



南開大學  
Nankai University

**multiswap**

基于 RDMA 的多节点异质内存系统

功能挑战赛开发文档

成员：曹骜天 朱奕翔 许宸

学校：南开大学

指导教师：宫晓利

2024 年 8 月 15 日

## 摘要

随着内存需求的不断增长，远程直接内存访问（RDMA）技术的引入为解决内存不足问题提供了一种可行的方案。RDMA 允许一台机器直接访问另一台机器的内存，在系统内存不足时，可以利用远程内存进行数据交换，从而扩展本地资源。尽管 Compute Express Link（CXL）技术也具备广阔的前景，但基于 RDMA 的交换机制由于其良好的兼容性，使得现有软件无需修改即可使用，因此在未来的计算机数据中心的仍将广泛存在。在这种背景下，通过 RDMA 所实现的远程内存池服务可以由多个内存节点同时提供，并且这些内存节点由于资源分配、内存效率、带宽预留以及节点内存带宽压力等因素的不同，可能存在性能差异。因此，如何在多个内存池节点中合理选择目标内存池进行内存交换操作变得尤为重要。

为了解决上述问题，我们首先利用 Linux 中 swap 子系统对多设备的原生支持和 front-swap 的可扩展性，基于 RDMA 实现了远程内存页面传输协议。而后构建了多节点混合远程内存池系统 MultiswapSys。该系统是拥有一个计算节点和两个内存节点的内存池集群，用于模拟数据中心的机器存在闲置内存的情况。最后通过实验进行流量监测与控制，分析了各个内存节点在不同带宽下系统的性能表现，并在计算节点上实现了的页面调度策略，通过为每一个换出的页选择合适的内存节点，显著提高了混合远程内存池系统的性能。

基于上述搭建的系统环境，我们分为了系统实现正确性与有效性两方面进行测试。在正确性测试方面，我们让负载产生内存压力以强制触发 swap 机制，证明了所实现远程内存节点的正确性。在有效性测试方面，我们首先在不限制带宽的情况下使一个计算节点匹配一个或两个内存节点分别进行测试，将特定工作集运行时长缩短了 50% 以上，同时在 Kmeans、tensorflow 等十分具有现实意义的测试集上进行相同测试，使其性能也获得了显著提升。然后对两个内存节点的带宽做了不同的限制，针对测试集运行稳定时每秒的吞吐量以及运行总时间，通过实现页面调度策略，成功地在带宽限制下的系统性能提升了 25% 以上。最后以此说明了我们本项研究工作的必要性，为未来大规模数据中心和高性能计算环境中的远程内存技术应用提供了有价值的参考。

**关键词：**远程内存，RDMA，内存池，页面传输，性能优化

# 目录

<b>1 项目介绍</b>	<b>5</b>
<b>2 背景知识</b>	<b>6</b>
2.1 基于 Frontswap 接口的页面置换机制	6
2.2 RDMA 通信原理介绍	7
2.2.1 传统网络与 RDMA 的对比	7
2.2.2 应用场景	8
2.2.3 RDMA 协议	8
2.2.4 RoCE 协议	8
2.3 RDMA 的通信方式	9
2.3.1 Send/Receive 操作	9
2.3.2 Read/Write 操作	9
2.4 NVIDIA OFED 栈结构	10
2.5 硬件支持	11
2.5.1 第五代 RDMA 网卡	11
2.5.2 NVIDIA Mellanox SN2700 交换机	12
<b>3 系统设计与开发计划</b>	<b>13</b>
3.1 项目目标	13
3.2 系统框架结构	14
3.3 系统开发环境	15
3.4 技术线路	15
<b>4 阶段一：实现远程内存页面传输机制</b>	<b>17</b>
4.1 驱动安装与环境配置	17
4.2 实现在用户态的 RDMA 通信	21
4.2.1 RDMA 编程	21
4.2.2 用户态 RDMA 的连接建立与断开	23
4.2.3 RDMA 连接双端交互过程	25
4.2.4 用户态程序运行	27
4.3 Frontswap 接口介绍及使用	28
4.4 Frontswap 的接口实现:multiswap	30
4.4.1 内核模块的编写与编译	30
4.4.2 为 Linux 内核添加补丁 1	32
4.5 使用 DRAM 作为 multiswap 的 BACKEND	32
4.5.1 read、write 接口函数的实现	32
4.5.2 对 far memory 的大小要求	34
4.5.3 DRAM BACKEND 正确性测试	37
4.5.4 为 Linux 内核添加补丁 2	37
4.6 使用 RDMA 作为 multiswap 的 BACKEND	38
4.6.1 内核态 RDMA 连接建立的交互过程	38

4.6.2	为远程内存节点编写 server 程序 . . . . .	39
4.6.3	内核态 RDMA 连接的建立与销毁 . . . . .	41
4.6.4	在内核态实现远端 CPU 无感知的 RDMA 通信 . . . . .	47
4.6.5	单个远程内存节点正确性测试 . . . . .	50
<b>5</b>	<b>阶段二：配置交换机并连接两个内存节点</b>	<b>51</b>
5.1	SN2700 交换机配置 . . . . .	51
5.2	配置两个远程内存节点 . . . . .	55
5.2.1	配置各端口 . . . . .	55
5.2.2	RDMA BACKEND 的修改与优化 . . . . .	56
5.2.3	尝试不同粒度的页面映射方式 . . . . .	58
<b>6</b>	<b>阶段三：构造异构内存节点</b>	<b>59</b>
6.1	使用交换机限制带宽 . . . . .	59
6.2	尝试进行页面调度以优化系统性能 . . . . .	63
6.3	缺点及改进方法 . . . . .	64
6.4	问题解决 (决赛第二阶段新工作) . . . . .	64
6.4.1	构建 pageid 到远程内存的映射 . . . . .	65
6.4.2	获取内存节点的可用 slot . . . . .	66
6.4.3	更新映射时机的选择 . . . . .	68
6.4.4	评估与改进 . . . . .	69
<b>7</b>	<b>正确性测试</b>	<b>70</b>
7.1	DRAM BACKEND 实现正确性测试 . . . . .	71
7.2	RDMA BACKEND 实现正确性测试 . . . . .	71
7.2.1	内存节点操作 . . . . .	71
7.2.2	计算节点操作 . . . . .	73
<b>8</b>	<b>性能测试</b>	<b>73</b>
8.1	硬件配置 . . . . .	73
8.2	benchmark 选择 . . . . .	73
8.3	环境搭建 . . . . .	74
8.4	端到端性能测试 . . . . .	74
8.4.1	测试工具 . . . . .	74
8.4.2	cgroup 限制内存 . . . . .	75
8.4.3	测试脚本使用 . . . . .	75
8.5	性能比较 . . . . .	75
8.6	异构节点测试 . . . . .	79
<b>9</b>	<b>遇到的困难以及解决方案</b>	<b>80</b>
<b>10</b>	<b>总结与展望</b>	<b>80</b>
10.1	项目总结 . . . . .	80
10.2	未来展望 . . . . .	80



11 比赛收获	81
12 致谢	81

# 1 项目介绍

在现代计算环境中，随着数据量的爆炸性增长，内存资源成为计算性能的重要瓶颈之一。为了应对内存不足的问题，传统的方法包括增加本地内存或使用磁盘交换。然而，这些方法都有各自的限制。远程直接内存访问（RDMA）技术的出现，为内存扩展提供了一种高效的解决方案。RDMA 允许计算节点直接访问远程节点的内存，从而显著减少数据传输的延迟和带宽消耗。

RDMA 技术通过简化数据路径，消除了多次数据拷贝和协议处理，使其特别适用于高性能计算和大数据应用。基于 RDMA 的交换机制具有极好的兼容性，因为它无需对现有软件进行修改，只需在计算节点的内核中增加相应的驱动模块，并在内存节点上运行服务程序。这种高兼容性使得基于 RDMA 的内存扩展方案在未来相当长的一段时间内会广泛应用于数据中心。

在提供远程内存池服务时，不一定只依赖于单个内存节点，多个内存节点可以同时为一个计算节点提供服务。然而，这些内存节点的资源划分、内存性能以及带宽压力可能各不相同。因此，如何在多个内存节点中合理选择目标内存池来完成内存交换操作，成为一个非常有价值的研究课题。

对于大型服务器集群来说，将已有的部分节点的空闲内存作为远程内存，往往比添加专用远程内存节点经济效益更高，资源利用率也更高，但这样可能会导致单个远程节点的内存不足以支持计算节点的内存需求，于是延伸出了单个计算节点连接多个远程内存节点的问题。由于这些节点的内存效率、带宽预留以及节点内存带宽压力的不同，它们之间存在性能差异，导致在访问中出现性能瓶颈从而影响访问效率。在本项目中，我们主要聚焦于混合远程内存池系统的搭建以及节点间的带宽差异问题。因此，我们希望探讨在单个计算节点通过交换机适配两个异构远程内存节点情况下的性能问题，真实构建了如图1.1所示的混合远程内存池系统。图中椭圆所标识位置为内存节点，方框所标识位置为计算节点，下方为以太网交换机。

鉴于目前 RDMA 驱动已在低版本内核中不再支持，我们决定在 Linux 6.1 版本的内核上进行开发，在 swap 子系统中实现基于 frontswap 的页面置换机制。本项目主要在 NVIDIA MLNX\_OFED DRIVER 提供的 verbs + CMA (ib\_core) 接口之上进行，利用 RDMA 技术实现远内存交换。通过 RDMA，我们能够直接访问远程节点的内存，而无需经过操作系统内核的干预，这显著提高了数据传输的效率和带宽。具体而言，verbs 接口提供了对 RDMA 硬件的低级访问，使得我们可以精确控制数据传输的每个细节；而 CMA (Connection Manager Assistant) 则简化了连接管理，确保了通信的可靠性和可扩展性。在我们的设计中，内核程序通过调用 verbs API 进行数据传输操作，例如读取和写入远程内存，同时使用 CMA 来建立和维护与远程节点的连接。这种方法不仅提高了系统的吞吐量和响应时间，还减少了 CPU 的使用（本项目中，本地 CPU 仅参与控制面操作，远端 CPU 不参与操作），使得资源能够被更有效地利用，从而满足高性能计算和大规模数据处理的需求。

本项目的目标是利用 RDMA 技术构建一个高性能的远程内存池系统，并优化其在带宽受限情况下的性能表现。具体来说，我们实现了基于 RDMA 的远程内存页面传输协议并证实了其可用性，构建了多节点混合远程内存池系统 MultiswapSys。通过实验分析不同带宽限制下系统的性能表现，提出了物理页面分配方法，从而在带宽受限情况下将系统性能提升了 25% 左右。我们的工作为未来大规模数据中心和高性能计算环境中的远程内存技术应用提供了有价值的参考。

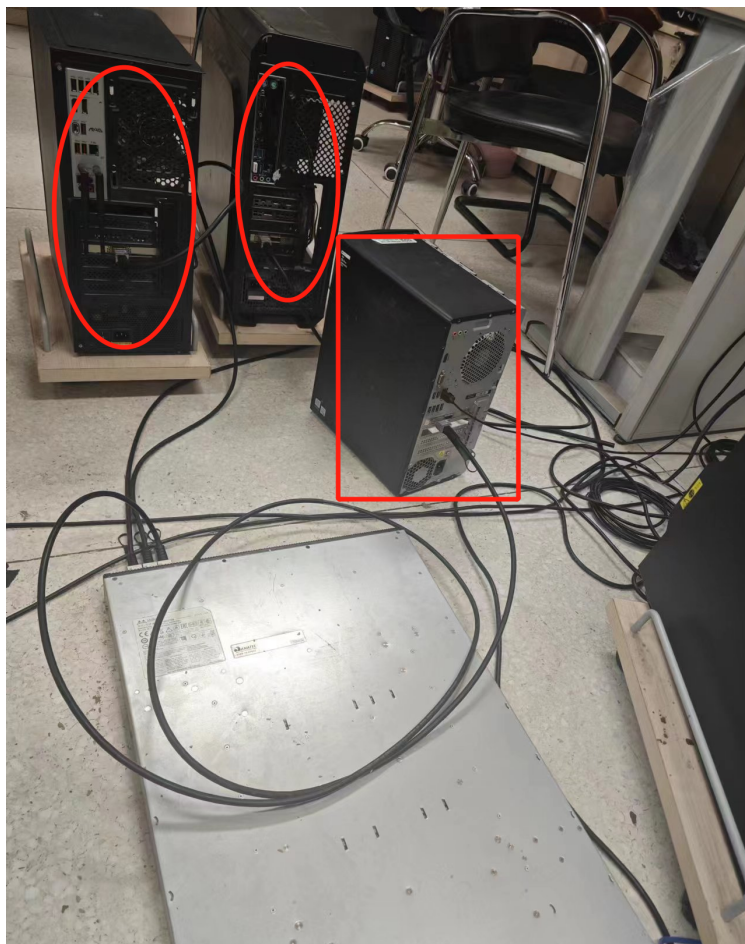


图 1.1: 系统实物连接图

## 2 背景知识

### 2.1 基于 Frontswap 接口的页面置换机制

在本项目的研究过程中，我们重点调研了几篇关键的论文，均通过 Frontswap 接口实现页面置换机制，这些研究为我们提供了重要的技术支持和参考。

首先，Amaro 等人在 2020 年提出的 *Fastswap*<sup>[1]</sup> 是一种优化的 Linux 交换系统。*Fastswap* 通过以下几种方式实现了基于 RDMA 的页面置换：

- **前置交换接口 (Frontswap)**：用于页面粒度的交换，减少上下文切换，提高性能。
- **多队列机制**：使用两个队列，一个用于关键操作，一个用于预取操作，避免头阻塞。
- **轮询机制**：关键操作完成时使用轮询机制，减少延迟，提升页面获取速率。

Wang 等人在 2022 年提出的 *Memliner*<sup>[8]</sup> 基于 *Fastswap* 进行了进一步的优化。*Memliner* 在 Linux 5.4 内核上开发，完成了对 *Fastswap* 的一些补充和改进，使其在更高版本的内核中能够更好地发挥作用。这些改进包括：

- **增强的页面置换算法**：进一步优化了页面置换过程中的性能。
- **改进的内存管理机制**：通过更有效的内存分配策略，提高了系统的整体效率。

通过这些相关论文的调研，我们深入理解了当前领域的研究现状和技术难点。这为我们的系统设计和优化提供了坚实的理论基础，确保了本项目能够采用最新的技术手段和最佳实践，从而提高系统性能并解决现有技术中的问题。这些研究表明，使用 RDMA 技术进行远程内存交换具有显著的性能优势。Fastswap 和 Memliner 的实现为我们提供了宝贵的技术参考，使我们能够在 Linux 6.1 内核上开发出基于 Frontswap 的页面置换机制。在此基础上，我们设计并实现了 Multiswap 系统，成功地将页面交换到多个异构的远程内存中，并优化了多节点混合远程内存池系统的性能。

## 2.2 RDMA 通信原理介绍

**RDMA (Remote Direct Memory Access)** 技术可以使一个节点直接访问远程节点的内存，该操作与访问本地内存相似。与传统的网络通信方式不同，RDMA 允许数据在两台计算机之间传输时绕过复杂的 TCP/IP 协议栈。在该过程中，远端 CPU 不会时刻参与传输，大部分工作直接由网卡和内存完成。

### 2.2.1 传统网络与 RDMA 的对比

在传统网络中，数据传输需要通过多次拷贝和协议栈的处理，这对 CPU 的依赖较强。而使用 RDMA 技术时，数据可以通过专用的网络接口卡 (NIC) 直接在两台机器的内存之间传输，大幅减少了 CPU 的参与。具体优势包括：

- **零拷贝 (Zero Copy)**：避免了数据在用户空间和内核空间之间的多次拷贝，减少了数据传输的总体延迟，因为数据传输过程减少了对 CPU 的依赖，CPU 资源可以被用于其他计算任务，从而提高了系统的整体效率，零拷贝使得数据传输过程更高效，网络带宽可以得到更充分的利用。
- **内核旁路 (Kernel Bypass)**：RDMA 提供一个专用的 Verbs interface 而不是传统的 TCP/IP Socket interface。应用程序可以直接在用户态执行数据传输，不需要在内核态与用户态之间做上下文切换
- **CPU 卸载**：传统网络通信需要操作系统内核参与大量的协议处理工作，包括数据包的封装、解析和校验等。RDMA 技术允许这些操作在硬件级别完成，极大地减轻了 CPU 的工作负担。硬件级别的数据包处理通常比软件实现更快，因为它可以并行处理多个操作，并专门针对这些任务进行优化。

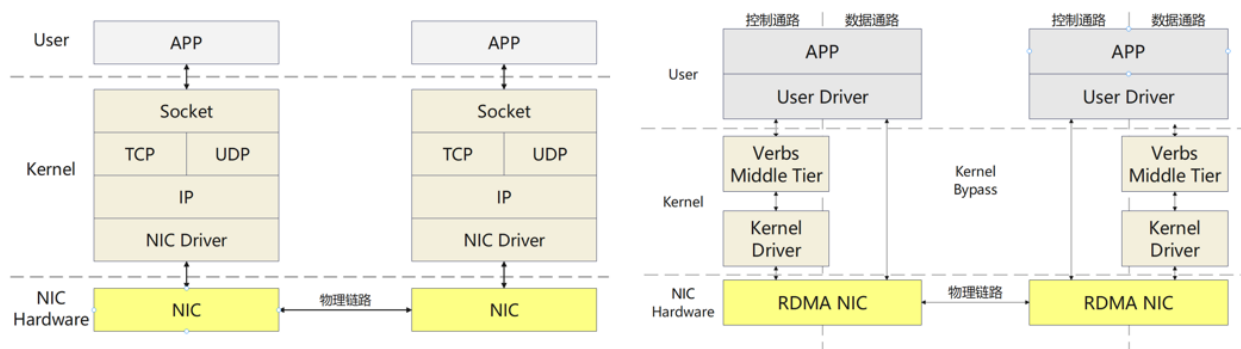


图 2.2: 传统以太网通信和 RDMA 通信的区别

图2.2体现了传统以太网通信和 RDMA 通信的区别。传统以太网通信流程中，应用程序通过 Socket 接口使用 TCP/IP 协议栈进行数据传输，这个过程中需要依赖 Linux 内核处理数据包的收发，并通过

驱动程序与物理网卡 (NIC) 进行交互, 最终通过物理链路进行数据传输。传统以太网通信流程中, 需要 CPU 参与的把数据从用户空间拷贝到内核空间, 以及同样需要 CPU 全程参与的数据包组装和解析, 数据量大的情况下, 这将对 CPU 造成很大的负担。这种通信方式存在协议栈处理和内核态与用户态切换的开销, 导致网络延迟相对较高。

而 RDMA 通信则不同, 在 RDMA 中存在额外的数据通路。具体体现在用户态驱动与 RDMA 网卡之间的数据传输路径。RDMA 通过绕过内核, 能够直接从用户空间访问 RDMA 网卡 (RNIC)。RNIC, 即 RDMA 网络接口卡, 内部包含缓存页表条目 (Cached Page Table Entry), 用于将虚拟页面映射到对应的物理页面。这种设计显著提高了数据传输的效率和性能, 减少了内存复制和延迟。这条直接的数据通路绕过了内核态, 减少了数据在内核和用户态之间切换的开销。此外, RDMA 通信中还保留了一条传统的数据通路用于控制和管理操作。例如, RDMA 设备的初始化和配置仍然需要通过内核驱动进行。这条路径确保了 RDMA 设备的正确管理和控制, 但不参与实际的数据传输。

从图中可以看出, 传统网络的结构包含多个复杂层次, 每一层都可能成为性能瓶颈。相比之下, RDMA 通过简化数据路径, 消除了多次数据拷贝和协议处理, 大大减少了延迟和带宽消耗。因此, RDMA 特别适用于高性能计算和大数据应用, 其中高速数据传输和低延迟是关键要求。

### 2.2.2 应用场景

RDMA 技术在高性能计算 (HPC) 和大型数据中心中广泛应用, 其主要优势在于提供了更高的带宽和更低的时延, 非常适合需要快速数据传输和同步的场景。

### 2.2.3 RDMA 协议

RDMA 协议包括:

- **Infiniband (IB)**: 定义了一整套从链路层到传输层的协议, 提供最高性能, 但需要专用硬件。
- **RDMA over Converged Ethernet (RoCE)**: 基于以太网的 RDMA 协议, 分为 RoCE v1 和 RoCE v2, 后者支持路由。
- **internet Wide Area RDMA Protocol (iWARP)**: 在现有的以太网基础上实现 RDMA 功能。

我们项目的 RDMA 交换设备主要采用 RoCE v2 协议。在下一小节中将其进行着重介绍:

### 2.2.4 RoCE 协议

RoCE (RDMA over Converged Ethernet) 是一种在以太网上实现 RDMA (远程直接内存访问) 的网络协议。它允许数据在服务器之间高效地直接传输, 绕过操作系统, 从而大幅减少延迟并提高数据中心的效率。RoCE 是一种重要的技术, 因为它结合了传统以太网的通用性和 RDMA 技术的高性能特点。

RoCE v2 在 RDMA 和以太网之间引入了 UDP/IP 层。这使得 RoCE v2 可以在任何支持 IP 路由的标准以太网基础设施上运行, 包括广域网。RoCE v2 通过 UDP 封装, 提供了更好的兼容性和灵活性, 使得 RDMA 技术可以应用于更广泛的网络环境。

## 2.3 RDMA 的通信方式

RDMA 技术支持两种主要的数据传输操作：**Read/Write** 和 **Send/Receive**，适用于不同的应用场景。

### 2.3.1 Send/Receive 操作

Send/Receive 操作是网络通信的基本操作，完成一次通信过程需要两端 CPU 的参与。

- **RDMA Send**：在此模式下，数据通过创建发送和接收队列的方式在两台主机之间传输，需要队列管理。
- **RDMA Receive**：接收方需要预先设置好接收队列，并准备好相应的内存缓冲区以接收数据。

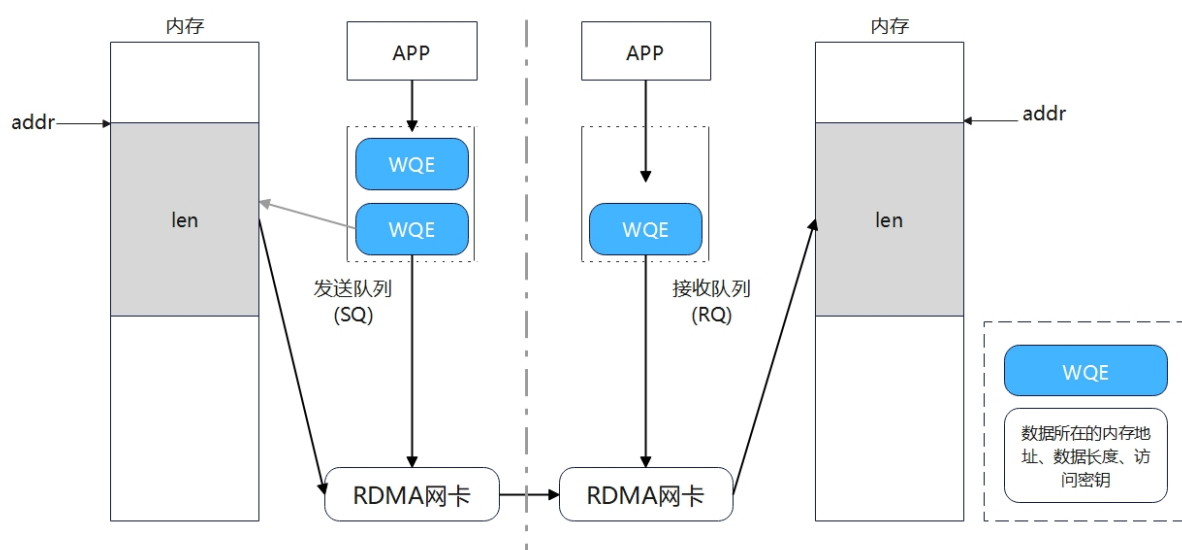


图 2.3: Send/Receive 操作

### 2.3.2 Read/Write 操作

在准备阶段，本端通过数据交换，从 WQE 中获得对端某个可用内存区域的地址和“钥匙”，即对该远端内存的读写权限。获得权限后，本端可以像访问自身内存一样，直接读写这一远端内存区域，这样就可以绕过 CPU 达成读写的目的。

- **RDMA Read**：允许一台主机直接从另一台主机的内存中读取数据，无需远程 CPU 的干预。
- **RDMA Write**：允许一台主机直接将数据写入另一台主机的内存，同样不需要远程 CPU 的干预。

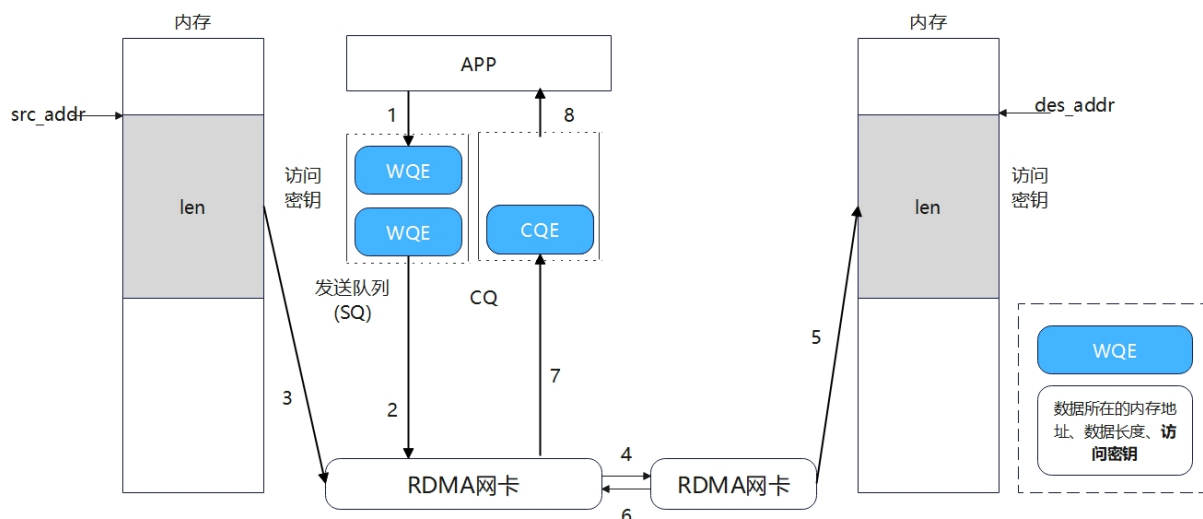


图 2.4: Write 操作示意图

Read/Write 操作的设备相同，只是操作的路径相反。Write 操作的主要过程如图2.4:

1. 请求端 APP 以 WQE (WR) 的形式下发一次 WRITE 任务。
2. 请求端硬件从 SQ 中取出 WQE，解析信息。
3. 请求端网卡根据 WQE 中的虚拟地址，转换得到物理地址，然后从内存中拿到待发送数据，组装数据包。
4. 请求端网卡将数据包通过物理链路发送给响应端网卡。
5. 响应端收到数据包，解析目的虚拟地址，转换成本地物理地址，解析数据，将数据放置到指定内存区域。
6. 响应端回复 ACK 报文给请求端。
7. 请求端网卡收到 ACK 后，生成 CQE，放置到 CQ 中。
8. 请求端 APP 取得任务完成信息。

## 2.4 NVIDIA OFED 栈结构

我们在 NVIDIA 提供的 MLNX\_OFED DRIVER 上进行开发，它是一个由 NVIDIA 提供的高性能网络软件包，主要用于增强 InfiniBand 和以太网 RDMA(RoCE) 的性能。它基于 OpenFabrics Alliance 提供的开源软件包，经过 NVIDIA 的优化和扩展 [6]。图2.5为 NVIDIA OFED 的栈结构，展示了各层如何交互，以及上层协议 (ULP) 如何与硬件、内核和用户空间进行接口。通过这种分层结构，能够适用于存储、高性能计算、数据中心、嵌入式系统和以太网管理等多种市场需求。



