中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES
ISCAS

RISC-V虚拟内存扩展的概念验证

李昕[1], 张梓悦[1], 胡博涵[1], 潘庆霖[2], 林泓宇[2]
[1]中科院计算所
[2]中科院软件所
2021年6月25日

RISC-V基金会发展伙伴

- **发展伙伴**：对RISC-V做出大量技术贡献的组织[1]

- **主要贡献**：通过技术投入推进RISC-V指令集扩展的确立
  - RISC-V指令集扩展的确立需要经历很多工程验证
  - 发展伙伴助力RISC-V基金会完成验证工作
  - 涵盖了硬件、操作系统、模拟器、编译器、形式化验证等

RISC-V DEVELOPMENT PARTNER

- **中科院计算所和中科院软件所是首批RISC-V基金会发展伙伴**

[1] https://riscv.org/risc-v-development-partner/

虚拟内存

- 虚拟内存是现代计算机系统中的一个关键技术
  - 在软件上提供了统一的虚拟地址空间
  - 在硬件上实现虚拟地址到物理地址的转换

- 工作组提出多种虚拟内存相关的指令集扩展[1]，分别处于不同状态

机制	对现有机制的调整	Svnapot	Svpbmt
内容	Sv57 推测更新PTE A位 新增PTE C位 地址翻译算法重构 放松PTW A/D位更新逻辑 } 曾用Zsa	自然对齐的二次幂地址转换(如64KB大小的页支持)	基于页的内存属性
状态	将合并进特权指令集1.12	将要稳定(stable)	仍在讨论中

[1] RISC-V Virtual Memory task group：https://github.com/riscv/virtual-memory

RISC-V虚拟内存扩展的概念验证

- 计算所和软件所共同承担了RISC-V虚拟内存扩展的概念验证工作
  - 计算所：硬件实现及概念验证
  - 软件所(PLCT实验室)：操作系统支持及概念验证

	Sv57	其他特性 (Zsa)	Svnapot
软件	Linux		Linux
硬件	NutShell[1]	rocket-chip[2]	NutShell

- 概念验证工作阶段性成果已经开源
  - 开源仓库地址：https://github.com/RV-VM

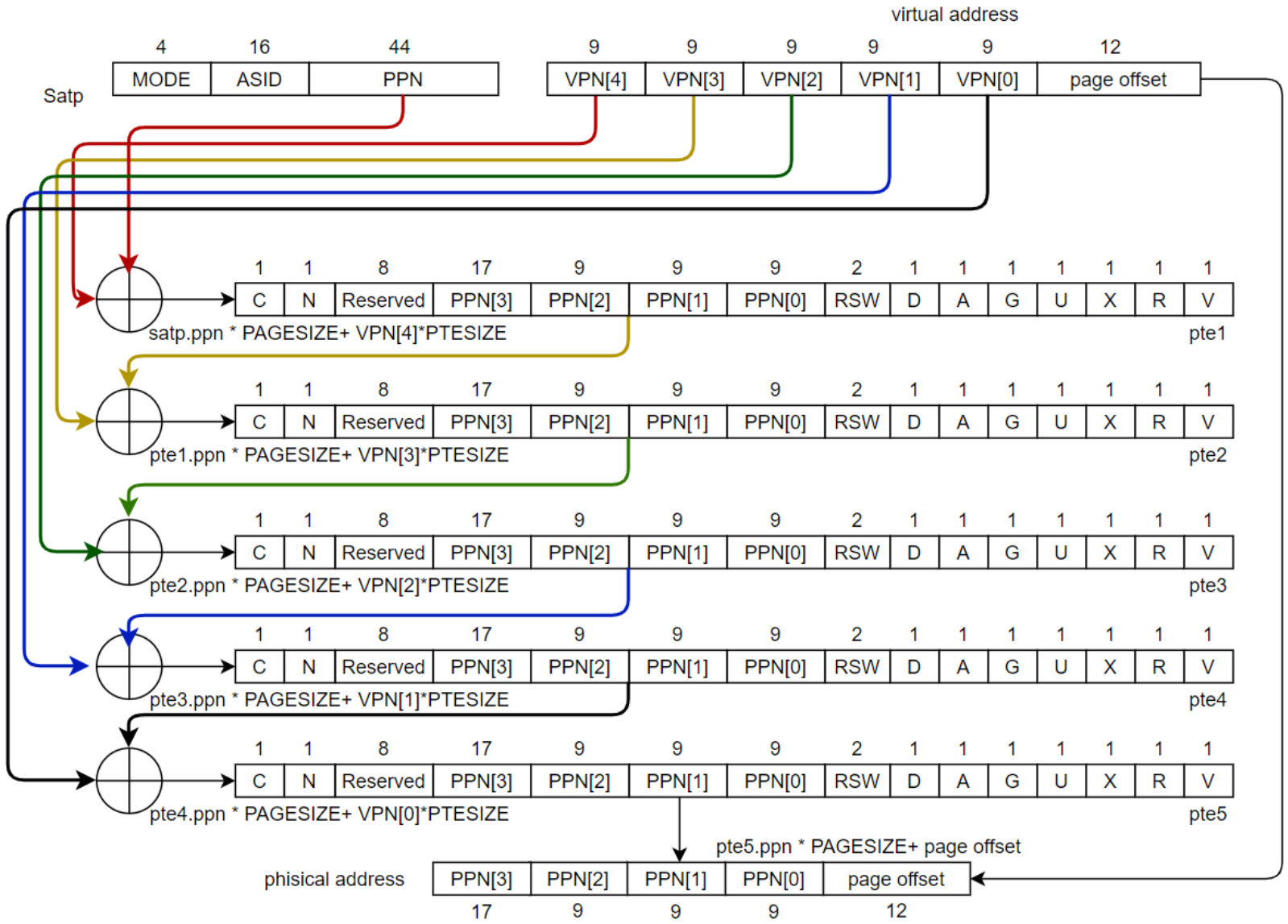
[1] NutShell：https://github.com/OSCPU/NutShell
[2] rocket-chip：https://github.com/chipsalliance/rocket-chip

Sv57：一种新的分页方案

- RISC-V 的分页方案以 Sv*N* 的模式命名，*N* 表示以位为单位的虚拟地址长度。现有的分页方案包括 Sv32、Sv39、Sv48
- Sv57分页方案：提供更大的虚拟地址空间，支持更大的数据规模

方案名称	虚拟地址宽度	虚拟地址空间	物理地址宽度	物理地址空间
Sv32	32	2^{32} Byte (4GB)	34	2^{34} Byte (16GB)
Sv39	39	2^{39} Byte (512GB)	56	2^{56} Byte (64PB)
Sv48	48	2^{48} Byte (256TB)	56	2^{56} Byte (64PB)
Sv57	57	2^{57} Byte (128PB)	56	2^{56} Byte (64PB)

Sv57的实现中对PTW的修改



Page Table Walk

从根部遍历页表，进行虚拟地址到物理地址的转换

- satp.ppn给出一级页表的基址，VPN[4]给出一级页号
- pte1.ppn给出二级页表的基址，VPN[3]给出二级页号
- ……
- pte4.ppn给出五级页表的基址，VPN[0]给出五级页号
- pte5.ppn和page offset字段组成了最终的物理地址

