RUNMAP设计方案

# 一、RUNMAP设计总览

本技术的设计目标是在混合内存系统上，通过hint page fault技术，追踪内存密集型应用的内存访问请求情况，根据内存访问局域性在异构内存系统上实现高效的数据分布和迁移。

本技术方案设计分为一下几个部分：

1. 更高效的页面扫描机制，用于捕获内存访问信息。同时为了降低扫描开销，设计了页面筛选机、热节点与热页面扫描频率控制机制，此为本设计的核心创新点。

2. 页面热度信息记录机制，用于存储内存访问信息。

3. 冷热页面识别机制，基于内存访问信息识别页面的冷热程度。与其他方法相比，本设计中的冷页面识别方案更为积极和准确。

4. 页面热度直方图机制。为了更直观地刻画各个内存节点的页面热度分布，便于动态调整冷热阈值，对页面的冷热信息以直方图的形式进行了统计。

5. 页面迁移机制，动态地调整冷热页面布局。同时为了应对内存压力非常紧张的场景，设计了候选页面机制。

# 二、基于物理页面的页面扫描机制

## 2.1 AutoNUMA扫描机制中的问题

AutoNUMA（也被称为numa balancing）机制是一种用于优化多处理器系统中NUMA架构的内存访问的机制。AutoNUMA的核心设计是定期扫描指定进程的地址空间，然后进行页表遍历，修改这些地址对应页表项（PTE，Page Table Entry）的一些权限位。后续CPU访问这些被标记的地址时会触发权限保护位错误，进入内核的缺页中断处理程序，这类缺页中断被称为numa hint page fault，或者hint page fault。通过hint page fault，内核可以成功捕获到CPU的访存信息。

理想情况下，AutoNUMA的扫描会标记页面然后获得访问信息，即使每次标记都要进行页表遍历操作，这也是可以接受的。然而，我们通过分析AutoNUMA的扫描机制发现其存在许多低效的扫描。如下图所示：