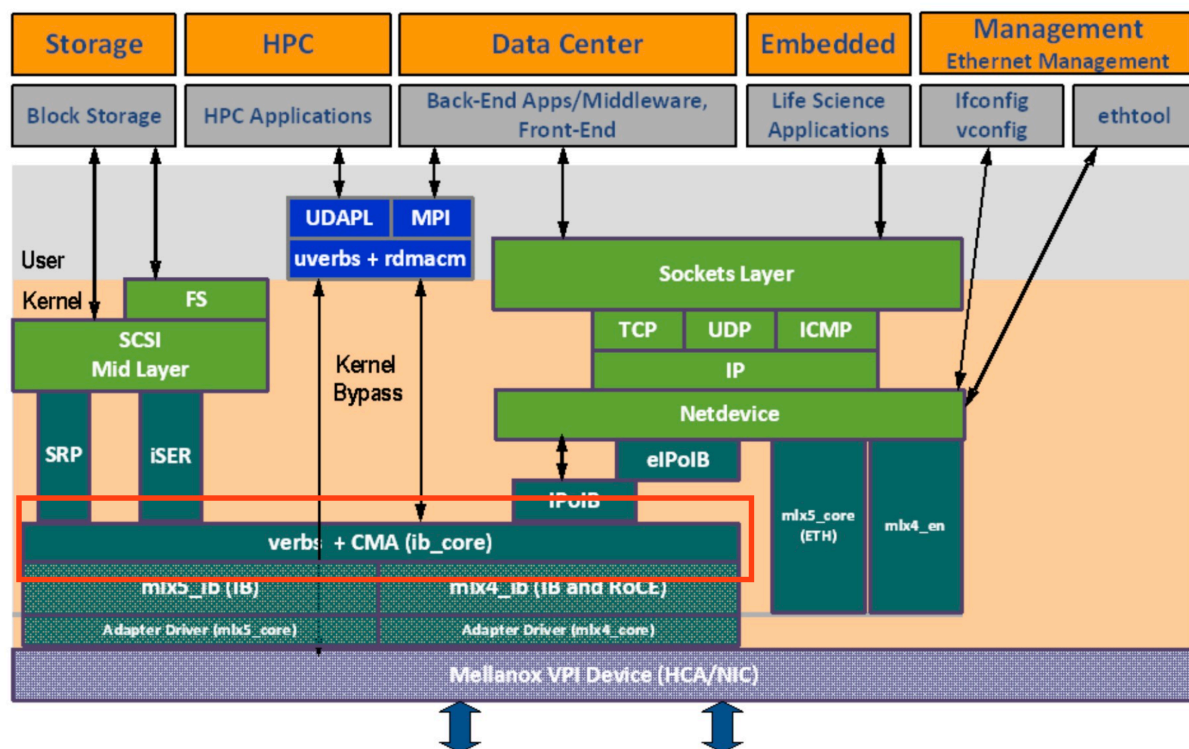


图 2.5: OFED 栈结构

下面对各层做简要介绍：应用层涵盖了存储、高性能计算、数据中心、嵌入式系统和管理模块，负责高效的数据传输、并行计算、网络配置和性能调优。

用户态层包括文件系统和内核旁路，确保数据高效传输和减少延迟，实现快速的网络通信。

核心协议层包括 Sockets 层、IP 层、Netdevice 和 eIPoIB，提供基本网络接口、数据路由和设备管理，支持高速网络连接。

驱动层包括 verbs + CMA、mlx5\_ib、mlx5\_core、mlx4\_ib 和 mlx4\_en 等，我们所做的开发工作就在于该层提供的接口之上进行（图2.5中的红色框部分）。

硬件层包括 Mellanox VPI 设备，如 ConnectX 和 BlueField 适配器，支持 InfiniBand 和以太网连接，提供高性能的网络通信能力。这些适配器可以在不同的网络环境中灵活使用，满足多样化的网络需求。

## 2.5 硬件支持

### 2.5.1 第五代 RDMA 网卡

经过调研，第五代 RDMA 网卡，Nvidia Mellanox ConnectX-5 RoCE 网卡成为本项目的理想选择。

ConnectX-5 系列网卡具有 100Gb/s 的带宽。通过低延迟和高吞吐量的特性提升应用程序性能。其端口可以根据需要在以太网模式和 RDMA 模式之间切换，满足不同的应用需求，这种灵活性使其在多种网络环境中均能发挥最佳性能。该系列网卡适用于多种典型应用场景，如高性能计算（HPC），利用 RDMA 的低延迟和高带宽特点，满足超级计算和科学计算的需求；在数据中心，通过高吞吐量和灵活的端口配置，优化数据中心网络性能和资源利用率；在云计算和虚拟化环境中，提供虚拟交换机加速和硬件卸载，提升云环境下的网络性能。



我们认为使用该系列网卡能够充分发挥其高性能、灵活性和多协议支持的优势，确保本项目所设计的远程内存交换系统在性能和可靠性方面达到最佳状态。其实物图如图2.6所示：

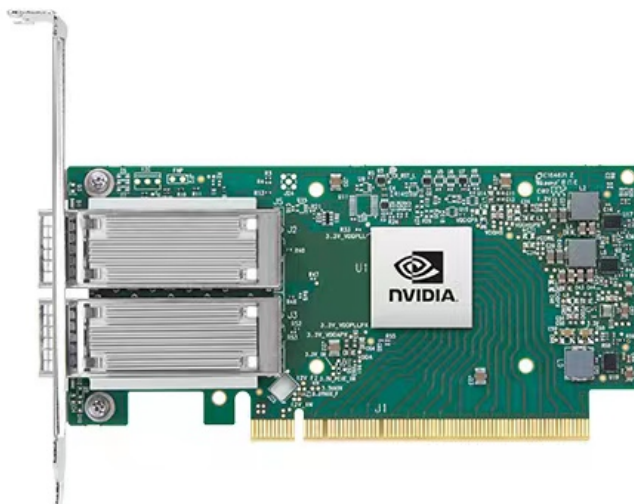


图 2.6: ConnectX-5 RoCE 网卡

我们所使用的直连线缆型号为 Mellanox MCP1600 IB EDR 100G DAC 铜缆，其实物图如图2.7所示，它提供 100Gbps 的带宽，能够满足 ConnectX-5 系列网卡的要求。



图 2.7: 连接线缆

### 2.5.2 NVIDIA Mellanox SN2700 交换机

以太网交换机可以通过网络实现不同计算节点之间内存资源的共享和动态分配。它在现代数据中心和高性能计算（HPC）环境中得到广泛使用，因其可以极大地提高资源利用率和系统性能。该项目所采用的以太网交换机为 NVIDIA Mellanox SN2700 交换机，实物如图2.8所示。NVIDIA Mellanox SN2700 交换机是一款高性能网络交换机，专为现代数据中心和高性能计算（HPC）环境设计。它主要采用 RDMA over Converged Ethernet (RoCE) 技术。这使得 SN2700 能够在以太网基础上实现远程直接内存访问（RDMA），提供低延迟和高带宽的网络连接。



图 2.8: SN2700 switch

技术规格

参数	详细说明
交换容量	12.8 Tbps
包转发率	9.52 Bpps
端口类型	32 个 QSFP28
支持的以太网速度	10GbE, 25GbE, 40GbE, 50GbE, 100GbE
延迟	< 300 ns
电源	双冗余，热插拔
风扇	热插拔，智能调速
尺寸	1U 机架安装

网络协议支持

支持广泛的网络协议，包括 IPv4、IPv6、OSPF、BGP 和 PIM 等；支持 VXLAN 和 NVGRE 等覆盖网络技术，实现数据中心网络的虚拟化。

硬件优势

- 高吞吐量：**提供高达 12.8 Tbps 的交换容量，能够支持大规模数据传输。
- 低延迟：**单跳延迟低于 300 纳秒，适用于需要快速数据交换的应用场景。
- 灵活的端口配置：**支持 32 个 100GbE QSFP28 端口，可配置为 40GbE 或 100GbE。端口可以通过分割线缆支持 10GbE 和 25GbE 接口，增加了灵活性。
- 高可靠性和可用性：**支持热插拔冗余电源和风扇模块，提高了设备的可靠性和可用性，无阻塞交换架构确保了高效的数据流通。

### 3 系统设计与开发计划

#### 3.1 项目目标

- 在 6.1 版本的 Linux 发行版内核上完成基于 Swap 子系统的 frontswap 接口的 RDMA 页面置换机制。
- 尝试为单个计算节点适配两个远程内存设备。

- 在远程内存设备性能出现差异时，分析原因与瓶颈，并尝试使用调度手段优化。

## 3.2 系统框架结构

我们最终构成的系统由三台主机构成，也就是拥有三个节点。其中一个节点为计算节点，剩余为远程内存节点。设备间的物理连接示意图如图3.9所示，将计算节点与内存节点安装上 ConnectX-5 RoCE 网卡后与交换机进行连接，交换机将会转发它们的流量，使三者能够互相访问。

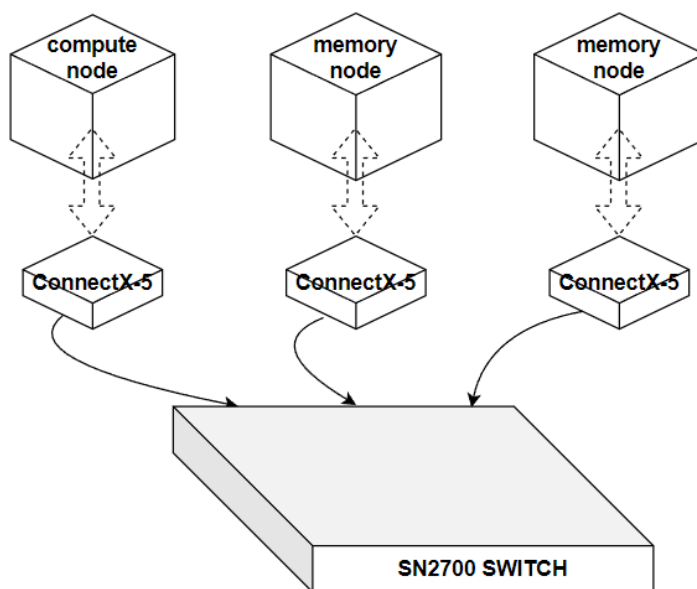


图 3.9: 物理连接示意图

图3.10为系统具体实现的框架结构，图中左右两部分为远程内存节点的架构，其余部分为计算节点的架构。我们希望通过为 frontswap 接口添加后端的实现，来将计算节点的页面换出到 SWAP 交换区的过程进行拦截，转而将其交换到远程内存节点当中，以此来提高系统的内存性能。其中计算节点的每一个 CPU 核心都构建了两个队列对 (qp)，分别对应换出到不同的远程内存节点。在我们的设计中，本地 CPU 仅需进行控制操作，而无需进行数据的传输；远程节点 CPU 不需要参与此交互过程。在建立系统后，我们尝试使用交换机限制远程节点的传输带宽来构造异质的远程节点，并在此基础上进行实验探索以及性能优化。

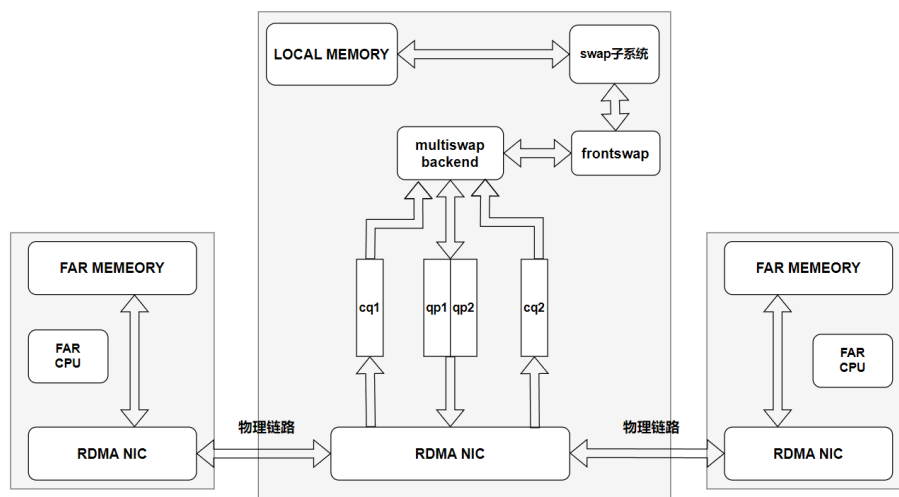


图 3.10: 系统架构图

### 3.3 系统开发环境

系统开发的软件环境如下表2所示：

	计算节点	内存节点	交换机
Linux Kernel	6.1.87	5.15	6.1.0
Nvidia MLNX OFED driver	23.10-3.2.2.0-LTS	5.8-4.1.5.0-LTS	—
OS	Ubuntu 22.04	Ubuntu 20.04	Cumulus Linux 5.9

表 2: 开发环境

### 3.4 技术线路

我们项目的开发计划如图3.11所示，总体上分为三个阶段进行：

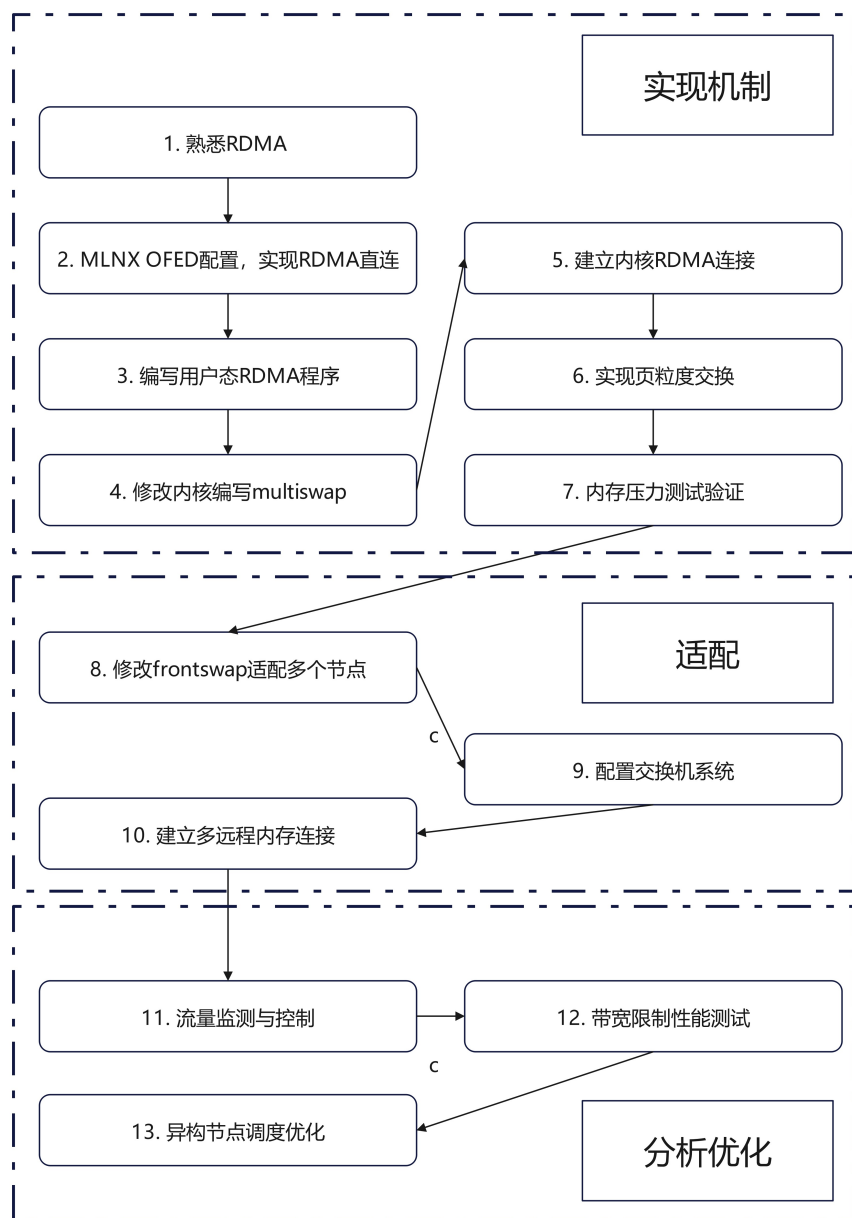


图 3.11: 项目开发计划

### 阶段一

在此阶段中，我们基于 RDMA 在一个计算节点与一个内存节点间实现远程内存页面传输机制：

- 认识 RDMA，熟悉 RDMA 通信原理以及基本流程。
- 安装 MLNX OFED 网卡驱动并配置网卡，尝试在两台机器间能够通过 RDMA 网卡收发信息。
- 编写用户态的 RDMA 程序，使用 uverbs 以及 cmaAPI 使两个节点间建立 RDMA 连接，计算节点为 client 端主动发起连接，远程内存节点作为 server 端接收其连接请求。
- 在计算节点修改内核并编写 multiswap module: 在 Linux 的 frontswap 接口中短路拦截 swap\_readpage 和 swap\_writepage，采用 frontswap\_load 和 frontswap\_store 操作处理换入换出的页面，并用本机的 DRAM 来模拟所实现的 frontswap 的正确性。

- 根据编写的用户态建链程序,在计算节点的内核模块 multiswap 中与远程内存节点之间建立 RDMA queue pairs, 实现两个节点之间连接的建立。
- 在计算节点实现以页为颗粒度的与远程内存之间的交换: 在本机测试通过的 DRAM 接口的基础上编写 RDMA 接口, 能够通过 RDMA 独有的 read 和 write 操作实现远端 CPU 无感知的页面换入换出过程。
- 指定程序进行内存压力测试以强制调用 swap 操作, 验证所实现接口的正确性, 实现计算节点与远程内存节点一对一的基于 RDMA 的远程内存。

## 阶段二

在此阶段中, 我们尝试配置交换机并连接两个远程内存节点:

- 在一对一节点的基础上对 frontswap 接口进行修改, 使计算节点能够适配多个远程内存, 并使得多个 far memory 之间负载均衡。
- 学习并配置交换机系统, 使其能够成功转发来自各个节点的流量。
- 使用交换机将计算节点与多个远程内存节点建立连接, 并将页面置换到多个远程内存中。

## 阶段三

在此阶段中, 我们尝试使用交换机进行流量监控与带宽限制, 尝试构建异构的远程内存节点, 分析其中可能的性能瓶颈后进行优化。

- 学习使用交换机进行流量监测以及流量控制, 构造出异构的远程内存节点。
- 对不同的计算节点限制不同带宽, 探索在不同带宽下的异构 far memory 的访存性能影响。
- 尝试对异构节点进行合理调度, 提升系统的整体性能。

接下来, 我们将按照上述的三个阶段具体描述项目构建的过程。

# 4 阶段一：实现远程内存页面传输机制

## 4.1 驱动安装与环境配置

最初, 我们尝试配置两台主机, 分别充当一个计算节点 (Linux 6.1 kernel) 与一个内存节点 (Linux 5.4 kernel)。Mellanox OFED (OpenFabrics Enterprise Distribution) 是由 Mellanox 提供的驱动程序包, 提供对 Mellanox InfiniBand 和以太网设备的支持, 包括 HCA (主机通道适配器) 和 NIC (网络接口卡)。我们需要在两台主机上分别安装该驱动, 在[Mellanox 官方网站](#)选择对应操作系统的驱动版本安装时, 容易出现操作系统版本对应正确但是不支持其对应的内核版本的问题, 这里找到的解决方案为在[Linux 内核官网](#)使用其对应的长期支持版本内核进行开发。

**经过无数次的试错与调试**, 建议 Ubuntu20.04 with Linux 5.15 generic 使用的 OFED 驱动版本为 5.8-4.1.5.0-LTS, Ubuntu22.04 with Linux 6.1 及以上使用的驱动版本为 23.10-3.2.2.0-LTS。

## 手动编译 Linux 内核

我们在计算节点使用的 Linux 内核版本为 6.1.87。此版本不是 Ubuntu22.04 的原生内核，下载后需要手动编译与安装此内核，下载后进入该内核目录下，运行 **make menuconfig** 命令生成配置文件，选择默认配置即可，保存后自动生成.config 文件即可进行编译，可根据主机 CPU 核心数选择对应的编译线程。由于我们使用的为 6 核 CPU，输入命令 **make -j6** 进行编译。在初次编译时通常会出现以下报错：

```
make[1]: *** No rule to make target 'debian/canonical-certs.pem', needed by
'certs/x509_certificate_list'. Stop.
make: *** [Makefile:1844: certs] Error 2
```

解决方案为编辑.config 文件，做如下修改：

```
CONFIG_SYSTEM_TRUSTED_KEYS=""
CONFIG_SYSTEM_REVOCATION_KEYS=""
```

即可成功编译内核。

编译完成后紧接着可以安装模块与内核：

```
sudo make modules_install
sudo make install
```

由于此时默认的启动项仍然是默认的内核并且 grub 处于自动选择内核状态，需要修改/etc/default/grub 文件，使得开机时能够手动选择内核。修改方式如下：将 GRUB\_TIMEOUT 设置为大于 0 的时间，并且可以通过修改 GRUB\_DEFAULT 来将新编译的内核作为默认选项，其中输入的数字为 GRUB 启动项菜单的每一级菜单索引（注意此索引从 0 开始），修改示例如图4.12：

```
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT="1> 4"
GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=[lsb_release -i -s 2> /dev/null || echo Debian]
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
GRUB_CMDLINE_LINUX=""
```

图 4.12: 修改 grub

接着更新 grub:

```
sudo update-grub
```

更新完毕重启后，选择我们刚才安装的内核进入，输入 **uname -r** 即可查看是否更新内核。

## 安装 MLNX OFED 驱动并配置网卡接口

下载驱动压缩包并解压后进入目录，运行自动安装脚本并加入选项 **-force-dkms**：

### MLNX 驱动安装

```
sudo ./mlnxofedinstall --force-dkms
```



```
/etc/init.d/openibd restart
```

安装完成后，在终端输入 `ibstat` 即可查看 infiniband 设备状态，我们在计算节点使用的为双口网卡，并连接上了其中一个网口，于是可以查看到两个设备，其中一个处于连接状态，如下图所示，其 HCAid 为 `mlx5_1`。

```
cat22@cat:~$ ibstat
CA 'mlx5_0'
  CA type: MT4119
  Number of ports: 1
  Firmware version: 16.35.3502
  Hardware version: 0
  Node GUID: 0x043f720300dc27fc
  System image GUID: 0x043f720300dc27fc
  Port 1:
    State: Down
    Physical state: Disabled
    Rate: 40
    Base lid: 0
    LMC: 0
    SM lid: 0
    Capability mask: 0x00010000
    Port GUID: 0x063f72fffedc27fc
    Link layer: Ethernet
CA 'mlx5_1'
  CA type: MT4119
  Number of ports: 1
  Firmware version: 16.35.3502
  Hardware version: 0
  Node GUID: 0x043f720300dc27fd
  System image GUID: 0x043f720300dc27fc
  Port 1:
    State: Active
    Physical state: LinkUp
    Rate: 100
    Base lid: 0
    LMC: 0
    SM lid: 0
    Capability mask: 0x00010000
    Port GUID: 0x063f72fffedc27fd
    Link layer: Ethernet
```

紧接着，进行 hca 测试，输入命令 `hca_self_test.ofed`，检测硬件设备，若输出图4.13结果，则说明测试通过（其中网卡端口有一个未进行连接，所以显示为 DOWN）。

```
cat22@cat:~$ sudo hca_self_test.ofed
[sudo] cat22 的密码:
---- Performing Adapter Device Self Test ----
Number of CAs Detected ..... 2
PCI Device Check ..... PASS
Kernel Arch ..... x86_64
Host Driver Version ..... MLNX_OFED_LINUX-24.04-0.6.6.0 (OFED-24.
04-0.6.6): 6.1.87
Host Driver RPM Check ..... PASS
Firmware on CA #0 NIC ..... v16.35.3502
Firmware on CA #1 NIC ..... v16.35.3502
Host Driver Initialization ..... PASS
Number of CA Ports Active ..... 1
Port State of Port #1 on CA #0 (NIC).... DOWN (Ethernet)
Port State of Port #1 on CA #1 (NIC).... UP 4X EDR (Ethernet)
Error Counter Check on CA #0 (NIC).... PASS
Error Counter Check on CA #1 (NIC).... PASS
Kernel Syslog Check ..... PASS
Node GUID on CA #0 (NIC) ..... 04:3f:72:03:00:dc:27:fc
Node GUID on CA #1 (NIC) ..... 04:3f:72:03:00:dc:27:fd
----- DONE -----
```

图 4.13: hca test

接下来，为了使两台机器间能够相互访问，需要为 ib 网卡配置一个内网 ip，这样能让其像其它网卡一样工作。输入命令 `ibdev2netdev`，这是安装的 MLNX OFED 驱动所提供的脚本，可以查看适



配器端口与 ib 网卡的对应关系，找到计算节点对应网络接口为 enp1s0f1np1，内存节点对应网络接口为 enp1s0f0np0，如下图4.14所示：

```
cat22@cat:~$ ibdev2netdev
mlx5_0 port 1 ==> enp1s0f0np0 (Down)
mlx5_1 port 1 ==> enp1s0f1np1 (Up)
```

图 4.14: 查找 ib 设备对应网口

确定了网络接口后，需要使用 **ifconfig** 命令为其分配一个局域网的 ip 地址，我们选择分配一个私有 IP 地址范围 (10.0.0.0 - 10.255.255.255)。为计算节点分配的 ip 为 10.10.10.1, 为内存节点分配的 ip 为 10.10.10.2，使用的具体命令如下：

为 ib 网卡配置 ip

```
#client node
sudo ifconfig enp1s0f1np1 10.10.10.1 netmask 255.255.255.0
#server node
sudo ifconfig enp1s0f0np0 10.10.10.2 netmask 255.255.255.0
```

我们在执行该操作时遇到了一些问题，输入命令配置 ip 并成功执行，但是却并没有实际配置成功，于是必须在配置完成后需要使用 ifconfig 命令查看是否配置成功，配置成功后的结果如下图4.15所示：

```
cat22@cat:~$ ifconfig
enp1s0f0np0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether 04:3f:72:dc:27:fc txqueuelen 1000 (以太网)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp1s0f1np1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.10.10.1 netmask 255.255.255.0 broadcast 10.10.10.255
    ether 04:3f:72:dc:27:fd txqueuelen 1000 (以太网)
    RX packets 244 bytes 34856 (34.8 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 348 bytes 44343 (44.3 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

图 4.15: IP config

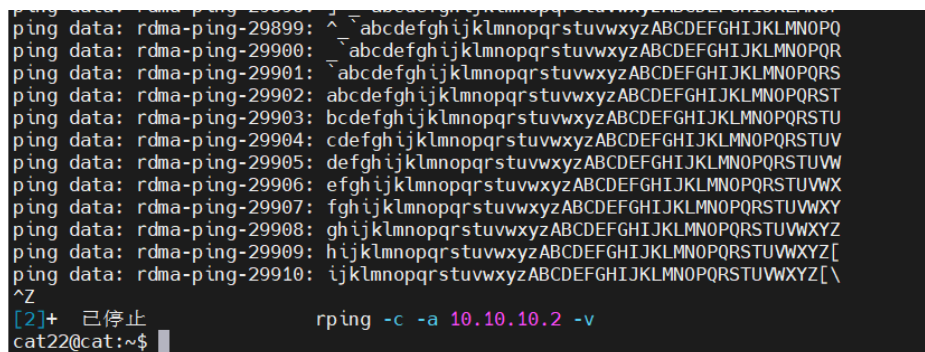
若没有显示相应的 ip，则需要再次输入配置命令进行配置。

接着我们需要测试两端之间是否能够通过 rdma 网卡相互通信，具体做法是使用 MLNX OFED 提供的 rping 命令建立一个 RC(Reliable Connection) 连接，它会使用 lib rdma\_cm 在两个节点之间建立 RDMA 连接并传递消息，在终端中会输出连接建立等消息，任选一台机器作为 server 端与 client 端，在 server 端和 client 端依次分别输入以下命令：

RDMA 连接测试

```
# server node, 其中ip地址都填写为server的ip
rping -s -a 10.10.10.2
#client node
rping -c -a 10.10.10.2 -v
```

在 client 端输入后，若在终端不断输出一系列的 ping data，效果如图4.16，则说明在两台机器中已经成功建立了 RDMA 连接，能够支持我们下一步的工作。