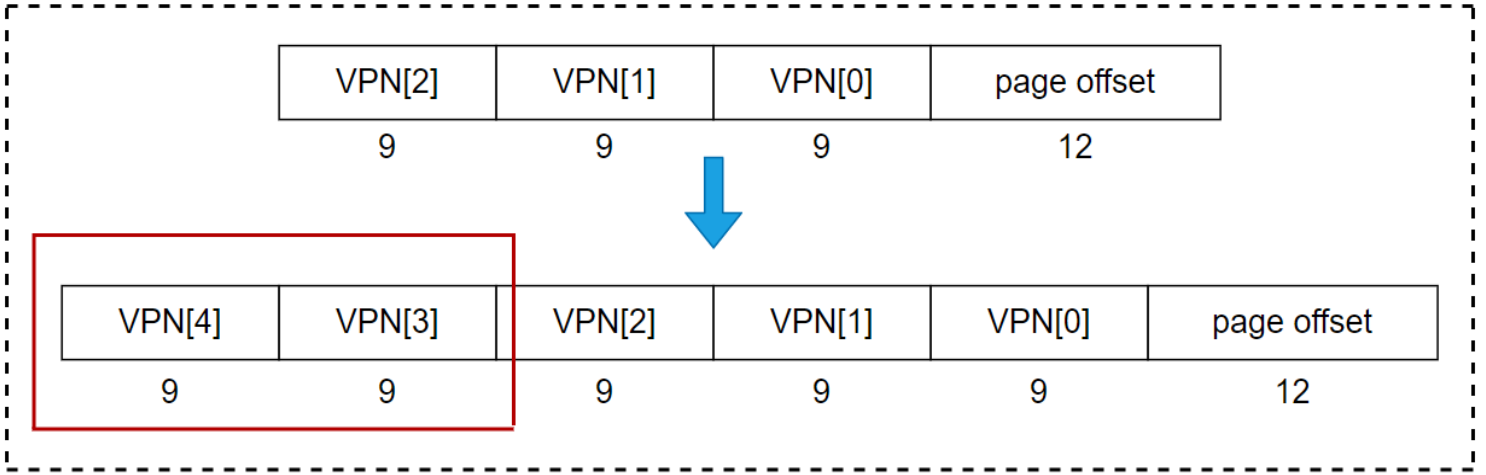
NutShell上实现Sv57

原先的NutShell

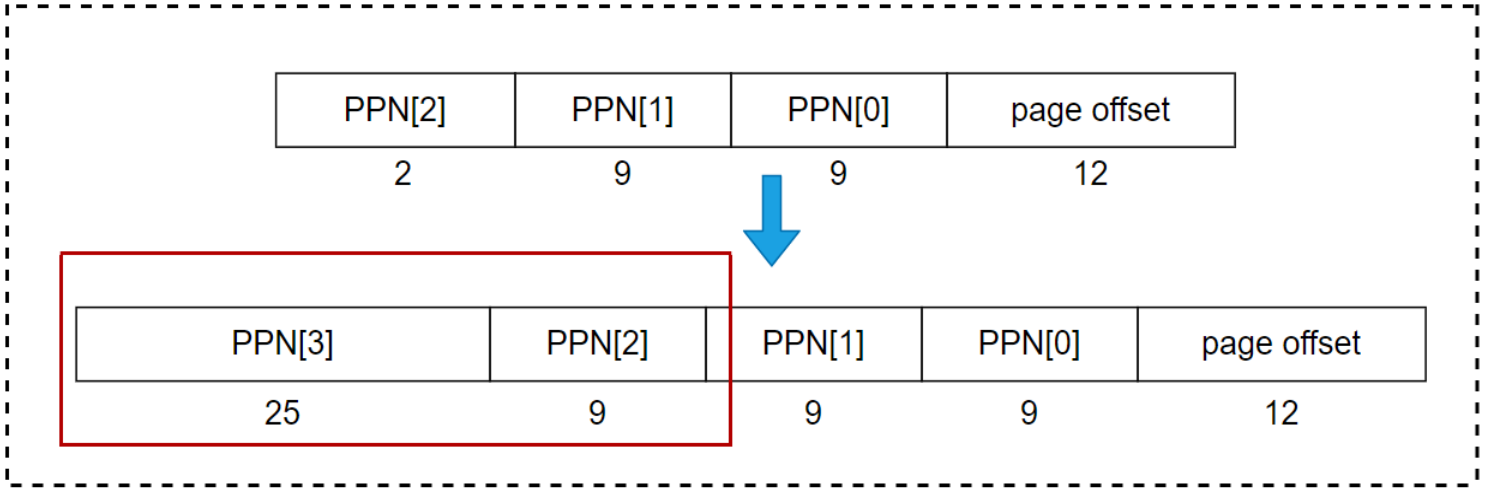
- 分页方案为Sv39
- 39位虚拟地址
- 32位物理地址
- PTW过程遍历一个三级页表

修改后的NutShell

- 分页方案为Sv57
- 57位虚拟地址
- 64位物理地址
- PTW过程遍历一个五级页表



虚拟地址的修改



物理地址的修改

NutShell上测试Sv57

- **初始化**
  - 在NutShell的内存中放置一个（采用Sv57方案的）5级页表
  - 设置satp寄存器的ppn字段
  - 开启NutShell虚拟地址到物理地址的转换
- **运行**
  - 在NutShell上运行C程序编译得到的二进制文件
  - 与模拟器nemu[1]进行寄存器行为比对[2]，从而验证实现的正确性

[1] 余子濠. NEMU：一个效率接近QEMU的高性能解释器. RVWC2021.
[2] 王凯帆, 王华强. SMP-MArch-Diff: 支持多处理器和RV微结构状态的差分测试方法. RVWC2021.

Sv57在Linux上的实现

- 基本背景
  - Linux内核**主线尚未支持**Sv48和Sv57
  - Alex Ghiti发布了关于Sv48的patch[1]，目前尚未合并到主线中
- **概念验证**
  - 以Sv48的补丁为基础开展关于Sv57的概念验证工作
  - 需要验证：实现可行性、功能正确性、运行效率等
- 目前进展
  - 在nemu中实现了对Sv48的支持[2]
  - 通过修改内核的编译配置，编译Sv48相关代码
  - 在nemu上成功启动支持Sv48的内核
  - 正在进行Sv57的实现，已完成部分功能

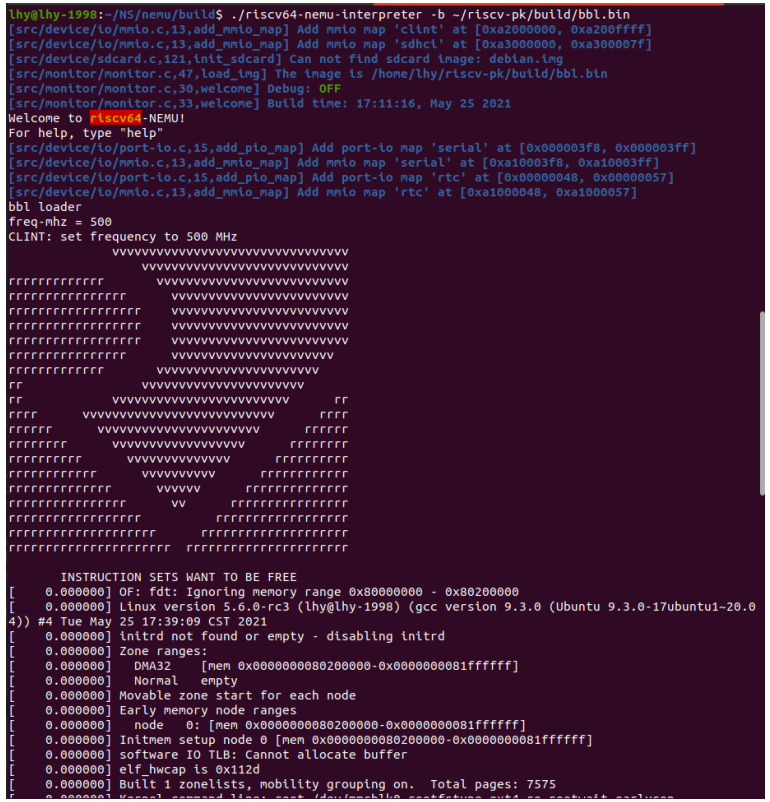
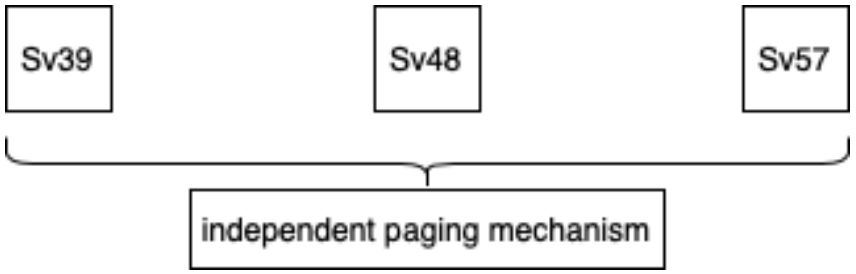


图 - 在nemu中运行Sv48内核

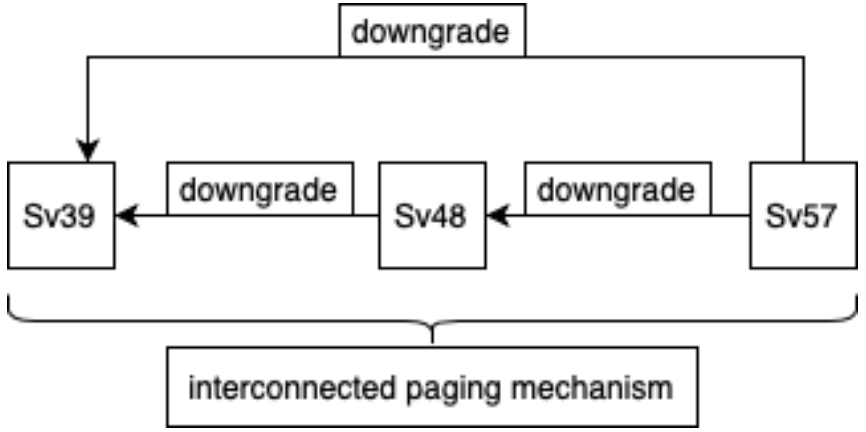
[1] Linux社区中关于Sv48的补丁：https://patchwork.kernel.org/project/linux-riscv/list/?series=408843
[2] 支持Sv48的nemu：https://github.com/LHY-24/nemu/tree/nemu-sv48

■ Sv57在Linux上的实现

- 实现思路
  1. 将Sv39、Sv48、Sv57作为三个独立的机制来实现
     - 无需考虑向前兼容，实现复杂度较低
     - 无法根据硬件环境作运行时调整，通用性较差



