CPU 必须通过相同的内存总线访问相同的内存资源，因此随着 CPU 数量的增加，内存访问冲突将迅速增加，最终会造成 CPU 资源的浪费，使 CPU 性能的有效性大大降低。实验证明， SMP 服务器 CPU 利用率最好的情况是 2 至 4 个 CPU 。



3.1.3.2 NUMA (Non-Uniform Memory Access)

由于 SMP 在扩展能力上的限制，人们开始探究如何进行有效地扩展从而构建大型系统的技术， NUMA 就是这种努力下的结果之一。利用 NUMA 技术，可以把几十个 CPU(甚至上百个 CPU) 组合在一个服务器内。其 CPU 模块结构如图 2 所示：



NUMA 服务器的基本特征是具有多个 CPU 模块，每个 CPU 模块由多个 CPU(如 4 个) 组成，并且具有独立的本地内存、 I/O 槽口等。由于其节点之间可以通过互联模块 (如称为 Crossbar Switch) 进行连接和信息交互，因此每个 CPU 可以访问整个系统的内存 (这是 NUMA 系统与 MPP 系统的重要差别) 。显然，访问本地内存的速度将远远高于访问远地内存 (系统内其它节点的内存) 的速度，这也是非一致存储访问 NUMA 的由来。由于这个特点，为了更好地发挥系统性能，开发应用程序时需要尽量减少不同 CPU 模块之间的信息交互。

利用 NUMA 技术，可以较好地解决原来 SMP 系统的扩展问题，在一个物理服务器内可以支持上百个 CPU 。比较典型的 NUMA 服务器的例子包括 HP 的 Superdome 、SUN15K 、IBMp690 等。

但 NUMA 技术同样有一定缺陷，由于访问远地内存的延时远远超过本地内存，因此当 CPU 数量增加时，系统性能无法线性增加。如 HP 公司发布 Superdome 服务器时，曾公布了它与 HP 其它 UNIX 服务器的相对性能值，结果发现， 64 路 CPU 的 Superdome (NUMA 结构) 的相对性能值是 20 ，而 8 路 N4000(共享的 SMP 结构) 的相对性能值是 6.3 。

从这个结果可以看到， 8 倍数量的 CPU 换来的只是 3 倍性能的提升。

3.1.4 中断亲和性 (IRQ Affinity)