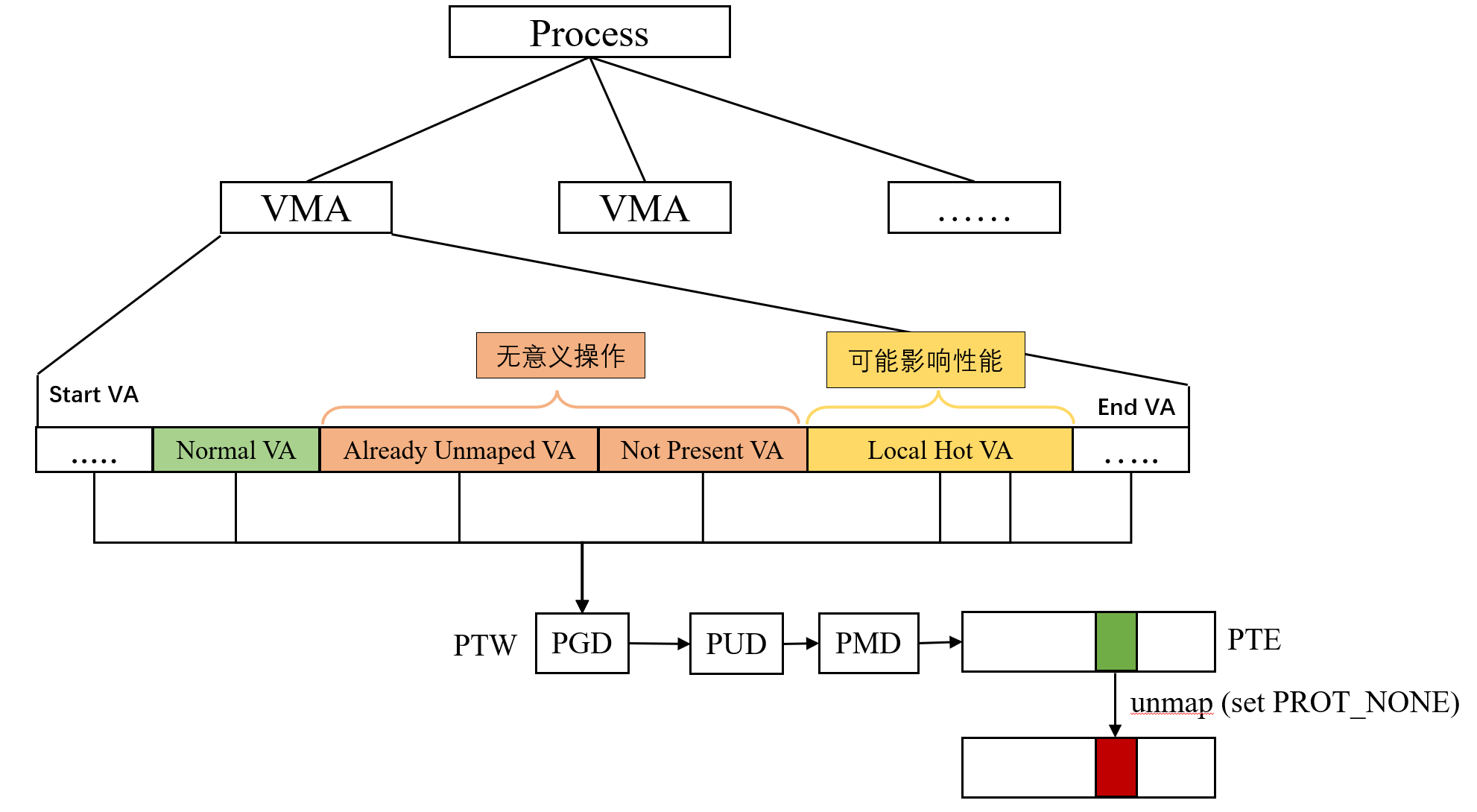


图 1 AutoNUMA的扫描机制中的问题

其中，

（1）“Normal VA”表示未被unmap的虚拟地址，对这类地址进行unmap操作是比较合理的，可以捕获他们的访问信息。

（2）“Already Unmapped VA”表示已经被unmap过的虚拟地址，这类地址已经处于unmap状态，所以AutoNUMA会直接跳过。

（3）“Not Present VA”表示未加载到内存中的虚拟地址，它没有对应的物理页面，因此AutoNUMA也会直接跳过。

（4）“Local Hot VA”表示已经被放到本地DRAM节点的热页面，对于这类热页面，它们已经被放到最快的local dram上了，频繁地对其进行unmap操作不会产生收益，只会造成page fault开销，打断cpu的正常访问。

进一步地，这些低效的扫描机制限制了AutoNUMA的扫描频率上限，即因为许多无效的扫描开销导致扫描频率无法进一步提高，而较低扫描频率会导致页面识别准确率下降，影响页面布局。

## 2.2 探究AutoNUMA扫描问题的解决方法

针对AutoNUMA扫描机制中的问题，有一些“逻辑上的”的解决方法，但是都存在一些问题：

（1）针对“Already Unmapped VA”，可以提前对每个VA设置一个flag，标记这个VA是否处于标记状态。但问题在于：进程的虚拟地址空间一般都特别大，除了pte，内核并没有为每个VA都维护一个数据结构，。

（2）针对“Not Present VA”，最好是可以直接跳过。但问题在于：VA本身没有该信息；

（3）针对“Local Hot VA”，

1）如果已经有大量的热页面，可以减少该节点的扫描频率，避免干扰进程的正常数据访问操作。但问题在于我们无法知道当前VA对应的页面是在哪个内存节点上，因此就无法控制某个内存结点的扫描频率；

2）对于过去刚被取消标记的页面，可以设置一个等待阈值，过了这个时间之后再进行标记，避免频繁标记同一个页面。但问题和前面的flag无法添加一样，除了pte，没有位置添加这个数据结构。

## 2.3提出基于物理页面的页面扫描机制

因此，本设计提出基于物理页面的页面扫描机制，即通过扫描物理页面来对各个页面对应的pte进行标记/unmap操作，它可以很容易地解决前面提到的问题：

（1）针对“Already Unmapped VA”，可以在struct page中添加flag信息；

（2）针对“Not Present VA”，struct page天然可以跳过，因为如果数据没有被加载到内存中，就不会存在对应的struct page；

（3）针对“Local VA”，struct page中包含了内存节点信息，且每个内存节点均有自己独有的lru list，因此我们可以对不同的lru list采取不同的扫描频率。

（4）针对“Hot VA”，我们可以在struct page中添加时间戳信息，用来记录页面热度信息以及扫描信息。

不过需要注意，由于内核中的struct page已经非常拥挤，因此在实现过程中，我们并没有直接基于struct page，而是通过struct page\_ext间接地扩充struct page内容。

## 三、页面热度信息记录机制

为了记录页面热度信息，RUNMAP基于struct page\_ext为每个page添加了一个page\_info元数据，page\_info里面主要包含了如下内容（此处仅描述热度信息相关内容，不包括其他机制用到的内容）：

