蜗窝科技

慢下来，享受技术。

TLB flush操作

作者：linuxer 发布于：2016-9-23 19:57 分类：内存管理

一、前言

Linux VM subsystem在很多场合都需要对TLB进行flush操作，本文希望能够把这个知识点相关的方方面面描述清楚。第二章描述了一些TLB的基本概念，第三章描述了ARM64中TLB的具体硬件实现，第四章描述了linux中和TLB flush相关的软件接口。内核版本依然是4.4.6版本。

二、基本概念

1、什么是TLB?

TLB的全称是Translation Lookaside Buffer，我们知道，处理器在取指或者执行访问memory指令的时候都需要进行地址翻译，即把虚拟地址翻译成物理地址。而地址翻译是一个漫长的过程，需要遍历几个level的Translation table，从而产生严重的开销。为了提高性能，我们会在MMU中增加一个TLB的单元，把地址翻译关系保存在这个高速缓存中，从而省略了对内存中页表的访问。

2、为什么有TLB?

TLB这个术语有些迷惑，但是其本质上就是一种cache，既然是一种cache，那么就没有什么好说的，当然其存在就是为了更高的performance了。不同于instruction cache和data cache，它是Translation cache。对于instruction cache，它是解决cpu获取main memory中的指令数据（地址保存在PC寄存器中）的速度比较慢的问题而设立的。同样的data cache是为了解决数据访问指令比较慢而设立的。但是实际上这只是事情的一部分，我们来仔细看看程序中的数据访问指令（例如说是把）的执行过程，这个过程可以分成如下几个步骤：

（1）将PC中的虚拟地址翻译成物理地址

（2）从memory中获取数据访问指令（假设该指令需要访问地址x）

（3）将虚拟地址x翻译成物理地址y

（4）从location y的memory中获取具体的数据

instruction cache解决了step （2）的性能问题，data cache解决了step （4）中的性能问题，当然，复杂的系统会设立了各个level的cache用来缓存main memory中的数据，因此，实际上unified cache同时可以加快step （2）和（4）的速度。Anyway，这只是解决了部分的问题，IC设计工程师怎么会忽略step （1）和step （3）呢，这也就是TLB的由来。如果CPU core发起的地址翻译过程能够在TLB（translation cache）中命中（cache hit），那么CPU不需要访问慢速的main memory从而加快了CPU的performance。

3、TLB工作原理

大概的原理图如下（图片来自Computer Organization and Design 5th）：

站内搜索

请输入关键词 搜索

功能

留言板
评论列表
支持者列表

最新评论

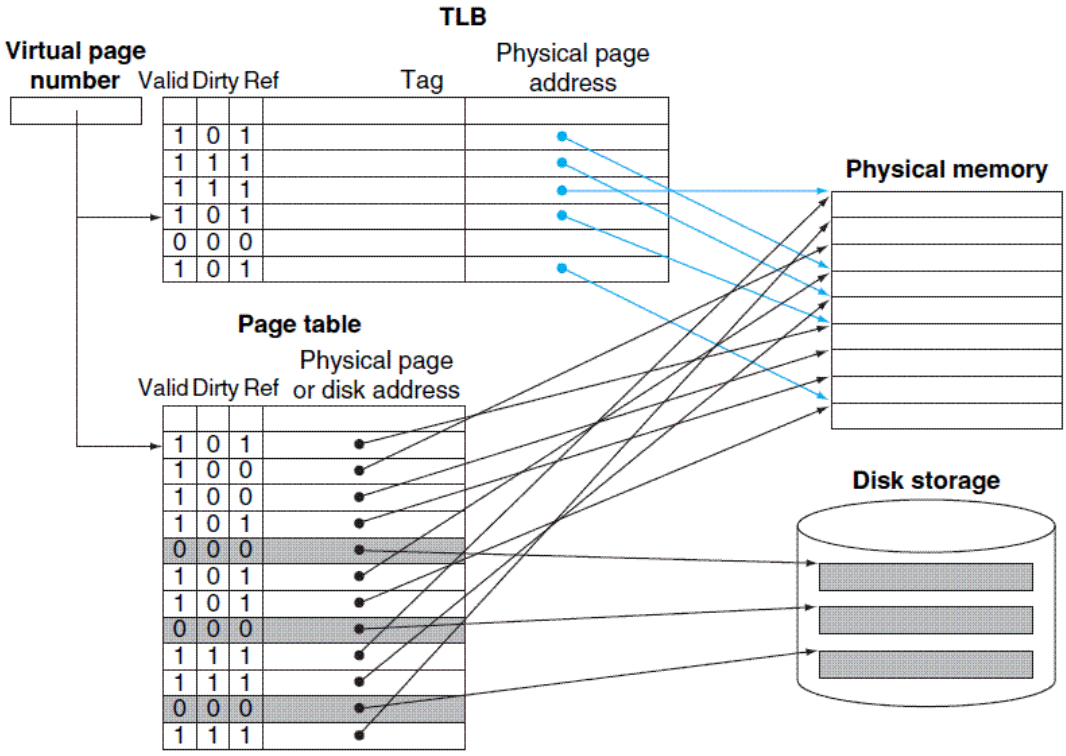
LLEo
感谢wowo 大佬
yz
@无非：group0和group1其中一个可以产生fiq，如...
xdwinter
聊表心意~感谢蜗窝,收益颇多。
xdwinter
聊表心意~感谢蜗窝
little_vage
向蜗蜗大佬致敬。不忘初心，牢记使命!
ttdevrs
图片很棒