  2. 将Sv39、Sv48、Sv57作为一个整体来实现
     - 支持向前兼容，实现复杂度较高
     - 根据硬件环境在不同的页表机制间灵活调整，通用性好



Sv57在Linux上的实现

页表结构

- 结构变化：Sv57比Sv48多使用了一级页表 —— P4D
- 实现思路：自定义了RV64_VA_BITS_57等配置项，可以在编译内核时显式开启5级分页

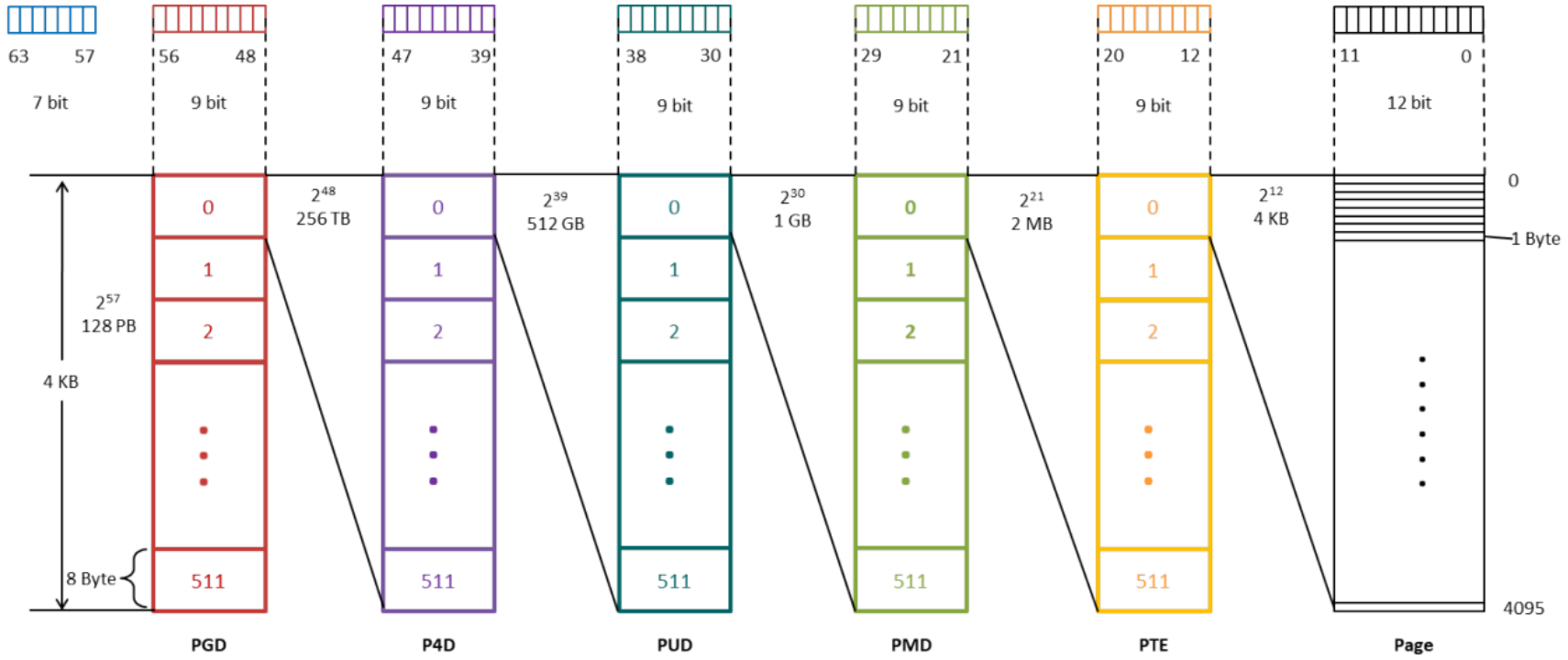


图 - Linux中的5级页表结构

Linux上的页表折叠

页表折叠

- 为什么折叠：

  编译内核时显式配置开启5级分页 ->

  硬件不支持5级分页[1] ->

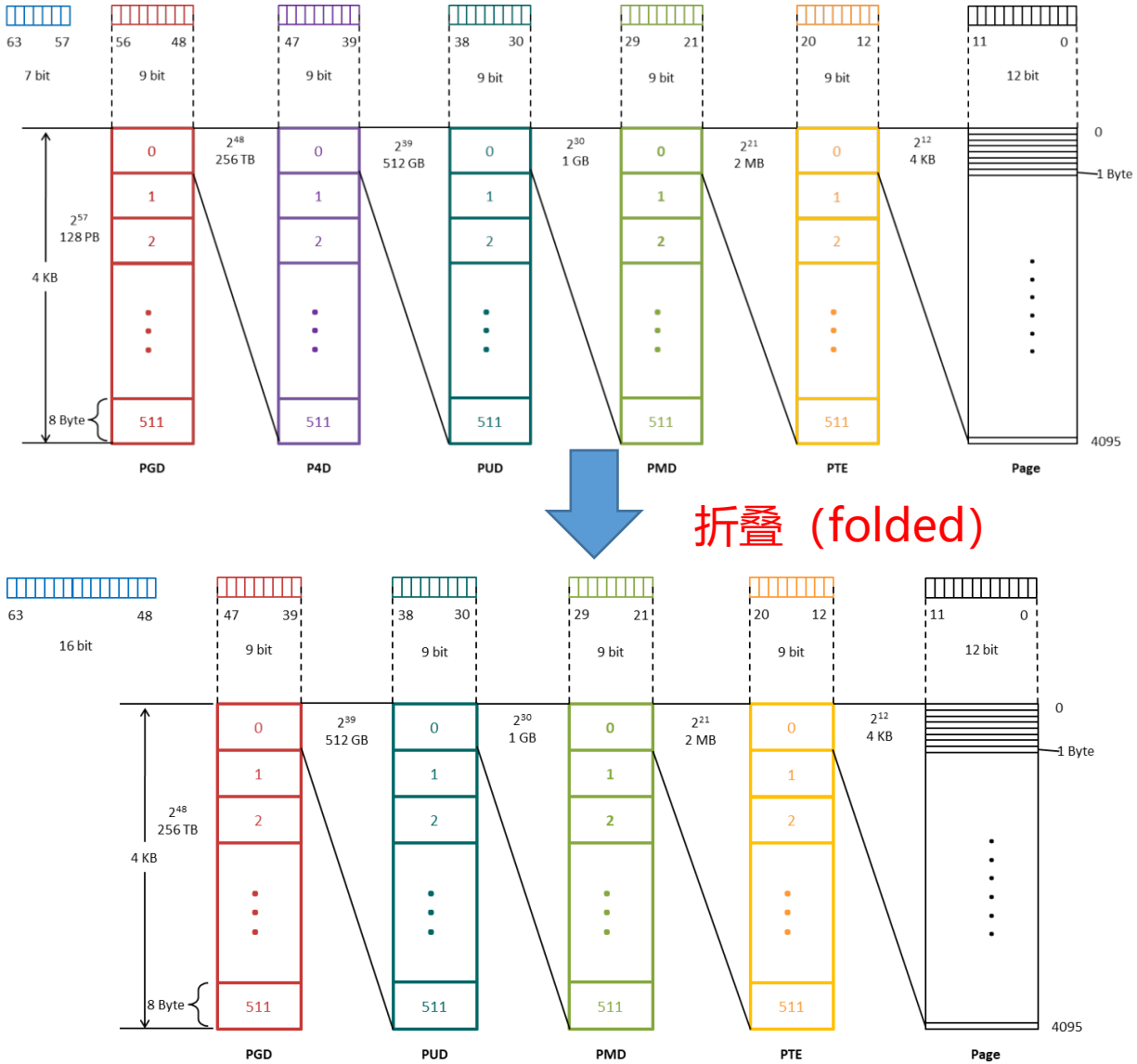
  内核回退到3级或者4级分页

- 如何折叠：

  将P4D合并为PGD ->

  修改P4D对应的页表项、函数和常量->

  令PGD指向PUD，使用4级分页寻址



[1] 在Alex的补丁中实现了一个在启动过程中利用satp寄存器MODE字段WARL特性来检测硬件是否支持4级页表的函数，这里借鉴了他的思想。

Sv57在Linux上的实现

地址空间

- 以官方文档中Sv39的布局为参照，给出了Sv57的虚拟地址空间布局
- 用户空间和内核空间的范围扩大，有效地址位数增加
- 内核的加载位置保持不变
- 各模块的相对顺序保持不变，但是具体地址范围有所变化

Start addr	Offset	End addr	Size	VM area description
0000000000000000	0	00ffffffffffffff	64 PB	user-space virtual memory, different per mm
0100000000000000	+64 GB	feffffffffffffff	~16K PB	... huge, almost 64 bits wide hole of non-canonical virtual memory addresses up to the -64 PB starting offset of kernel mappings.
				Kernel-space virtual memory, shared between all processes:
ffffffc000000000	-256 GB	ffffffc7ffffffff	32 GB	kasan
ffffffcefee00000	-196 GB	ffffffcefeffffff	2 MB	fixmap
ffffffceff000000	-196 GB	ffffffceffffffff	16 MB	PCI io
ffffffcf00000000	-196 GB	ffffffcfffffffff	4 GB	vmemmap
ffffffd000000000	-192 GB	ffffffdfffffffff	64 GB	vmalloc/ioremap space
ffffffe000000000	-128 GB	fffffffff7ffffff	124 GB	direct mapping of all physical memory
ffffffff00000000	-4 GB	ffffffff7fffffff	2 GB	modules
ffffffff80000000	-2 GB	ffffffffffffffff	2 GB	kernel, BPF