```

ping data: rdma-ping-29899: ^\ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQ
ping data: rdma-ping-29900: ^\ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQR
ping data: rdma-ping-29901: ^\ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRS
ping data: rdma-ping-29902: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRST
ping data: rdma-ping-29903: bcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRSTU
ping data: rdma-ping-29904: cdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRSTU
ping data: rdma-ping-29905: defghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRSTUW
ping data: rdma-ping-29906: efghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRSTUWX
ping data: rdma-ping-29907: fghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRSTUWXY
ping data: rdma-ping-29908: ghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRSTUWXYZ
ping data: rdma-ping-29909: hijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRSTUWXYZ[
ping data: rdma-ping-29910: ijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZKLMNOPQRSTUWXYZ[\
^Z
[2]+ 已停止                  rping -c -a 10.10.10.2 -v
cat22@cat:~$

```

图 4.16: rping 输出

我们在此过程中有事会遇到无法连通两台设备的情况，发现是因为先前在第一次为网络接口配置的 ip 地址因不明原因消失，只需要再次配置 ip 地址即可。至此，基本的用户态环境配置完成，能够进行用户态 RDMA 通信程序的开发。

## 4.2 实现在用户态的 RDMA 通信

因用户态程序调试的方便性，我们尝试在用户态先为两个节点间建立 RDMA 连接，其中一端作为 client 请求连接，另一端作为 server 接受请求，而后将 client 端程序移植进计算节点的内核当中，建立内核态的 RDMA 连接。

### 4.2.1 RDMA 编程

#### Verbs

OpenFabric Alliance (OFA) 推动 RDMA 技术的贡献之一就在于 Verbs API，广义的 Verbs API 主要由两大部分组成：

IB\_VERBS 和 RDMA\_CM 两套 API。

两者之间的关系是 RDMA\_CM 提供了一种更高层次的抽象，用于简化 RDMA 连接的建立和管理过程，但它底层依然使用 IB Verbs API 来执行实际的操作。因此在实际的用户态编程中，更倾向于使用两套 API 结合的方式进行编程，即 RDMA\_CM API 管理连接（建立连接和销毁）+IB\_VERBS API 管理收发。IB\_VERBS 接口以 ibv\_xx（用户态）或者 ib\_xx（内核态）作为前缀，是最基础的编程接口，使用 IB\_VERBS 就足够编写 RDMA 应用了。RDMA\_CM API 是基于 IETF 发布的 RDMA 连接管理协议（RFC 4364）实现的。它定义了一组用于建立、管理和终止 RDMA 连接的函数和消息格式。通过使用 RDMA\_CM API，应用程序可以透明地处理连接管理任务，从而专注于实现应用逻辑。RDMA\_CM 主要又可以分为两个功能

- CMA (Connection Management Abstraction) 是在 Verbs API 基础上实现的，用于 CM 建链并交换信息的一组接口。CM 建链借鉴了 Socket 的流程和 API，底层是基于 QP 实现的，从用户的角度来看，是在通过 QP 交换之后数据交换所需要的 QPN，Key 等信息。
- CM VERBS，是用于连接管理和数据交换的接口，相当于在 Verbs API 上又封装了一套数据交换接口，也可以用于数据交换。

在实际的编程中，通常不会只使用某一套 API 而是可以灵活地对 API 中的 function 取舍，因为直接使用 IB\_VERBS API 可以获得更细粒度的控制而使用 RDMA\_CM API 可以简化连接管理的复杂性，从而写出可读性和功能性俱佳的代码。在进行的编程中，我们主要参考 [4] 以及 [3] 中的内容以及示例。

## RDMA 有关名词和概念

PD	Protection Domain，在 RDMA 中，PD 像是一个容纳了各种资源（QP、MR 等）的“容器”，将这些资源纳入自己的保护范围内，避免他们被未经授权的访问。一个节点中可以定义多个保护域，各个 PD 所容纳的资源彼此隔离，无法一起使用。
WQE	Work Queue Element，WQE 可以认为是一种“任务说明”，这个工作请求是软件下发给硬件的，这份说明中包含了软件所希望硬件去做的任务以及有关这个任务的详细信息。
WQ	Work Queue，就是用来存放“任务书”的“文件夹”，WQ 里面可以容纳很多 WQE。
WC	Work Completion，其实是 CQE 在用户层的“映射”。因为 APP 是通过调用协议栈接口来完成 RDMA 通信的，CQE 本身并不对用户可见，是驱动中的概念。用户真正通过 API 收到的是 WC。
CQ	Completion Queue，硬件通过 CQ 中的 CQE/WC 来告诉软件某个 WQE/WR 的完成情况。
QP	Queue Pair，RDMA 通信的基础，其中包含了一个 RQ 和一个 SQ。基于连接的 QP 只能一对一交流，而基于数据报的非连接通信能实现 QP 一对多通信。
RQ	Receive Queue，存储了接收的 WQE。
SQ	Send Queue，存储了发送的 WQE。

## RDMA 连接管理

RDMA 的每一个连接是通过一个 event channel 来管理的，rdma\_create\_event\_channel() 创建一个 event channel。rdma\_create\_id() 根据 event channel 创建一个 rdma\_cm\_id。每个 rdma\_cm\_id 对应一个连接（与 POSIX socket 中的文件描述符类似）。接下来具体到 Server 端和 Client 端进行介绍。

对于 Server 端，首先需要使用 rdma\_bind\_addr() 为一个 rdma\_create\_id() 绑定一个 sock-addr。然后使用 rdma\_get\_cm\_event() 监听 event channel 中的关于连接管理的事件。每个事件都是一个 rdma\_cm\_event 结构，每个事件的 rdma\_cm\_id 对应了事件关联的连接；每个事件需要用 rdma\_ack\_cm\_event() 进行释放并从 event queue 中移除，所以一般使用一个拷贝的对象来处理事件。每个事件的 event 属性表示了事件的类型。

对于 Client 端，我们需要调用 rdma\_resolve\_addr() 将目标地址 (addrinfo 结构中的 ai\_addr 作为参数) 和可选的源地址解析成 RDMA 地址，当解析完成时，会在 event queue 中生成一个 RDMA\_CM\_EVENT\_ADDR\_RESOLVED 事件。然后同样使用 rdma\_get\_cm\_event() 监听 event queue 中的关于连接管理的事件。

**重要结构体** ibv\_sge 是定义 SGE 的结构体，每一个 SGE 就是一个 Data Segment。RDMA 支持 Scatter/Gather 操作，具体来讲就是 RDMA 可以支持一个连续的 Buffer 空间，进行 Scatter 分散到多个目的主机的不连续的 Buffer 空间。Gather 指的就是多个不连续的 Buffer 空间可以 Gather 到目的主机的一段连续的 Buffer 空间。

```
struct ibv_sge {
    uint64_t    addr;
    uint32_t    length;
    uint32_t    lkey;
```

```
};
```

表格中展示了 `ibv_sge` 结构体关键的成员变量及其意义：

<code>addr</code>	数据段所在的虚拟内存的起始地址
<code>length</code>	数据段长度
<code>lkey</code>	该数据段对应的密钥

`ibv_send_wr` 是定义发送到 QP 的发送队列的工作请求的结构体。

Send 工作请求结构体

```
struct ibv_send_wr {
    uint64_t      wr_id;
    struct ibv_send_wr *next;
    struct ibv_sge *sg_list;
    int          num_sge;
    enum ibv_wr_opcode opcode;
    unsigned int  send_flags;
};
```

表格中展示了 `ibv_send_wr` 结构体的成员变量及其意义：

<code>wr_id</code>	唯一标识一个发送工作请求的 id。
<code>next</code>	指向链表中的下一个发送工作请求，如果是 NULL 则表示是最后一个。
<code>sg_list</code>	由一个或多个 SGE 构成的数组。
<code>num_sge</code>	<code>sg_list</code> 的长度。
<code>opcode</code>	指定数据传输的方式、数据流的方向以及使用的 <code>wr</code> 中的属性。
<code>send_flags</code>	控制发送行为的操作和特性。

`ibv_recv_wr` 成员变量的含义与 `ibv_send_wr` 中的基本相同，区别在于 `ibv_recv_wr` 是定义发送到 QP 的接收队列的工作请求的结构体，所以相应的工作请求也是接收工作请求。另外接收操作不需要执行复杂的原子操作等，所以成员变量相对也更少。

recv 工作请求结构体

```
struct ibv_recv_wr {
    uint64_t      wr_id;
    struct ibv_recv_wr *next;
    struct ibv_sge *sg_list;
    int          num_sge;
};
```

#### 4.2.2 用户态 RDMA 的连接建立与断开

建立连接，断开连接的过程，都是由 `rdma_get_cm_event` 获取事件，再交由 `on_event` 分情况由不同的事件处理函数进行处理。

图4.17和图4.18展示了客户端以及服务器端处理事件的流程，各个节点代表了调用过程的关键函数，序号代表了在该端的函数执行顺序，由于连接和断开的请求都是由客户端发起的，所以在服务器端的流程图里有两个客户端的函数作为节点，背景为红色，方便描述流程。在 `client` 端中，第一个事

件是由 `rdma_resolve_addr` 触发的，随后按照图中顺序依次进行识别和处理，可以看到一个事件处理函数里往往都会调用一个 RDMA\_CM API 的函数，这个函数又会触发新的事件，然后在主函数的循环中被获取，最终形成图中流程。server 端是相对被动的一方，无论是建立连接还是断开连接的事件，都是由 client 方触发的，所以 server 端需要一直处于获取事件的状态，直到 client 端触发了特定事件，server 端才能获取到事件并进行处理，图中所示的两个红色背景的节点也就是 client 端的函数就是可以触发相应事件的功能，在这两个函数的作用下，server 端的连接和断开连接得以形成，流程中也清晰地表明了中间过程。

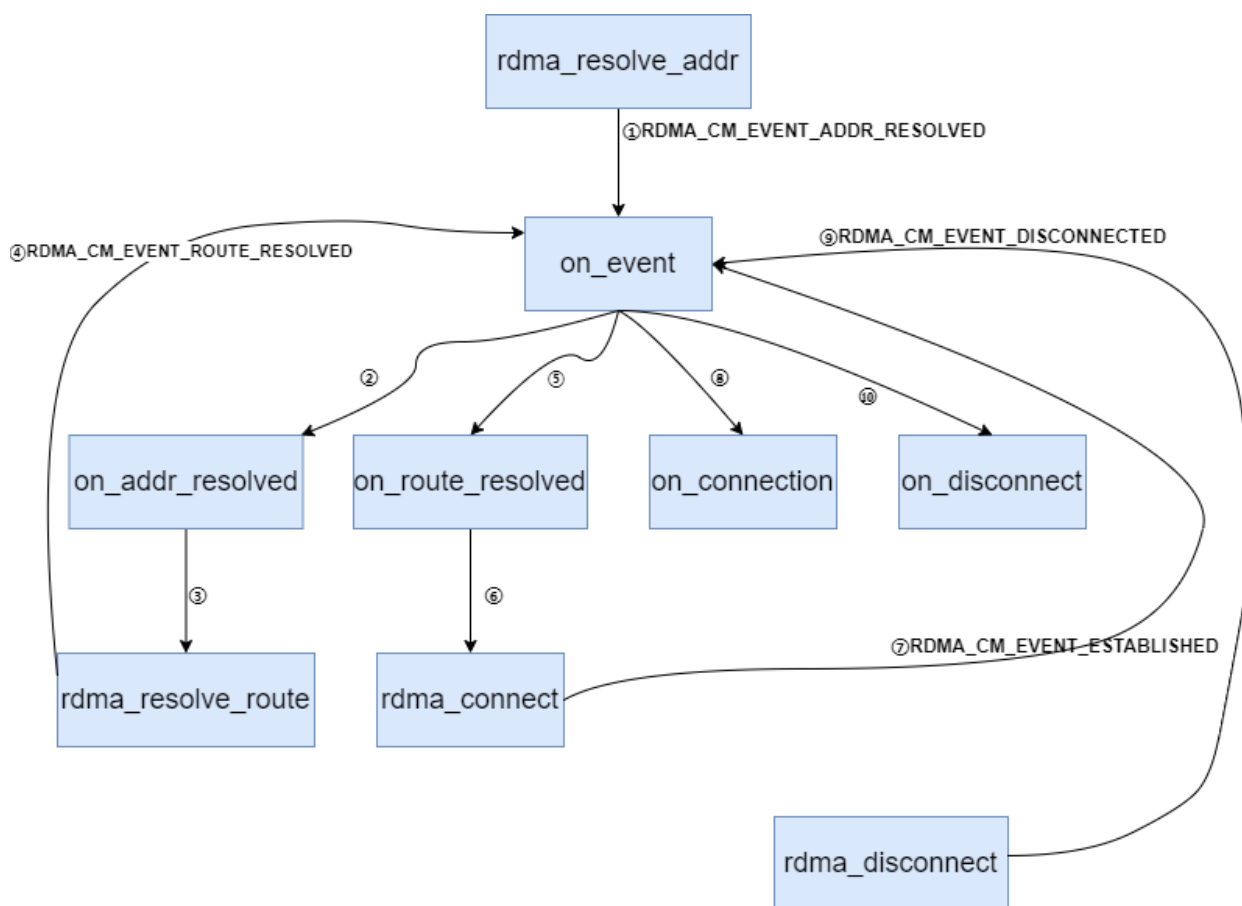


图 4.17: Client 端事件触发与处理

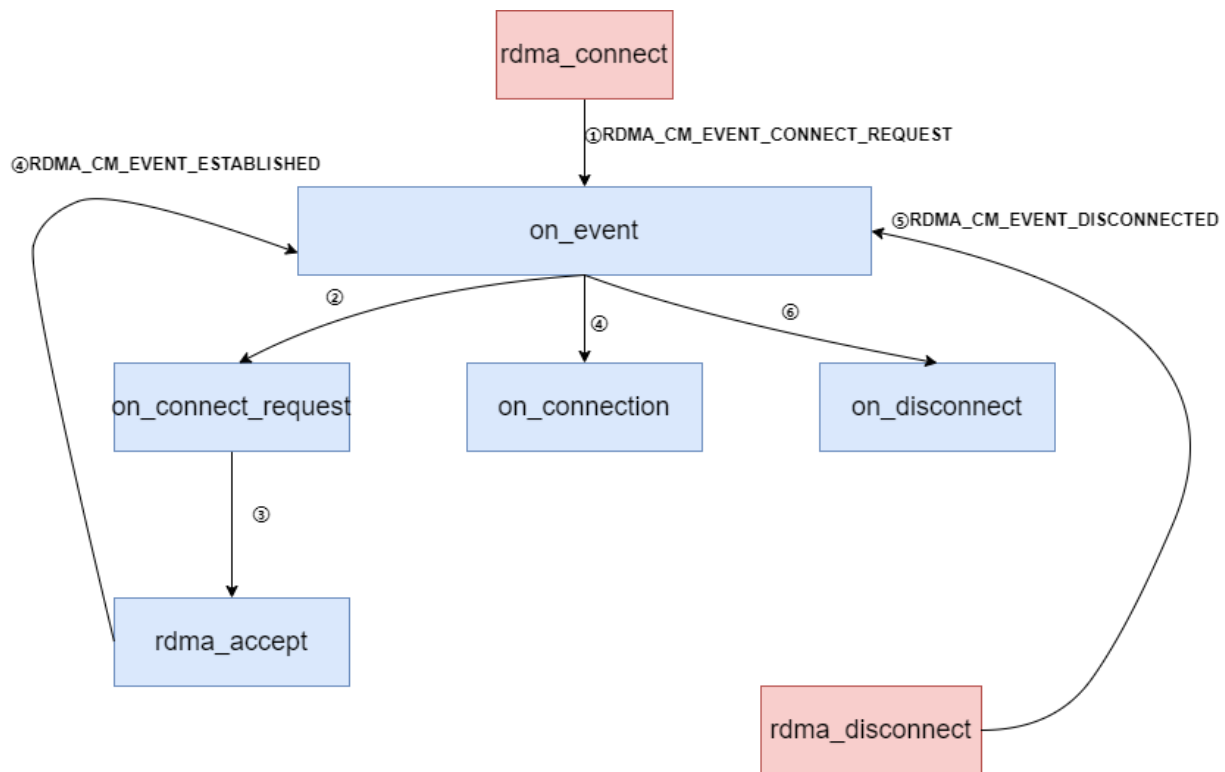


图 4.18: Server 端事件触发与处理

#### 4.2.3 RDMA 连接双端交互过程

图4.19展示了用户态程序中 Server 和 Client 的交互过程，包括建立连接，消息收发和断开连接，图中的序号代表了执行的顺序，下面将对每个步骤进行详细说明：

1. **rdma\_resolve\_addr** 解析地址并触发 **RDMA\_CM\_EVENT\_ADDR\_RESOLVED** 事件，该事件被 **rdma\_get\_cm\_event** 获取。
2. **ibv\_poll\_cq** 是在一个单独的线程里进行的，它会不断轮询完成队列获取完成事件。因此当新的完成事件进入完成队列后，会立即被处理。
3. **ibv\_reg\_mr** 会设置好 **send\_region** 和 **recv\_region** 的内存的大小以及 RDMA 网卡对这两块内存的权限，使这两块内存区域从常规的只能由 CPU 访问的内存变为了能通过 RDMA 网卡访问的内存（前提是设置的权限允许 RDMA 网卡这么做）。
4. 在 RDMA 的消息 Send/Recv 过程中，接收方应该在消息到来之前就提前 post recv 请求，这样当真正收到消息后，不需要占用 CPU 将消息从网卡的缓存区域存入内存区域中。
5. **rdma\_resolve\_addr** 将目标地址和可选源地址从 IP 地址解析为 RDMA 地址，指定的 **rdma\_cm\_id** 标识将与本地设备相关联。
6. **rdma\_resolve\_addr** 触发 **RDMA\_CM\_EVENT\_ROUTE\_RESOLVED** 事件后被 **rdma\_get\_cm\_event** 获取。
7. **rdma\_connect** 发起 RDMA 连接请求。

8. `rdma_listen` 会监听来自客户端的请求，所以当客户端执行 `rdma_connect` 后，服务器端便接收到了连接请求。
9. 服务器端收到连接请求会触发 `RDMA_CM_EVENT_CONNECT_REQUEST` 事件，被 `rdma_get_cm_event` 获取。
10. 与第三步相同。
11. 与第四步相同。
12. `rdma_accept` 接受来自客户端的连接请求，连接建立成功。
13. 连接建立成功后，服务器端 `rdma_get_cm_event` 获取到 `RDMA_CM_EVENT_ESTABLISHED` 事件。
14. 客户端同样获取到 `RDMA_CM_EVENT_ESTABLISHED` 事件。
15. 服务器端发送消息到客户端。
16. 客户端发送消息到服务器端。
17. 以上两步完成之后，客户端便处理了两个完成事件，而第二步中，客户端单开了一个轮询完成队列的线程，这个线程在处理了两个完成事件后会发起断开连接的请求。
18. 服务器端 `rdma_get_cm_event` 获取到 `RDMA_CM_EVENT_DISCONNECTED` 事件后进行资源销毁和清理。
19. 客户端 `rdma_get_cm_event` 获取到 `RDMA_CM_EVENT_DISCONNECTED` 事件后进行资源销毁和清理，连接断开成功。



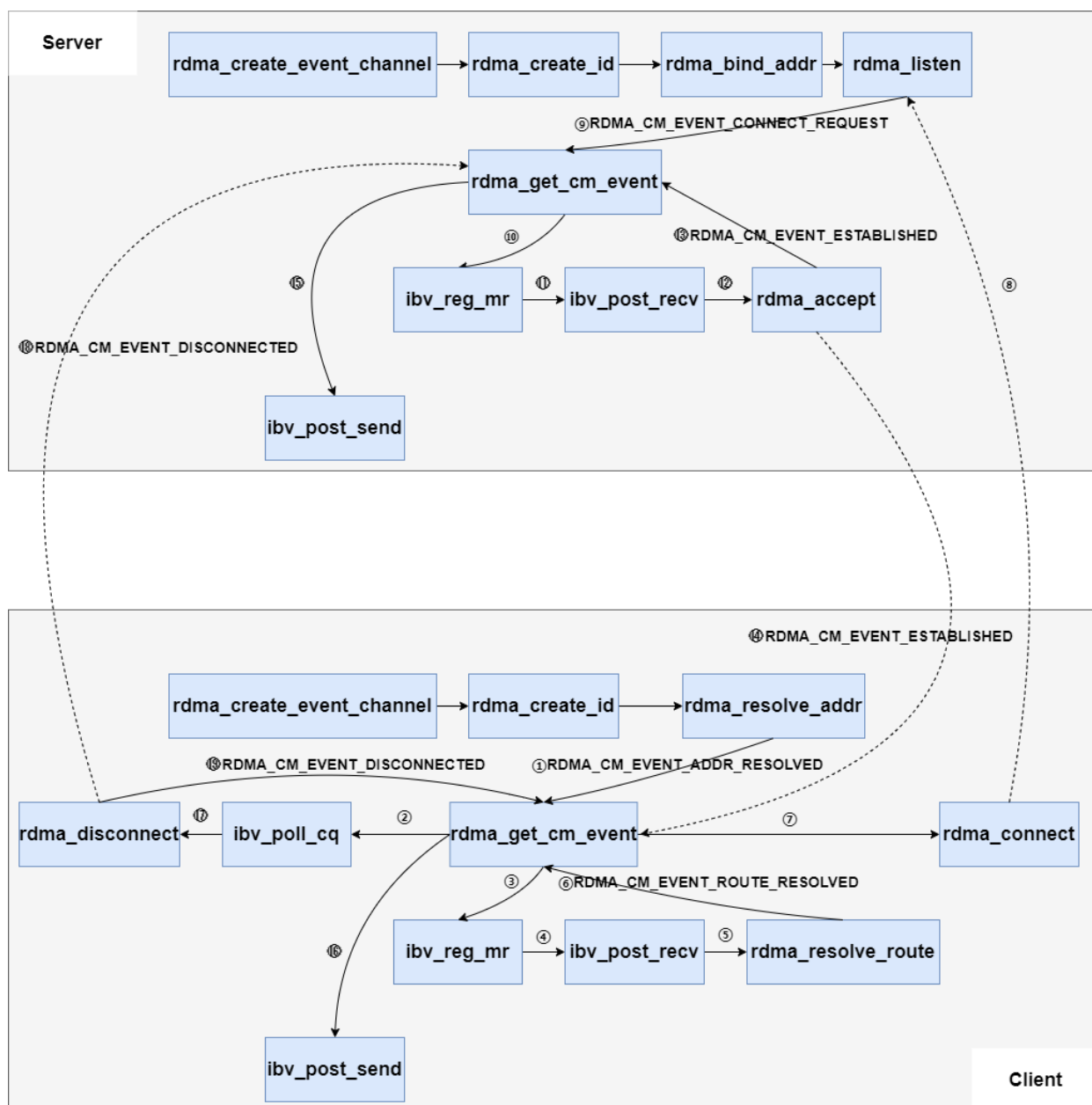


图 4.19: 交互过程

#### 4.2.4 用户态程序运行

在实际的开发过程中，不是先将硬件环境等都配置好再进行用户态程序运行的，如前期设计所提到的，用户态程序编写和运行的目的就是用来检测 RDMA 设备是否配置成功以及能否正常工作，而后将其移植进入内核态中，建立内核态 RDMA 连接。所以编写完代码后的运行用户态程序本身当然并不复杂，想办法在客户端和服务端将 RDMA 用户态程序运行起来并且得到正确的预期结果才是难点，从两台非常普通的主机到他们能够进行用户态的 RDMA 通信是这个项目真正意义上从 0 到 1 的突破。这个过程难点也就是设备以及环境的配置的过程在前文已经介绍，这里介绍一下两个程序的运行方法。

运行 server 端程序之前，更改 server 端代码中的 IP 地址以及端口号，使其作为监听的对象，编译后运行：



## server 端运行命令

```
./server
```

运行结果如图4.20所示，由于当前只运行了 server 端的用户程序，所以只能看到打印了监听端口号。

```
cat22@cat:~/gitRepo/farmemserver$ ./server
listening on port 8888.
```

图 4.20: server 端程序运行结果

运行 client 端程序不需要修改源代码，只需要运行编译好后的 client 可执行程序，运行时输入 server 端的 IP 地址和端口号，就能成功运行。

## client 端运行命令

```
./client <server-address> <server-port>
```

运行结果如图4.21所示，从输出中可以看出得到了预期的结果，依次完成了解析地址，路由，建立连接和断开连接，这说明 RDMA 连接能够成功建立并且能够顺利断开连接。还注意到 client 端成功接收到了来自 server 端的消息，说明消息的收发也能够正确完成。

```
farmemserver ./client 10.10.10.5 8888
address resolved.
route resolved.
connected
received message: message from passive/server side with pid 249712
disconnected.
```

图 4.21: client 端程序运行结果

同时还要关注 client 程序运行后，server 端的情况，运行结果如图4.22所示，也得到了预期的输出，收到了建立连接的请求，成功建立连接，成功发送消息，成功断开连接。

```
cat22@cat:~/gitRepo/farmemserver$ ./server
listening on port 8888.
received connection request.
connected. posting send...
send completed successfully.
peer disconnected.
```

图 4.22: client 运行后 server 端的输出结果

### 4.3 Frontswap 接口介绍及使用

在 Linux 的 swap 子系统中，当系统的内存不足需要将匿名页写回到磁盘时，首先会通过 `address_space` 调用 `swap_writepage` 函数将页面异步回写到交换空间中，等待该页面需要被访问时，会触发 `pagefault`，通过 `swap_readpage` 重新加载进内存中。而 Frontswap 可以在这两个过程中进行拦

截，使内核将要交换的页存储在比交换空间（磁盘）更快的存储介质上，以减少 swap 操作的延迟，提高系统的性能。在本项目中，更快的存储介质即为使用 RDMA 进行连接的远程内存。于是，我们需要做的就是对 Frontswap 接口进行一种实现。

在 `/include/linux/frontswap.h` 中定义的 `frontswap_ops` 描述了 frontswap 接口：

#### frontswap\_ops

```
struct frontswap_ops {
    void (*init)(unsigned); /* this swap type was just swapon'ed */
    int (*store)(unsigned, pgoff_t, struct page *); /* store a page */
    int (*load)(unsigned, pgoff_t, struct page *); /* load a page */
    void (*invalidate_page)(unsigned, pgoff_t); /* page no longer needed */
    void (*invalidate_area)(unsigned); /* swap type just swapoff'ed */
};
```

这是 Linux 内核中典型的**机制 (mechanism)-策略 (policy)** 的关系，实现一种 `frontswap_ops` 并让内核采用此策略来完成本部分的工作，frontswap 接口可以通过调用函数 `frontswap_register_ops` 更换策略。frontswap 的核心操作在于 `store` 和 `load`，其中都包含了三个参数，分别代表了 swap area(type)，换出的页相对于此交换设备的偏移位置 (pgoffset) 以及换出的页 (page)。在 `/mm/page_io.c` 中进行 `swap_writepage` 和 `swap_readpage` 时，以 `write` 操作为例，会先对 frontswap 接口进行尝试，当 frontswap 接口被启用并成功调用将页面换入换出时，将会直接返回，否则进行 `writepage` 到 swap 交换区的操作。**我们能够清晰地知道，此接口的作用是在页换出换入到交换区时，拦截这个过程并抢先换页到 frontswap 接口中所准备的更快的存储介质中去。**

同样以 `write` 操作为例，若采用了 frontswap 路径，`frontswap_store` 函数在 frontswap 启用时会进一步调用 `__frontswap_store` 函数，在此时，如果我们在先前注册 (register) 了一种 `frontswap_ops` (即对 frontswap 进行了实现并且加载到内核中)，就会调用该 ops 所实现的 `store` 函数，将页换出到所实现的一种策略当中处理，`read` 操作同理。总体来说，从 frontswap 接口进行页的换出和换入过程分别如下图 4.23 和 4.24 所示：

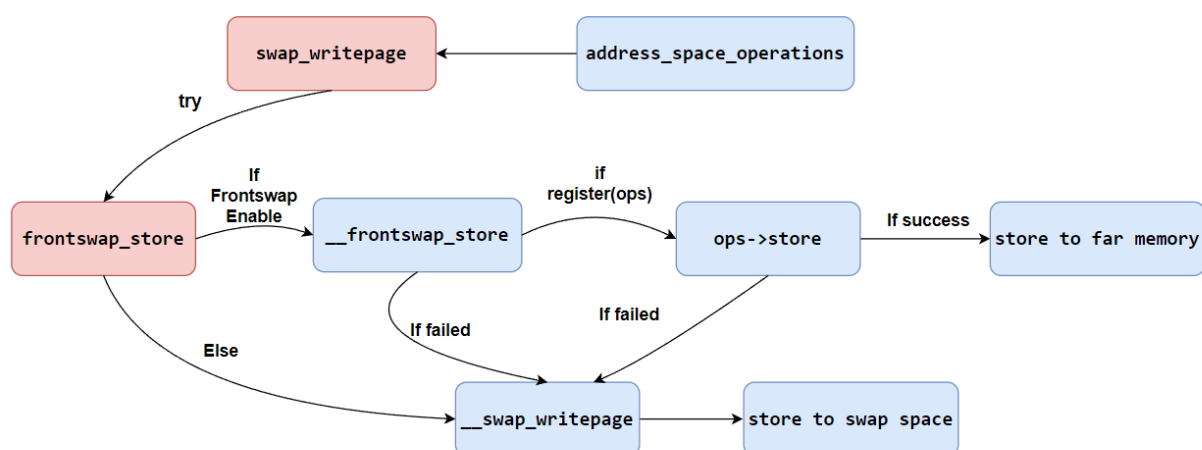


图 4.23: frontswap store

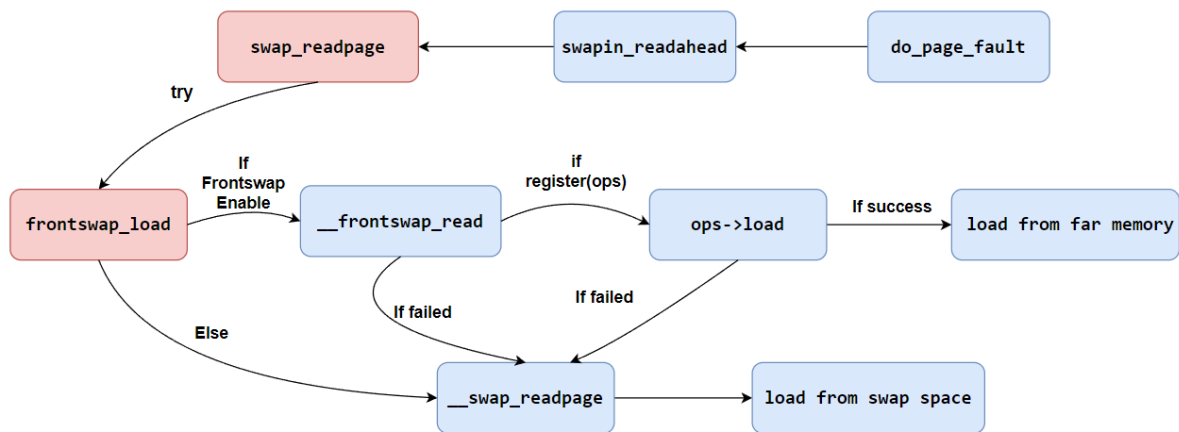


图 4.24: frontswap load

#### 4.4 Frontswap 的接口实现:multiswap

我们参考了 *FASTSWAP*[1] 这篇文章在 Linux 4.11 内核版本上的工作以及 *Memliner*[8] 在 Linux 5.4 内核版本上的工作。在 Linux 6.1 的内核版本上，实现了 frontswap 基本的 read 和 write 功能，并将其以 module 的方式安装在内核中。此部分内容在 multiswap.c 中完成。根据之前描述的 frontswap 接口功能，我们将其进行初步实现，结构如下：

##### frontswap 接口实现