文章分类

Linux内核分析(23)
- 统一设备模型(15)
- 电源管理子系统(43)
- 中断子系统(15)
- 进程管理(29)
- 内核同步机制(22)
- GPIO子系统(5)
- 时间子系统(14)
- 通信类协议(7)
- 内存管理(31)
- 图形子系统(2)
- 文件系统(5)
- TTY子系统(6)

u-boot分析(4)
Linux应用技巧(13)
软件开发(6)
基础技术(13)
- 蓝牙(16)
- ARMv8A Arch(15)
- 显示(3)
- USB(1)

基础学科(10)
技术漫谈(12)



当需要转换VA到PA的时候，首先在TLB中找是否有匹配的条目，如果有，那么我们称之TLB hit，这时候不需要再去访问页表来完成地址翻译。不过TLB始终是全部页表的一个子集，因此也有可能在TLB中找不到。如果没有在TLB中找到对应的item，那么称之TLB miss，那么就需要去访问memory中的page table来完成地址翻译，同时将翻译结果放入TLB，如果TLB已经满了，那么还要设计替换算法来决定让哪一个TLB entry失效，从而加载新的页表项。简单的描述就是这样了，我们可以对TLB entry中的内容进行详细描述，它主要包括：

（1） 物理地址（更准确的说是physical page number）。这是地址翻译的结果。

（2） 虚拟地址（更准确的说是virtual page number）。用cache的术语来描述的话应该叫做Tag，进行匹配的时候就是对比Tag。

（3） Memory attribute（例如：memory type，cache policies，access permissions）

（4） status bits（例如：Valid、dirty和reference bits）

（5） 其他相关信息。例如ASID、VMID，下面的章节会进一步描述。

三、ARMv8的TLB

我们选择Cortex-A72 processor来描述ARMv8的TLB的组成结构以及维护TLB的指令。

1、TLB的组成结构。下图是A72的功能block：

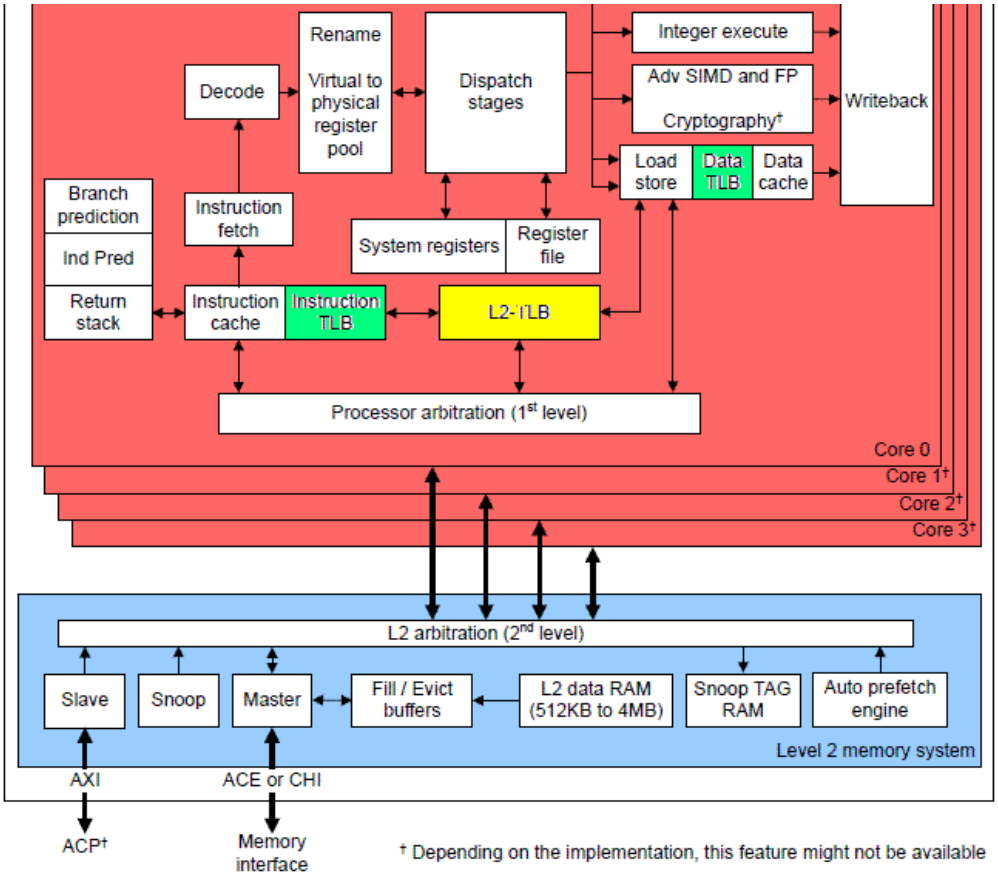
项目专区(0)
X Project(28)

随机文章

Linux PM QoS framework(1)_概述和软件架构
Linux kernel内核配置解析(1)_概述(基于ARM64架构)
蓝牙协议分析(2)_协议架构
eMMC 原理 1 ： Flash Memory 简介
程序员的"纪律性"（《程序员杂志》署名文章）