图 - Linux中的开启5级分页后的虚拟内存布局

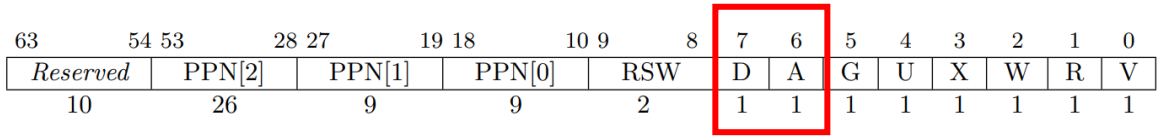
Sv57在Linux上的实现

深入探讨

内容	设想
P4D页表项相关	将架构无关的部分（赋值、清零、有效性判断等） 同X86和ARM64中的相关代码提取出来 增加内聚性，减少耦合度
Fixmap中的FIX_P4D	RISC-V中fixmap的布局与ARM64有所不同，具体细节还有待探究
pgtable.h文件	该文件包含定义较多，可以参照ARM64将其拆分为以下3个文件 pgtable.h（页表相关） pgtable-64.h（RV64中页表相关） pgtable-prot.h（页表项权限相关）

地址翻译改进算法的实现

- 旧版标准中，进行地址翻译时的Page Table Walk（页表访问）步骤对于页表A (Accessed, 访问) / D (Dirty, 写脏) 位的管理，提供了两种方案：
  - 软件管理：若初次访问页表叶节点，A/D位未设置，报异常，由软件处理
  - 硬件设置：若初次访问页表叶节点，A/D位未设置，则由硬件**原子地**设置
- 开源RISC-V处理器（Rocket, BOOM, Nutshell等）并未实现硬件管理的A/D位
- 在**Draft 1.12版本的spec**中，提出了地址翻译的改进算法，改进了硬件管理页表项A/D位的机制
  - 允许A位的**推测更新**
  - 放松了对于A位和D位更新的原子性需求



63 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

- 对于硬件实现而言，更加友好
  - 允许A位的推测更新，对取指阶段的页表A位管理更加友好：取指一般都是推测的，其对页表的访问也是推测的，因此取指时确定地更新A位并不现实
  - 放松了对于A/D位更新的原子性要求，可以使用类似于LR/SC的形式实现：LR/SC是一对原子操作，可用于监听对某个地址的更改
- **无需增加软件层面的支持**

地址翻译改进算法的实现

```
1. a = satp.ppn * PAGESIZE; i = LEVELS - 1
2. pte = LR(a+va.vpn[i]*PTESIZE)
        save the address of pte into the LR register
3. if (pte.v == 0 || (pte.r == 0 & pte.w == 1))
        Signal Page Fault
   else if (pte.r == 1 || pte.x == 1)
        It's a leaf PTE, goto step 4
   else i = i - 1
        if (i < 0)
                Signal Page Fault
        else
                a = pte.ppn * PAGESIZE
4. Check the legality of the memory access
5. If (pte.a == 0 && isLoad ||
       pte.d == 0 && isStore && isLegal )
        new_pte = pte with a/d bits set
        Try( SC(new_pte, a+va.vpn[i]*PTESIZE) )
                if SC success
                        return new_pte
                if SC fails
                        goto step 2
```

❶ 修改PTW状态机及关键逻辑

- **将“读”操作从Load变为LR**

Load Reserved将PTE地址注册到本核心的保留站（LR Register）中，并监控所有核心对其的修改

- **返回的PTE置A/D位**

- **增加“更新”步骤，操作为SC**

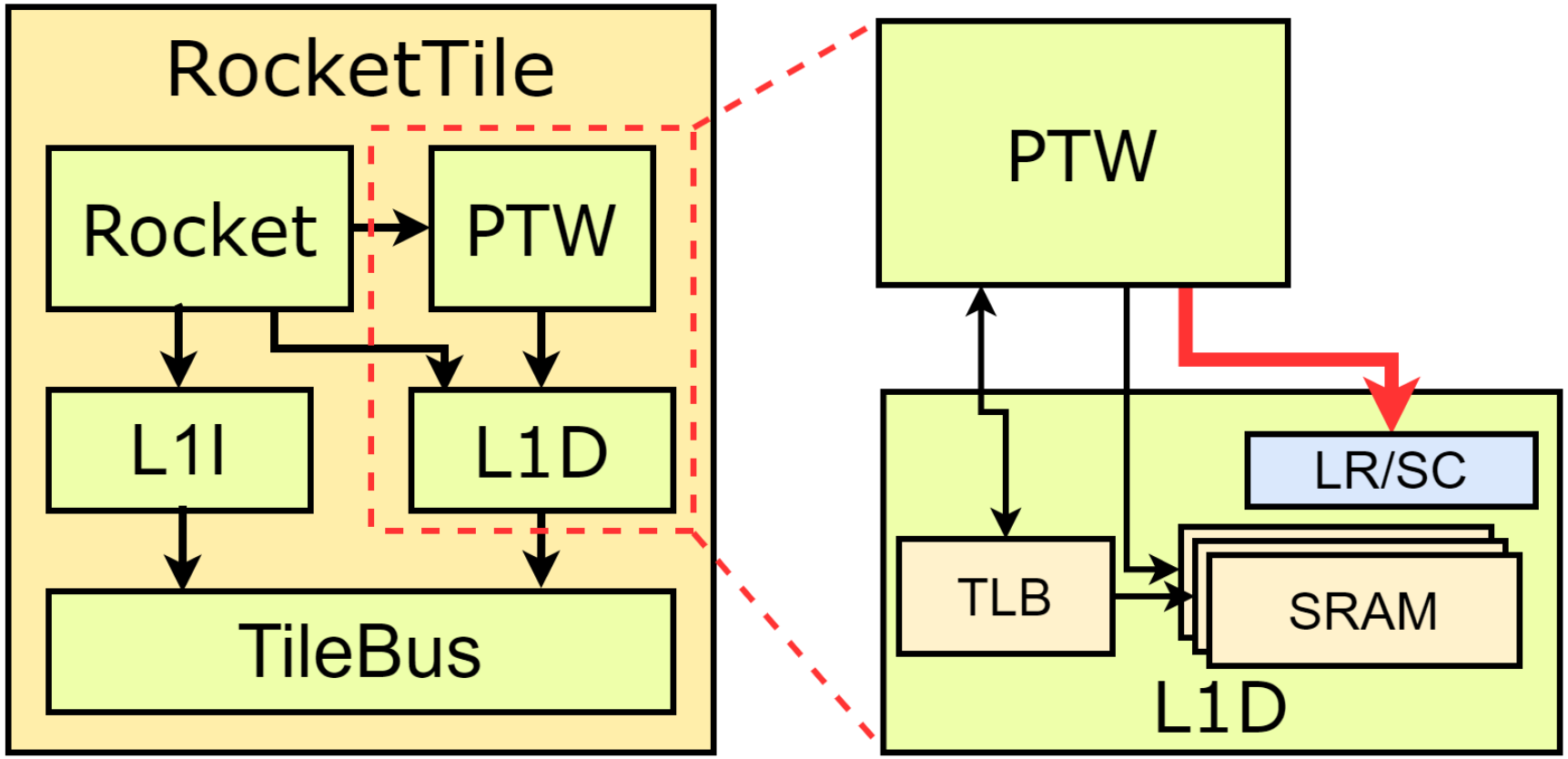
Store conditional操作，若第2步中LR注册的地址没有被写入过，则SC操作可以成功