将特定的网卡中断绑定到特定的 ARM 核心，避免核心间频繁的上下文切换。

3.2 软件层的优化

软件层的核心在于减少操作系统内核与用户应用程序之间的数据交换开销。

3.2.1 DPDK (Data Plane Development Kit):

绕过 Linux 内核协议栈（Kernel Bypass），直接在用户态处理数据包，极大减少了系统调用和中断产生的开销。

3.2.2 XDP (eXpress Data Path)

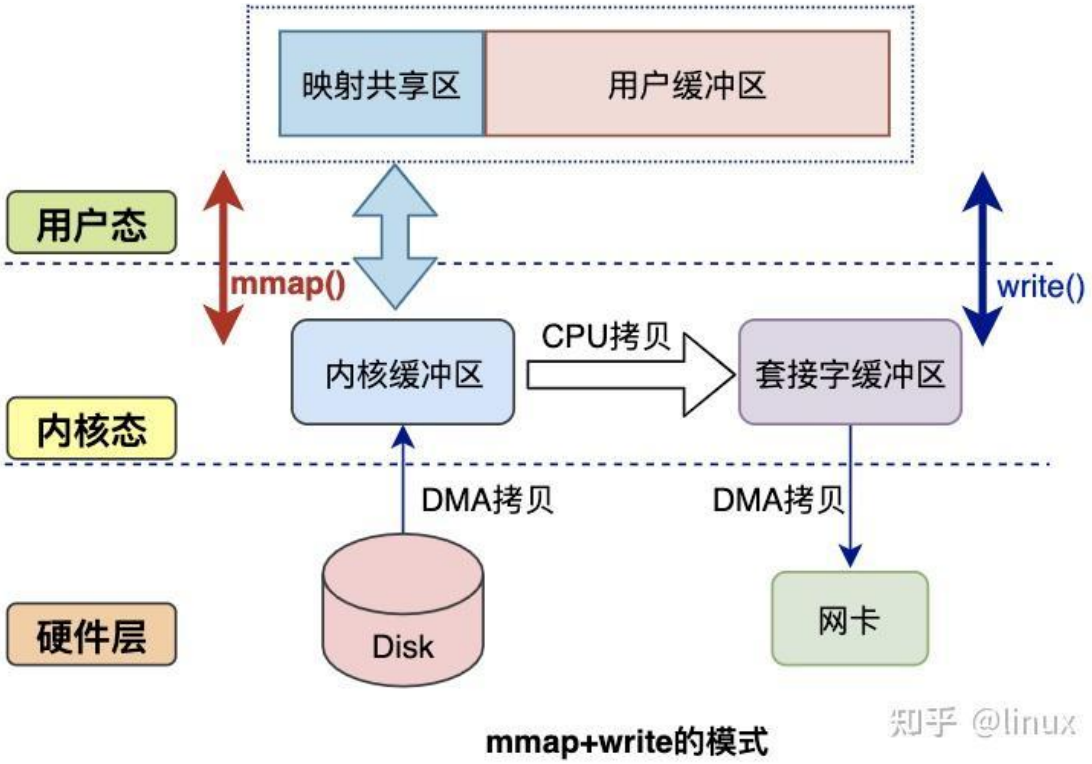
在 Linux 内核中更早地拦截数据包，利用 eBPF 技术在 ARM 核心上实现极速转发。

3.2.3 零拷贝技术 (Zero-copy)

我们可以看到，如果应用程序不对数据做修改，从内核缓冲区到用户缓冲区，再从用户缓冲区到内核缓冲区。两次数据拷贝都需要 CPU 的参与，并且涉及用户态与内核态的多次切换，加重了 CPU 负担。
我们需要降低冗余数据拷贝、解放 CPU，这也就是零拷贝 Zero-Copy 技术。目前来看，零拷贝技术的几个实现手段包括：mmap+write、sendfile、sendfile+DMA 收集、splice 等。

3.2.3.1 mmap 方式

mmap 是 Linux 提供的一种内存映射文件的机制，它实现了将内核中读缓冲区地址与用户空间缓冲区地址进行映射，从而实现内核缓冲区与用户缓冲区的共享。
这样就减少了一次用户态和内核态的 CPU 拷贝，但是在内核空间内仍然有一次 CPU 拷贝。

