**PRV464S/464处理器**

**技术手册**

**第一部分 总体介绍**

**1.1 总体概览**

PRV464S 是一款全新的兼容RISCV指令集的处理器，将会具备4KByte L1指令缓存，设计工作频率66+MHz，工作频率远高于PRV332的33MHz，在同频下可以达到PRV332十二倍+的性能。

PRV332（33MHz） 5MIPS（不开启mmu），1.5MIPS（开启MMU）

PRV464 (66MHz） 66MIPS（不开启MMU），66MIPS（开启MMU）

\*MIPS：百万条指令每秒。

\*以上性能参数是理论估计最佳值。

PRV464将会有4级流水线，4KB+ 4KB的全速L1缓存（16-way，256Byte/way），L1带宽为240MB/s，支持RV32IMA（可能会有C）其流水线分别为：

IF (取指令) ID(译码）EX（执行）WB（写回）

**1.2 PRV332处理器实现教训**

PRV332使用状态机实现，执行周期为6T，速度缓慢，并且由于逻辑设计失误导致扇出过高，总结下来有以下问题：

1、内存访问速度非常缓慢，没有cache机制，4T的时间浪费在总线上的数据传输，尤其是当开启MMU时，页表的多次查询导致速度进一步下降。

2、执行周期设置不合理，总体上还是类似PRV32的多执行周期，

3、CSR部分的RTL代码设计问题，导致CSR部分扇出太高，频率无法提升。

4、异常处理机制设置不正确，导致处理器遭遇异常后白白浪费周期。

**1.3 PRV464S实现方案**

PRV464S处理器将会使用四级流水线，IF，ID，EX，WB四级，其中IF可以在1个clk内完成，ID需要1个clk（当需要对一条指令解码成两条），EX可能需要1个或更多周期（当移位，内存访问，乘除法，原子指令等）。WB需要一个clk。

PRV464采用四级流水线，巧妙地规避了流水线中的一些问题：如当取数据发生异常，取指令应该是在等待状态，因为cache控制器一次只能处理一边的缓存不命中。在有访存指令存在在流水线里的时候，处理器流水线里最多包含两条指令（EX一条，ID一条)

**1.3.1 PRV464S解决数据相关性问题**

为了解决数据相关性问题，在ID阶段就会对RS1，RS2，RD，CSR进行译码，获得当前指令的RS1，RS2，RD，CSR等信息，如果上述的RS1，RS2，CSR与当前流水线中的正在执行的指令的RD，CSR重合，则在ID和EX间插入气泡，即valid（指令有效位）=0，同时IF将不会取指令，但是IF输出数据将保持，时序如下图所示：

**1.3.2 PRV464S解决异常和中断问题**

为了解决异常和中断处理的问题，PRV464S在每级流水线之间传输的信号加入了控制段，控制段包含（但不限于）：

exc组：

ecall： 环境调用

ebreak： 环境断点

ins\_page\_fault: 指令页面错误。

ins\_acc\_fault: 指令访问错误。

ins\_addr\_mis: 指令地址不对齐。

ill\_ins: 错误指令。

load\_page\_fault: 加载页面错误。

load\_acc\_fault: 加载访问失败。

load\_addr\_mis: 加载地址不对齐。

store\_page\_fault: 存数页面错误。

store\_acc\_fault: 存数访问错误。

store\_addr\_mis: 存数地址不对齐。

soft\_int: 软件中断

timer\_int: 定时器中断

ext\_int: 外部中断

value组：

pc[31:0] : 目前指令PC值

ins[31:0]：目前指令的编码

addr[31:0] : 目前指令访问的内存地址

当发生中断，控制段的exc组中的某个位将会被置1，例如当在IF阶段发生了指令页面错误，那么ins\_page\_fault这个位将被置1，同时vlid位将会被置0，然后传递给ID，ID随后传给EX，EX随后传给WB。在从IF传递到WB的过程中，valid位始终控制着指令是否有效，若valid位为0，则不管其他位如何，这一级流水线都不做任何操作（除了WB级）。 当指令传递给WB级之后，WB级将会自动更改相关寄存器的值，包括mepc，metval等和异常相关的寄存器。