文章存档

2022年2月(2)
2022年1月(1)
2021年12月(1)
2021年11月(5)
2021年7月(1)
2021年6月(1)
2021年5月(3)
2020年3月(3)
2020年2月(2)
2020年1月(3)
2019年12月(3)
2019年5月(4)
2019年3月(1)
2019年1月(3)
2018年12月(2)
2018年11月(1)
2018年10月(2)
2018年8月(1)
2018年6月(1)
2018年5月(1)
2018年4月(7)
2018年2月(4)
2018年1月(5)
2017年12月(2)
2017年11月(2)
2017年10月(1)
2017年9月(5)
2017年8月(4)
2017年7月(4)
2017年6月(3)
2017年5月(3)
2017年4月(1)
2017年3月(8)
2017年2月(6)
2017年1月(5)
2016年12月(6)
2016年11月(11)
2016年10月(9)
2016年9月(6)
2016年8月(9)
2016年7月(5)
2016年6月(8)
2016年5月(8)
2016年4月(7)
2016年3月(5)
2016年2月(5)
2016年1月(6)
2015年12月(6)
2015年11月(9)
2015年10月(9)
2015年9月(4)
2015年8月(3)
2015年7月(7)
2015年6月(3)



A72实现了2个level的TLB，绿色是L1 TLB，包括L1 instruction TLB（48-entry fully-associative）和L1 data TLB（32-entry fully-associative）。黄色block是L2 unified TLB，它要大一些，可以容纳1024个entry，是4-way set-associative的。当L1 TLB发生TLB miss的时候，L2 TLB是它们坚强的后盾。

通过上图，我们还可以看出：对于多核CPU，每个processor core都有自己的TLB。

2、如何确定TLB match

整个地址翻译过程并非简单的VA到PA的映射那么简单，其实系统中的虚拟地址空间有很多，而每个地址空间的翻译都是独立的：

（1）操作系统中的每一个进程都有自己独立的虚拟地址空间。在各个进程不同的虚拟地址空间中，相同的VA被翻译成不同的PA。

（2）如果支持虚拟化，系统中存在一个host OS和多个guest OS，不同OS之间，地址翻译是不同的，而对于一个guest OS内部，其地址空间的情况请参考（1）。