```
static struct frontswap_ops sswap_frontswap_ops = {
    .init = sswap_init,
    .store = sswap_store,
    .load = sswap_load,
    .invalidate_page = sswap_invalidate_page,
    .invalidate_area = sswap_invalidate_area,
};
```

接下来需要实现策略中的各个函数，其中查阅 Linux 内核手册可知，`invalidate_page` 将会从远内存中移除某特定页，`invalidate_area` 将会移除所有和此交换设备相关的所有页，是一种类似于 `swapon` 的操作。我们暂时没有使用到这两个接口，于是直接简单在函数中返回即可，同时，`init` 操作也暂时没有需要做的工作，它将会在 `frontswap_init` 中被调用，这里只让它打印信息告知初始化成功即可。剩余两个关键操作 `store` 和 `load` 函数，我们先让其简单返回为-1(表示调用 frontswap 失败，继续将页交换到 swap 交换区中)。至此完成了一个简单的 frontswap 接口的测试用例。

##### 4.4.1 内核模块的编写与编译

Linux 的内核模块编写有着完整的一套规范，首先需要包含头文件 `<linux/module.h>`，并输入相关的模块信息，包括 `author`, `license`, `description` 等，但是更为重要的是编写 `module_init` 与 `module_exit` 函数，这两个函数分别在模块加载和卸载时调用，用于初始化模块内容与释放所占用相关资源。所以我们需要在模块初始化时注册我们的 `frontswap_ops` 实现，在退出时，由于系统没有准备 `unregister` 函数，所以不方便直接从内核中移除该 ops，此过程后续可能触发内核 bug，这一点我们将在后续说明。总之，这部分编写的模块代码如下：

## 模块的初始化与退出

```
#include <linux/module.h>
static int __init init_sswap(void)
{
    pr_info("begin init sswap\n");
    if(frontswap_register_ops(&sswap_frontswap_ops))
    {
        pr_err("sswap register ops failed!\n");
        return -1;
    }
    pr_info("sswap module loaded\n");
    return 0;
}

static void __exit exit_sswap(void)
{
    pr_info("unloading sswap\n");
}
module_init(init_sswap);
module_exit(exit_sswap);
```

接着尝试编译该内核模块，为其编写 Makefile 文件。在 /usr/src/ofa\_kernel/default 中包含了 Mellanox OFED 驱动相关的内核模块和驱动程序的源代码和配置文件，我们在构建内核模块时需要包含此路径，分别使用了三个变量：OFA\_DIR、OFA\_INCLUDE、OFA\_SYMVERS 分别包含此路径下的源代码目录，头文件目录，以及内核模块符号版本文件的路径，定义如下：

```
OFA_DIR ?= /usr/src/ofa_kernel/default
OFA_INCLUDE := $(OFA_DIR)/include
OFA_SYMVERS := $(OFA_DIR)/Module.symvers
```

为了在编译模块时可以使用 OFED 驱动相关的符号文件，需要将 OFA\_SYMVERS 导出到环境变量 KBUILD\_EXTRA\_SYMBOLS 中，并将 KDIR 变量设置为当前内核版本的构建目录（通过 `uname -r` 获取）。做了以上这些准备后，就可以开始正式编译模块。我们只需为 `obj-m` 变量定义模块对象（在这里定义为 `multiswap.o`），并指定模块源代码目录为当前目录，系统将会自动查找当前目录对应的同名 C 文件，切换到指定的内核构建目录中进行模块构建，这部分的 Makefile 文件编写如下：

```
export KBUILD_EXTRA_SYMBOLS=$(OFA_SYMVERS)
KDIR ?= /lib/modules/`uname -r`/build
obj-m := multiswap.o
default:
    $(MAKE) -C $(KDIR) M=$$PWD
clean:
    $(MAKE) -C $(KDIR) M=$$PWD clean
```

上述文件中，`-C` 选项用于切换到指定的目录进行编译，`$(KDIR)` 是先前我们指定的内核构建目录，`M` 选项用于指定模块源代码所在的目录，`$$PWD` 是当前工作目录。这告诉内核构建系统，模块的源代码在当前目录中，并且需要在这个目录中查找和编译模块源文件。

至此，可以通过 `make` 命令编译并生成以 `.ko` 结尾的模块文件进行编译。

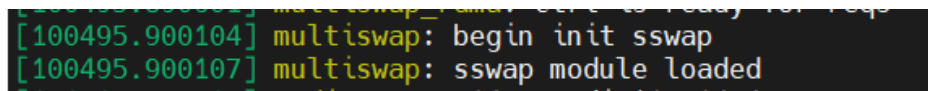
#### 4.4.2 为 Linux 内核添加补丁 1

在 Linux 6.1 内核进行模块编译的过程中，模块编译发生报错，大致信息为 `frontswap` 的接口函数未定义。这是在头文件中包含了 `frontswap.h` 后还产生的报错，并且前期在 Linux 5.4 的内核中进行相关尝试时并未发现此问题，于是我们比较了 5.4 和 6.1 内核的 `frontswap.c` 文件，并查阅了相关资料得知，在 Linux 内核中，一个模块的编译如果需要调用内核中的函数，需要依赖内核编译时生成的 `Module.symvers` 文件，在这个文件中保存了大量通过 `EXPORT_SYMBOL` 导出的全局符号，只有导出了这些符号才能为外部模块所用。于是我们定位错误在于在 **6.1 内核中没有 `EXPORT frontswap` 的相关接口**到全局符号中，而在自定义模块中却使用了 `register_ops` 等函数导致出错（这部分在 Linux 5.4 的内核中是默认操作）。于是在内核文件 `frontswap.c` 中相关的函数后做类似的 `EXPORT_SYMBOL` 操作即可，具体操作可以参考我们 `gitHub` 仓库里的**补丁**。添加补丁过后，由于修改了全局符号表，需要重新编译并且安装内核，具体操作详见 4.1 节。在修改了全局符号表并执行 `make install` 命令时，会与 **MLNX OFED 驱动冲突**而报错，原因为驱动引用的全局符号表与当前不一致，此问题的解决方案为先卸载驱动安装修改后的内核，在重装内核完成后再安装驱动。

修改过后，能够正常进行模块编译。在编译成功后，输入命令进行模块安装和卸载：

```
sudo insmod multiswap.ko
sudo rmmod multiswap.ko
```

安装完成后，`dmesg` 查看内核缓冲区中是否输出了正确的信息，如图 4.25 所示，则安装成功：



```
[100495.900104] multiswap: begin init sswap
[100495.900107] multiswap: sswap module loaded
```

图 4.25: multiswap 安装信息

## 4.5 使用 DRAM 作为 multiswap 的 BACKEND

### 4.5.1 read、write 接口函数的实现

在连接远程内存节点之前，我们先尝试将本地的 DRAM 充当远程内存使用，将本要交换到 swap 交换区的页面交换到一块我们事先准备好的 DRAM 中，用于测试 `frontswap` 接口的可行性，并且该本地 DRAM 是理论上远程内存的性能上限<sup>1</sup>。在此过程中，我们能够测量得到远程内存的理论上限并且为支持此功能对内核做一些可能需要的修改。

实现该功能即只需实现 `frontswap` 的 `read` 和 `write` 接口，由于后续还需要进行 RDMA 远程内存的实现，我们不再 `multiswap` 模块中直接编写该接口，而是再进行一层函数封装：

`sswap` 的 `read` 和 `write` 操作

```
static int sswap_store(unsigned type, pgoff_t pageid,
    struct page *page)
{
    if (sswap_rdma_write(page, pageid << PAGE_SHIFT)) {
        pr_err("could not store page remotely\n");
    }
}
```

<sup>1</sup>相比于远程内存，使用本地内存少了一段网络传输的过程。

```

    return -1;
}
return 0;
}

static int sswap_load(unsigned type, pgoff_t pageid, struct page *page)
{
    if (unlikely(sswap_rdma_read_sync(page, pageid << PAGE_SHIFT))) {
        pr_err("could not read page remotely\n");
        return -1;
    }
    return 0;
}

```

对于该函数的具体实现，我们使用一个新的模块来完成，命名为 `multiswap_dram`。在模块的 `init` 函数中，需要分配一块内存区域作为换出页的存放区域，对该块区域的大小有相关要求，在后续我们将说明，在模块的 `exit` 函数中，只需要释放这一块内存区域即可。

#### multiswap\_dram 的初始化与退出

```

static void __exit sswap_dram_cleanup_module(void)
{
    vfree(drambuf);
    pr_info("DRAM backend is cleaned up\n");
}

static int __init sswap_dram_init_module(void)
{
    pr_info("start: %s\n", __FUNCTION__);
    pr_info("will use new DRAM backend");

    drambuf = vzalloc(REMOTE_BUF_SIZE);
    pr_info("vzalloc'ed %lu bytes for dram backend\n", REMOTE_BUF_SIZE);

    if(SWAPFILE_SIZE > REMOTE_BUF_SIZE) {
        pr_info("warning: swapfile size is larger than remote buffer size\n");
        pr_info("this may cause a part of pages are not use frontswap\n");
    }

    pr_info("DRAM backend is ready for reqs\n");
    return 0;
}

module_init(sswap_dram_init_module);
module_exit(sswap_dram_cleanup_module);

```

此模块的关键部分在于实现我们封装的两个函数（此部分代码在后续一同给出）：

- `sswap_rdma_write(struct page *page, u64 roffset)`:



使用 `kmap` 将 `page` 进行临时映射，并由此获取其对应的地址，得到该地址后使用 `copy_page` 函数将其复制到 `drambuf + roffset` 的偏移位置，最后利用 `kunmap` 解除该临时映射即可。

- `sswap_rdma_read(struct page *page, u64 roffset):`

页面的换入过程和换出过程进行 `copy_page` 的方向相反，在这之前需要更小心地判断该页是否可以从远端读取（即是否在 `swapcache` 中，是否是锁定页面，是否非 `uptodate` 状态），若通过了这些检查，和 `write` 操作做一个方向相反的复制页内容后解除临时映射。完成页面的读取之后还需要进行解锁该页，更新该页的状态。这一部分更新页的状态的操作原本是在内核中 `page_io.c` 中 `swap_readpage` 调用成功 `frontswap_load` 执行的，为了后续开发异步读操作（`frontswap` 接口函数成功返回后并不一定真正会读取到该页，这在 `RDMA` 操作中很常见），我们将其放在模块内此处更新，同时内核需要做一定的修改，参见4.5.4。

#### 4.5.2 对 far memory 的大小要求

对于充当远程内存的区域大小我们也有一定的要求。Linux 内核中采用 `swap_info_struct` 来管理 `SWAP` 分区，其中一个 `swap_info_struct` 对应一个 `swap` 分区，这些 `swap` 分区对应的结构体被放置到一个全局的 `swap_info` 数组上便于管理。`swap` 分区内部以 `page` 为单位划分出多个 `swap slot`，当发生内存回收时，一页会被回收到一个 `slot` 里，如图4.26所示，在图中我们同时也描述了 `frontswap` 接口拦截页面置换到 `swapfile` 中的过程：在 `swap_readpage` 和 `write_page` 的中，将物理页直接换入到准备好的 `frontswap backend` 中。

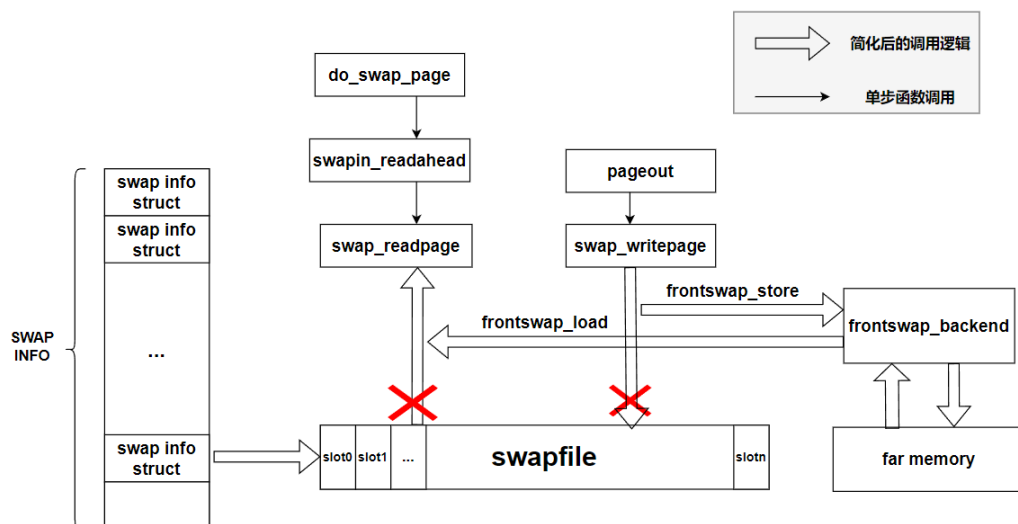


图 4.26: SWAP 机制

`/mm/frontswap.c` 中的 `frontswap_load` 和 `store` 函数的执行流程相似，这里以 `store` 为例，其实现如下：

```

frontswap_store

int __frontswap_store(struct page *page)
{
    int ret = -1;

```

```

swp_entry_t entry = { .val = page_private(page), };
int type = swp_type(entry);
struct swap_info_struct *sis = swap_info[type];
pgoff_t offset = swp_offset(entry);

VM_BUG_ON(!frontswap_ops);
VM_BUG_ON(!PageLocked(page));
VM_BUG_ON(sis == NULL);

if (__frontswap_test(sis, offset)) {
    __frontswap_clear(sis, offset);
    frontswap_ops->invalidate_page(type, offset);
}

ret = frontswap_ops->store(type, offset, page);
if (ret == 0) {
    __frontswap_set(sis, offset);
    inc_frontswap_succ_stores();
} else {
    inc_frontswap_failed_stores();
}

return ret;
}

```

虽然没有被交换到 swap 分区中去，但是对于要操作的页，首先会获取对应的 `swp_entry_t`，这是一个 64 位的结构体，其中 2-7 位存放了 swap type(Linux 将所有的 `swap_info_struct` 保存在一个数组 `swap_info` 当中，type 为数组下标，用于索引具体的 swap 分区)，8-57 位存放其将要被回收到的 slot 在 swap 分区内的页偏移 offset，而在我们的实现中，将 `drambuf + addroffset` 直接作为该页在远程内存中的保存位置，于是要求**远程内存的大小大于或者等于该页对应的 swapfile 的大小**，否则将会可能某些页溢出远程内存。

接下来我们将保证这一点，首先采用 `swapon` 命令查看本地 swap 交换区的大小，发现该大小为 32G，本地内存总大小为 32GB，显然不能全部分配给 dram backend 用来换出页面，我们需要重新添加一个 swap 分区，并且为了方便，我们希望有且仅有一个 swap 分区，其大小为 8GB。

#### 修改 swap 分区

```

# 删除旧的 swap 分区:
sudo swapoff /dev/nvme0n1p3
sudo swapoff /dev/nvme0n1p6
sudo rm /dev/nvme0n1p3
sudo rm /dev/nvme0n1p6

# todo: 注释在/etc/fstab文件中这两个swap分区的自动挂载

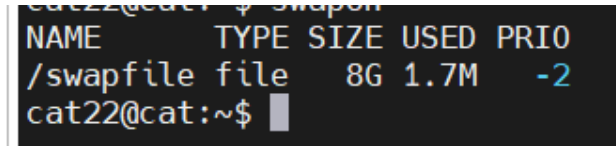
# 添加新的 swap 分区并初始化，大小为 8GB
sudo dd if=/dev/zero of=/swapfile bs=1024 count=8388608
ll -h /swapfile

```



```
#在/etc/fstab中设置开机自动挂载，具体格式参考文件中的描述
/swapfile none swap sw 0 0
```

执行完毕后，再次 swapon，可以看到 swap 分区已经修改为 8GB，如图4.27所示：



NAME	TYPE	SIZE	USED	PRIO
/swapfile	file	8G	1.7M	-2

图 4.27: swap 分区大小

除此之外，为了不引起不必要的麻烦，multiswap\_dram.c 中添加了宏 **SWAPFILE\_SIZE**，在 write 操作中进行了判别：即使需要换出的 offset 超过了我们分配出的 REMOTE\_BUFFER 的大小，也能够直接返回-1，跳出 frontswap 接口，在内核中进行后续的正常处理，而不会在 copy\_page 中因为索引超出大小而报错，对于后续的 RDMA BACKEND 的实现，也采用了此方法。最终两个接口实现的代码如下所示，注意需要用 EXPORT\_SYMBOL 导出以供 multiswap.c 使用：

multiswap\_dram read and write

```
int sswap_rdma_write(struct page *page, u64 roffset)
{
    //this part is not use frontswap
    if(roffset >= REMOTE_BUF_SIZE)
        return -1;
    void *page_vaddr;
    page_vaddr = kmap_atomic(page);
    copy_page((void *) (drambuf + roffset), page_vaddr);
    kunmap_atomic(page_vaddr);
    DEBUG_PRINT("write over\n");
    return 0;
}
EXPORT_SYMBOL(sswap_rdma_write);

int sswap_rdma_read_sync(struct page *page, u64 roffset)
{
    //this part is not use frontswap
    if(roffset >= REMOTE_BUF_SIZE)
        return -1;
    void *page_vaddr;

    VM_BUG_ON_PAGE(!PageSwapCache(page), page);
    VM_BUG_ON_PAGE(!PageLocked(page), page);
    VM_BUG_ON_PAGE(PageUptodate(page), page);

    page_vaddr = kmap_atomic(page);
    copy_page(page_vaddr, (void *) (drambuf + roffset));
    kunmap_atomic(page_vaddr);

    SetPageUptodate(page);
}
```

```
unlock_page(page);
DEBUG_PRINT("read over\n");
return 0;
}
EXPORT_SYMBOL(sswap_rdma_read_sync);
```

### DRAM BACKEND 的编译与安装

在之前的 Makefile 的基础上，只需要将 multiswap\_dram.o 加入到要编译的模块对象列表中去，就能自动搜索对应的 c 文件进行编译了。

#### 修改 makefile

```
obj-m := multiswap.o
obj-m += multiswap_dram.o
```

在安装模块时，由于 multiswap 模块中调用了 multiswap\_dram 的接口函数，所以需要先安装后者，卸载模块时则相反：

#### 模块安装与卸载

```
# 模块安装
sudo insmod multiswap_dram.ko
sudo insmod multiswap.ko
# 模块卸载
sudo rmmod multiswap.ko
sudo rmmod multiswap_dram.ko
```

### 4.5.3 DRAM BACKEND 正确性测试

对于项目的每一步实现，我们都做了相应的正确性测试，这里安排一小节是为了展示项目完成过程中的完整性，具体测试的内容请参见第7节。

### 4.5.4 为 Linux 内核添加补丁 2

在测试过程中，我们尝试在每一次 SWAP 关键路径上打印出相关信息来检验相关实现的正确性。在初次测试时，multiswap 模块能够成功安装，但是具体的交换路径并没有按照我们所想象的一样被 frontswap 拦截，而是继续交换到 swap 分区中。在先前我们对 Linux 5.4 版本的内核中进行相关测试时，却没有产生此问题。经我们探索后发现，这是因为老版本内核和 6.1 版本内核对注册的 frontswap\_ops 所采用的管理机制不同所导致的。

在老版本的内核中，每一个新注册的 frontswap\_ops 都会加入一个全局的链表结构用于管理，在进行相关 store 或者 load 操作时，会从头开始遍历该链表，尝试链表中每一个注册对象的 store 和 load 接口，直到有一个能够返回成功，这无疑在某些时候减慢了 swap 的速度。于是在 Linux 6.1 内核中，进行了相关修改，简化了这个过程：定义一个全局的 frontswap\_ops 变量，在初始化时空。在进行注册时如果该全局变量为空，则将要注册的对象赋值给它，否则直接退出注册流程并返回失败。在后续的 store 和 load 操作中，直接调用该注册对象的 frontswap 接口操作。

但是我们手动注册的 ops 并非总是第一个注册的 ops，例如在编译内核前产生的.config 文件中包含 ZSWAP 相关选项，zswap 一种 frontswap 的 backend 实现，应用它会导致 frontswap\_ops 被抢先注册导致注册流程失败，并且在我们后续测试时需要频繁对模块进行安装与卸载，于是决定修改这部分内核函数，注释掉条件判断语句，使新注册的 frontswap\_ops 直接替代旧的操作，修改后如下：

#### 修改内核 register 函数

```
int frontswap_register_ops(const struct frontswap_ops *ops)
{
    //this only enable one ops! diff from kernel 5.4
    // if (frontswap_ops)
    // return -EINVAL;

    frontswap_ops = ops;
    static_branch_inc(&frontswap_enabled_key);
    return 0;
}
EXPORT_SYMBOL(frontswap_register_ops);
```

除此之外，在4.5.1节所提到的更新页状态，原本操作是在内核中 page\_io.c 中的 swap\_readpage 中进行的，由于我们将其修改到了模块中完成，我们需要将原处注释掉：

#### 取消更新页状态

```
int swap_readpage(struct page *page, bool synchronous,
                  struct swap_iocb **plug)
{
    //省略...
    if (frontswap_load(page) == 0) {
        // SetPageUptodate(page);
        // unlock_page(page);
        goto out;
    }
    //省略...
}
```

至此，DRAM BACKEND 能够成功安装并运行，接下来我们将介绍 RDMA BACKEND 的开发过程。

## 4.6 使用 RDMA 作为 multswap 的 BACKEND

### 4.6.1 内核态 RDMA 连接建立的交互过程

远程内存节点作为 server 端为计算节点提供远程内存，需要在其之上运行一个用户态程序建立连接，并且发送给计算节点自身的 memory region 以及 remote key，这样计算节点才能指定数据发送到内存节点的位置。发送完这些相关信息后，计算节点就能通过 DMA 与 RDMA 技术实现两端 CPU 无感知的交换工作。在4.2节中，我们已在用户态实现了两台机器 RDMA 的连接建立，接下来需要将 client 端建立连接的过程移植到内核态当中，尝试计算节点在内核态建立 RDMA 连接，远程内存节点建立连接的过程不需要过多改变。在内核态中，请求建立连接的函数和其相关参数有很大改变，下

图4.28为内核态程序 (计算节点) 与用户态程序 (内存节点) **建立连接时的交互过程**：

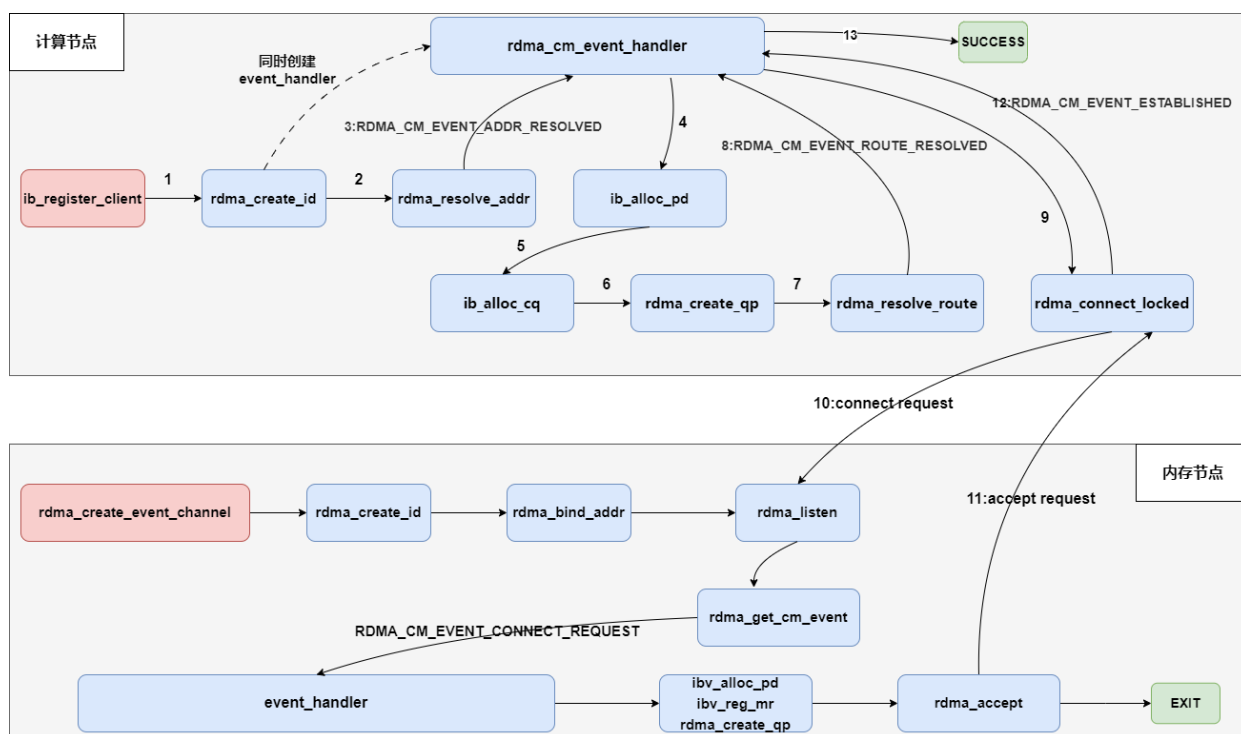


图 4.28: 交互过程

接下来我们将分为内存 server 端和计算 client 端两部分，分别介绍其连接建立的过程。

#### 4.6.2 为远程内存节点编写 server 程序

在 FASTSWAP<sup>[1]</sup> 的工作中，编写了一个运行在 Linux 4.1 版本上的用户态程序，可以进行 RDMA 连接并发送信息，其接口随内核版本并没有发生名称和功能上的改变，我们可以直接使用，其初始化和连接过程如图4.28下半部分所示，简要介绍一下其工作过程：

1. 首先创建一个全局队列管理器 (global controller)，用来统一管理所有的队列连接，同时保存这些队列所共享的保护域 (pd) 以及所绑定 RDMA 设备的上下文 (verbs)，其中包含了与该设备相关的所有资源。
2. 接着，和我们在4.2中所做的操作一样，创建一个事件通道 (event\_channel)，并将其事件通道通过 rdma\_create\_id 绑定到一个通信标识符 (cm\_id) 中。最后将其绑定到指定的监听端口用来监听事件即可。至此，服务端完成了等待连接前的必要操作，可以在该事件通道中等待并获取相关事件。
3. 在监听到连接请求后，为接收请求做准备：首先为队列所对应的 RDMA 设备分配 pd，接着在该 pd 中注册一块允许远程读写的内存区域 (memory region)，最后为队列对 (qp) 的相关属性进行初始化后即可通过 rdma\_create\_qp 创建队列对，创建完成后的队列对会返回在 cm\_id 结构体的相关属性中。于是可以使用此 cm\_id 接受远端的连接请求。

连接建立之后的 client 节点并没有 server 节点的 memory region 以及 rkey 等相关信息，只能采用 send 与 receive 的方式进行通信，此通信方式需要占用 CPU 进行处理。于是我们需要在服务端主

动发送这些信息，在计算节点主动接收这些信息并保存，此操作后，计算节点就能够通过 RDMA 的 read 和 write 操作向内存节点进行数据的传输。

落实到实际实现上，我们需要向 send queue 中添加 work request。专用于发送的 work request 结构体成员如下：

发送请求结构体