\*注意，RISCV定义中，发生ecall和ebreak的时候，xepc里面存的值为下一条指令，其他时候都是当前指令。

注意在上述过程中，IF将异常指令传递给ID之后，IF将会空拍，ID传递给EX之后ID也继续空拍，直到该指令被传递给了WB，控制段里的exc段没有任何一位是1的时候，IF才会再次开始取指令。

当处理器执行了会影响PC的指令（如RET指令），处理器会停止继续取指直到RET指令被正确写回（修改了PC和权限成功）

**1.3.3 PRV464S解决fence指令问题**

当处理器IF阶段得到fence指令的时候，处理器将会等待流水线中每一条指令执行完成之后（即fence指令走到了WB阶段），再继续开始取指令。

为此新增加每级流水线之间传输的段： fence段

包含信号：

fence.i

fence

fence.vma

例如，当处理器取到了一条fence.i指令，经过ID之后发现是一条fence.i指令，在下一个周期IF将会暂停取指令，当fence.i指令走到ex阶段，cache控制器将会同步L1I和内存。等待fence.i指令走到WB阶段之后，IF继续取指令。

当处理器取到了一条fence指令，处理器会停止取指令，经过ID解码之后发现是一条fence指令，则fence执行到ex阶段时，缓存会进行刷新（和内存同步），缓存行中所有tag=1的行（被修改过）都会被重新刷入内存，同时tag被清零。

当处理器取到了一条fence.vma指令，则处理器会停止取指令，等到fence.vma指令走到EX阶段时，cache控制器会根据当前缓存的VA-PA映射表重新在内存中找到对应的PT再次刷新L1.

解决fence指令，若处理器执行了fence指令，则cache控制器将会把cache中所有tag=1的行写回内存，并将tag重新置0以表示没有被弄脏。

fence指令处理方式同上。

**1.3.4 PRV464S处理器解决L1替换问题**

PRV464S处理器具备512\*64bit的L1指令缓存和512\*64bit的L1数据缓存，一次恰好存放一个页面，这4KB的L1I被分为16\*32\*64bit的区域，共32行（命名为I00-I31，D00-D31），每行128Byte。

当处理器取指令的时候，L1缓存控制器根据IF给出的VA，如果命中，则直接给指令，若没有命中，chache控制器将查询主存中的页表，根据页表翻译出当前指令页面，然后使用32次burst将当前指令页全部存入L1中。 对于较短的程序可以获得较高的指令执行速度提升。

当处理器遭遇了fence.i指令，处理器的IF会停止直到fence.i指令被完全执行成功。

当处理器EX段要访问内存时，首先向cache控制器给出需要访问数据的地址，如果cache控制器命中（数据可能在L1I中，也有可能在L1D中），则在相应cache的行上的脏位置1（I00到I31,D00到D31一共64个tag），并立即将数据送给EX。若没有命中，cache控制器给出等待信号，首先访问内存页表，进行VA-PA转换后判断该地址是否为可缓存地址，如果是可缓存页则先查询L1D中tag为1的行，将tag为1的行写回内存，然后缓存该页面。如果是不可缓存的地址，则L1D不动，直接写入内存。