（3）如果支持TrustZone，secure monitor、secure world以及normal world是不同的虚拟地址空间。

当然，我们可以在TLB匹配过程中，不考虑上面的复杂情况，比如在进程切换的时候，在切换虚拟机的时候，或者在切换secure/normal world的时候，将TLB中的所有内容全部flush掉（全部置为无效），这样的设计当然很清爽，但是性能会大打折扣。因此，实际上在设计TLB的时候，往往让TLB entry包括了和虚拟地址空间context相关的信息。在A72中，只有满足了下面的条件，才能说匹配了一个TLB entry：

（1）请求进行地址翻译的VA page number等于TLB entry中的VA page number

（2）请求进行地址翻译的memory space identifier等于TLB entry中的memory space identifier。所谓memory space identifier其实就是区分请求是来自EL3 Exception level、Nonsecure EL2 Exception level或者是Secure and Non-secure EL0还是EL1 Exception levels。

（3）如果该entry被标记为non-Global，那么请求进行地址翻译的ASID（保存在TTBRx中）等于TLB entry中的ASID。

（4）请求进行地址翻译的VMID（保存在VTTBR寄存器中）等于TLB entry中的VMID

3、进程切换和ASID(Address Space Identifier)

如果了解OS的基本知识，那么我们都知道：每个进程都有自己独立的虚拟地址空间。如果TLB不标识虚拟地址空间，那么在进程切换的时候，虚拟地址空间也发生了变化，因此TLB中的所有的条目都应该是无效了，可以考虑invalidate all。但是，这么做从功能上看当然没有问题，但是性能收到了很大的影响。

一个比较好的方案是区分Global pages （内核地址空间）和Process-specific pages（参考页表描述符的nG的定义）。对于Global pages，地址翻译对所有操作系统中的进程都是一样的，因此，进程切换的时候，下一个进程仍然需要这些TLB entry，因而不需要flush掉。对于那些Process-specific pages对应的TLB entry，一旦发生切换，而TLB又不能识别的话，那么必须要flush掉上一个进程虚拟地址空间的TLB entry。如果支持了ASID，那么情况就不一样了：对于那些nG的地址映射，它会有一个ASID，对于TLB的entry而言，即便是保存多个相同虚拟地址到不同物理地址的映射也是OK的，只要他们有不同的ASID。

切换虚拟机和VMID的概念是类似的，这里就不多说了。

4、TLB的一致性问题

TLB也是一种cache，有cache也就意味着数据有多个copy，因此存在一致性（coherence）问题。和数据、指令cache或者unified cache不同的是，硬件并不维护TLB的coherence，一旦软件修改了page table，那么软件也需要进行TLB invalidate操作，从而维护了TLB一致性。

5、TLB操作过程

我们以一个普通内存访问指令为例，说明TLB的操作过程，在执行该内存访问指令的过程中，第一件需要完成的任务就是将要访问的虚拟地址翻译成物理地址，具体操作步骤如下：

（1）首先在L1 data TLB中寻找匹配的TLB entry（如果是取指操作，那么会在L1 instruction TLB中寻找），如果运气足够好，TLB hit，那么一切都结束了，否者进入下一步

（2）在L2 TLB中寻找匹配的TLB entry。如果不能命中，那么就需要启动hardware translation table walk了

（3）在执行hardware translation table walk的时候，是直接访问main memory还是通过L2 cache 访问呢？其实都可以的，这和系统配置有关（具体参考TCR_ELx）。如果配置的是Normal memory, Inner Write-Back Cacheable，那么可以在L2 cache中来寻找page table。如果配置的是Normal memory, Inner Write-Through Cacheable或者Non-cacheable，那么hardware translation table walk将直接和external main memory。

6、维护TLB的指令

我们将在下一章，配合linux的标准TLB flush接口来描述。

四、TLB flush API

和TLB flush操作相关的接口API主要包括：

1、 void flush_tlb_all(void)。

这个接口用来invalidate TLB cache中的所有的条目。执行完毕了该接口之后，由于TLB cache中没有缓存任何的VA到PA的转换信息，因此，调用该接口API之前的所有的对page table的修改都可以被CPU感知到。注：该接口是大杀器，不要随便使用。