```
struct ibv_send_wr {
    uint64_t      wr_id;
    struct ibv_send_wr  *next;
    struct ibv_sge      *sg_list;
    int      num_sge;
    enum ibv_wr_opcode  opcode;
    unsigned int      send_flags;
    //...
}
```

该结构体是一次传输中任务的集合，也是最小单位。集合中每个具体的传输内容保存在其中的 sge 对象中，这些对象组成链表 sg\_list 保存在 wr 里，并使用 next 指针来逐个访问。由于我们明确需要传输的内容为注册的 memory region 的起始地址以及它的 rkey，将其封装后放置到一个 sge 中即可。最后调用 ibv\_post\_send，指定要发送的队列对以及 wr 传输数据，其实现如下：

发送远端信息

```
int send_message(struct queue *q)
{
    struct ibv_send_wr wr = {};
    struct ibv_send_wr *bad_wr = NULL;
    struct ibv_sge sge = {};
    struct memregion servermr = {};

    printf("connected. sending memory region info.\n");
    printf("MR key=%u base vaddr=%p\n", ctrl->mr_buffer->rkey, ctrl->mr_buffer->addr);

    servermr.baseaddr = (uint64_t) ctrl->mr_buffer->addr;
    servermr.key = ctrl->mr_buffer->rkey;

    wr.opcode = IBV_WR_SEND;
    wr.sg_list = &sge;
    wr.num_sge = 1;
    wr.send_flags = IBV_SEND_SIGNALED | IBV_SEND_INLINE;

    sge.addr = (uint64_t) &servermr;
    sge.length = sizeof(servermr);

    TEST_NZ(ibv_post_send(q->qp, &wr, &bad_wr));
}
```

### 4.6.3 内核态 RDMA 连接的建立与销毁

#### 内核态 RDMA 连接的建立

此部分内容为图4.28的上半部分的细节实现，参考了 Linux 6.1 内核文件中 `/drivers/nvme/host/rdma.c` 目录下的实现。首先，我们需要调用 `ib_register_client` 函数向 InfiniBand 中间层注册 `ib_client`，在这之后，开始相关的初始化操作。和 server 端类似，为了方便管理，也定义了一个全局的 `ctrl` 结构体管理所有队列和连接相关的内容，该结构体定义如下所示：

ctrl 结构体

```
struct sswap_rdma_ctrl {
    struct sswap_rdma_dev *rdev;
    struct rdma_queue *queues;
    struct sswap_rdma_memregion servermr;

    union {
        struct sockaddr addr;
        struct sockaddr_in addr_in;
    };

    union {
        struct sockaddr srcaddr;
        struct sockaddr_in srcaddr_in;
    };
};
```

在相关结构 `rdma_dev` 中，包含了 protect domain 以及 rdma 设备的相关信息，`queues` 指针用来管理在连接中所创建的所有 RDMA 队列，`rdma_queue` 结构体是对底层的各种类型的 `ib_queue` 的封装，其数据结构实现如下：

rdma\_queue 结构体

```
struct rdma_queue {
    struct ib_qp *qp; //queue pair
    struct ib_cq *cq; //completion queue
    spinlock_t cq_lock;
    enum qp_type qp_type;

    struct sswap_rdma_ctrl *ctrl; //rdma controller

    struct rdma_cm_id *cm_id;
    int cm_error;
    struct completion cm_done; //完成变量，用于同步

    atomic_t pending; //队列中未处理完的cq数量
};
```

每一个队列结构体都会与 server 端创建一个队列连接，因此需要单独为其分配一个 `cm_id`，该结

构体除了包含了 queue pair 以及 complete queue 等信息之外，还包含了可用于在连接时同步队列<sup>2</sup>操作的一些变量，使用的是内核提供的 completion 机制。

controller 控制器除了上述以外，还保存了两端网络地址和远端 memory region 和 rkey 等信息。

我们在模块的初始化函数里进行连接的建立。为每一个 CPU 核心都分配了两个独立的队列，分别用于 rdma 的读和写操作，并为这些队列依次建立连接。初始化有以下工作需要完成：

1. 为全局控制器分配空间，将传入的本地与远端 ip 地址 (此 ip 即在4.1中提到的网卡设备的 ip 地址) 字符串转换成二进制形式并保存在 sockaddr\_in 结构体中。
2. 为和全局控制器进行关联的所有队列结构体分配空间，并初始化其中的完成变量以及 complete queue element 计数器，这一部分需要使用原子操作来进行。
3. 为每一个队列分配 rdma\_cm\_id，此过程和用户态有很大不同，主要体现在 rdma\_create\_id 函数的执行上，该内核函数的原型如下：

```
/**
 * rdma_create_id - Create an RDMA identifier.
 *
 * @net: The network namespace in which to create the new id.
 * @event_handler: User callback invoked to report events associated with the
 *   returned rdma_id.
 * @context: User specified context associated with the id.
 * @ps: RDMA port space.
 * @qp_type: type of queue pair associated with the id.
 *
 * Returns a new rdma_cm_id. The id holds a reference on the network
 * namespace until it is destroyed.
 */
#define rdma_create_id(net, event_handler, context, ps, qp_type) \
    __rdma_create_kernel_id(net, event_handler, context, ps, qp_type, \
        KBUILD_MODNAME)

//调用
queue->cm_id = rdma_create_id(&init_net, sswap_rdma_cm_handler, queue,
    RDMA_PS_TCP, IB_QPT_RC);
```

net 参数需要指定网络命名空间，我们直接使用内核中的初始网络命名空间 init\_net 即可，第二个参数需要我们手动指定事件处理句柄，这一点和用户态程序有很大不同。在捕获到事件后，会自动触发 callback，调用该事件处理句柄进行相应处理。context 参数指定绑定在此 cm\_id 上的队列相关信息，我们将封装好的队列结构体传入即可。另外还有两个参数需要说明：port space 参数指定准备使用的 RDMA 端口空间，我们选用 RDMA\_PS\_TCP 提供可靠、面向连接的 QP 通信 (其与 TCP 不同，RDMA 端口空间提供基于消息的通信，而不是基于流的通信)。qp type 指定使用的 RDMA 队列类型，我们选用 RC(reliable connection) 类型。

在准备了上述工作后，就能够正式开始连接的建立，如图4.28中所示，将本地转换后的 ip 地址与远端地址传入 rdma\_resolve\_addr 函数中解析，解析过程中为相应的 cm\_id 识别到 RDMA 网卡设

<sup>2</sup>注：在此后提到的队列，如果没有特别说明，都指的是封装后的结构体



备，保存在其中的 device 成员里。此后控制流转移到事件处理句柄中，在此处只需要同步等待连接建立成功即可。需要利用 wait\_for\_completion 来等待所设置的完成变量在连接建立后的完成。

在事件处理句柄捕获到解析地址完成事件后，可以对 RDMA 相关资源进行分配操作：

1. 根据队列结构体绑定的 cm\_id 获取 RDMA 设备，并将其加入到全局控制器中（只有一张网卡，所以所有的队列共用一个设备），然后对此设备使用内核函数 ib\_alloc\_pd 分配 protect domain，我们让所有队列共享一块 pd，所以只需要执行一次后加入到全局控制器中即可。
2. 创建完成队列 complete queue，内核函数 ib\_alloc\_cq 提供快速且简化的完成队列创建，仅仅需要传入我们需要的上下文数据并指明轮询该队列的上下文。有两种常用的上下文如下所示：
  - IB\_POLL\_DIRECT: 直接轮询，直接在调用者的上下文中轮询，需要使用轮询函数 ib\_process\_cq\_direct 直接处理其中的 callback。
  - IB\_POLL\_SOFTIRQ: 在软中断上下文中轮询，不需要手动进行轮询调用处理，而是在切换到软中断上下文中自动处理。

由于对 page 的处理优先级较高，我们对读写队列的完成队列处理方式都采用前者。

3. 在做好了上述准备后，只需要为队列对准备好初始化属性，将创建好的完成队列以及相关初始化参数传入，即可成功创建队列对。

在进行了上述操作后，即可调用 rdma\_resolve\_route 函数进行路由解析，事件处理句柄接收到对应的事件后，设置好连接参数，即可尝试连接了。在此过程中我们遇到了一些困难。初次尝试使用与用户态一致的 rdma\_connect 函数连接，但是陷入了死等待的状态且无法卸载驱动，但是使用该函数在 Linux 5.4 版本的内核中执行却能连接成功。在 6.1 版本下进入内核/drivers/infiniband/core/cma.c 查看该函数的实现如下：

#### rdma\_connect 函数的实现

```
int rdma_connect(struct rdma_cm_id *id, struct rdma_conn_param *conn_param)
{
    struct rdma_id_private *id_priv =
        container_of(id, struct rdma_id_private, id);
    int ret;

    mutex_lock(&id_priv->handler_mutex);
    ret = rdma_connect_locked(id, conn_param);
    mutex_unlock(&id_priv->handler_mutex);
    return ret;
}
```

该函数首先会根据 cm\_id 在结构体 rdma\_id\_private 中的位置来获取 priv 结构体，再为其加上锁，接着调用函数 rdma\_connect\_locked 来真正进入连接。通过在内核中加入打印信息确认了问题在于此处 id\_priv 一直无法上锁。我们对比了 5.4 与 6.1 的内核，发现在 5.4 的内核版本中并没有 rdma\_connect\_locked 函数，仅有 rdma\_connct 函数。再详细查看内部实现，其与 6.1 版本中 rdma\_connect\_locked 函数的实现一致，于是判断此处上锁这是 6.1 内核的新特性。我们仅仅需要调用老版本的内核函数 rdma\_connect\_locked 进行连接即可，这部分的代码实现如下：

## 建立连接

```
static int sswap_rdma_route_resolved(struct rdma_queue *q,
    struct rdma_conn_param *conn_params)
{
    struct rdma_conn_param param = {};
    int ret;

    param.qp_num = q->qp->qp_num;
    param.flow_control = 1;
    param.responder_resources = 16;
    param.initiator_depth = 16;
    param.retry_count = 7;
    param.rnr_retry_count = 7;
    param.private_data = NULL;
    param.private_data_len = 0;

    // 进行addr_resolved和route_resolved后, 进行connect,
    ret = rdma_connect_locked(q->cm_id, &param);
    if (ret) {
        pr_err("rdma_connect failed (%d)\n", ret);
        sswap_rdma_destroy_queue_ib(q);
    }
    return 0;
}
```

至此连接建立成功，在事件处理函数捕捉到此信息后，唤醒最初进行等待的完成变量，连接建立部分结束。

## 接收远程节点的相关信息

在两端连接建立后，计算节点还需要获得内存节点的内存区域以及 remote key，这样才能在 RDMAread 和 write 操作时指定从远端的特定地址拿取或者存储数据。本部分内容参考了 nvme/host/rdma.c<sup>[2]</sup> 中的实现。

远程内存节点部分，我们使用了 rdma\_send 操作发送了其相关 memory region 以及 rkey 到计算节点，我们需要创建工作请求（用自定义 rdma\_req 结构体来表示）进行 rdma receive 操作接收这些有关信息。我们将 rdma\_req 结构体定义如下：

```
struct rdma_req {
    struct completion done;
    struct ib_cqe cqe;
    u64 dma;
    struct page *page;
};
```

这是一个用于封装工作请求的结构体，主要用于队列处理完毕后在回调函数中的资源释放以及同步操作，其中包含了一个完成变量，64 位的无符号地址等成员，在后续使用时我们接着说明。

交互过程示意图如图4.29所示：

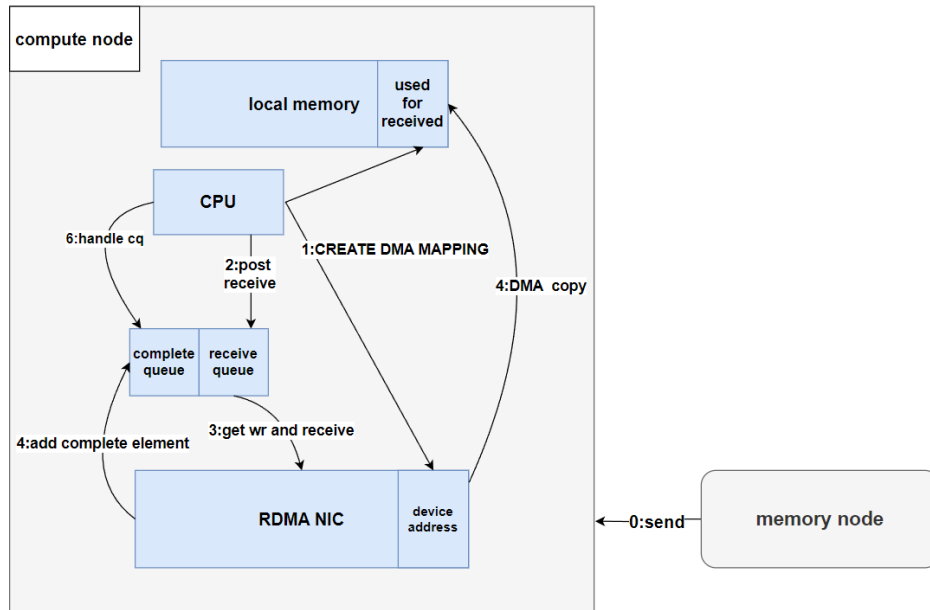


图 4.29: 通过 recv 接收远端信息

在用户态接收数据时，我们在 send/receive 时将传输内容的起始虚拟地址以及长度传入 sge(传输数据的最小单元)中即可做到传输。在内核中则麻烦一些，需要使用 `ib_dma_map_single` 函数，此函数会根据对应的 ib 设备是否支持虚拟地址的 DMA 来决定是否执行 DMA 映射。若支持虚拟地址的 DMA，则直接返回传入的虚拟地址并使用即可，若不支持，则需要将其映射到 IB 设备的物理地址空间中，以便在 RDMA 传输中使用。

#### ib\_dma\_map\_single 内核函数

```

/**
 * ib_dma_map_single - Map a kernel virtual address to DMA address
 * @dev: The device for which the dma_addr is to be created
 * @cpu_addr: The kernel virtual address
 * @size: The size of the region in bytes
 * @direction: The direction of the DMA
 */
static inline u64 ib_dma_map_single(struct ib_device *dev,
                                     void *cpu_addr, size_t size,
                                     enum dma_data_direction direction)
{
    if (ib_uses_virt_dma(dev))
        return (uintptr_t)cpu_addr;
    return dma_map_single(dev->dma_device, cpu_addr, size, direction);
}
  
```

该函数返回一个 64 位的设备物理地址，参数包括需要映射的设备，内存地址及大小以及 DMA 传输的方向。由于我们只需要接收来自远程内存的消息，所以只需要创建由设备向内存区域单方向传输的 DMA 映射即可。在创建完相关映射后，需要先进行这两个映射区域的同步操作，使得设备的这块物理地址与内存地址的内容完全一致，这样就完成了简易版创建内存区域的过程。此后，若在设备的映射

区域接收到了内容，DMA 控制器将会自动将其复制到内存映射位置上：

#### 映射设备地址到内存区域

```
inline static int get_req_for_buf(struct rdma_req **req, struct ib_device *dev,
    void *buf, size_t size,
    enum dma_data_direction dir)
{
    int ret;
    ret = 0;
    *req = kmem_cache_alloc(req_cache, GFP_ATOMIC);
    if (unlikely(!req)) {
        pr_err("no memory for req\n");
        ret = -ENOMEM;
        goto out;
    }
    init_completion(&(*req)->done);

    (*req)->dma = ib_dma_map_single(dev, buf, size, dir);
    if (unlikely(ib_dma_mapping_error(dev, (*req)->dma))) {
        pr_err("ib_dma_mapping_error\n");
        ret = -ENOMEM;
        kmem_cache_free(req_cache, req);
        goto out;
    }
    ib_dma_sync_single_for_device(dev, (*req)->dma, size, dir);
out:
    return ret;
}
```

在做好映射之后，还需设置好 work request 中事件完成后的**回调函数**，该函数会在工作请求完成时被调用，用于资源释放解除 DMA 映射等操作，即在图4.29中通知 CPU 任务已完成，可以进行资源释放的部分，事实上，捕获完成事件是 CPU 主动轮询等待完成队列进行的。

做好了以上准备后，可以开始最终的 recv 操作，使用 ib\_post\_recv 函数发送接收请求。与用户态接收不同之处在于需要提供本地的 local key（在 pd 中保存），且传入接收的地址为先前与内存创建 DMA 映射的 RDMA 设备的物理地址。发送请求后，硬件会开始接收内容到指定的地址中，DMA 也开始将其往映射的内存中搬运。可以隔一段时间查询完成队列中是否出现了新的完成工作（在请求被成功处理后，硬件会为完成队列中添加元素）。若出现了完成事件，则调用 ib\_process\_cq\_direct 函数直接处理。处理时会调用之前设置的回调函数，在回调函数中，根据完成队列中的数据上下文查找到该完成队列所对应的 queue 结构体，进行相关资源的释放，同时将初始化工作请求时设置的 completion.done（完成变量，用于同步）+1，结束对完成队列的轮询，内存中的对应区域也已经由 DMA 控制器完成了内容的传输。这部分的函数调用流程如下图4.30所示：

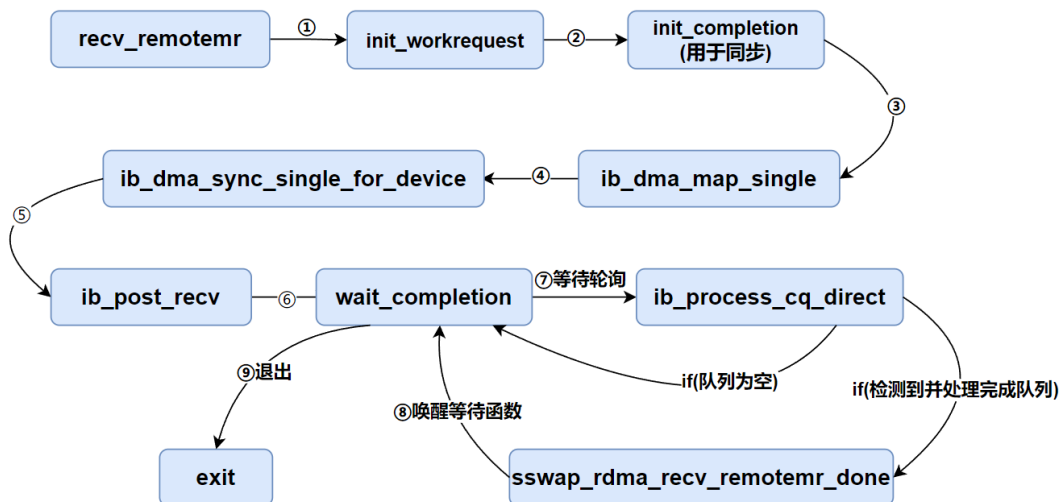


图 4.30: 采用 recv 接收远端信息

### 内核态 RDMA 连接的销毁

销毁连接的流程相对固定：首先调用 `rdma_disconnect` 函数关闭连接，由于我们在本地没有注册相关 memory region，不需要对其资源进行释放，只需要依次释放队列的相关资源即可，使用的为内核态 (以 `ib` 开头的) 相关接口。

#### 销毁连接

```

static void sswap_rdma_free_queue(struct rdma_queue *q)
{
    rdma_destroy_qp(q->cm_id);
    ib_free_cq(q->cq);
    rdma_destroy_id(q->cm_id);
}
  
```

### 4.6.4 在内核态实现远端 CPU 无感知的 RDMA 通信

在2.3节中我们提到，RDMA 有两种通信方式，分别为需要两端 CPU 参与控制操作的 `send` 和 `recv` 方式，以及只需本地 CPU 参与控制操作的 `read` 和 `write` 方式，在本地获取到远端的 memory region 地址以及 remote key 后，我们将可以使用此方式来完成以 page 为粒度的页面交换。这部分的内容我们通过重写 `sswap_rdma_read` 和 `sswap_rdma_write` 函数来实现。读和写操作总体的流程相差不大，我们以先以写操作为例（页面总是要被先换出再换入）。

#### RDMA:write 操作实现页换出

首先，和 DRAM 中一致，需要做越界大小检查，保证换出页的偏移地址在远程内存所提供的范围内，接着找到当初为每一个 CPU 准备的写队列，后续将使用其向 send queue 发送请求，流程和 `recv` 操作很类似，由以下几个部分组成：

1. 流量检测并控制：在每一次开始传输之前，需要查看此队列的未完成工作数量，若当前的 request 较多，则先轮询处理完部分请求，以防队列阻塞过久。

2. 创建 DMA 映射后进行同步操作，并且初始化完成变量：与 recv 操作一样，我们需要为准备换出的物理页和 RDMA 设备的物理地址进行 DMA 映射并返回该设备地址，此刻该 DMA 传输方向应设置为 DMA\_TO\_DEVICE，在内核中有专用函数 `ib_dma_map_page` 来创建物理页的映射，这样 DMA 控制器能够自动将该物理页的内容传输到 RDMA 设备中。在调用 `ib_dma_sync_single_for_device` 进行同步后，此设备地址空间上已经包含了我们所想要换出页的内容。除此之外，同样也需要初始化完成变量，用于在传输完成后进行同步以及释放资源。
3. 设置好回调函数，在处理到对应完成事件后会自动调用该函数解除 DMA 映射、更新状态。
4. 接下来可以向 send queue 队列发送请求，使用的是 RDMA 的 write 操作，发送此类请求需要使用专用 work request: `ib_rdma_wr`。查看其数据结构可以发现，是由普通的工作请求结构体 `ib_send_wr` 封装形成的：

rdma 请求结构体

```
struct ib_rdma_wr {
    struct ib_send_wr wr;
    u64      remote_addr;
    u32      rkey;
};
```

其中增加了两个成员，用来表示将要进行操作的远程内存地址，以及远程访问权限 `rkey`。其中的 `wr` 也同样包含基本的 `sge`，在 `sge` 中指定本地进行交互的地址（即为先前建立好映射的 RDMA 设备地址）。由于 send 操作请求可能有 read 和 write 两种，需要在 `wr` 的 `opcode` 字段指明操作类型为 `IB_WR_RDMA_WRITE`。对于单个内存节点而言，远程访问的地址 `remote_addr` 即为接收到的 `remotemr` 起始地址加上原本该页在 swap device 中的偏移量，即将远程内存当作更快的远程交换设备来使用。最后，在将队列中的操作计数器增加后，使用 `ib_post_send` 发送该请求，具体实现如下：

发送 rdma write 请求

```
inline static int sswap_rdma_post_rdma(struct rdma_queue *q, struct rdma_req *qe,
    struct ib_sge *sge, u64 roffset, enum ib_wr_opcode op)
{
    struct ib_rdma_wr rdma_wr = {};
    int ret;

    BUG_ON(qe->dma == 0);

    sge->addr = qe->dma;
    sge->length = PAGE_SIZE;
    sge->lkey = q->ctrl->rdev->pd->local_dma_lkey;

    rdma_wr.wr.next = NULL;
    rdma_wr.wr.wr_cqe = &qe->cqe;
    rdma_wr.wr.sg_list = sge;
    rdma_wr.wr.num_sge = 1;
    rdma_wr.wr.opcode = op;
    rdma_wr.wr.send_flags = IB_SEND_SIGNALED;
```

```

rdma_wr.remote_addr = q->ctrl->servermr.baseaddr + roffset;
rdma_wr.rkey = q->ctrl->servermr.key;
DEBUG_PRINT("roffset is: %llu\n", roffset);

atomic_inc(&q->pending);

ret = ib_post_send(q->qp, &rdma_wr.wr, NULL);
if (unlikely(ret)) {
    pr_err("ib_post_send failed: %d\n", ret);
}

return ret;
}

```

5. 在发送完成后，若队列中还有未完成的工作，即 `q->pending` 大于 0，我们则轮询完成队列并处理，在处理完成队列时的回调函数会将 `q->pending` 减 1，直至它为 0<sup>3</sup>。

此部分的函数调用流程如图4.31：

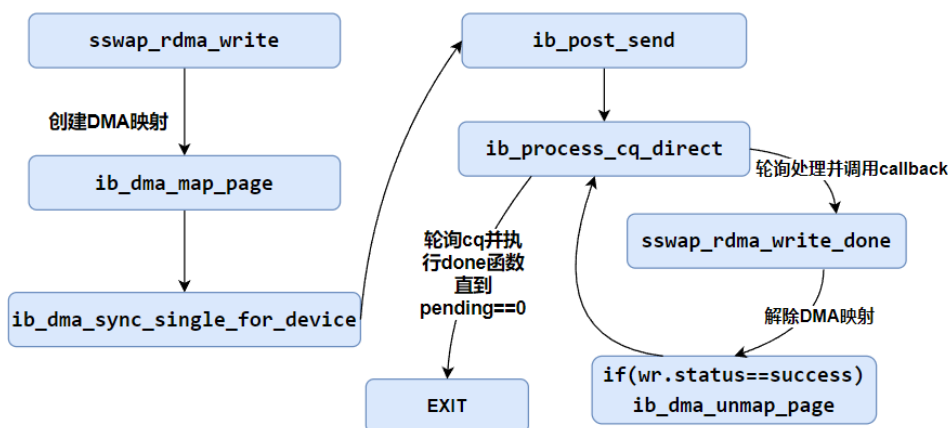


图 4.31: write 操作执行流程

此过程中双端软硬件交互的过程如图4.32所示，本地 CPU 仅参与控制面，远端 CPU 完全不参与，总结描述如下：

1. 本地 CPU 为换出页与 RDMA 网卡设备的地址进行 DMA 映射，并同步到设备中。
2. 本地 CPU 向队列中发起 write 请求。
3. RDMA 网卡处理该请求，向远端进行 write 操作。
4. 远端网卡处理请求后直接将内容写入 memory region 中，并返回确认信息。
5. 本地 RDMA 网卡在收到确认后，向完成队列当中添加元素，代表此请求已经完成。

<sup>3</sup>注：此处为简单实现，我们假设 rdma 网卡处理队列的速度足够快，否则在此处将花费大量时间，而我们后面的实验证明，即使这样实现，也足以体现其优势，在后续可考虑优化。



6. CPU 轮询到完成队列中的元素，处理回调函数，工作完成。

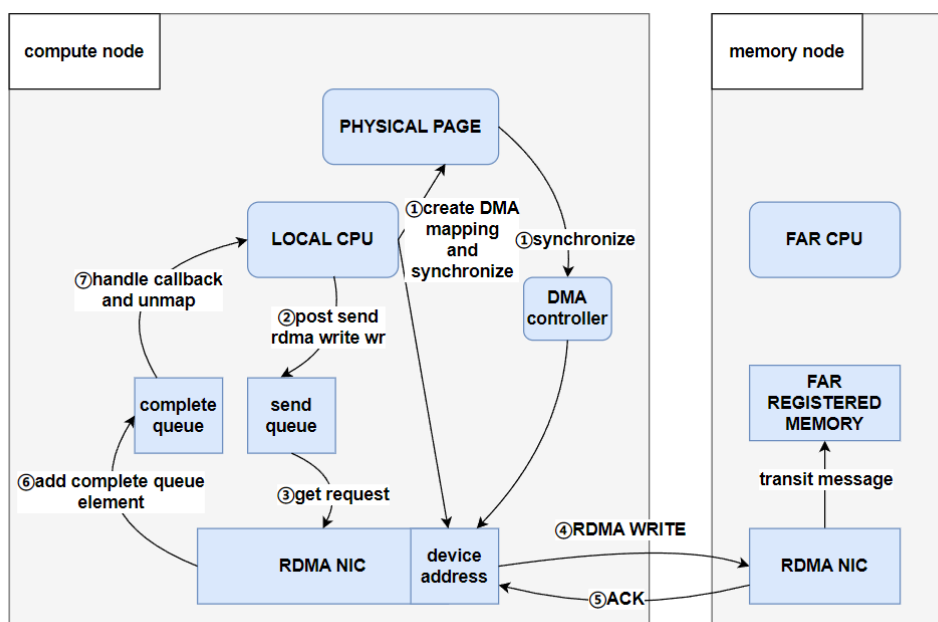


图 4.32: write 操作系统交互图

### RDMA:read 操作实现页换入

read 操作与 write 操作相差不大，只是需要另外增加一些条件判断：在读取页面之前，需要做一些检查：保证该页在 SwapCache 且被锁定，并且不是 uptodate 的状态。除此之外，还有一些不同：

1. 在与 RDMA 设备物理地址空间进行 DMA 映射以及解除映射时，由于是从 RDMA driver 读入到内存中，传入的 direction 参数为 DMA\_FROM\_DEVICE。
2. 在发送请求时，仍然调用的是 `ib_post_send` 函数，只是其中 `wr.opcode` 参数需要设置为 `IB_WR_RDMA_READ`。
3. 解除 DMA 映射时，为保证一致性，需要将内容同步到内存中<sup>4</sup>。
4. 轮询 cq 并执行 callback，在 wr 执行成功后，需要解锁该页面，并将该页面的状态设置为 uptodate。这部分的操作原本是在 `frontswap.c` 中 `frontswap_load` 函数调用完 `ops->load` 函数中执行的，在同步读的情况下这么做没问题，但若是在异步读的情况下，`ops->load` 函数返回并不代表页面真正读取完成，所以为了后续扩展方便，将这两个操作统一放在模块驱动中执行。

程序执行流程与 write 操作基本一致，在此处不再赘述。至此，单内存节点 RDMA BACKEND 实现。

#### 4.6.5 单个远程内存节点正确性测试

对于项目的每一步实现，我们都做了相应的正确性测试，这里安排一小节是为了展示项目完成过程中的完整性，具体测试的内容请参见第7节。

<sup>4</sup>我们使用 `ib_dma_sync_single_for_cpu` 函数同步到内存中，但是即使不使用该函数在实际实验中也并没有出错，查阅资料得知 `unmap` 函数在特定情况下能够将缓冲内容 flush 到内存中，但是为了保证严谨性，还是进行了同步操作

## 5 阶段二：配置交换机并连接两个内存节点

### 5.1 SN2700 交换机配置

#### 交换机系统配置

交换机的系统为 Cumulus Linux 4.7。Cumulus Linux 是一种专为网络交换机设计的开放式网络操作系统，提供了一种灵活、高效的网络管理方式。基于 Linux 内核，Cumulus Linux 支持标准的 Linux 工具和网络协议，使得网络管理人员能够利用熟悉的 Linux 命令和脚本进行网络配置和管理。这种开放性使得用户可以定制网络环境以满足特定需求，而不依赖于专有的硬件和软件解决方案。

由于 Cumulus Linux 4.7 并不兼容现有的 RDMA 网卡，我们需要将其手动升级至 Cumulus 5.9 版本。

首先，用 USB 转 DB9 的 console 线连接 SN2700 的 console 接口和电脑的 USB 口，用串口连接交换机。串口连接的配置配置如下：

Parameter	Setting
Baud Rate	115200
Data bits	8
Stop bits	1
Parity	None
Flow Control	None

默认开机后会进入 ONIE 平台，选择第一个选项” Install OS”：



图 5.33: ONIE 平台

默认 ONIE 会自动寻找脚本去自动安装，等待进入 ONIE 界面，使用如下命令让 ONIE 停止自动发现脚本的行为：