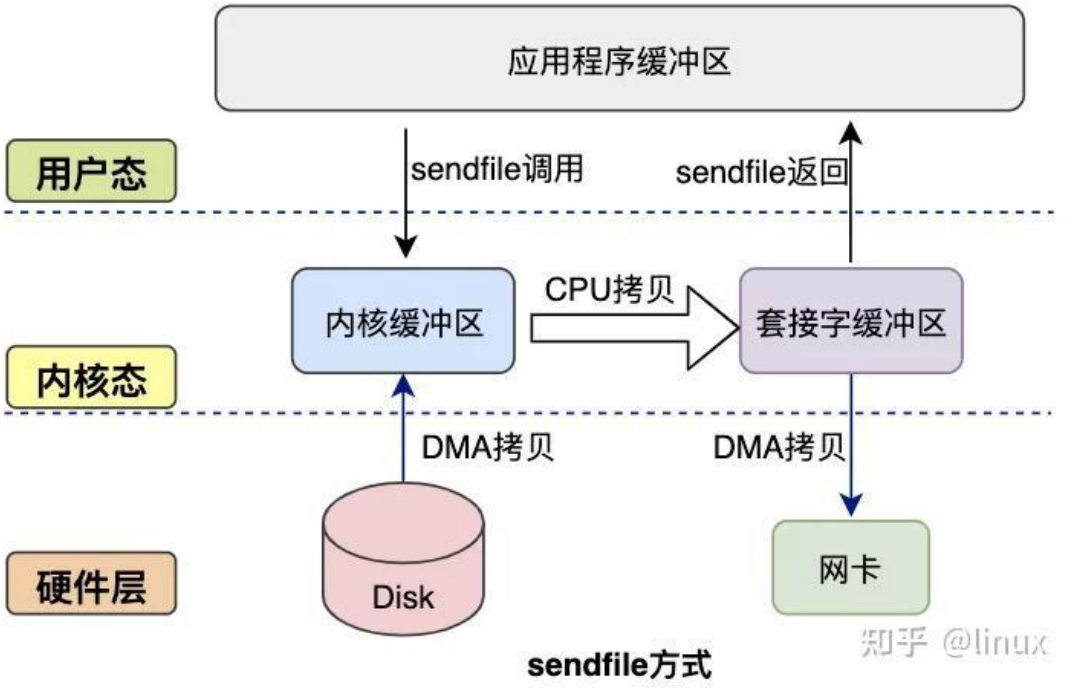


mmap+write的模式

mmap 对大文件传输有一定优势，但是小文件可能出现碎片，并且在多个进程同时操作文件时可能产生引发 coredump 的 signal。

3.2.3.2 sendfile 方式

mmap+write 方式有一定改进，但是由系统调用引起的状态切换并没有减少。
sendfile 系统调用是在 Linux 内核 2.1 版本中被引入，它建立了两个文件之间的传输通道。
sendfile 方式只使用一个函数就可以完成之前的 read+write 和 mmap+write 的功能，这样就少了 2 次状态切换，由于数据不经过用户缓冲区，因此该数据无法被修改。

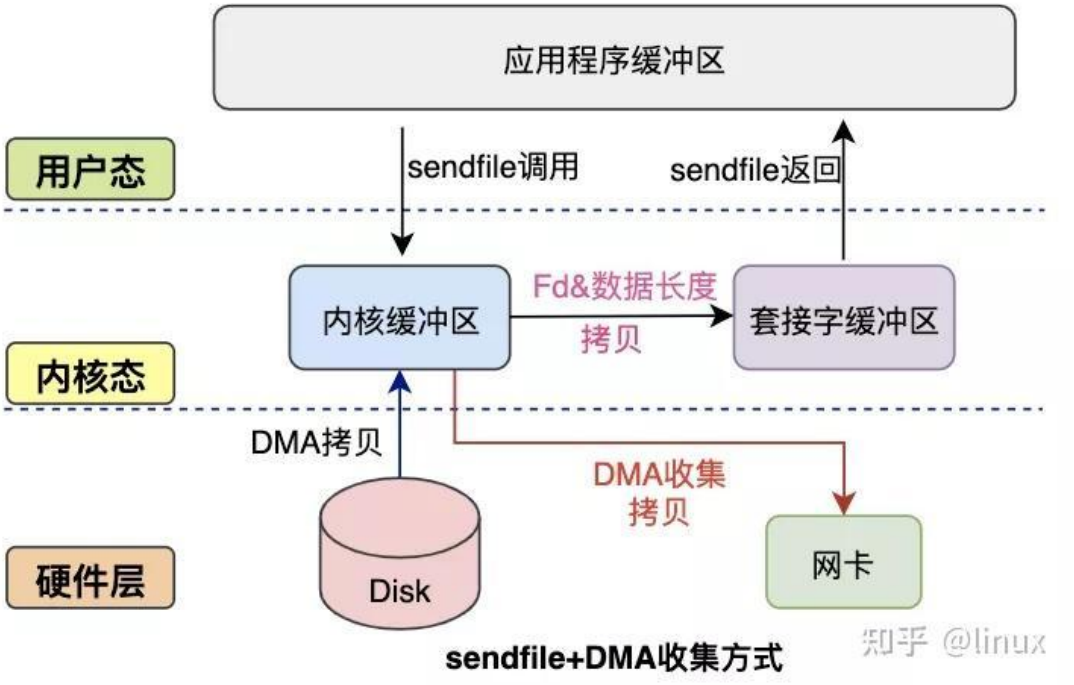


sendfile方式

从图中可以看到，应用程序只需要调用 sendfile 函数即可完成，只有 2 次状态切换、1 次 CPU 拷贝、2 次 DMA 拷贝。但是 sendfile 在内核缓冲区和 socket 缓冲区仍然存在一次 CPU 拷贝，或许这个还可以优化。