L1控制器外部将会留配置接口，配置接口可以配置处理器在哪些地址下可以缓存，哪些地址不能缓存

**1.3.5 PRV464S处理器解决跳转指令(包含Bxx，JALx）**

**解决方案1：**PRV464S处理器使用静态预测来预测分支，预测每一次跳转为跳，同时增加控制段： BP段：

增加信号：

take： 表示这条指令是否跳转（只有当执行beq，bne等指令的时候这个位有效）

当执行分支指令时，若在EX阶段发现并无跳转，则take位置一，当分支指令进入WB阶段时，将会重新更改PC值，同时进行流水线冲刷使预测错误的指令全部失效。

**解决方案2：**不使用分支预测，遇到分支指令延迟一个周期，IF之后空一拍，ID获得了RS1和RS2之后就可以快速比较数据生成PC地址，然后马上修改PC值取指令。 这种方法需要延迟一个周期，在最好/最坏情况下都会固定的消耗一个CLK。

**1.3.6 PRV464S解决原子拓展指令的问题**

为了支持原子指令，PRV464S选择在ID阶段对A指令进行解码拆分成两条指令：读内存（指令甲）+写内存（指令乙）。甲乙两条指令在内部会被判定为有数据相关，因此只有当甲指令被执行完成之后乙指令才会被执行。

当处理器正在执行原子指令时，是不允许被打断的，在处理器进行atom操作时，处理器不会受理任何中断。因此在程序看来，该指令被原子的执行了。

**1.3.7 PRV464S解决burst指令数据大小的问题**

AHB-LITE总线支持未定义长度的burst操作，本处理器实现里面将会使用单次传输（NONSEQ）+未定义长度的burst传输。单次传输会使用8/16/32/64位访问，未定义长度的burst传输本处理器也只会使用增量为8的burst。

**1.3.8 PRV464解决处理器状态转换问题**

目前已知的可能造成处理器权限变更的情况是修改CSR，为了简化实现方案，降低可能出现的漏洞，464处理器对所有csr修改指令（csrrw ，ret，ecall，ebreak等）都当作单次执行处理。可以遇见的是，处理器修改csr指令只会在流水线中存在很小一段时间，因此带来的性能降低可以忍受。

**1.3.9 PRV464处理器解决PMP检查**

为了使用RISCV架构中定义的PMP检查，PRV464在cache控制器中加入物理地址检查，当处理器访问L1时，使用的VA首先被送到L1，L1包含了该VA对应的PA，然后送往PMP检查，如果PMP检查出错，则BIU给出

(目前暂不实现PMP，pmp寄存器被保留但是不生效）

**第二部分 模块介绍**

**2.1 BIU单元**

**2.1.1 cache\_ctrl单元**

cache\_ctrl单元主要任务是负责L1I/D的存取，当发生cache未命中的时候向mmu提交内存访问请求，如果mmu反馈可以进行缓存，则cache\_ctrl先将L1中tag=1的行输出给mmu写回内存，然后使用burst将数据页载入cache。如果该页面不能被缓存，则cache\_ctrl直接送出要访问的数据直接写内存。

**2.1.1.1 cache\_ctrl单元与mmu之间的通信**

当IF/EX单元尝试访问L1缓存而没有命中，或者是执行了fence指令需要同步缓存和内存时候，cache\_ctrl会向mmu提出内存访问需求。信号如上。

以上信号主要被分为三个部分：全局信号，读通道信号，写通道信号。其中**命令通道信号**包含： paddr(物理地址指示），wr（写请求），rd（读请求），sync\_mem(内存同步），pa\_cov(物理地址转换)，cache\_rdy(cache准备好信号），