```
onie-stop
```

```

Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-x86_64-mlnx_msn2100-r0 ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-x86_64-mlnx_msn2100-r0.bin ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-x86_64-mlnx_msn2100 ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-x86_64-mlnx_msn2100.bin ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-mlnx_msn2100 ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-mlnx_msn2100.bin ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-x86_64-mlnx ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-x86_64-mlnx.bin ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-x86_64 ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer-x86_64.bin ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer ...
Info: Attempting http://[fe80::a94:efff:fe10:17f9%eth0]/onie-installer.bin ...
Info: Attempting tftp://onie-server/pxelinux.0 ...
Info: Attempting tftp://10.209.0.50/pxelinux.0 ...

```

图 5.34: ONIE 平台

```

ONIE:/ # onie-stop
discover: installer mode detected.
Stopping: discover... done.
ONIE:/ #

```

图 5.35: ONIE 平台

将启动盘插入 SN2700 的 USB 口（建议将 U 盘格式化成 fat32 格式）：



图 5.36: SN2700 USB 口示意图

在 onie 运行命令：

```
fdisk -l
```

查看 USB 的 disk 目录名，一般为”/dev/sdb1”。运行下面的挂载命令将 USB 设备挂载交换机本地的/mnt/usb 目录：

```
mount /dev/sdb1 /mnt/usb
```

```

ONIE:/mnt/usb # fdisk -l

Disk /dev/sda: 30.0 GB, 30016659456 bytes
255 heads, 63 sectors/track, 3649 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1    *           1         3650     29313143+  ee  EFI GPT

Disk /dev/sdb: 15.7 GB, 15787360256 bytes
255 heads, 63 sectors/track, 1919 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1    *           1         1920     15416320   c   Win95 FAT32 (LBA)

```

图 5.37: ONIE 平台

进入/mnt/usb 目录，将” cumulus-linux-5.9.0-mlx-amd64.bin” 文件拷贝到/tmp 目录下，此时可以将 USB 拔出。

1 分钟后交换机会自动重启并进入下面菜单，此时无需操作，等待其进入安装即可：

```

GNU GRUB version 2.02

+-----+
| CUMULUS-INSTALL |
| CUMULUS-INSTALL [Interactive] |
| ONIE |
+-----+

Use the ^ and v keys to select which entry is highlighted.
Press enter to boot the selected OS, 'e' to edit the commands
before booting or 'c' for a command-line.
The highlighted entry will be executed automatically in 1s.

```

图 5.38: GRUB 界面

接着，进入约 15 分钟的安装过程（该过程无需任何操作）：

```

Step 3/7: Installing System Image
Copying...
13%

```

图 5.39: ONIE 平台

安装完成后，会进入图5.40所示的 login 画面，第一次登陆的默认用户名是与密码均为 cumulus，

登陆成功后会进入密码更新界面：

```
[ OK ] Reached target Multi-User System.
Starting Bootlog Service...
Starting Update UTMP about System Runlevel Changes...
[ OK ] Started Clag Reboot Notifier.
[ OK ] Started Update UTMP about System Runlevel Changes.
[ OK ] Started Bootlog Service.

Debian GNU/Linux 10 ttyS0
login: █
```

图 5.40: ONIE 平台

更新密码，Cumulus Linux 刷机完成。

```
You are required to change your password immediately (administrator enforced)
Changing password for cumulus.
Current password:
```

图 5.41: ONIE 平台

若观察到 RDMA 网卡指示灯正常，那么证明连接成功，可以开始配置桥接等工作。

## 配置桥接

在这一节中，我们将进行进一步设置，在交换机端配置桥接：

### 配置桥接

```
nv set interface swp1-32 bridge domain br_default access 100
nv set interface swp1-32 link state up
nv set system config auto-save enable on
nv config apply -y
```

- `nv set interface swp1-32 bridge domain br_default access 100`
  - **作用：** 这条命令将接口 `swp1-32` 加入到 `br_default` 桥接域中，并将其设置为访问模式 (access mode)，使用 VLAN ID 100。
  - **bridge domain br\_default：** 将接口分配到名为 `br_default` 的桥接域。这通常用于在交换机中对数据流量进行隔离。
  - **access 100：** 将接口设置为访问模式的接口，这意味着该接口只处理与指定 VLAN ID（此处为 100）相关的流量。这种模式通常用于连接终端设备。
- `nv set interface swp1-32 link state up`
  - **作用：** 这条命令将接口 `swp1-32` 的链接状态设置为 “up”，即启用该接口。这是为了确保接口处于活动状态并能够进行数据传输。
- `nv set system config auto-save enable on`

- **作用：** 这条命令启用自动保存系统配置的功能。这意味着在进行配置更改后，系统会自动保存配置，以确保重启或故障后配置不会丢失。

- `nv config apply -y`

- **作用：** 这条命令应用之前设置的配置更改。`-y` 参数通常用于在命令行中自动确认操作，而无需用户交互确认，确保配置能够立即生效。

现在可以开始为各个节点分配 ip。该步骤在各机器的操作系统上进行配置，这里不多赘述。

在交换机上运行下面的指令查看配置文件，若机器型号相同，则配置文件应该与[配置文件](#)中所给出的配置文件完全一致：

```
sudo nano /etc/network/interfaces
```

## 配置 RoCE

在配置端口后，我们需要配置网络接口的 QoS（服务质量）设置：启用 RoCE 的 QoS 配置，并将 RoCE 模式设置为无损模式。使其在传输过程中不会产生丢包，适用于对延迟和数据完整性要求较高的场景。接着，为默认的有损流量池分配 10% 的内存。这些流量在拥塞时可能会被丢弃。最后为 RoCE 无损流量池分配 90% 的内存，些流量在传输过程中不会丢包，能够确保数据的完整性和低延迟。运行如下的代码配置 RoCE：

### RoCE

```
nv set qos roce enable on
nv set qos roce mode lossless
nv set qos traffic-pool default-lossy memory-percent 10
nv set qos traffic-pool roce-lossless memory-percent 90
nv config apply -y
```

## 5.2 配置两个远程内存节点

### 5.2.1 配置各端口

我们将交换机端口 swp5 连接计算节点，swp1 和 swp3 连接内存节点。并为各个节点如4.1节所述为 RDMA 网卡配置 ip，配置后的设备 ip 如图所示

配置完后在交换机用 `ethtool` 指令查看端口状态，能够显示端口的带宽等信息，证明连接正常，此时可以开始后续开发工作。

```

cumulus@cumulus:mgmt:~$ sudo ethtool swp1
Settings for swp1:
    Supported ports: [ FIBRE ]
    Supported link modes:   1000baseT/Full
                           10000baseT/Full
                           40000baseCR4/Full
                           40000baseSR4/Full
                           40000baseLR4/Full
                           56000baseKR4/Full
                           25000baseCR/Full
                           50000baseCR2/Full
                           100000baseSR4/Full
                           100000baseCR4/Full
                           100000baseLR4_ER4/Full
                           50000baseSR2/Full
                           10000baseCR/Full
                           10000baseSR/Full
                           10000baseLR/Full
                           10000baseLRM/Full
                           10000baseER/Full
    Supported pause frame use: Symmetric
    Supports auto-negotiation: Yes
    Supported FEC modes: None      RS      BASER
    Advertised link modes:  1000baseT/Full
                           10000baseT/Full
                           40000baseCR4/Full
                           40000baseSR4/Full
                           40000baseLR4/Full
                           56000baseKR4/Full
                           25000baseCR/Full
                           50000baseCR2/Full
                           100000baseSR4/Full
                           100000baseCR4/Full
                           100000baseLR4_ER4/Full
                           50000baseSR2/Full
                           10000baseCR/Full
                           10000baseSR/Full
                           10000baseLR/Full
                           10000baseLRM/Full
                           10000baseER/Full
    Advertised pause frame use: Symmetric
    Advertised auto-negotiation: Yes
    Advertised FEC modes: Not reported
    Speed: 100000Mb/s
    Duplex: Full
    Auto-negotiation: on
    Port: None
    PHYAD: 0
    Transceiver: internal
    Link detected: yes

```

图 5.42: 查看端口状态

### 5.2.2 RDMA BACKEND 的修改与优化

在先前的实现中，我们使用一个全局控制器 (gctrl) 来管理计算节点与一个远程内存的连接，并在其中记录连接过程中的重要内容。为了管理多个远程内存节点的连接，我们将全局控制器添加了相关信息，并拓展为链表，链表的节点结构如下所示：

全局控制链表

```

struct gctrl_entry {
    struct sswap_rdma_ctrl *gctrl;
    struct list_head list;
    int serverport;
    char serverip[INET_ADDRSTRLEN];
    char clientip[INET_ADDRSTRLEN];
};

```

扩展的思路很简单，只需要为每个链表元素记录下连接端与本地端的相关信息即可。实现该链表使用 Linux 内核为我们准备好的 list 结构体，我们为每一个链表节点都添加 list\_head 元素，并设置好链表



头，接着就能很方便使用内核提供的对链表的操作函数来操作链表。接着仅需要将我们原先的初始化函数进行嵌套，转化为对多个节点的初始化即可。从结构上说，我们并没有让不同的 CPU 分别负责交换到不同远程节点的操作，而是让每个 CPU 都有对任意节点进行交换的能力，即为每个 CPU 核心分配了双倍的队列资源，使其能够更灵活高效地进行分配任务，实现如下所示：

#### 生成多个全局控制器

```
static int sswap_rdma_create_gctrl_list(int num)
{
    int i;
    struct gctrl_entry *entry;
    int ret;

    for (i = 0; i < num; ++i) {
        entry = kzalloc(sizeof(struct gctrl_entry), GFP_KERNEL);
        if (!entry) {
            pr_err("no memory for entry\n");
            return -ENOMEM;
        }

        //bind port and ip
        entry->serverport = serverport[i];
        strncpy(entry->clientip, clientip, sizeof(entry->clientip));
        strncpy(entry->serverip, serverip[i], sizeof(entry->serverip));

        ret = sswap_rdma_create_ctrl(&entry->gctrl);
        if (ret) {
            pr_err("could not create ctrl, number %d\n", i);
            return -ENODEV;
        }

        list_add(&entry->list, &gctrl_list);
    }

    return 0;
}
```

初始化阶段需要进行的修改比较简单，但在资源释放方面的过程中有一些问题需要注意：在单节点连接建立过程中，若在创建过程中资源分配失败，我们使用的处理方式是在对应的资源分配函数中使用 goto 语句直接跳转释放该资源，所以在初始化函数执行到最后只需要释放最初创建的 ib\_client 即可。而对于多节点的远程内存，若第一个远程创建了相关队列资源并连接成功，而后续连接的节点出现了意外，那么按照最初的资源释放逻辑，将会直接释放后续节点的队列资源并且直接注销 ib\_client。在注销以后，第一个远程节点所占用的队列资源并没有被释放，但是对两个远程节点的连接都已经关闭，于是出现了资源泄露问题（队列资源在连接中是宝贵的）。我们给出的一种简单的解决方案为：在每一次退出函数是都直接遍历整个全局控制链表，检测其资源释放情况后在注销 ib\_client。

在模块的卸载函数里，同样进行资源释放函数的嵌套即可，注意在嵌套释放后要释放我们为此手动申请的链表资源，否则也会造成内存泄露。

## 修改后的模块退出函数

```
static void __exit sswap_rdma_cleanup_module(void)
{
    //sswap_rdma_stopandfree_queues(gctrl);
    struct gctrl_entry *entry, *tmp;
    list_for_each_entry(entry, &gctrl_list, list) {
        sswap_rdma_stopandfree_queues(entry->gctrl);
    }
    ib_unregister_client(&sswap_rdma_ib_client);
    //kfree(gctrl);
    list_for_each_entry_safe(entry, tmp, &gctrl_list, list) {
        list_del(&entry->list);
        kfree(entry->gctrl);
        kfree(entry);
    }
    kfree(gctrl);
    if (req_cache) {
        kmem_cache_destroy(req_cache);
    }
}
```

除此之外，还需进行读写接口的修改。我们简单地读写接口的实现增加了 `int` 型参数，用于指定交换到的远程节点，并在获取队列时获取和该节点进行连接的相应队列。也就是说该读写接口只负责交换页面到指定偏移量 (`offset`) 和设备 (`dev`) 的远程内存中。至于确定交换的位置以及设备等工作，交给上层封装的函数，也就是 `muliswap` 模块来决定。

### 5.2.3 尝试不同粒度的页面映射方式

在封装简化后，我们所有调度到远程节点的剩余工作都可在 `multiswap` 模块中实现。目前思路是实现一个调度器 (`scheduler`) 结构，将记录传入的每一页的 `pgoffset`，并根据不同远程节点的带宽以及使用负载情况进行动态调度：在执行 `write` 操作时记录每一页传入 `pgoffset` 到远程节点 `memory offset` 的映射关系并将页调度，在执行 `read` 操作时查找该映射关系表，读取到页面之后删除该关系表，并标记远程节点的对应位置为可用状态。这么做虽然会增长一部分的内存开销，但是能最精确地对每一个物理页进行操作。在此节，我们只实现了一个简化版本的调度器进行一部分基准测试。

首先，对于两个具有相同带宽的远程节点，在两者不会出现带宽上的性能差异时（进行换页的过程占不满带宽），我们尝试探讨其不同页面分配方式的差异。

此时，我们实现的调度器是根据对 `pageoffset` 的范围所进行的不同粒度分割，如图5.43所示。在4.5.2节中我们提到，对于传入的页其对应的 `pageoffset` 有大小上的限制，于是想到两种朴素的划分方式：将所有的页按照 `pageoffset` 进行奇偶分割：我们称其为**细粒度分割**，或是确定了页面偏移的最大范围后按照此范围的前后分割：我们称其为**粗粒度分割**。在单节点的实现中，我们直接将此 `pageoffset` 转换为地址偏移量，并以此偏移量作为在远程内存中的位置。为了不在远端内存中产生大量碎片，在建立双节点远程内存时，我们还需要进行 `offset` 映射后整理成连续区域传入远程内存。对于粗粒度分割，做法为除以远程内存节点的数量数后取整，将此结果作为远程内存中的地址偏移；对于细粒度分割，做法为对可以分配到这个节点的总页数取模，即若有两个 4G 的远程节点，最多 8G 的地址偏移，则前 4G 分配到分配到远程内存 1 中，位于后 4G 范围的页将其偏移地址除以 4G 后的余数作为其在远程

内存中的 offset。

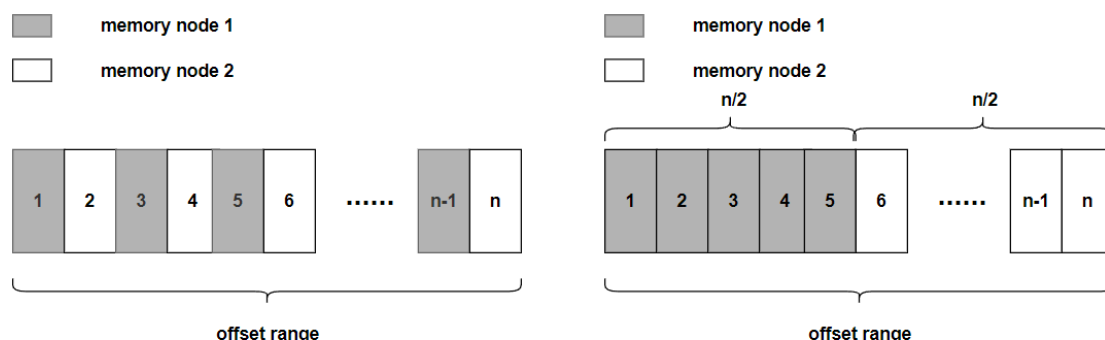


图 5.43: 划分方式

在后续的进展中，我们还对此页面分配是否均匀产生了疑问，并在实验中证实了：对于稍微有一定工作量的测试集而言，总体上这两种分配方式分配到远程节点的页面是均匀的，且两者的性能相近，请参见8.5节。

## 6 阶段三：构造异构内存节点

### 6.1 使用交换机限制带宽

#### 单远程节点带宽限制

连接单个远程内存节点后，我们尝试使用交换机实时监测每个网络接口的带宽（swp1 连接远程内存节点，swp5 连接计算节点），在交换机的 Cumulus Linux 系统下使用输入如下命令，启动 bwm-ng 带宽监测工具并设置刷新间隔为 2s：

```
bwm-ng -t 2000 -A 10 -u bits -I swp1,swp3,swp5 -d
```

并在计算节点运行满线程的负载<sup>5</sup>，得到带宽统计如图6.44所示：

```
bwm-ng v0.6.3 (probing every 2.000s), press 'h' for help
input: /proc/net/dev type: rate
/
```

iface	Rx	Tx	Total
swp1:	14.50 Gb/s	6.70 Gb/s	21.20 Gb/s
swp3:	0.00 b/s	255.87 b/s	255.87 b/s
swp5:	6.70 Gb/s	14.50 Gb/s	21.21 Gb/s
total:	21.20 Gb/s	21.21 Gb/s	42.41 Gb/s

图 6.44: 带宽监控

从带宽监控图中我们发现，swp1 接收到的流量 Rx(从远程内存中接收) 可以达到约 15Gb/s，转发的流量 Tx(从计算节点转发) 可以达到 7Gb/s。这里接收流量代表着从内存节点传送回计算节点的

<sup>5</sup>这里使用的是后续测试中采用8.2节的 stream 测试集，满线程运行以达到最大的带宽占用，并限制其使用 50% 的本地内存加上 50% 的远程内存

页面 (read 操作), 转发流量代表着从计算节点发送到内存节点的页面 (write 操作)。write 操作一般只将特定的页换出内存, 而在处理 page fault 时, Linux 带有预取功能, 进行 read 操作时往往会接着预取连续的页面, 所以对于此程序而言, 造成了 read 操作产生的流量大于 write 操作的局面。我们使用 perf 工具生成的火焰图如下图6.45所示<sup>6</sup>, 紫色部分分别为 store 和 load 操作所占用的时间比例, 显然 load 操作的时间占比要久一些, 也验证了我们的说法。

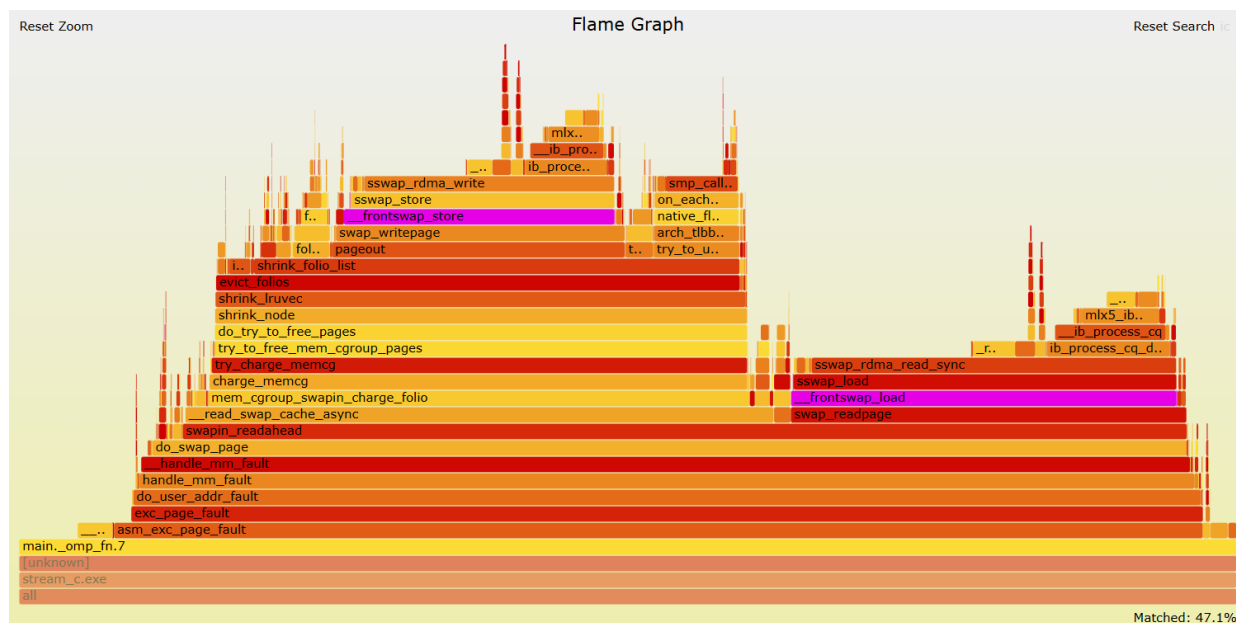


图 6.45: FlameGraph

后续我们将根据获得到的带宽数据进行带宽限制。

对于限制带宽的操作, 我们参考了 nvidia 提供的 Cumulus Linux 文档 [7], 尝试在 Cumulus Linux 上配置一个访问控制列表 (ACL), 并将其应用到与内存节点连接的两个接口中。值得一提的是, 在使用 acl 时, 需要分别设置入站流量 (inbound) 和出站流量 (outbound), 并且该流量控制是以 packet (以太网数据包) 为单位设置的。于是在此之前, 我们需要查看对应接口的 MTU 大小:

```
cumulus@cumulus:mgmt:~$ ifconfig swp1
swp1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 9216
    ether 04:3f:72:79:d4:fc txqueuelen 1000 (Ethernet)
    RX packets 21779957515 bytes 20623111508499 (18.7 TiB)
    RX errors 0 dropped 19 overruns 0 frame 0
    TX packets 16723130404 bytes 13478119067545 (12.2 TiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

cumulus@cumulus:mgmt:~$
```

图 6.46: 确认 swp1 的 MTU 大小

如图6.46所示, 其 MTU 大小为 9216 字节, 即 packet 的最大大小为 9216 字节。根据图6.44, 接收流量速率小于 2GB/s, 发送流量小于 1GB/s, 于是在理论上, 只满足需要:

$$packetnumber = avgflow/MTU \quad (1)$$

<sup>6</sup>生成的具体方式参见8.4.1节

即可基本满足带宽需求，不会发生阻塞。由此我们计算出 swp1 的每秒接收数据包为 233016 个，每秒转发的数据包为 116508 个，并初次尝试以此为带宽限制进行应用。在交换机上编写并运行如下脚本：

```
nv set acl example1 type ipv4

nv set acl example1 rule 10 action police
nv set acl example1 rule 10 action police mode packet
nv set acl example1 rule 10 action police burst 233016
nv set acl example1 rule 10 action police rate 233016

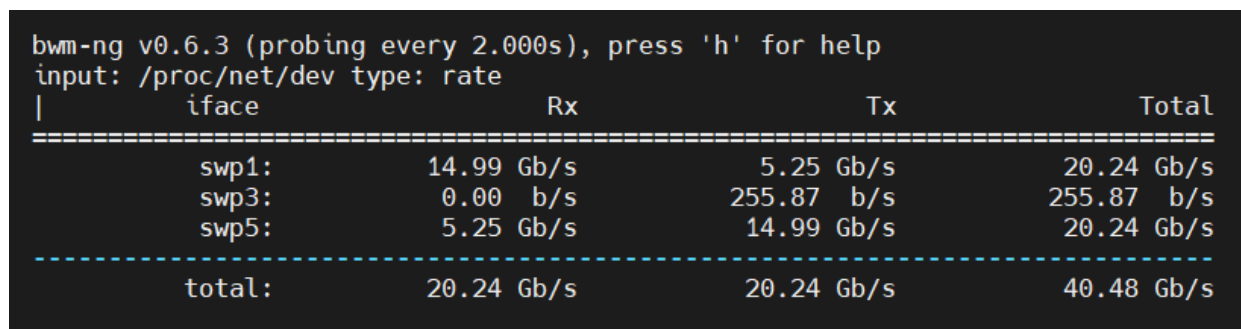
nv set acl example2 type ipv4
nv set acl example2 rule 10 action police
nv set acl example2 rule 10 action police mode packet
nv set acl example2 rule 10 action police burst 116508
nv set acl example2 rule 10 action police rate 116508

nv set interface swp1 acl example1 inbound
nv set interface swp1 acl example2 outbound

nv config apply
```

大致内容为创建两个 ACL，其中的流量控制分别设置为先前计算好的平均每秒接收和转发的数据包个数，最后绑定在 swp1 的接收和转发端，最后将其应用。

应用该设置后，我们大失所望，因为此时监测到的流量只有 1Gb/s 左右，远远达不到我们的预期（预期和不限带宽时的流量相差不大）。我们对此进行分析，或许是数据包大小出现问题（每个数据包比预估要小，导致限制其带宽过小），于是使用传输过程的（峰值流量）/（数据包最大个数）计算出**真正的可能数据包大小在 850-1000 字节附近**，与 MTU 所示的 9216 字节相差甚远，今后就以 925 字节作为传输过程中数据包的平均大小进行限制带宽。在重新计算后，我们得出，若不限带宽，每秒接收数据包个数为 2321603 个，转发的数据包个数为 1160801 个。在重新设置后，监测到的流量如图 6.47 所示，满足了我们的要求，后续将进行多远程节点的开发工作，为每个远程节点限制不同的带宽来手动构造异构节点。



iface	Rx	Tx	Total
swp1:	14.99 Gb/s	5.25 Gb/s	20.24 Gb/s
swp3:	0.00 b/s	255.87 b/s	255.87 b/s
swp5:	5.25 Gb/s	14.99 Gb/s	20.24 Gb/s
total:	20.24 Gb/s	20.24 Gb/s	40.48 Gb/s

图 6.47: 带宽限制后的监测流量

### 搭建混合远程内存池系统

首先，在未限制带宽时，运行同样的负载程序并使用粗粒度划分，在交换机系统上对两个远程内存节点的带宽进行监测，如图所示：