（1）PageUnmapLRU标志位。用来标记页面是否处于unmap状态；

（2）时间戳字段。

1）对于标记状态的页面，该字段用来记录标记时刻；

2）对于刚被取消标记状态的页面，该字段用来记录上次标记的时刻，用来控制其标记频率，即避免同一个页面被频繁标记。

## 四、冷热页面识别机制

## 4.1 tiering介绍

tiering是基于intel基于AutoNUMA设计的一个优化tieried memory的方案。tiering中识别冷热页面的机制如下：

1、在unmap pte时，在page结构体中记录初始时间戳；

2、CPU访问unmap状态的pte时，触发page fault，进入OS的中断处理程序；

3、获取当前时间戳，计算时间间隔，小于一定阈值将其识别为hot page。

这个流程即如下图所示：

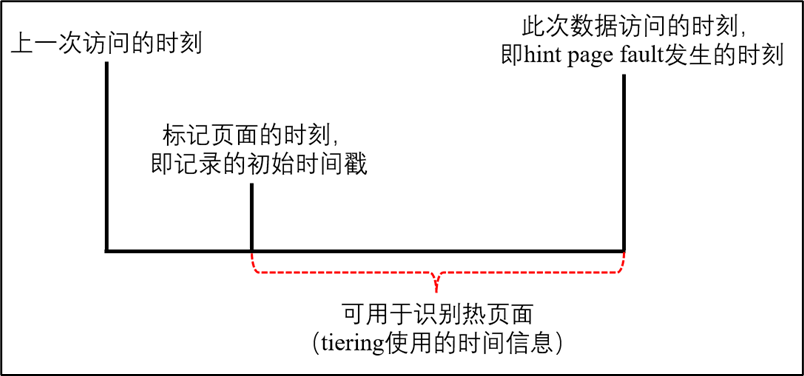


图 2 基于时间戳识别热页面（tiering）

## 4.2 tiering存在问题以及优化方案

然而，tiering中存在一些问题：

1、tiering依赖于AutoNUMA的unmap机制，这部分问题同前面所述；

2、tiering的demote机制仍依赖kswapd。这存在两个问题，一方面，kswapd只在内存压力紧张的情况下才会触发，此时做demote操作比较晚；另一方面，kswapd基于粗粒度的lru list。

关于第二个问题，我们注意到，其实时间戳信息除了可以用来识别热页面外，还可以用来识别冷页面。如下图，当页面一直处于标记状态时，说明这个页面一直没被访问过，那么我们定期扫描页面，如果这个页面处于标记状态，就可以计算时间间隔，来判断该页面的冷热程度。

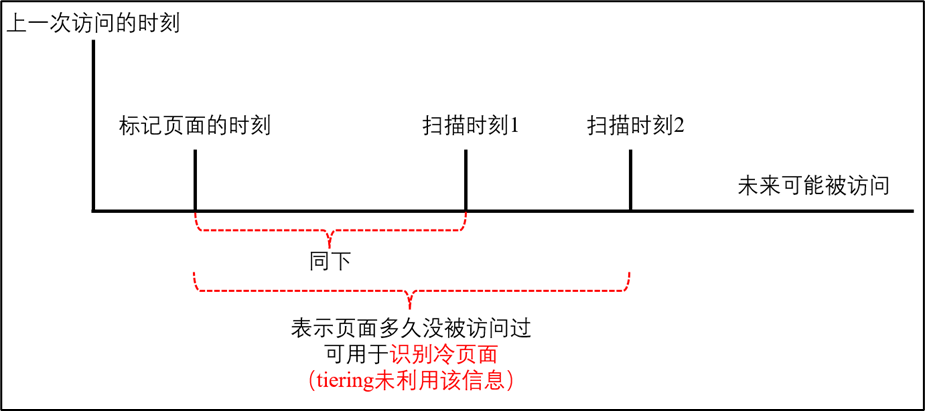


图 3 基于时间戳识别冷页面（提出优化）

## 4.3 冷热页面识别机制

本方案在tiering利用时间戳识别“热页面”的基础上，添加了“冷页面”的识别机制。该机制与“扫描access bit的相关方案”类似，但是得益于前面提出的PageUnmapLRU页面标志，我们无需扫描pte来读取access bit，就可以直接通过struct page得到页面的冷度信息。

## 五、页面热度直方图机制

## 5.1直方图的数据结构

为了对各个内存节点上所有页面有一个整体的热度分布有一个直观的数据感受，我们统计了内存节点上各个页面的时间间隔（包含两种时间间隔：hint page fault得到的时间间隔，以及多次扫描得到的时间间隔），然后通过直方图这种数据结构来展现。

