# **低时延交易设计 之 大页内存**

原创 愚夫 [愚夫一得](javascript:void(0);)

 2025年03月01日 23:37 上海

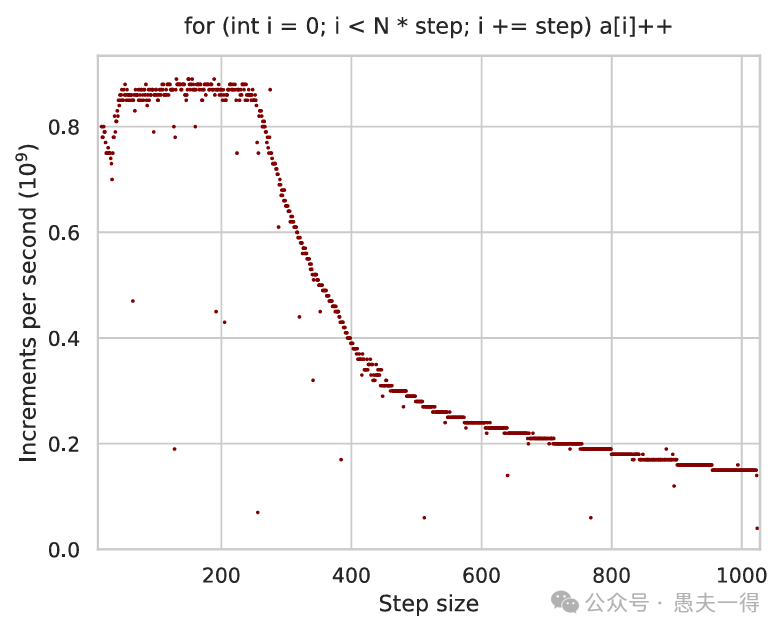
与内存相关的低延迟技术当中，大家伙谈的较多的是CPU缓存、亲缓存性、内存对齐、数据预取、数据共享等话题。而今天要聊一个偏冷门但也不容忽视的内存话题：大页内存。

# 从一个例子谈起

先来看一个简单的循环：

const int N = (1 << 13);  
int a[D \* N];  
**for** (int i=0; i<D\*N; i+=D)  
    a[i] += 1;

问题来了，我们改变D （步长），数组大小也跟着变，但总的迭代次数 N 不变。执行性能有何变化？



当步长D从16逐步增加到1024时，理论上每次迭代都访问新的缓存行（64字节），总数据量恰好能塞进L2缓存。但实际测试中，当D≥256时性能突然暴跌——这背后的元凶不是CPU缓存，而是 **TLB (Translation Lookaside Buffer)**，寻址的速查小本本。

TLB就像个小小的缓存，专门记着 **虚拟地址** 和 **物理地址** 的对应关系。 每次程序要访问内存，先查查这小本本，看看有没有记录。 有记录，那就快得很，直接用物理地址去访问； 没记录，就得费劲地去查**页表**（Page Table），还是分层的，找到物理地址才能访问。

TLB是虚拟地址到物理地址的"速查表"。如上程序所在机器上CPU的L1 TLB仅有64个条目，L2 TLB约2048个条目。以默认4KB页计算：

* • L1 TLB覆盖范围：64×4KB=256KB
* • L2 TLB覆盖范围：2048×4KB=8MB

当D=256时，数组大小正好是8K(N)×256(D)×4B(int)=8MB——这正是L2 TLB的极限。超过这个阈值后，每次内存访问都要穿透多层页表，相当于在电话簿里逐页翻找地址，而非直接查看速记本。

# 大页内存

**大页内存** 的出现就是为了解决TLB缓存失效的问题。顾名思义，大页内存使用更大的内存页尺寸，例如 **2MB 甚至 1GB**，而不是传统的 4KB。这就像将邮编从街道为单位，换成了以省为单位，记录数一下子就变少了。

使用大页内存的优势是显而易见的：

1. 1. **减少页表条目**: 同样大小的内存，使用大页后，页表所需的条目数量大大减少，页表尺寸也随之减小。
2. 2. **提高 TLB 命中率**: 由于每个 TLB 条目可以覆盖更大的内存范围，TLB 的效率更高，命中率也随之提升。这意味着 CPU 可以更快地完成地址转换，减少内存访问延迟。
3. 3. **简化页表查找**: 即使发生 TLB Miss，更小的页表也意味着更快的页表遍历速度。

简而言之，大页内存通过**化繁为简**的方式，显著降低了虚拟内存管理的开销，从而减少了内存访问延迟，提高了应用程序的性能，尤其是在内存密集型和延迟敏感型应用中，如交易系统。