```
bwm-ng v0.6.3 (probing every 2.000s), press 'h' for help
input: /proc/net/dev type: rate
```

\	iface	Rx	Tx	Total
=====	=====	=====	=====	=====
	swp1:	7.21 Gb/s	3.61 Gb/s	10.82 Gb/s
	swp3:	7.17 Gb/s	3.80 Gb/s	10.97 Gb/s
	swp5:	7.41 Gb/s	14.39 Gb/s	21.80 Gb/s
-----	-----	-----	-----	-----
	total:	21.80 Gb/s	21.79 Gb/s	43.59 Gb/s

图 6.48: 对两个远程节点进行带宽监测

可以看到，粗粒度进行页面置换时两个内存节点的转发与接收带宽几乎相同，且均为单节点时的一半，说明此页面置换算法能够较为均匀地与两个远程节点进行页面交换，后续的测试章节还证明了此情况下交换到两个内存节点的数量是相当的。

紧接着，我们对 swp1 与 swp3 这两个连接到不同远程内存节点的 interface 进行内存限制。限制 swp3 的带宽为均匀分配页面的情况下所占带宽的 2/3，swp1 的带宽为 swp3 两倍后再次运行负载程序，进行带宽监视，如图所示，发现了意想不到的结果，如图6.49所示：

```
bwm-ng v0.6.3 (probing every 2.000s), press 'h' for help
input: /proc/net/dev type: rate
```

/	iface	Rx	Tx	Total
=====	=====	=====	=====	=====
	swp1:	5.67 Gb/s	1.85 Gb/s	7.51 Gb/s
	swp3:	5.54 Gb/s	2.08 Gb/s	7.62 Gb/s
	swp5:	3.95 Gb/s	11.21 Gb/s	15.16 Gb/s
-----	-----	-----	-----	-----
	total:	15.16 Gb/s	15.14 Gb/s	30.30 Gb/s

图 6.49: 监测限制带宽后的节点流量

我们仅仅限制了 swp3 的带宽为其需求的 2/3，但是发现 swp1 节点在程序执行过程中的流量也降低到了和 swp3 节点相同！也就是说，在这种情况下，造成了类似木桶效应的结果：**慢节点的带宽成为此时的性能瓶颈**。从结果上看，程序的运行时间与不限制带宽相比变慢了 28% 左右。

简单分析一下此情况出现的原因：我们实现的换页机制中，采用的为同步更新操作，即只有在换页请求发送后，CPU 检测到了硬件传递到完成队列中的完成事件，函数才进行返回。下面结合图示6.50说明。假设对于计算节点的每一个 CPU 来说，往两个内存节点之间的页面交换是近似交替的，即在处理交换到高带宽节点的页后，处理交换到低带宽节点的页，以此反复。我们用宽度来表示执行此换页操作所需的时间。最初，两个内存节点的带宽都未被占满，因此交换过程的时间相当；随着交换需求的增加，低带宽节点的带宽被占满，所需完成任务的时间增加，而与两个节点间的交换任务又是交替进行的，因此在高带宽节点处理事件完毕后，会向占满带宽的低带宽节点接着发送任务，待其处理完后再向高带宽节点发送下一次任务，期间的间隔事件我们记录为 timespend1，而低带宽节点两次任务的间隔事件为 timespend2，显然此时这两段时间间隔相等，并同时大于最初未占满带宽时的时间间隔。timespend1 与 timespend2 相同即说明了一段时间内往两个节点分配的任务个数相同，且任务完成后才可分配下一个任务，因此两节点的流经流量相等。

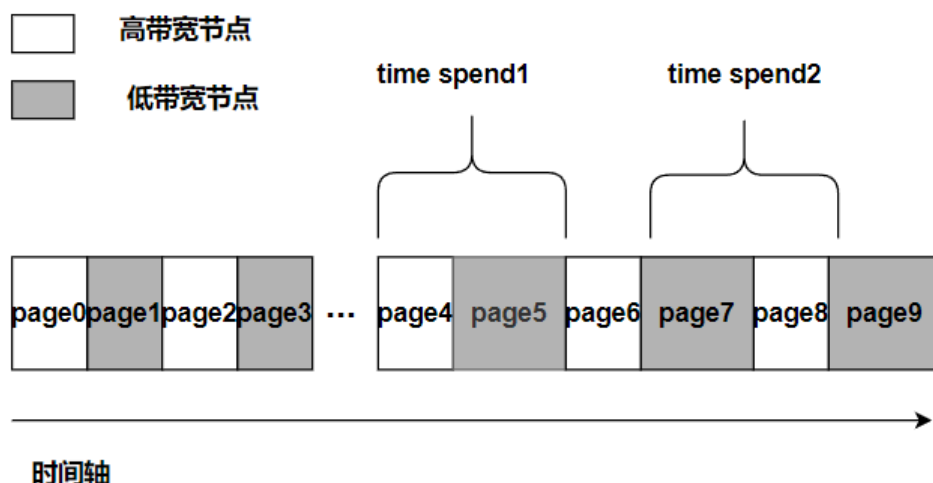


图 6.50: 交替换页

## 6.2 尝试进行页面调度以优化系统性能

我们从上一小节的内容中受到启发，在进行 write 操作低带宽节点的带宽趋于饱和时，尝试调度这些本该交给低带宽节点操作的一部分页到高带宽节点中，等其工作压力减小后再向低带宽节点中调度。由于换出到低带宽节点的页面减少，同时也减小从其中 read 的压力。

为实现此功能，需要实现在5.2.3节中所述的调度器，我们相信，合理地对两个内存节点进行页面分配能够解决这个问题。页面调度机制主要聚焦于以下问题：内核提供了一个页以及一个页对应的 pageoffset，需要将该页换入到哪一个远程内存中，换入到该远程内存的哪个位置中？

在本节中，我们考虑到内核换页操作的延迟敏感性，并不考虑过于复杂的页面调度机制。在 multiswap 中，我们为 read 和 write 函数保留了接口，在进行读写时只需传入对应的远程内存编号以及换入远程内存的偏移地址，即可将该页送入相应的队列中进行处理。

### 封装后的读写函数

```
int sswap_rdma_read_sync(struct page *page, u64 roffset, unsigned int dev);
int sswap_rdma_write(struct page *page, u64 roffset, unsigned int dev);
```

对于换入远端的偏移地址，我们暂时不做修改，仅考虑如何调度以缓解慢节点的带宽占用问题。我们尝试的思路很简单，对 frontswap 接口传入的 pageid 进行分割，分别以 n:1 的形式根据两个节点的带宽差距分别分配给快慢两个节点。例如两个节点的带宽比为 2:1，则所有的页以 2:1 的比例分配到两个节点当中。我们证明了对于有一定负载的程序来说，依据 pageid 为划分的页面分配是均匀的（参见8.5节），所以我们只需要根据 pageid 简单分割尝试即可，实现一个简单的调度器：

### scheduler

```
#define RATING 3
static int schedule_node(pgoff_t pageid)
{
    if(pageid % RATING == 0)
        return 0;
    else
```



```

    return 1;
}
sswap_rdma_write(page, pageid << PAGE_SHIFT, schedule_node(pageid))
sswap_rdma_read_sync(page, pageid << PAGE_SHIFT, schedule_node(pageid))

```

使用同样的程序与带宽限制，我们再次运行程序测试，带宽检测如下图6.51:

```

bwm-ng v0.6.3 (probing every 2.000s), press 'h' for help
input: /proc/net/dev type: rate
\
=====
      iface      Rx              Tx              Total
=====
      swp1:      9.46 Gb/s      3.31 Gb/s      12.77 Gb/s
      swp3:      4.73 Gb/s      1.66 Gb/s      6.38 Gb/s
      swp5:      4.97 Gb/s      14.19 Gb/s     19.15 Gb/s
-----
      total:     19.16 Gb/s     19.16 Gb/s     38.33 Gb/s

```

图 6.51: 页面调度优化后进行带宽检测

将图6.51对比图6.48发现，经过调度后，每秒统计的总流量已经十分接近，我们接着统计了三种不同情况下测试程序如下表3:

	每秒平均流量和 (Gb/s)	程序运行时长
未限制带宽	43.59	3:04.5
限制带宽	30.30	4:25.21
限制带宽后进行调度优化	38.33	3:23.87

表 3: 优化前后测试对比

在进行优化后，无论是流量分配还是在运行时长上，都有了明显的提升，在我们所关注的系统每秒平均流量和方面，我们调度优化后的方法相比于未进行优化性能提升了 25%，初步验证了此方法的可行性，在后续的项目中，我们还会进一步使用更多测试集做更为详细地测试，以验证该方法能切实部署在真实系统上的可能性。

### 6.3 缺点及改进方法

很明显，如此朴实的方法对于性能有很大的提升，但是距离未限制带宽的性能还有一定的距离，并且需要人为地根据已知带宽来手动设定分配策略，我们希望在后续能够构建自适应的页面分配策略，以更灵活、高效地方式提高系统整体的性能。

### 6.4 问题解决 (决赛第二阶段新工作)

我们在决赛的第二阶段尝试构建自适应的页面分配策略。需要完成这个任务，有两个关键的方面需要考虑:

- 在某一时刻，如何选择一个特定的远程内存节点进行传输?
- 此时，我们换出的页应该放在该远程内存节点所提供的内存池的哪个位置?

第二个问题是第一个的基础，只要能够实时获取到换出页的位置，剩下的工作只需要考虑调度到哪一个节点即可。我们尝试解决了上述的第二个问题。在先前的实现中，我们简单地将传入的 `pageid`(原

本要换入的 swap 交换区中的页偏移量) 通过移位运算转换为 offset, 并将此 offset 作为相对于远程内存区域起始地址的偏移。在进行异构内存的调度时, 也只是简单地将 pageid 做了取模运算, 映射到不同的远程内存节点中, 也就是说, 是一种静态的映射方式。若我们想要实现实时的调度, 则不能使用静态的映射方式 (除非两个节点都准备了多余的内存来保证 pageid 移位后的 offset 在总内存范围之内, 但这样无疑会有大量内存碎片与空闲内存)。所以我们需要实现动态的映射: 每个时刻传入的 pageid 都能映射到任意远程内存的对应位置, 我们需要手动分配并记录这个映射关系, 并使得远程内存尽量紧凑<sup>7</sup>。总之, 我们的任务是实时寻找远程内存的一个空闲位置, 用于页换出; 在页换入时能够找到该位置, 实现页换入, 同时远程内存尽量紧凑。

为了实现此功能, 我们新构建了 remote\_schedule 模块。该模块主要解决了在多线程资源竞争下的下列问题:

1. 如何记录 pageid 对应哪一个 memory node, 该存储到该 memory node 的哪个位置 (slot)?
2. 对于每一个 memory node, 如何存储并获取到其可用的空闲 slot? 选择哪一个 slot 来保存要换出的页呢?
3. 映射关系什么时候能够释放, memory node 的 slot 什么时候能够恢复可用呢?

接下来, 我们将结合开发的过程, 依次回答这些问题。

#### 6.4.1 构建 pageid 到远程内存的映射

此操作存在我们换页的关键路径上, 为了尽可能快地完成该操作, 我们参考了内核中 swap\_address\_space 的实现, 利用其中的成员 xarray 结构体来保存映射。xarray[5] 是一种高效的键值对数据结构, 基于 radix tree 实现, 用于管理大规模的数据集。其查找速度仅和存储在其中的元素个数有关, 且内核为我们封装了接口, 直接使用不会产生数据一致性问题。为了加速这个过程, 我们仅让每一个 xarray 最多存储  $2^{14}$  项, 也就是对应 64M 的页。也就是说, 我们需要事先构建一个 address\_space 的数组来存储所有 pageid 的映射, 初始化过程如下所示:

remote\_address\_space 的实现

```
struct address_space *remote_spaces __read_mostly;
static unsigned int nr_remote_spaces __read_mostly;

static int init_remote_address_space(unsigned long nr_pages)
{
    struct address_space *spaces, *space;
    unsigned int i, nr;
    nr = DIV_ROUND_UP(nr_pages, REMOTE_ADDRESS_SPACE_PAGES);
    spaces = kvcalloc(nr, sizeof(struct address_space), GFP_KERNEL);
    if(!spaces)
        return -ENOMEM;
    for(i = 0; i < nr; i++) {
        space = spaces + i;
        //init xarray
        xa_init_flags(&space->i_pages, XA_FLAGS_LOCK_IRQ);
        atomic_set(&space->i_mmap_writable, 0);
    }
}
```

<sup>7</sup>这里指的是远程内存的总和与 pageid 的范围相等, 例如 swapfile 的大小为 8G, 则 pageid 可能的范围为 (0-8G), 我们希望远程内存的大小总和也为 8G, 这样不会产生冗余内存。

```

space->a_ops = &remote_aops;
mapping_set_no_writeback_tags(space);
}

nr_remote_spaces = nr;
remote_spaces = spaces;

return 0;
}

```

在 k-v 键值对中，我们在 value 中需要保存换入到远程内存的 id 以及换入到该远程内存中的位置。我们使用一个 64 位的 unsigned long 型变量来保存这些内容。低四位用于保存换入到的远程内存 id，也就是说最多支持 16 个内存节点，剩余位用来表示换入到远程内存中 slot 的页偏移量。

内核中已经为我们准备好了 xarray 的有关操作函数，所以我们可以很轻松地完成映射的构建。值得一提的是，xarray 中键值对的 value 可以存储指针和数据，并在保存 (xa\_store) 时会进行类型检查。当保存的是数据时，需要调用 xa\_make\_value 函数，该函数将 value 左移一位并将最低位置 1。在检查时检测最后一位是否为 1 来确定 value 是否合法。进行内容的读取 (xa\_load) 时，则会调用 xa\_to\_value 将内容右移恢复。所以我们原先数值的最高位中不能保存有效数据，也就是说总共只有 59 位来保存远端的 slot 偏移，其结构示意如图 6.52 所示。由于使用的是页偏移量，所以可保存的内容远远超过现实中的内存大小，因此理论上不会产生错误。至此，我们的映射可以顺利的构建完成。

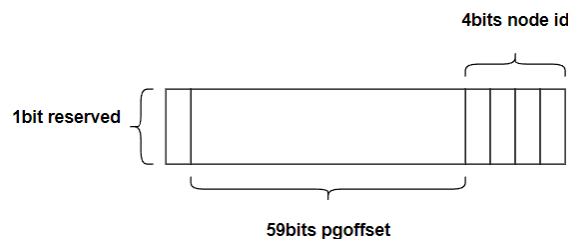


图 6.52: xarray entry

#### 6.4.2 获取内存节点的可用 slot

我们采用位图 (bitmap) 来构建每一个远程节点的远程内存页所占用的情况，若正在使用中，则将对对应位置为 1；若空闲，则置为 0。在实际的实现过程中，我们首先为每一个内存节点准备了一个 unsigned long 类型的数组，将其作为存储用的 bitmap，并始终寻找第一个空闲的 bit 作为插入的位置，其示意图如下图 6.53 左侧所示。但是在后续的测试过程中，使用 perf 分析发现，光是寻找第一个空闲的 bit 这个位查找过程，就花费了整个程序运行时长的 10%，这是由于 bitmap 数组过长导致的，也是我们所不允许容忍的，于是考虑对此进行优化。我们不单只保存 bitmap，而是将其封装成如下所示的结构体：

remote\_bitmap

```

struct remote_bitmap {
    unsigned long *mapping;
    spinlock_t w_lock;
    unsigned long nr_pages;
    unsigned long free_pages;
}

```

```
};
```

我们采用分段的思想，将一个远程内存节点中的内存切分成以页为单位，由最初的一个 bitmap 映射改为由若干个 remote\_bitmap 结构体所构成的数组来表示，如图6.53所示。每一个 remote\_bitmap 结构体包含四个成员，分别是用于作为 bitmap 存储的 unsigned long 数组，用于保证一致性的自旋锁，该结构体所对应的总页数以及当前剩余的空闲页数。我们允许每一个结构体最多记录  $2^{14}$  个页，也就是说最多包含 256 个 unsigned long 的变量。在远程页被读取或是映射释放时，将 free\_pages 做相应的增减操作，这个操作过程需要持有锁。当 free\_pages 减小为 0 时，则说明此 bitmap 无对应的页可用，将跳转到下一个 bitmap 中寻找，而判断一个 remote\_bitmap 结构体是否有空闲页的操作即为读取 free\_pages 变量是否为 0，这样省去了大量的位查找操作，极大地加快了查找时间。最后返回时，只需要将查找过程中跳过的 remote\_bitmap 的 nr\_pages 相加并加上当前 bitmap 中的偏移，即可得到该页距离远程内存起始位置的偏移。需要注意的是，在此过程中，任何对于 remote\_bitmap 中可能改变的成员进行读或是写操作时，都需要持有锁。

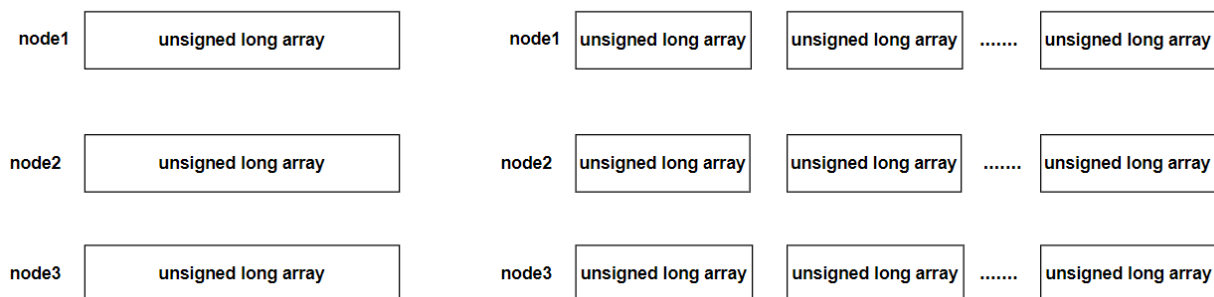


图 6.53: bitmap 前后对比

## 分段 bitmap

```
static struct remote_bitmap* remote_map[MAX_REMOTE_NODES];
static unsigned long nr_remote_maps[MAX_REMOTE_NODES];
static unsigned long sswap_select_slot(unsigned int type, struct remote_mapping_flags
    *flags)
{
    unsigned long nr_pages, slot;
    struct remote_bitmap *map, *temp;
    unsigned int i, offset;

    temp = remote_map[type];
    nr_pages = temp->nr_pages;
    offset = 0;
    for(i = 0; i < nr_remote_maps[type]; i++)
    {
        map = temp + i;

        spin_lock_irq(&map->w_lock);
        if(map->free_pages != 0)
        {
            slot = find_first_zero_bit(map->mapping, nr_pages);
```

```

        flags->bits.mapping_success = 1;
        //set bit, update bitmap
        set_bit(slot, map->mapping);
        map->free_pages--;
        spin_unlock_irq(&map->w_lock);
        return slot + offset;
    }
    spin_unlock_irq(&map->w_lock);

    offset += nr_pages;
}

pr_info("all remote bitmap is full\n");
flags->bits.mapping_full = 1;
return 0;
}

```

### 6.4.3 更新映射时机的选择

何时可以删除 xarray 与 bitmap 中的映射？这是个很值得考究的问题，不过对于 xarray 中所对应的 pageid 与 remote slot 之间的映射，不需要手动更新，因为 xa\_store 操作会自动覆盖先前的内容，所以只需要选定 remote node 与 slot，直接覆写上去即可，且 xarray 的接口有锁保护，我们不需要关心此部分内容。我们要关注的问题只有如下：何时能够将 bitmap 中的位置清零来表示这个位置重新可用？首先我们想到的是在触发一次 read 操作后将该映射关系清空，因为 read 操作成功代表已经成功从远程内存中获取到了换出的页，于是映射自然也就没有用了。我们将此想法付诸实现，在单线程程序上测试通过，但是在多线程程序中发生段错误。我们分析许久并没有发现错误原因，于是转换思路，采用懒更新 (lazy update) 机制，在有第二次相同 pageid 的 write 操作发生时，说明内核中此时已经准备将新的内容写到 swapfile 中的 pageid 处了，此时旧的内容一定为内核所抛弃。于是在进行 write 操作时我们可以查看 xarray 映射中是否有 pageid 对应的远程内存 slot，若有，则说明存在旧的映射，我们将 bitmap 中对应位置清零，让其重新恢复可用。后续再对需要换出的新页寻找新的 slot，这样做也不会影响调度手段的实施。综上，在进行写操作前，我们尝试从 xarray 中读取 pageid 是否有对应的 value，若有，则在 bitmap 中删除旧映射后重新选择分配；若没有，则直接尝试分配。这部分实现的代码如下：

#### 懒更新机制

```

u64 sswap_add_mapping(unsigned int type, pgoff_t pageid, struct remote_mapping_flags
    *flags)
{
    struct address_space *address_space;
    unsigned long slot, xa_msg, old_msg, offset;
    unsigned int old_type;
    void *entry;

    address_space = &remote_spaces[pageid >> REMOTE_ADDRESS_SPACE_SHIFT];
}

```

```

//if old mapping exists, we should delete it in bitmap first
if((entry = xa_load(&address_space->i_pages, pageid)) != NULL)
{
    //pr_info("replace old mappings\n");
    old_msg = xa_to_value(entry);
    old_type = REMOTE_TYPE(old_msg);
    slot = REMOTE_SLOT(old_msg);
    offset = slot / REMOTE_ADDRESS_SPACE_PAGES;
    slot = slot % REMOTE_ADDRESS_SPACE_PAGES;

    spin_lock_irq(&remote_map[old_type][offset].w_lock);
    clear_bit(slot, remote_map[old_type][offset].mapping);
    remote_map[old_type][offset].free_pages++;
    spin_unlock_irq(&remote_map[old_type][offset].w_lock);
    //pr_info("replace old mappings complete\n");
}

slot = sswap_select_slot(type, flags);

//no available slot, find another type
if(flags->bits.mapping_full == 1)
{
    pr_info("mapping fulls\n");
    return 0;
}

xa_msg = MAKE_XA_MSG(slot, type);
entry = xa_mk_value(xa_msg);
//xa_load can overwrite the old mapping, so we don't need to delete it
if(xa_err(xa_store(&address_space->i_pages, pageid, entry, GFP_KERNEL)))
{
    pr_err("failed to add mapping\n");
    VM_BUG_ON(1);
}
return slot << PAGE_SHIFT;
}

```

#### 6.4.4 评估与改进

我们安装上 remote shedule 模块并测试多线程程序 (pagewalker7节), 使用 perf 分析得到的火焰图如下图6.54所示:







```

echo $$
if [ ! -d "/sys/fs/cgroup/yuri/" ];then
    mkdir /sys/fs/cgroup/yuri
else
    echo "cgroup yuri already exists"
fi
echo "+memory" >> /sys/fs/cgroup/yuri/cgroup.subtree_control

if [ ! -d "/sys/fs/cgroup/yuri/pagerank_150M/" ];then
    mkdir /sys/fs/cgroup/yuri/pagerank_150M
else
    echo "cgroup yuri/pagerank_150M already exists"
fi

echo 134217728 > /sys/fs/cgroup/yuri/pagerank_150M/memory.max
echo "set memory.max to"
cat /sys/fs/cgroup/yuri/pagerank_150M/memory.max
echo "adding current shell to pagerank_150M"
echo $$ | sudo tee /sys/fs/cgroup/yuri/pagerank_150M/cgroup.procs
gcc pagewalker.c -lm -O0 -o pagewalker
./pagewalker
echo "memory.peak is:"
cat /sys/fs/cgroup/yuri/pagerank_150M/memory.peak

```

## 7.1 DRAM BACKEND 实现正确性测试

执行命令安装模块到内核中并进行测试，进入项目文件夹下：

### 安装 DRAM BACKEND

```

#dev-rdma 文件夹下
make BACKEND=DRAM
sudo insmod multiswap_dram.ko
sudo insmod multiswap.ko
#test 文件夹下
sudo ./memory_limit_test.sh

```

在执行完模块安装后，dmesg 在内核缓冲区中能够看到相关的输出。运行完测试程序后，可以看到每一轮的迭代时间以及 cgroup 中的 memory.peak，达到了我们预期的 128M，如下图7.55所示。打开 DEBUG\_MODE 宏进入内核缓冲区中也能发现有关键路径上的输出，可以说明确实发生了 swap 操作。

## 7.2 RDMA BACKEND 实现正确性测试

### 7.2.1 内存节点操作

在进行实验之前，我们需要保证 server 的队列个数大于或等于 client 端，一组队列之间建立一个连接。我们将访问 server 端的 50000 号端口，在 server 端执行命令：

```
epoch 64
epoch 65
epoch 66
epoch 67
epoch 68
epoch 69
epoch 70
epoch 71
epoch 72
epoch 73
epoch 74
Time measured: 34.195 seconds.
memory.peak is:
134217728
```

图 7.55: 测试程序输出

## 开启 server

```
#farmemserver 文件夹下
make rmserver
./rmserver 50000
```

此时 server 端会出现监听信息，表示等待连接状态。注意，若此时手动 `ctrl+C` 终止程序，由于端口 50000 正在等待连接，再次重新运行程序会报如图 7.56 的错误：

```
xc@xc-HP-280-Pro-G2-MT:~/fastswap/farmemserver$ ./rmserver 50000
error: rdma_bind_addr(listener, (struct sockaddr *)&addr) failed (returned non-zero). - errno: 99
```

图 7.56: server-error

表示 `rdma_bind_addr` 出现错误，因为 50000 号端口在先前已经被绑定并仍然在等待连接，之前的程序并没有完全被杀死。我们查看先前程序的进程 id：

```
ps -aux | grep rmserver
```

搜索结果如图 7.57，表明确实存在该进程，id 为 7923：

```
xc@xc-HP-280-Pro-G2-MT:~/fastswap/farmemserver$ ps -aux | grep rmserver
xc      7923  0.0  0.0  4084  2428 pts/2    T   20:34   0:00 ./rmserver 50000
xc     10381  0.0  0.0  12108   656 pts/2    S+  20:42   0:00 grep --color=auto rmserver
```

图 7.57: 查找 rmserver 进程 id

我们需要手动 kill 该进程：

```
sudo kill -9 7923
```

而后就可以重新使用相同端口监听。

在使用双远程内存时，我们将第二个远程内存节点的监听端口号设置为 50001，在计算节点程序 `multiswap_rdma.c` 中指定了 50000 与 50001 这两个端口号，若有需要可自行修改。

### 7.2.2 计算节点操作

在计算节点项目文件夹下执行命令：

#### 安装 RDMA BACKEND