**应答信号**包含：cacheable（可以缓存）trans\_rdy（传输准备好） trans\_over（传输完成信号） bus\_error（总线错误）, page\_fault（页面错误）。

**读通道信号**：read\_data[31:0]（读数据）

**写通道信号：**write\_data[31:0](写数据)

目前定义的命令如下：

**pa\_cov**：物理地址转换，maddr传输的是虚拟地址，转换完成后将会返回物理地址，页表项，和是否缓存信号。

**pa\_cov+paddr**：物理地址检测，将会检查当前的物理地址是否可以缓存。

**wr+paddr**：写内存，单次访问内存时候。

**rd+paddr**：读内存，单次访问内存使用。

**rd+paddr+cache\_rdy**：使用物理地址，读内存，批量读入。

**wr+paddr+cache\_rdy**：使用物理地址，写内存，批量写入。

**trans\_over**：传输结束，在批量传输的过程中才有效，可以结束当前的SEQ传输。

\*其余组合将会保留用途

下面以一次读内存为例：T1：cache控制器给出rd信号，同时在maddress给出地址（使用paddr来标志该地址是否为物理地址，默认情况下这时应给出VA），然后等待mmu响应。假设Tn时mmu给出响应，page\_fault和bus\_error会直接中断当前传输，如果页面正常可以传输，mmu给出trans\_rdy信号并给出cacheable信号，以及转换后的PA，从read\_data中送出当前页面属性。此时cache控制器会有三个选择：

1、给出rd信号，然后送出size信号，mmu直接根据刚才译出的PA进行单次访问，访问完成后给出trans\_rdy并通过read\_data送出数据，cache控制器收到数据后通过trans\_over信号结束传输。

2、给出sync\_mem信号和cache\_rdy信号，同时通过maddress给出当前cache映射内存的PA，表示cache已经准备好要和内存同步。mmu随即操作AHB总线进入批量模式。每当一行传输完成（即传输完成32次burst，mmu自动的停止批量传输，等待cache控制器再次给出sync\_mem或者其他传输），cache控制器会再次给出sync\_mem信号表示需要再次开启一个新的批量传输，以此类推。 当传输完成后会给出trans\_over信号以完成当前传输。结束传输后，cache需要重新载入内存，此时cache控制器给出rd+paddr+cache\_rdy信号表示进入批量读内存模式，mmu直接使用该PA(该PA是刚才步骤1中获得的PA）开启AHB的批量传输，传输开始后给出trans\_rdy信号开始AHB的刷新。当一行（32个4byte传输）结束后，cache控制器会重新开始一次新的批量传输（trans\_rdy拉低以等待开启新的传输）。当传输完成后cache控制器会给出trans\_over以结束整个访问。