- **增加“重试”步骤**

若发现在取回PTE和更新PTE之间，原始PTE被修改（SC失败），则尝试重新获得新的PTE

地址翻译改进算法的实现

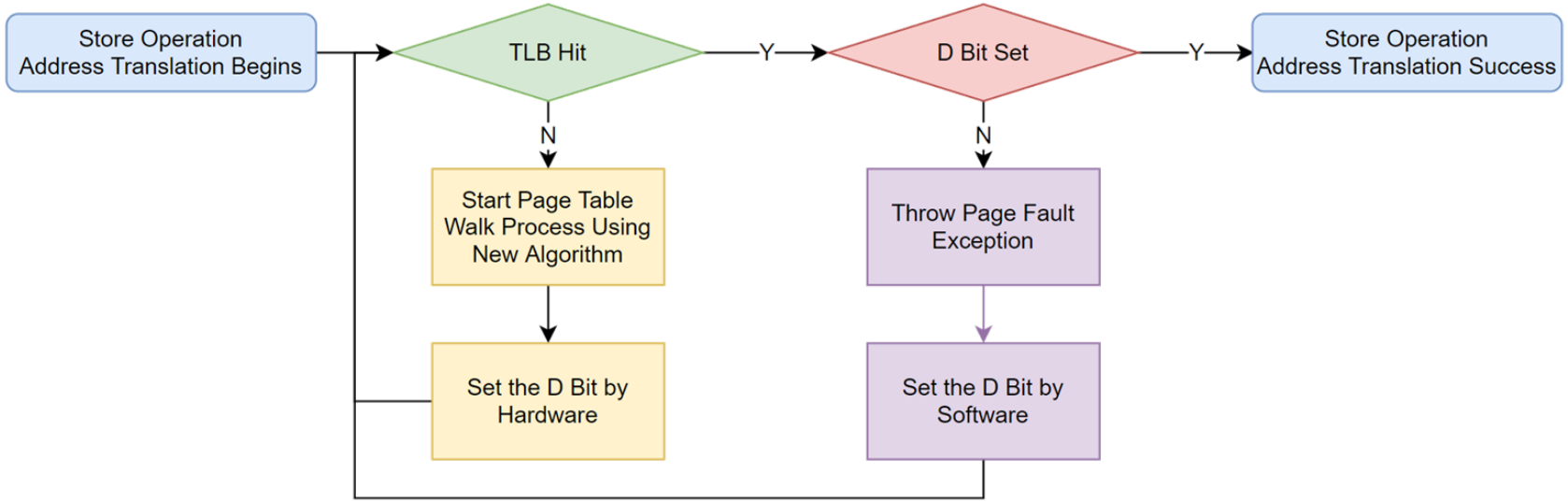
❷ 修改数据通路



- 添加PTW到LR/SC保留站之间的通路

地址翻译改进算法的实现

- 实现代价：只需添加从PTW到LR/SC保留站的数据通路
- 灵活性高：使用软/硬件混合的方式来管理A/D位，减小硬件实现成本
  - 对于Store操作而言：仅TLB Miss后进行Page Table Walk时，由硬件设置D位；当TLB Hit的时候，正常报Page Fault并由软件置位D

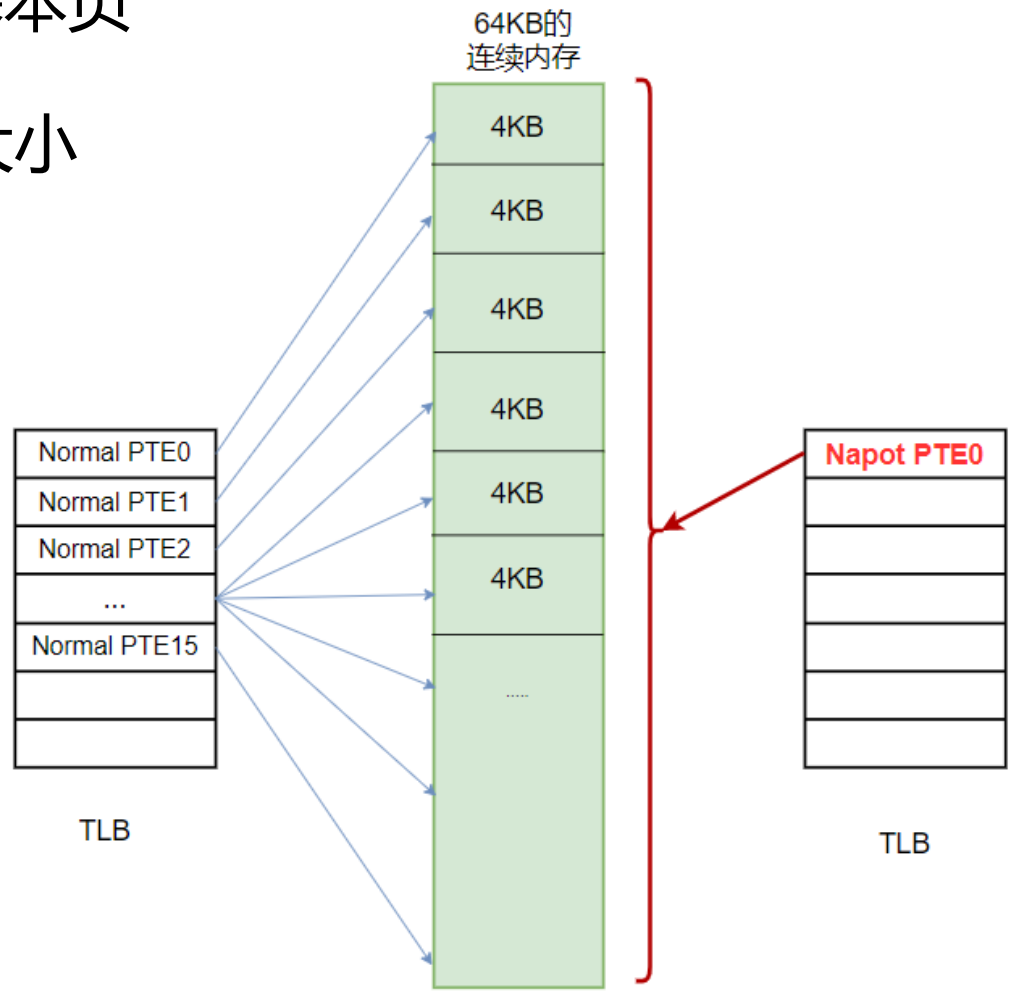


- 兼容性强：无需软件额外修改

Svnapot扩展：支持自然对齐的二次幂地址翻译

- RISC-V 现有的分页机制仅支持 4KB 大小的基本页
- Svnapot 允许**以兼容的方式支持**更多基本页大小
  - 目前的方案中，Svnapot 仅支持 64KB 的基本页
  - 未来可能会增加 128KB, 256KB 等情况
- 优势
  - 逻辑上扩充了TLB的大小
  - 降低了缺页率

描述64KB的物理地址空间使用**普通的PTE需要16个**，而使用**napot扩展的PTE只需要一个**



Svnapot扩展的实现

- ❶ 在原来的PTE基础上新增加C/N位

C位表示是否使用自定义的编码格式

N位表示是否开启Svnapot扩展

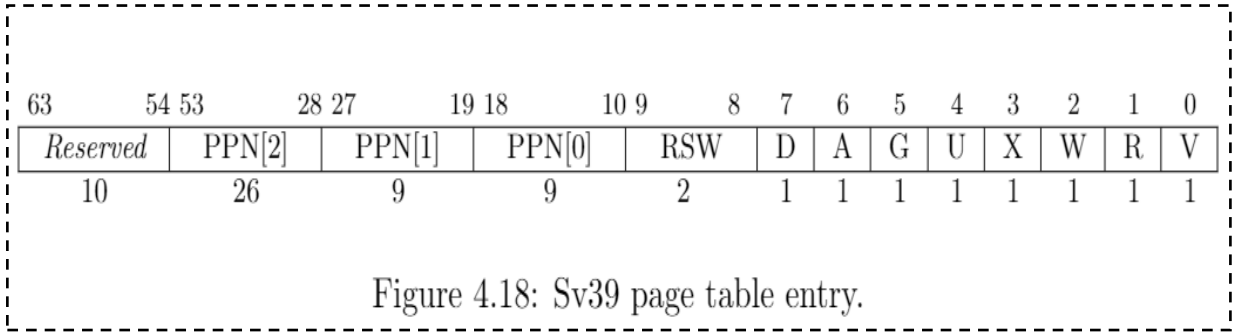


Figure 4.18: Sv39 page table entry.