对于ARM64，flush_tlb_all接口使用的底层命令是：tlbi vmalle1is。Tlbi是TLB Invalidate指令，vmalle1is是参数，指明要invalidate那些TLB。vm表示本次invalidate操作对象是当前VMID，all表示要invalidate所有的TLB entry，e1是表示要flush的TLB entry的memory space identifier是EL0和EL1，regime stage 1的TLB entry。is是inner shareable的意思，表示要invalidate所有inner shareable内的所有PEs的TLB。如果没有is，则表示要flush的是local TLB，其他processor core的TLB则不受影响。

flush_tlb_all接口有一个变种：local_flush_tlb_all。flush_tlb_all是invalidate系统中所有的TLB（各个PEs上的TLB），而local_flush_tlb_all仅仅是invalidate本CPU core上的TLB。local_flush_tlb_all对应的底层接口是：tlbi vmalle1，没有is参数。

2、 void flush_tlb_mm(struct mm_struct *mm)。

这个接口用来invalidate TLB cache中所有和mm这个进程地址空间相关的条目。执行完毕了该接口之后，由于TLB cache中没有缓存任何的mm地址空间中VA到PA的转换信息，因此，调用该接口API之前的所有的对mm地址空间的page table的修改都可以被CPU感知到。

对于ARM64，flush_tlb_mm接口使用的底层命令是：tlbi aside1is, asid。is是inner shareable的意思，表示该操作要广播到inner shareable domain中的所有PEs。asid表示该操作范围是根据asid来进行的。

3、void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr)。

flush_tlb_page接口函数对addr对应的TLB entry进行flush操作。这个函数类似于flush_tlb_range，只不过flush_tlb_range操作了一片VA区域，涉及若干TLB entry，而flush_tlb_page对range进行了限定（range的size就是一个page），因此，也就只是invalidate addr对应的tlb entry。

对于ARM64，flush_tlb_page接口使用的底层命令是：tlbi vale1is, addr。va参数是virtual address的意思，表示本次flush tlb的操作是针对当前asid中的某个virtual address而进行的，addr给出具体要操作的地址和ASID信息。l表示last level，也就是说用户修改了last level Translation table的内容（一般而言，PTE就是last level，在某些情况下，例如section map，last level是PMD），那么我们仅仅需要flush最后一级的页表（page table walk可以有多级）。

4、 void flush_tlb_range(struct vm_area_struct *vma, unsigned long start, unsigned long end)。

flush_tlb_range接口是一个flush强度比flush_tlb_mm要弱的接口，flush_tlb_range不是invalidate整个地址空间的TBL，而是针对该地址空间中的一段虚拟内存（start到end-1）在TLB中的entry进行flush。

ARM64并没有直接flush一个range的硬件操作接口，因此，在ARM64的代码中，flush一个range是通过flush一个个的page来实现的。

五、参考文献

1、Cortex A72 TRM

2、ARM ARM

3、Documentation/cachetlb.txt

原创文章，转发请注明出处。蜗窝科技

标签: TLB flush

« Linux TTY framework(2)_软件架构 | Linux TTY framework(1)_基本概念»

评论:

fear
2019-01-31 16:38

这篇文章太经典了，解答了很多问题，感谢大佬。

回复

linux小白
2017-07-06 11:30

我理解tlb中的table硬件不会修改，页表内容修改是操作系统来执行的，当tlb和linux中页表内容不匹配时候应该是invalid tlb中的表项，怎么叫flush_tlb_all(flush)呢， 这个函数名意思是用tlb中页表内容替换linux中页表?

回复

linuxer
2017-07-07 09:28

@linux小白：在这里，flush就是invalid的含义。

回复

linux小白
2017-07-07 11:15

@linuxer：@linuxer :命名不严谨啊，容易误导人，应该叫invalid_tlb_all....

回复

pursue4meaning
2017-02-02 00:04

写的非常清楚，涉猎广泛，博而不杂，佩服!

回复

madang

2017-01-14 11:36

IC设计工程师怎么会忽略step （1）和step （2）呢
--》 这里应该是step(1) 和 step（3）吧

回复

linuxer

2017-05-15 17:27

@madang：多谢指正。
呵呵，今天突然自己重新阅读了这篇文档，发现了这个错误，觉得很奇怪：为何这么明显的错误没有人提出来，拉到评论区，发现已经有网友提出了，只是我之前可能是错过了而已，呵呵～～～

回复

发表评论：

昵称

邮件地址 (选填)

个人主页 (选填)

发表评论