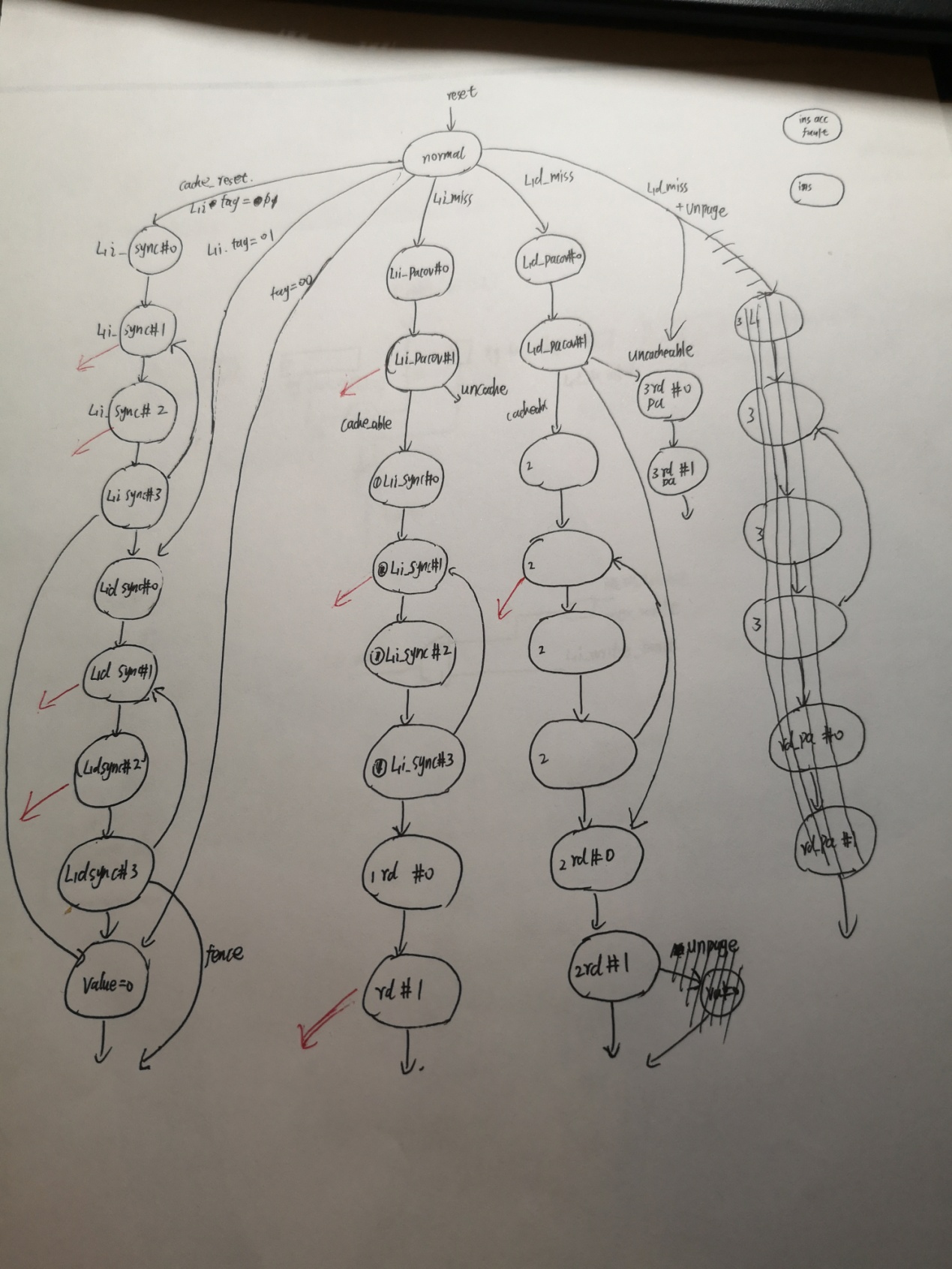
3、直接给出rd+cache\_rdy信号，并给出PA，直接开启一个新的burst传输，每行传输完成之后，mmu自动的开启下一行的传输，同时cache控制器也自动的开始下一行的输入。

cache控制器在每个clk会检查IF需要的地址和当前正在查询的地址的区别，如果IF给出的地址发生了改变（有可能处理器在IF的时候处理器更改了PC)

\*注意，在上述批量读入内存的过程中，cache中地址计数器和mmu中的传输应该保持一致，需要多加注意。

**2.1.1.2 cache控制控制逻辑**

cache控制器逻辑包含如下状态转换图：



\*normal: 正常模式，L1控制权在IF和EX手中，cache控制器监控。

\*l1isync0: 内存同步状态0。

当L1I和L1D都命中时，cache控制器逻辑在normal（正常）状态，此时L1I和L1D都交给CPU内核控制，cache控制器只负责检查当前内存访问时的地址和访问权限。

当ex给出cache\_flush信号时，cache控制器转到内存同步（l1isync、l1dsync）上，此举可以将L1中的数据全部同步到内存之中，根据L1I D 中被修改的情况不同（tag不同） cache控制器会选择跳到哪一段进行刷新。

当进行完L1D写回之后，根据是否有cache\_reset信号（即是否需要重新载入页面）调整，如果需要重新载入页面，cache控制器跳到valu\_clr状态上，此状态可以清除当前映射有效位，下次处理器访问内存时候会引起重载入缓存。

红线部分为可能出现问题的时候的状态转移，根据不同状态下遭遇的异常，状态会转移到ipf，iaf，lpf，laf，spf，sf六种状态，分别为：指令页面错误，指令访问错误，load页面错误，load访问错误，store页面错误，store访问错误。