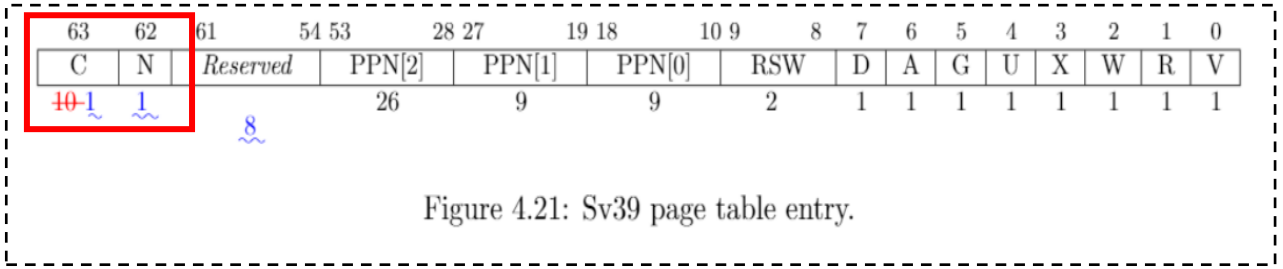
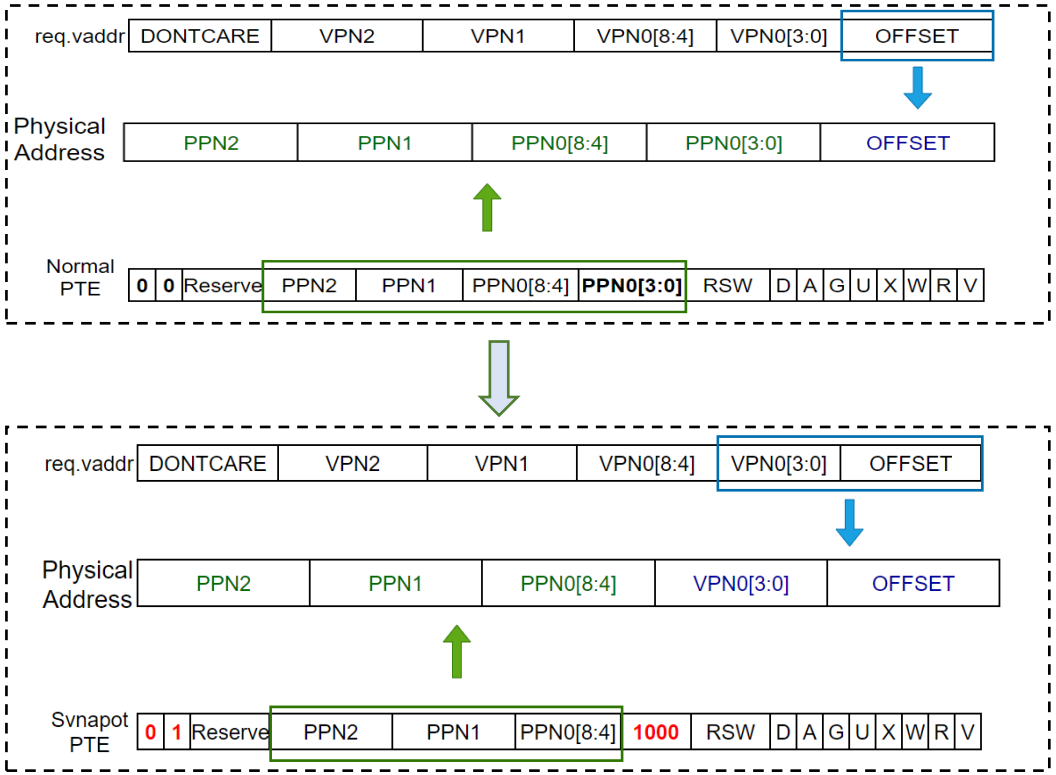


Figure 4.21: Sv39 page table entry.