3.2.3.3 sendfile+DMA 收集

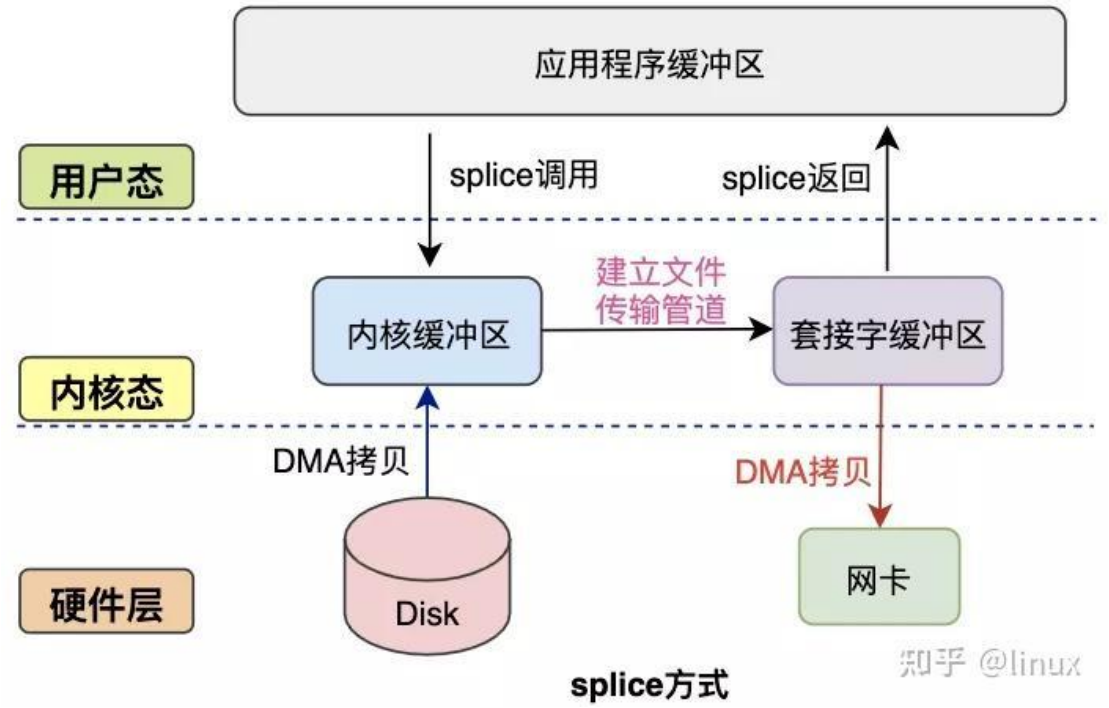
Linux 2.4 内核对 sendfile 系统调用进行优化，但是需要硬件 DMA 控制器的配合。升级后的 sendfile 将内核空间缓冲区中对应的数据描述信息（文件描述符、地址偏移量等信息）记录到 socket 缓冲区中。DMA 控制器根据 socket 缓冲区中的地址和偏移量将数据从内核缓冲区拷贝到网卡中，从而省去了内核空间中仅剩 1 次 CPU 拷贝。



这种方式有 2 次状态切换、0 次 CPU 拷贝、2 次 DMA 拷贝，但是仍然无法对数据进行修改，并且需要硬件层面 DMA 的支持，并且 sendfile 只能将文件数据拷贝到 socket 描述符上，有一定的局限性。

3.2.3.4 splice 方式

splice 系统调用是 Linux 在 2.6 版本引入的，其不需要硬件支持，并且不再限定于 socket 上，实现两个普通文件之间的数据零拷贝。



3.2.4 异步 I/O 与事件驱动模型

利用 ARM 平台下高效的 epoll 机制，配合异步编程模型（如 Node.js, Go 协程），实现单核处理数万个并发长连接，充分利用 ARM 核心的并发优势。splice 系统调用可以在内核缓冲区和 socket 缓冲区之间建立管道来传输数据，避免了两者之间的 CPU 拷贝操作。