默认的，当处理器缓存同步到内存时不发生错误，因为当前缓存中的内容是此前被载入到缓存中的，若此前载入的时候没有发生错误那么在写回的时候也不应该发生错误（使用同样的物理地址）如果发生了异常，那么这个异常直接会造成当前访问异常。

cache控制器负责检查当前访问的内容是否在缓存内，因此cache控制器里面有L1IVA，L1DVA两个寄存器标识当前缓存内映射的内容。

cache控制逻辑同时负责检查当前指令/数据访问的页面检查，页面检查分为两种：未命中的时候查询主存页表时的页面检查、命中之后的利用缓存快表进行页面检查。

**2.2 IF单元**

IF单元输入输出信号:

module ins\_fetch(

input wire clk,

input wire rst,

//后级模块准备好信号

input wire ready\_id,

input wire ready\_ex,

input wire ready\_wb,

//插空信号

input wire nop\_if,

//fence指令请求刷新信号

input wire cache\_flush\_input,

//对cache控制器信号

output wire [31:0]addr,

output wire rd, //取指令信号

input wire [31:0]ins\_in,

input wire acc\_fault, //访问失败信号

input wire cache\_ready, //cache准备好信号

output wire cache\_flush, //cache刷新信号

//对下一级（ID）信号

//指令输出

output wire [31:0]ins\_out,

//指令对应的PC值输出

output wire [31:0]pc,

//机器控制段

output reg ins\_acc\_fault, //指令访问失败

output reg ins\_addr\_mis, //指令地址错误

output reg ins\_page\_fault //指令页面错误

output reg int\_acc, //中断接收信号

output reg valid, //指令有效信号

IF单元主要负责指令的存取，它与cache控制器直接通信

**第四部分 Sv39CT分页方案**

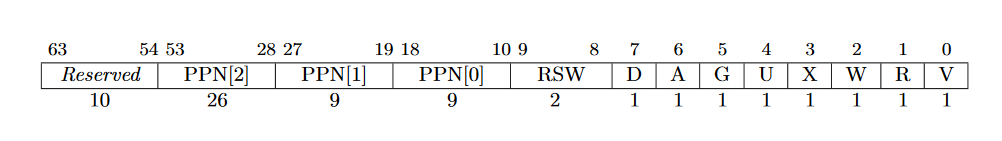
**4.1 概述**

为了解决现有的Sv39分页方案（以下简称原方案）无法动态配置可缓存页面和无法动态改变cache更新策略（类似X86保护模式下的PCD和PWT），本处理器提出更新的分页方案：Sv39CT（以下简称新方案）。

SiFIVE原有的分页方案中并没有增加这两个选项，将cacheability作为PMA的一份特性固定在了处理器设计中，这样就会导致处理器在使用PnP设备（如显卡）的时候，cache映射可能会被更改的PnP设备的存储器的时候（如映射显存，显卡可能会更改显存）的时候，需要cache也及时的更新。虽然这个特性可以使用更新的总线特征（如SiFIVE公司tilelink总线，可以支持向所有从设备广播自己修改了什么地方以保证cache一致），但是目前并没有多少IP支持这种总线，目前内置设备中广泛使用的AHB AXI总线等并不支持这个特性。而X86/ARM使用动态可缓存，写透两个位来保证cache一致已经经历了长久的工程实践。综上所述我们需要更新的分页方案来支持这个特性。

**4.2 技术特点**

新分页方案增加C，T两个位以增加更多的页面特征。原方案中的63-54位均为保留位，因此我们可以将63位设为C，62位设为T。



C: cacheable，表示该页面可以进行缓存，为了保持与之前的方案兼容，即使用原方案的操作系统可以无缝在使用新方案的系统上运行，当C位为1的时候表示该页面可以缓存。

T: Write Through，为1时表示该页面需要写透。为0时则使用原有的cache更新策略。

注意，新方案并不与原有的固定cacheability方案冲突，页面的缓存特性使用逻辑表达式为：C位为1 | PMA规定可缓存。同时，T位也与处理器实现方案采用的更新策略不冲突，也就是如果某处理器实现的cache采用WB方案，则当T位为1时强制要求WT，为0时不强制要求。如果某处理器实现只使用了WT方案，则T位不起作用。