```
#dev-rdma 文件夹下  
make BACKEND=RDMA  
sudo ./insmod.sh
```

执行成功后，计算节点的内核缓冲区与远程内存进行监听的程序都会输出相关信息，表示连接建立完成。运行测试程序的具体操作与 DRAM BACKEND 一致，同样通过了测试。

## 8 性能测试

### 8.1 硬件配置

对于双方节点，均使用 Nvidia Mellanox ConnectX-5 RoCE 网卡、交换机型号为 SN2700、计算节点 CPU 型号为 Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz，核心数为 6。

### 8.2 benchmark 选择

选取了六种测试程序，其中前三种是常见的实际工作，一种是评估运算性能的基准测试程序，另外两种是内存带宽测试的基准测试程序。因此前四种的访存操作占比少于后两种。

**Kmeans** 随机生成 4000 万个样本点，之后执行 Kmeans 算法将样本数据分成 10 个簇。

**Quicksort** 生成一个指定大小的数组随机生成数组元素，使用 STL 中的 `sort()` 对其进行排序，所以虽然名为 quicksort，实际上并非普通的快速排序。

**Tensorflow** 用 Tensorflow 训练框架训练 CNN，调整 `batch_size` 参数的大小就可以调节使用的内存大小

**Linpack** Linpack 是评估计算机系统浮点性能的基准测试程序，完成的任务是用高斯消元法求解 N 元一次稠密性线性方程组。

**Stream** 内存带宽测试的常用基准测试程序，进行四种操作，分别计算内存带宽。具体来说，COPY 会进行数组元素的复制，Scale 会将数组元素乘一个常数存入另一个数组元素中，Add 将两个数组元素相加赋值给另一个数组元素，Triad 是将一个数组元素与一个常数乘积的结果加上另一个数组元素的结果存入一个数组元素中。在该测试集中，可以指定相应内存操作的线程数量，非常适合用于评测多线程执行换页操作时的 RDMA 网卡带宽占用。

**Mbw** 内存带宽测试的常用基准测试程序，通常用来评估用户层应用程序进行内存拷贝操作所能达到的带宽，可测试在内存拷贝、字符串拷贝、和内存块拷贝三种方式下的内存拷贝速度。

### 8.3 环境搭建

一个 python 环境并不能适用于所有 python 程序，因此除了本机原本的 python 环境，还需要针对不能在本机的 python 环境下运行的测试用例创建虚拟环境来运行。以 Kmeans 和 tensorflow 为例，两个测试用例都是执行.py 文件来运行的

**Kmeans** Kmeans 是在本机的 python3.10 的环境下运行的，安装好对应的包就能运行，由于在测试时都是用.sh 脚本运行的，在输入命令时要在 sudo 后面加-E 的选项，因为 sudo 会重置大部分环境变量，而加上-E 可以保留当前环境变量，使得脚本能够正常运行 Kmeans 代码。

**Tensorflow** 这个测试程序的仓库里说明了需要使用的是 tensorflow 1.14，而这个版本的包在 python3.10 中是已经被淘汰了的，因此为了使用不同的 python 环境，创建一个 venu 的虚拟 python 环境，虚拟环境使用 python3.7，下载好对应的包后，在运行这个程序的脚本中加入激活虚拟环境的命令就能正常运行。

在分析性能瓶颈时还使用了 perf 工具，并绘制了火焰图。由于使用的是自己编译的内核，perf 工具并不能直接通过 apt 直接下载。需要在内核源码目录 tools/perf 下 make install 并把编译完成的 perf 可执行文件拷贝到 usr/bin 目录下，才能成功使用 perf 工具。绘制火焰图也需要安装可视化生成器，该工具的安装地址为 [FlameGraph](#)。

### 8.4 端到端性能测试

#### 8.4.1 测试工具

perf 是一款 Linux 性能分析工具，内置在 Linux 内核的一个 Linux 性能分析框架中，利用 CPU、PMU、和软件计数等进行性能分析。

具体使用，perf record 会采集数据并写入.data 数据文件中，这个文件是后续生成火焰图所使用的文件。

火焰图是 svg 格式的矢量图，是将 perf 得到的结果生成的具有不同层次且支持互动的图片，列出了所有可能导致性能瓶颈的调用栈，从火焰图中可以更清楚地分析性能的瓶颈以及各部分函数占用的时间。perf record 得到了.data 文件后需要运行下面三条命令来生成火焰图。

生成火焰图

```
sudo perf script -i ./perf.data &> ./tmp/perf.unfold
sudo $flamebase/stackcollapse-perf.pl ./tmp/perf.unfold &> ./tmp/perf.folded
sudo $flamebase/flamegraph.pl ./tmp/perf.folded > ./perf.svg
```

先解析 perf.data 并将结果保存在文件中，接着将解析的结果的符号进行折叠，最后生成火焰图。

依据发生 pagefault 时交换到的区域类型 (backend) 分为了 swapfile (磁盘交换区)，dram (使用本地内存作为交换区)，single\_node (交换到作为远程一台机器的内存)，2node\_fine-grained (交换到两个远程内存上并且使用细粒度的调度方式)，2node\_coarse-grained (交换到两个远程内存上并且使用粗粒度的调度方式)，测试时在不同的 backend 上进行，观察性能结果是否和预估一致。

还需要测试在不同的本地内存限制的百分比下，它们的表现的差别。首先测试不同测试程序在不限制内存的情况下的 Maximum resident set size，即使用的最大内存总数 (表4所示)，以这个 size 作

为基准, 乘以 50%-90% 的比例作为限制的内存, 这样至少可以确保在使用内存最多时, 一定能够发生 1 减去该比例的内存的交换。

Benchmark	Memory(GB)
Kmeans	4.6
Quicksort	8.0
Tensorflow	2.1
Linpack	1.5
Stream	4.0
Mbw	5.9

表 4: Benchmark 使用的内存大小

### 8.4.2 cgroup 限制内存

cgroup 是 Linux 内核的一项功能: 在一个系统中运行的层级制进程组, 简单来说, 可以对进程的可用资源进行限制, 也就可以按照设定的限制的内存运行, 一旦超出限制, 就会发生 pagefault。

具体到使用方法, 在脚本中使用 cgroup, 首先需要将脚本的进程 ID 添加到根 cgroup 的进程列表中, 然后将 memory 子系统挂在到 cgroup 上, 接着为了精细化管理和隔离, 创建了一个子 cgroup 并将当前 shell 的进程 ID 添加到这个子 cgroup, 这样这个 shell 进程及其子进程都会受到这个子 cgroup 的资源限制。最后设定这个子 cgroup 中的 memory.max 就可以限制内存。

### 8.4.3 测试脚本使用

由于测试需要进行的次数比较多, 为每一个测试程序都编写了脚本, 运行脚本可以将结果输出到保存的文件中, 并且可以一次测出不同的内存比例下的结果。脚本让每个测试都循环三次是为了能够求平均值减小误差。输出的结果重定向到相应的文件中。测量命令执行时间所使用的工具为 Linux 提供的 time 工具, 可通过 `usr/bin/time -v` 来执行程序, 可以得到程序运行的详细数据, 包括用户态时间、内核态时间、major page fault 总次数等。

得到所有结果后, 我们将执行结果通过脚本保存到文件中, 关注的是 Elapsed time 即总运行时间数据以及 major page fault 数据, 因此编写一个 python 脚本提取所有的 Elapsed time 并将三次结果取平均值。至此便得到了测试方法中所说的所有结果三次的平均值, 也就是最终的结果, 依据该结果就可以进行性能比较。

## 8.5 性能比较

我们按照上面提到的测试方法测试程序运行的总时间, 并将其用时结果求倒数, 以方便比较性能。首先, 我们比较单个远程内存节点与本地 DRAM, 磁盘交换区这三者的性能。在比较开始前, 我们可以有一个基本的认知, 即远程内存的性能不会高于本地 DRAM 的性能, 且不会低于磁盘交换区的性能, 以此确定了它的性能上限和性能下限, 我们在表5列举了单个远程内存节点相比于磁盘交换区所带来的性能提升, 并将详细比较使用图来说明。

Test Program \ Percentage	50%	60%	70%	80%	90%
Kmeans	18.85%	24.13%	14.84%	9.03%	4.74%
Quicksort	11.55%	7.94%	7.06%	6.23%	4.32%
Tensorflow	7.48%	7.44%	4.98%	0.73%	0.24%
Linpack	54.74%	53.58%	50.89%	48.53%	38.65%
Stream	57.29%	59.00%	57.49%	62.59%	65.00%
Mbw	63.95%	69.83%	69.52%	70.85%	67.20%

表 5: 使用单个远程内存相比于 SWAP 磁盘所获取的性能提升

如图8.60所示, 可以看到, 无论是具有实际应用意义的工作测试集, 还是进行内存带宽测试的基准测试集, 我们的远程内存对程序性能都有不同规模的提升。使用单个远程内存节点的表现都介于 dram 和 swapfile 之间, 但对于不同任务来说, 其提升的效果与该任务是否为计算密集型或是内存需求型任务相关。以 quicksort 测试集为例, 使用 perf 绘制 FlameGraph 如下图所示, 触发 swap 并进入 multiswap 模块进行处理的部分为图8.58示紫色位置, 占程序运行时间的比例统计约为 23.2%, 而 stream 测试集触发 swap 并进入 multiswap 模块进行处理的部分为图8.59的紫色部分, 占程序运行时间的比例为 50%, 即相当于我们只对这部分的操作进行了优化, 而程序执行的其它部分与我们的优化无关, 取决于程序自身性质与计算节点的计算能力。

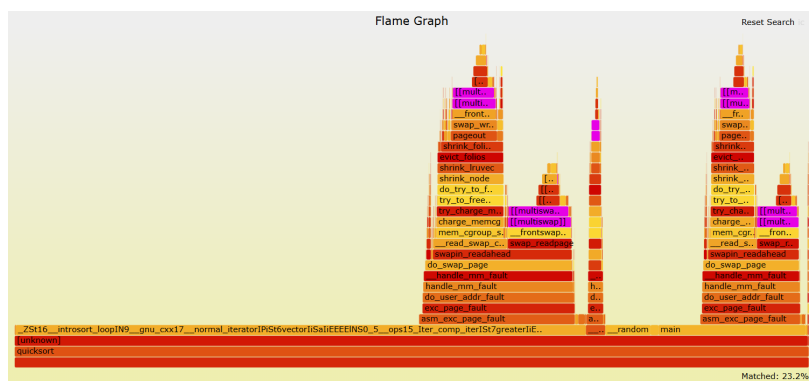


图 8.58: quicksort 测试集函数分析

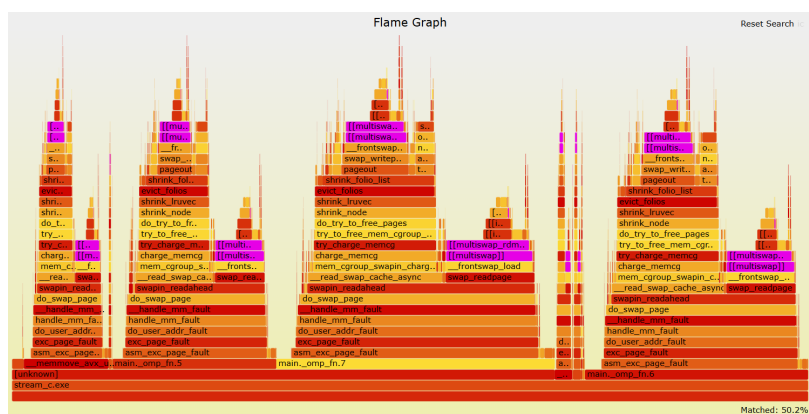
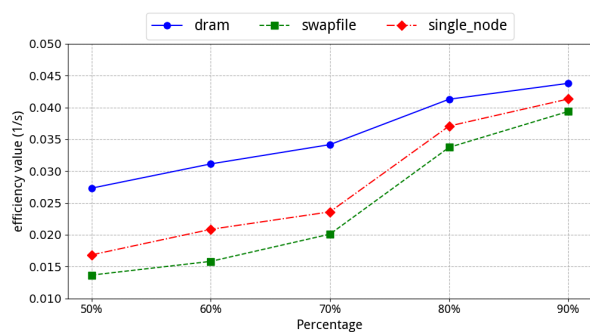
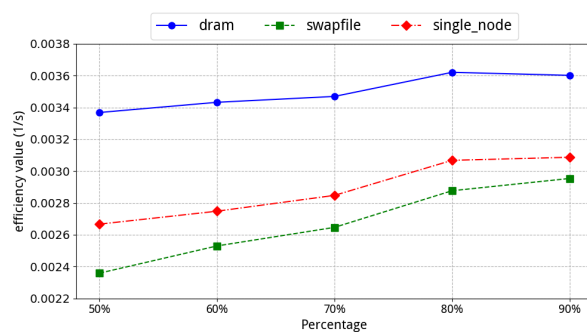


图 8.59: stream 测试集函数分析

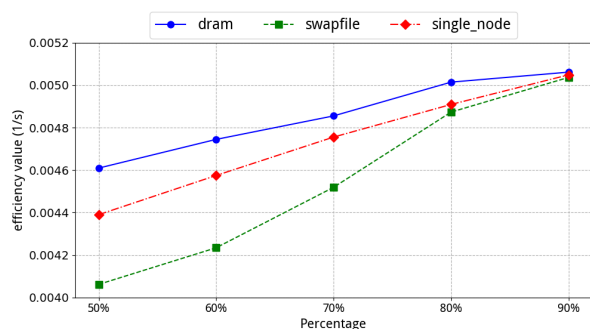
由此可知, 使用远程内存作为发生 pagefault 时的交换区有实际意义, 能够提升性能, 尤其是涉及到大量内存操作占比的程序, 使用该技术能够显著降低运行时间。



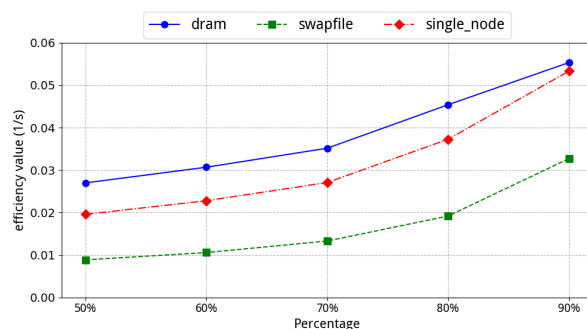
(a) Kmeans



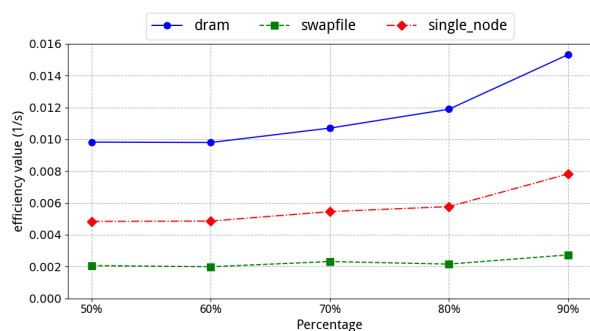
(b) Quicksort



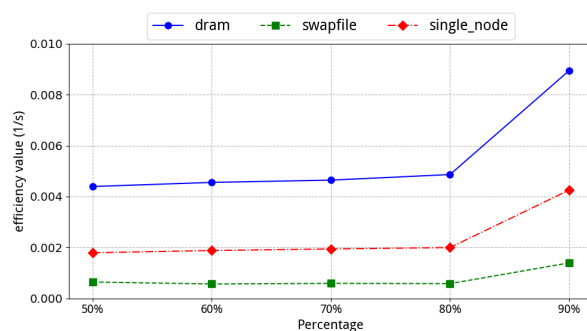
(c) Tensorflow



(d) Linpack



(e) Stream



(f) Mbw

图 8.60: 使用单个远程内存存在六个测试程序下与 dram 和 swapfile 的性能比较

接下来, 要比较使用两个相同带宽的远程内存节点时的性能。使用两个内存节点就涉及到调度方式的问题, 不同粒度会有不同的性能 (关于粗粒度和细粒度的解释参见5.2.3节)。图8.61展示了不同粒度在六个测试程序下的性能表现并与单节点和 swapfile 进行对比, 可以看出, swapfile 依然是性能下限。重点关注图8.61e和图8.61f, 这两个测试程序为我们先前提到的内存操作占比较大的程序, 能更好地反映性能差异, 可以看出, 细粒度的调度方式在此情况下的性能比单节点与粗粒度要差, 而粗粒度和单节点的性能近似。说明两个远程内存节点采用合适的调度方式能够一定程度上提高性能, 甚至比使用单节点的性能更好。



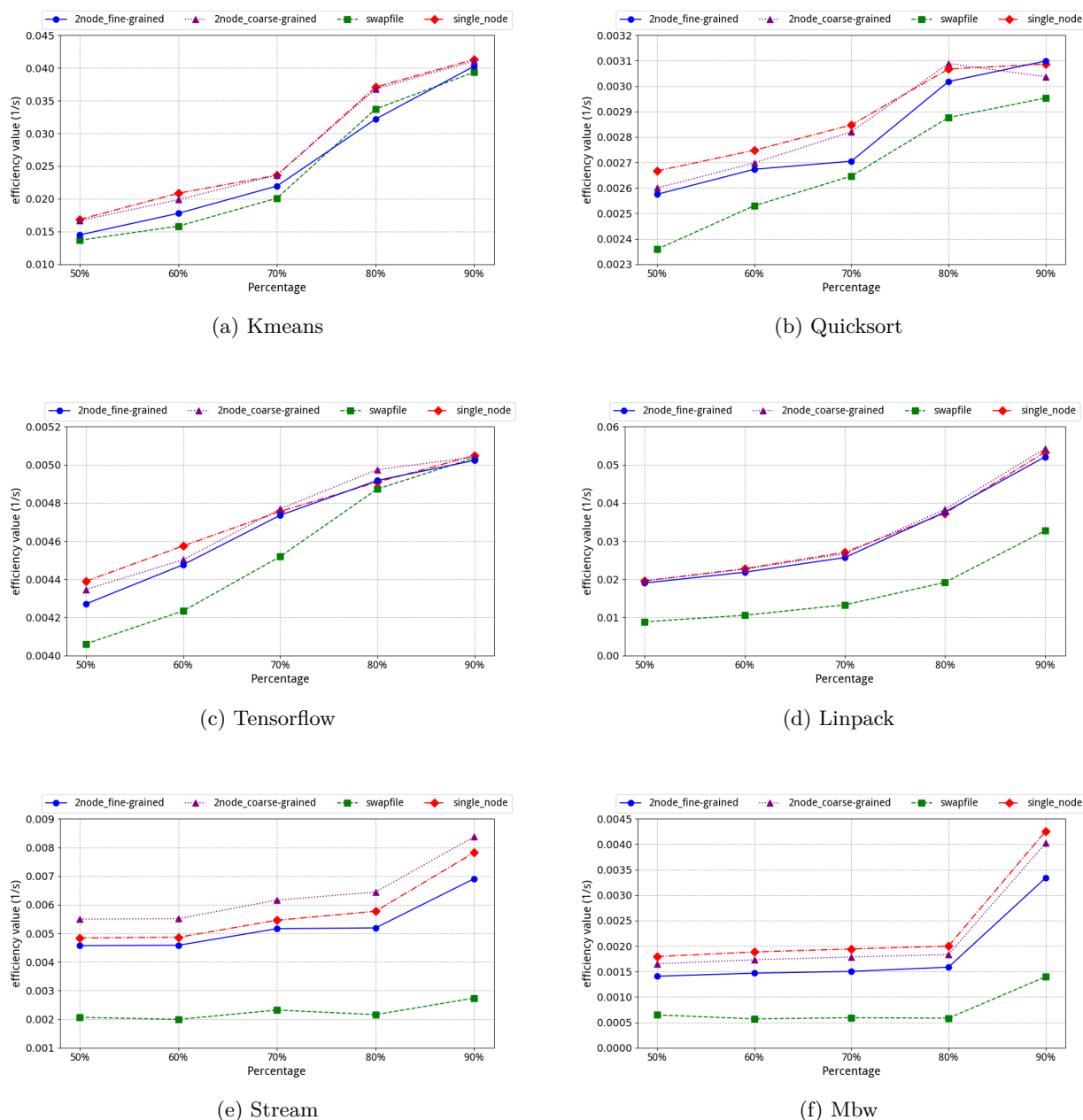


图 8.61: 使用两个远程内存节点并采用不同调度方法与单节点和 swapfile 的性能比较

最后, 还剩余了先前的一个疑惑没有解答, 我们是如何知道两种粒度的调度方式在一定程度以上的工作负载是均匀的呢? 我们通过为模块中的每一个 gctrl 控制器添加了计数器, 当页面被置换到控制器所对应的远程内存上时, 将该计数器 +1, 最后在模块的卸载函数中输出该计数器的值即可。于是只需要进行安装模块-> 进行程序测试-> 卸载模块的操作就能够得到这段时间内向两个远程内存节点中触发的 swap 操作的个数并统计, 得到结果如图8.62所示, 这证实两种粒度调度到两个远程内存的页面是均匀的! 同时也从侧面证明了, 若使用了这两种分配方式, 当一个远程内存节点的带宽受到限制并占满时, 流经另外一个远程节点的流量也会变小。

```
[973647.919525] multisswap_rdma: CM connection closed 10
[973647.919553] multisswap_rdma: cm_handler msg: disconnected (10) status 0 id 000000009d3221b2
[973647.919557] multisswap_rdma: CM connection closed 10
[973647.948853] multisswap_rdma: sswap_rdma_removeone()
[973647.948860] multisswap_rdma: sswap_rdma_removeone()
[973647.948869] page_count_dev[0]=9269319, page_count_dev[1]=9263728
```

图 8.62: 页面计数器

## 8.6 异构节点测试

在本节中，我们对带宽限制所形成的异构节点进行测试，主要比较测试用例在调度前后运行稳定时每秒的吞吐量以及运行总时间来彰显我们所提供策略的有效性。除了基本测试集 stream 之外，我们还将 pagewalker 程序（详见第7节）修改，采用 openMP 进行并行化以加大带宽占用。其余的测试集由于无法达到很好的带宽占用，我们在此不使用。

在6节中，我们在提出调度策略时，已经对 stream 测试集在未限制带宽、限制带宽、带宽限制后进行优化等三种情况进行了测试，下面我们将沿用此思路，对并行化后的 pagewalker 进行相应测试。首先，在未限制带宽时，使用粗粒度分配方式的带宽监测结果如下图8.63所示：

```
bwm-ng v0.6.3 (probing every 2.000s), press 'h' for help
input: /proc/net/dev type: rate
```

iface	Rx	Tx	Total
swp1:	4.99 Gb/s	4.73 Gb/s	9.72 Gb/s
swp3:	5.00 Gb/s	4.73 Gb/s	9.72 Gb/s
swp5:	9.45 Gb/s	9.99 Gb/s	19.44 Gb/s
total:	19.44 Gb/s	19.44 Gb/s	38.89 Gb/s

图 8.63: 未限制带宽时进行带宽监测

可以看到，与 stream 测试集相当，在使用所有 CPU 核心后最大能达到的系统吞吐量为 40Gb/s。接着，采用和之前一样的带宽限制方式，模拟异构节点。监测得到的数据如图8.64所示，并测试得到程序运行时长为 24.948s：

```
bwm-ng v0.6.3 (probing every 2.000s), press 'h' for help
input: /proc/net/dev type: rate
```

iface	Rx	Tx	Total
swp1:	2.85 Gb/s	2.68 Gb/s	5.53 Gb/s
swp3:	2.85 Gb/s	2.70 Gb/s	5.55 Gb/s
swp5:	5.42 Gb/s	5.69 Gb/s	11.11 Gb/s
total:	11.12 Gb/s	11.08 Gb/s	22.20 Gb/s

图 8.64: 限制带宽后进行监测

在此异构节点下，受到了低带宽节点的影响，系统的性能急转直下，系统吞吐量由原来的 40Gb/s 变为了 23Gb/s，运行时长也降低为 43.561s。最后我们将优化后的调度策略应用，再一次监测异构系统的带宽，得到结果如图8.65所示，同时程序运行时长提升到 28.219s：

```

bwm-ng v0.6.3 (probing every 2.000s), press 'h' for help
input: /proc/net/dev type: rate
-
=====
      iface                Rx                Tx                Total
=====
      swp1:                6.24 Gb/s                5.88 Gb/s                12.12 Gb/s
      swp3:                3.12 Gb/s                2.94 Gb/s                 6.06 Gb/s
      swp5:                8.81 Gb/s                9.36 Gb/s                18.17 Gb/s
-----
      total:               18.18 Gb/s               18.18 Gb/s               36.36 Gb/s

```

图 8.65: 优化策略后进行监测

通过 stream 和 pagewalker 两个高带宽占用用例的测试，我们将得到的测试结果汇总为表6：

	stream 吞吐量 (Gb/s)	stream 运行时长	pagewalker 吞吐量 (Gb/s)	pagewalker 运行时长
未限制带宽	43.59	3:04.5	38.89	0:24.948
限制带宽	30.30	4:25.21	22.20	0:43.561
调度优化	38.33	3:23.87	36.36	0:28.219

表 6: 优化前后测试对比

能够看到，在我们关注的关键指标上，相比于未优化的测试，优化后都取得了 25% 以上的性能提升，说明了我们所实现调度策略的有效性。同时，也说明了对这方面的工作进行研究的必要性。

## 9 遇到的困难以及解决方案

在项目实现的三个阶段中，详细记录了在本项目中遇到的主要问题以及解决方法，在此处不再赘述。

## 10 总结与展望

### 10.1 项目总结

在本项目中，我们通过 RDMA 技术成功地在 Linux 6.1 版本的内核上完成了基于 frontswap 接口的换页机制，将计算节点的物理页置换到两个异构的远程内存节点中，配置了一个拥有一个计算节点与两个内存节点的可扩展内存池集群，模拟数据中心集群中部分机器存在闲置内存的真实情况。我们进行了一系列实验测试，证明了基于此方法所实现的远程内存的可行性，还测试了在带宽限制下系统性能的表现，发现了异构内存节点中存在的带宽瓶颈在于慢节点的带宽，并针对该瓶颈问题完成了性能优化，以此提高了系统整体的性能。最后，设计了本地到远程内存节点的高效页面映射机制，为未来可实现的自适应页面调度策略打下了基础。此项目为未来大规模数据中心和高性能计算环境中的远程内存技术应用提供了有价值的参考。

### 10.2 未来展望

在我们所实现的方法中，仍然有一些未考虑全面的部分：

1. 对于 Linux 的页面换入过程来说，存在两种换入的页面：一是 pagefault 缺失的页面，该页面要求以最快的可能换入，二是 Linux 通过 swapin\_readahead 里预取得到的连续几个页面，这些页

面的优先级低于 pagefault 中的页面。但是我们的实现将它们统一进行处理：只要是读取的页面，都采用同步的方式读取到数据后返回，此方式很可能拖慢了系统的整体读取性能。

2. 我们所实现针对带宽的页面调度机制优缺点都很显然，优点是足够简单，对每一页进行处理的代价小，因此拥有较小的调度代价；缺点是需要拥有对节点带宽的先验知识，人工设置分配比例，并且动态调整能力较差。
3. 我们主要考虑了计算节点端的调度方式，有时需要考虑内存节点自身的瓶颈而非网卡带宽。例如在实际中，内存节点采用的往往是大型数据中心中拥有多余内存资源的节点，这些节点满足相应的空闲内存要求，但 RDMA 网卡使用的是 PCIe 接口，当 PCIe 总线带宽占用过高时，其可能代替网卡带宽成为传输过程中的另一个瓶颈。

在后续的工作中，我们将尝试修改内核中的 frontswap 接口，针对读取到的页面类型进行同步异步处理，具体来说，使用同步读操作处理 pagefault 中的页，异步读操作处理预取页，以此提高读取效率。我们还希望研究计算节点面对异构的远程内存节点时更加细粒度的页面调度机制，以更好地利用带宽并提升系统整体性能。除此之外，我们可以模拟真实服务器的集群环境，在远程内存节点上同时运行负载程序，例如使用 GPU 编程的 Cuda 程序，并尽量触发显存交换以占用远程内存节点更多的 PCIe 总线带宽。由于 RDMA 网卡使用的也是 PCIe 接口，我们希望探究远程内存节点运行何种负载时，PCIe 总线带宽会成为系统的性能瓶颈。

## 11 比赛收获

在本项目的实现过程中，得到了以下收获：

1. 从 0 开始了解并学习 RDMA 通信原理与编程技术以及网络通信相关知识，对 RDMA 技术有了深刻的认知。
2. 了解并学习如何修改、编译、安装 Linux 内核，并学习了如何构建 Linux 内核模块。
3. 深入了解学习了 Linux 系统的换页机制，特别是对于 SWAP 系统的相关结构有了充分的理解。
4. 学习了解了内核态 RDMA 连接与用户态 RDMA 连接的构建区别，成功实现了构建内核态 RDMA 连接并进行读写，掌握了其不同的通信方式，了解并使用了其优势所在。
5. 学习了对内核以及内核模块的 debug 工具和方式，为后续的开发工作提供了便利。
6. 从 0 开始认识、了解到使用交换机，学会了在交换机上配置交换机操作系统、监控带宽和控制带宽等相关操作。
7. 学会使用 Linux 下高效的性能分析工具 perf，并使用 FlameGraph 等辅助工具进行系统的性能测试。
8. 根据网卡带宽的限制实现了页面分配机制，更加充分利用了异构节点的性能，并提升了总体系统的性能，为我们提供了宝贵的科研经历。

## 12 致谢

感谢宫晓利老师的指导以及实验室董佳霖、刘兴泽、王理治等师兄的帮助！在老师与师兄们的帮助下，我们完成了本项目的实现！

## 参考文献

- [1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [2] Linus Torvalds. Nvme over fabrics rdma host code. <https://github.com/torvalds/linux/blob/master/drivers/nvme/host/rdma.c>, 2024.
- [3] Linux Documentation. Linux kernel 6.1 documentation online. <https://www.kernel.org/doc/html/v6.1/>, 2024.
- [4] Linux Man Pages. Linux man pages online. <https://linux.die.net/man/>, 2024.
- [5] Matthew Wilcox. Linux xarray. <https://www.kernel.org/doc/html/latest/core-api/xarray.html>, 2024.
- [6] nvidia. Nvidia mlnx ofed docs. <https://docs.nvidia.com/nvidia-mlnx-ofed-documentation-v23-07.pdf>, 2024.
- [7] nvidia. Nvidia networking docs. <https://docs.nvidia.com/networking-ethernet-software/cumulus-linux-59/System-Configuration/>, 2024.
- [8] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. {MemLiner}: Lining up tracing and application for a {Far-Memory-Friendly} runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 35–53, 2022.