Svnapot关于PTE作出的修改

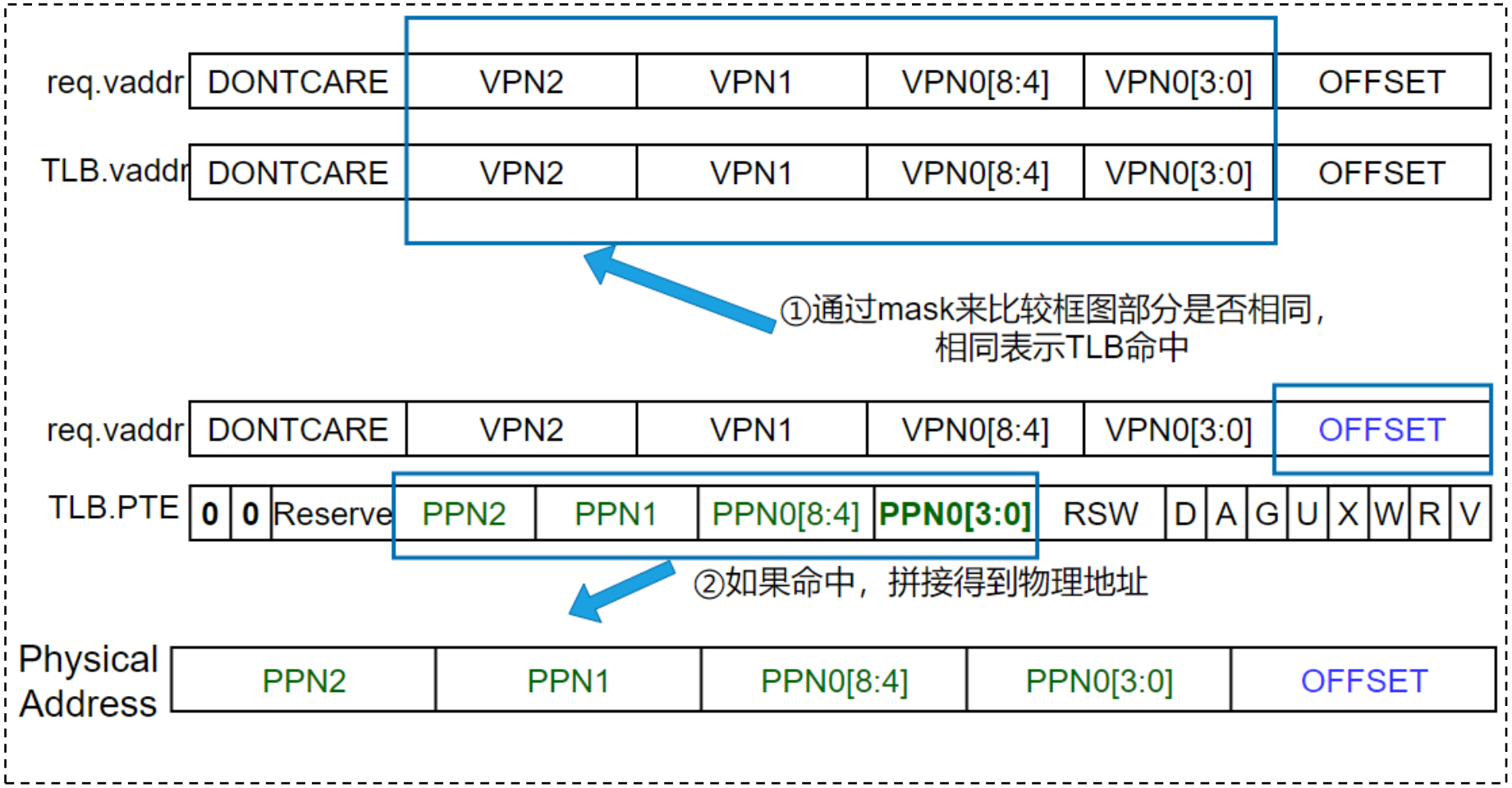
- ❷ 得到新的物理地址

一个开启Svnapot扩展的PTE(C=0,N=1)表示64KB大小的基本页



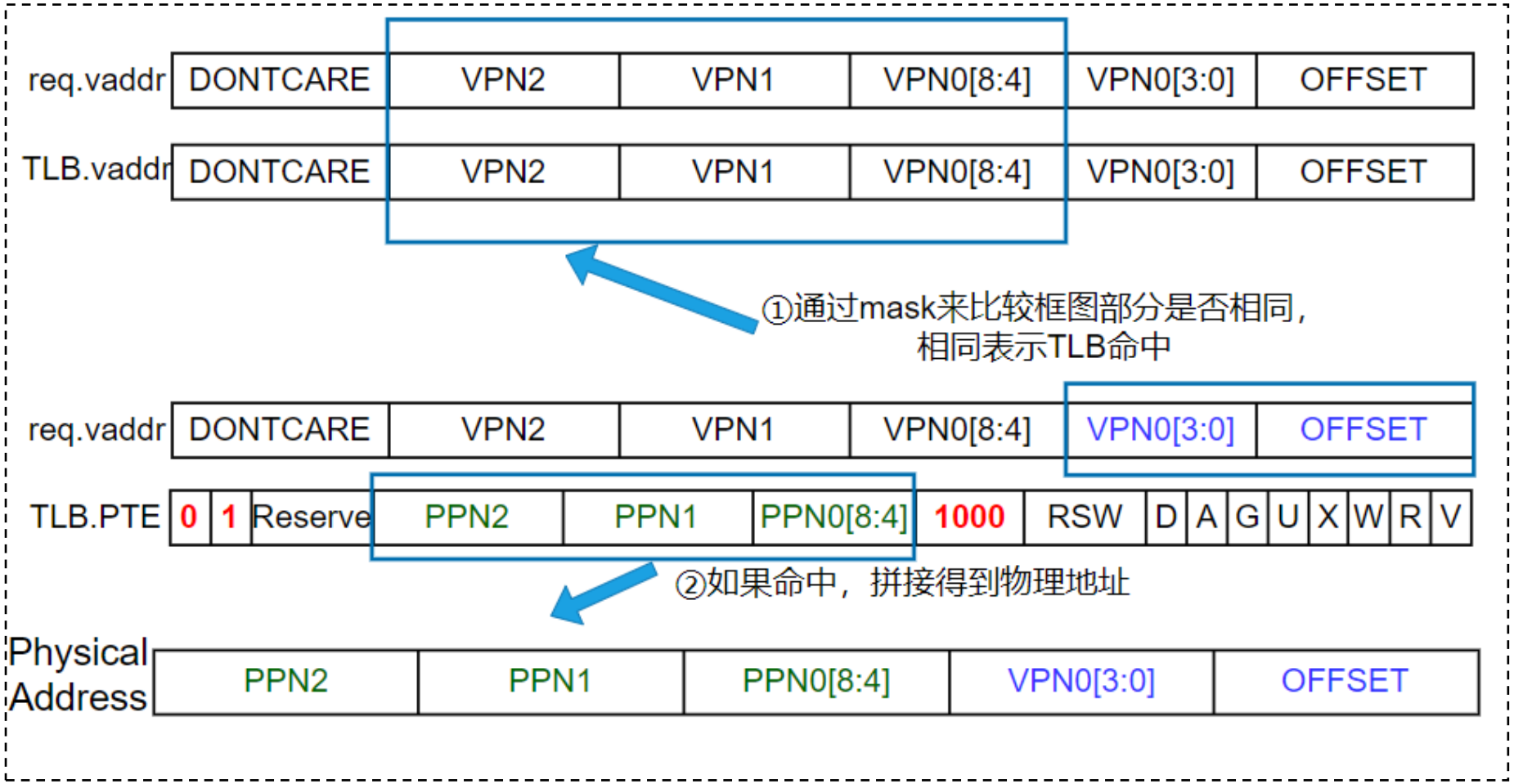
Svnapot关于地址转换作出的修改

■ NutShell上实现Svnapot扩展



- NutShell的TLB为**全相连**结构
- 使用3个不同的mask分别表示1GB,2MB,4KB的页面

■ NutShell上实现Svnapot扩展



- 新增一个mask用来表示64KB大小的页面

Linux Kernel支持Svnapot拓展

- （可能的）应用场景
  - mmap映射的匿名内存（已支持）
  - 内核中的文件缓存内容（完善中）
  - ……

- 需要进行的修改
  - 允许用户态应用程序通过mmap syscall启用Svnapot
  - VMA（Virtual Memory Address）管理：根据napot size进行对齐
  - page_cache使用napot size作为write out/read in基本单位

匿名内存mmap

- 借用已有的HUGETLB_*定义
- Svnapot拓展同时对VMA和物理页帧提出对齐要求
  - 起始地址对齐
  - 长度对齐
- 批量修改PTE需要加锁!
- Luckily，直接使用普通4K页的锁机制
  - 目前实现也是仅对PT所在的page加锁

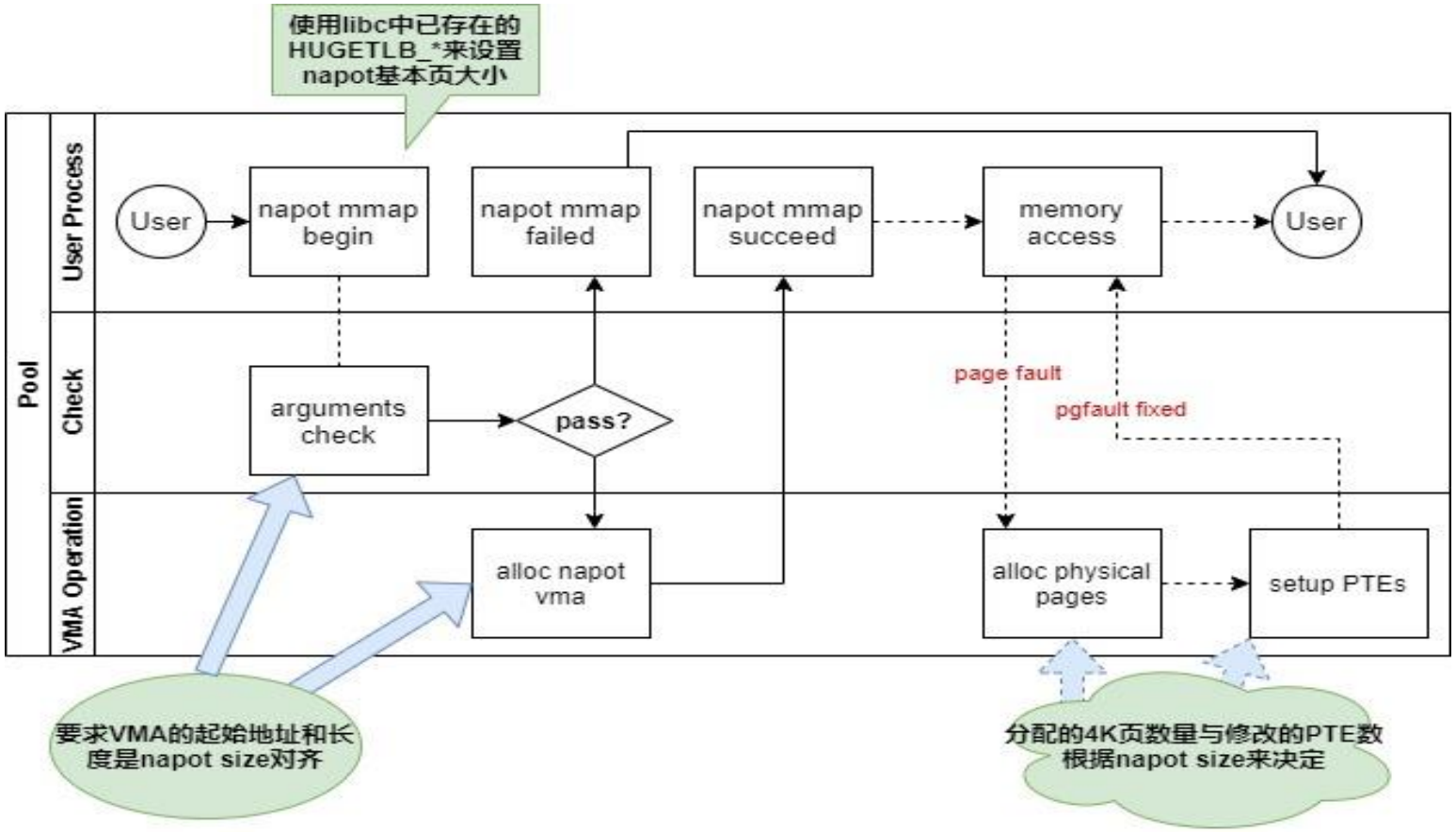


图 - 支持Svnapot拓展的mmap匿名内存使用场景

内核API适配

- 页面分配（compound page）
  - page_alloc_vma
- PTE批量设置
  - pte_set_at
- 可能导致某些基础API变慢
  - pte_pfn

```
+ #define _PAGE_NAPOT      (1UL << _PAGE_NAPOT_SHIFT)
+ #define _PAGE_CUSTOM     (1UL << 63)
+ #define _PAGE_PFN_MASK   (_PAGE_RESERVED_0 - (1UL << _PAGE_PFN_SHIFT))
+ #define __NAPOT_BIT_TO_BOOL(x) (((x & _PAGE_NAPOT) >> _PAGE_NAPOT_SHIFT) & 1UL)

  /* Yields the page frame number (PFN) of a page table entry */
  static inline unsigned long pte_pfn(pte_t pte)
  {
+ #ifdef CONFIG_NAPOT_SUPPORT
+       unsigned long temp_pfn = (pte_val(pte) & _PAGE_PFN_MASK) >> _PAGE_PFN_SHIFT;
+       return (temp_pfn & (temp_pfn - __NAPOT_BIT_TO_BOOL(pte_val(pte))));
+ #else
        return (pte_val(pte) >> _PAGE_PFN_SHIFT);
+ #endif
  }
```

图 – pte_pfn代码修改

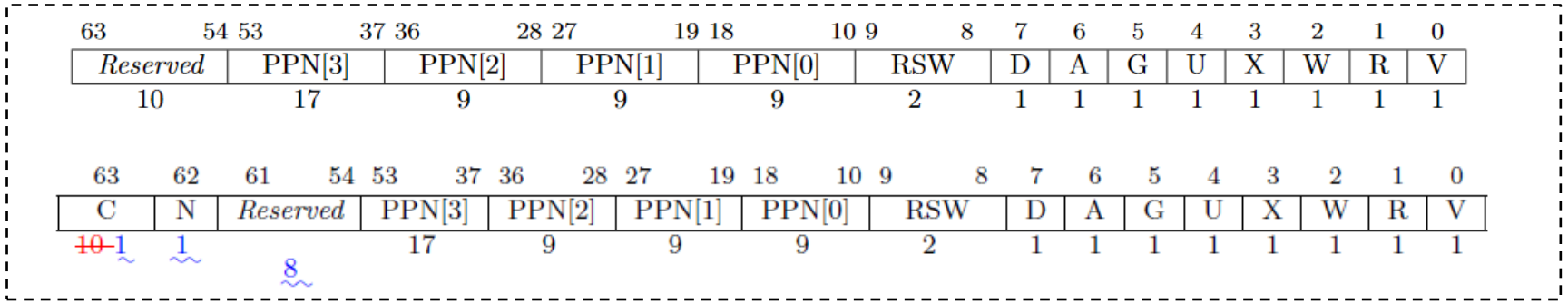


图 - PTE各字段定义的变动

文件缓存

- page_cache的napot size
  - 打开文件时设置一个**napot缓存块大小**
  - 这也是page_cache支持的**最大napot size**（比如256K）
  - 对fd进行mmap时，可以选择**不大于**设定大小的napot选项（16K or 64K or 128K）
- readahead算法适配
  - 原有的readahead算法需要对napot进行适配（或完全**禁用**原有算法）
  - 各文件系统提供的readahead接口并不统一……