具体来说，

（1）用一个计数数组来表示，索引为i的数字表示“访问时间间隔为2^i秒的页面有多少个”（通过使用指数计数来限制直方图宽度）。

（2）在计算得到上述两种时间间隔后，将其存到之前提到的page info中，然后用一个page flag（PageHistogram）表示这个页面是否被直方图记录。

如下图所示：

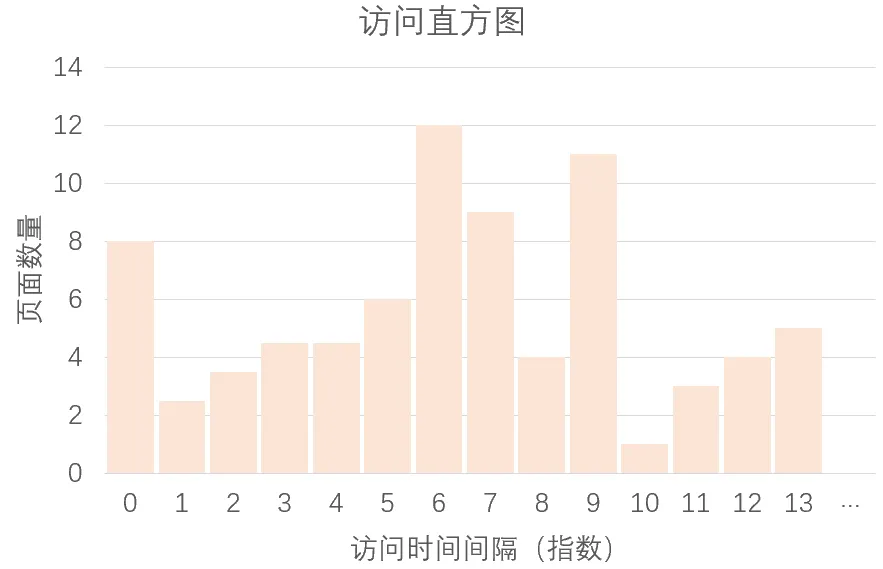


图 4 访问直方图示例

## 5.2 直方图的更新

在得到时间间隔后，通过PageHistogram来判断页面是否位于直方图中。如果该页面没有被加入，就加入到直方图；如果已经存在于数据直方图中，就根据间隔是否发生变化来调整直方图的计数情况。

## 5.3 访问直方图的使用

在判断页面热度情况的时候，可以计算一下当前页面的热度是否超过了数据直方图中一定比例的页面（目前设定的静态比例为7/8）。例如，（逻辑上来说）假设当前页面访问时间间隔为x，计算一下直方图中访问间隔大于这个间隔的页面的数目，然后计算一下比例，进而确定冷热情况。

这种方法的优点在于，这是一种相对热度，可以获得整个页面在整个节点中的热度情况，弥补静态热度阈值的不足。

## 六、页面迁移机制

## 6.1 常规的页面promote机制

常规页面在触发hint page fault后，RUNMAP会获取其节点信息，同时判断其页面是否是热页面。如果该页面是热页面，且位于PM或者远端内存节点，就会尝试将其promote到local DRAM中。

## 6.2 积极的页面demote机制

正如前文提到的，我们通过时间间隔信息可以确定一个页面多久没有被访问过，如果确定这个页面是冷页面，那么可以积极地将其demote到PM或者远端内存节点，而无需等待kswapd中的demote机制。

积极的demote机制可以移除local dram中的冷页面，从而空出足够多的空间给热页面进行promote操作。

## 6.3 内存压力紧张时的候选页面延迟迁移机制

如果内存压力非常紧张，紧张到PM节点和远端内存节点几乎都没有空闲空间，那么demote机制也无济于事，因为demote本身也属于迁移页面操作，如果目标节点（PM或远端节点）内存已满，将无法成功demote。因此设计了一个候选页面延迟迁移机制，在local DRAM无法成功demote页面时，将页面放入一个待迁移的队列中。然后在有热页面触发hint page fault，需要promote时，就执行两次迁移操作，即将热页面promote到local DRAM，同时将选择一个候选页面demote到这个热页面原本位于的节点。

具体地，我们设计了在page info中添加了一个struct list\_head字段，用来将各个候选页面组成一个链表，同时增加了一个PageCandidate标志位来标志其是否被选为候选页面。