3.3 网络协议优化

针对不同的应用场景，通过调整协议参数和更换轻量级协议来降低延迟。

优化维度	技术手段	核心效益
TCP/IP 调优	调整 tcp_rmem / tcp_wmem 缓冲区大小	提升高带宽延迟积（BDP）环境下的吞吐量
拥塞控制	启用 BBR 算法	在丢包率较高的弱网环境下显著提升传输速率
轻量级协议	使用 QUIC (HTTP/3)	基于 UDP 减少握手延迟（0-RTT），解决队头阻塞问题

优化维度	技术手段	核心效益
数据压缩	使用 Zstandard (Zstd) 或 Protobuf	配合 NEON 指令集，在不增加延迟的前提下压缩传输体积

四、挑战与未来方向

虽然 ARM 架构在网络处理领域表现出色，但在向高性能服务器和复杂网络环境迈进的过程中，仍面临一系列技术难题。

4.1 现存挑战

1. 高并发网络请求处理的功耗问题：尽管 ARM 以能效比著称，但在处理“C10M”（千万级并发）请求时，CPU 需要长时间维持在高频率。随着核心数增加，多核间的热量堆积和漏电流问题依然严峻。在边缘计算节点中，如何在有限的散热条件下平衡处理性能与峰值功耗，是架构设计的难点。

2. 异构计算与内核优化难点：ARM 生态中存在大量的定制化方案（如 big.LITTLE 架构、特定的加速模块）。这种生态碎片化导致通用的 Linux 内核优化很难在所有 ARM 平台上达到最优。开发者往往需要针对特定芯片（如 AWS Graviton 或 Ampere Altra）进行深度的底层适配，增加了研发成本。

3. 内存和缓存带宽瓶颈：网络 I/O 是典型的访存密集型任务。虽然 ARM 核心数在增加，但如果 L3 缓存容量或 内存通道带宽（Memory Wall）跟不上核心增长速度，就会导致 CPU 在等待数据包从内存传输时出现严重的“空转”现象。

4.2. 未来发展方向

- ARM 服务器在数据中心的应用（Cloud-Native ARM） 随着 AWS、Google、阿里等云厂商推出自研 ARM 处理器，未来网络请求将更加“云原生化”。

1. TCO（总拥有成本）优化： 利用 ARM 的高集成度，在单台机架内实现更高的算力密度。
2. 定制化网络卸载： 结合 DPU（数据处理单元），将网络虚拟化、防火墙逻辑彻底从通用 CPU 卸载到硬件层。

- 网络智能加速（AI + 网络）

1. 流量预测与调度： 利用 ARM 处理器中集成的 NPU（神经网络处理器），实时分析

流量模式，动态调整频率和路由策略以节省功耗。

2. 智能防御： 在硬件层集成 AI 算法，实现毫秒级的 DDoS 攻击检测与过滤。

- 新型网络协议与 ARM 优化的结合

1. SVE/SVE2 指令集应用： 利用 ARM 的**可伸缩矢量扩展 (SVE) **进一步加速 QUIC 和 HTTP/3 的复杂加密运算。
2. 软硬一体化设计： 协议栈（如协议解析、重传逻辑）将更多地固化到 SoC 的逻辑电路中，实现“协议即硬件”，从而彻底消除软件处理带来的延迟。

参考文献

[1] ARM 与神经网络处理器的通信方案设计, EEWorld, 2022 (或页面最后更新时间如可查), 可用: https://www.eeworld.com.cn/mcu/hisic531405.html（访问日期：2025-12-20）

[2] NEON 加速，博客园，作者 tibetanmastiff，https://www.cnblogs.com/tibetanmastiff/p/16876668.html（访问日期：2025-12-20）

[3] SMP、NUMA、MPP 体系结构介绍，腾讯云开发者社区，https://cloud.tencent.com/developer/article/1527941（访问日期：2025-12-20）

[4] 【Linux】图文并茂 | 彻底搞懂零拷贝（Zero-Copy）技术，知乎专栏，https://zhuanlan.zhihu.com/p/362499466（访问日期：2025-12-20）