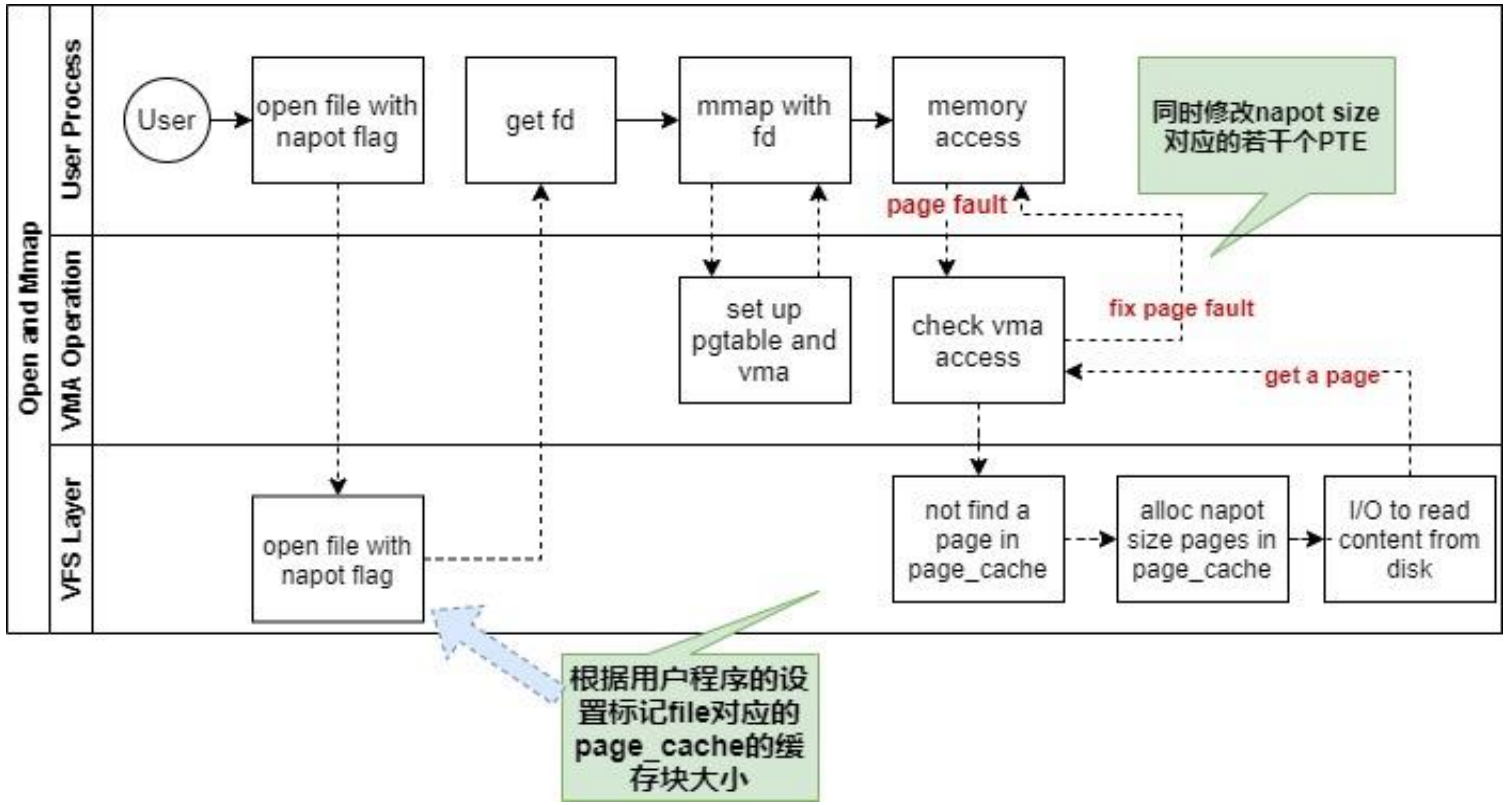


图 - 支持napot拓展的文件缓存场景

文件缓存

- page_cache以napot size为单位进行read in/write out
- **文件大小不是napot size的整数倍**?
  - ①不足部分换用4K页
  - ②page_cache中round up
- napot size设置较大时，I/O操作（比4K页）可能出现更明显**抖动**

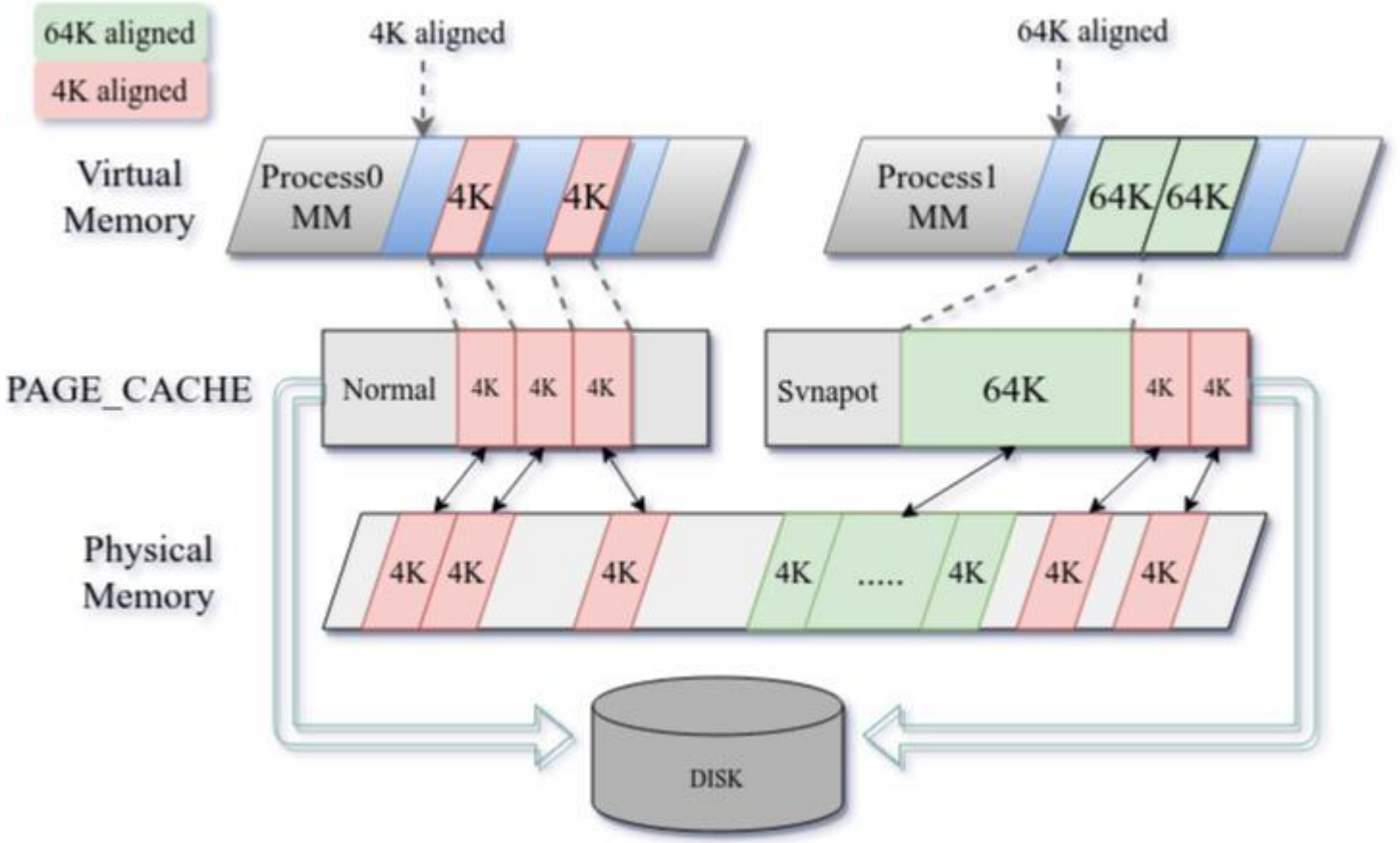


图 – 使用Svnapot与未使用Svnapot

总结

- Sv57
  - 硬件：修改地址宽度以及PTW遍历的页表级数，所需改动较小
  - Kernel：兼容已有的Sv48实现，实现方案清晰，工作量可控
- 地址翻译的改进算法（曾用名Zsa）
  - 硬件：只需改变Page Table Walker的状态机和数据通路，在保证兼容性的前提下，更高效地实现硬件对页表项A/D位的管理
  - Kernel：无需软件额外支持
- Svnapot
  - 硬件：只需为napot PTE增加新的mask，实现非常便利
  - Kernel：需要改动的内容涉及Kernel的虚存、文件系统模块，所需改动较大

谢谢！
欢迎批评指正