# 如何在Linux下使用

Linux 系统提供了多种方式来使用大页内存，主要分为以下两种：

### 1. 透明大页 (Transparent Huge Pages, THP)

**透明大页 (THP)** 是一种相对简单易用的机制，相当于汽车的**自动挡驾驶模式**。配置后，Linux 内核会自动尝试将普通的小页内存“升级”为大页内存，无需应用程序显式请求。

启用 THP 非常简单，只需一条命令：

echo always > /sys/kernel/mm/transparent\_hugepage/enabled

不过，全局开启巨页，也不是万能灵药。 它会降低内存的粒度，提高进程的最小内存消耗。 有些时候，内存碎片多，小进程多，反而会适得其反。 就像用大碗吃饭，吃得痛快，但饭量小的，就容易浪费。所以，除了 always （总是）和 never （从不），还有第三个选项： madvise 。

echo madvise > /sys/kernel/mm/transparent\_hugepage/enabled

在 madvise 模式下，应用程序需要通过 madvise() 系统调用显式地告知内核哪些内存区域可以使用大页。例如，在 C++ 中可以这样使用：

#**include** <sys/mman.h>  
  
void\* ptr = std::aligned\_alloc(page\_size, array\_size);  
madvise(ptr, array\_size, MADV\_HUGEPAGE);

**THP 的优点是易用性**，对应用程序代码的侵入性较小。**但缺点是控制力不足**，THP 的 “透明” 性也意味着你无法完全掌控大页的分配和使用时机。

不管是采用哪种模式，THP 都不保证能申请大页内存成功，在某些情况下可能会引入**延迟抖动 (Latency Spikes)。** 例如，当系统需要分配新的整片大页内存时，要划拉出一整块地儿。当存在内存碎片时，操作系统就需要“腾挪”空间，可能会像JVM垃圾回收时“stop the world”一样，触发昂贵的**内存整理 (Defragmentation)** 操作。

但是，对延迟极其敏感的高频交易系统，THP似乎不是最好的选择。

### 2. Hugetlbfs (Huge Page TLB File System)

**Hugetlbfs** 提供了另一种使用大页内存的方式，相当于汽车**手动挡驾驶模式**。与 THP 不同，它通过一个伪文件系统来实现，需要应用程序显式地分配和管理大页内存。

使用 hugetlbfs 的步骤如下：

1. 1. 挂载 hugetlbfs 文件系统 (如果尚未挂载):

mount -t hugetlbfs hugetlbfs /mnt/huge

1. 2. 预留大页内存池: 在系统启动时，需要预留一定数量的大页内存。这可以通过修改 Grub 引导配置或者在运行时调整 /proc/sys/vm/nr\_hugepages 来实现。
2. 3. 应用程序使用 mmap() 在 hugetlbfs 文件系统下分配大页内存: 这需要修改应用程序代码，使用 mmap() 映射 hugetlbfs 文件系统中的文件来获得大页内存。

**Hugetlbfs 的优点是控制力强，性能更稳定**，因为它避免了 THP 的动态分配和回收带来的不确定性。**缺点是使用相对复杂**，需要修改应用程序代码，并且需要提前规划和预留大页内存池。

对于追求极致低延迟的高频交易系统来说，使用确定性更高的hugetlbfs， 往往是更合适的选择。

### 在程序中使用大页

一般来说，程序用到的内存，大部分都是所谓的 “匿名内存”（anonymous memory）。 大都是程序调用内存分配器“要”来的。 现在那些高性能的内存分配器，像 jemalloc，tcmalloc，mimalloc 这些， 都支持分配 2MB 的大页内存，可以用 THP，也可以用 hugetlbfs。 有些程序，比如 Java VM 和 postgresql，自己就支持用 hugetlbfs 来分配内存。

# 调教 TLB

需要注意的是，**TLB 调优是一个精细活**，它需要你深入理解硬件架构和程序行为，并通过大量的基准测试来验证优化效果。

### 了解CPU TLB

就像 CPU 缓存一样，TLB 也分层级。以 x86 架构为例，通常有 L1 TLB 和 L2 TLB。L1 TLB 速度更快但容量更小，L2 TLB 容量更大但速度稍慢。

更细致地看，L1 TLB 还分为 **iTLB (Instruction TLB)** 和 **dTLB (Data TLB)**，分别缓存指令和数据的地址转换。而且，不同层级的 TLB 可能支持不同的页尺寸 (4KB, 2MB, 1GB)。可以用 cpuid 工具指令查看，如下是一个示例：

1 cache and TLB information (2):2  0x63: data TLB: 2M/4M pages, 4-way, 32 entries3        data TLB: 1G pages, 4-way, 4 entries4  0x03: data TLB: 4K pages, 4-way, 64 entries5  0x76: instruction TLB: 2M/4M pages, fully, 8 entries6  0xb6: instruction TLB: 4K, 8-way, 128 entries7  0xc3: L2 TLB: 4K/2M pages, 6-way, 1536 entries

例如，在上面 Cascade Lake CPU 上，L1 数据TLB 有 32 个条目缓存 2MB 页地址，4个条目缓存1G 页的地址，64 个条目缓存 4KB 页。L2 TLB 则拥有 1536 个条目，指令和数据的 4KB 和 2MB 页都算在里面。

### 将程序纳入大页

将程序的可执行文件或代码段映射到 hugetlbfs 文件系统中。这可以通过以下方式实现：

* • 将程序的代码段加载到 hugetlbfs 中的文件。
* • 使用 mmap 系统调用将该文件映射到进程的地址空间。

### 将数据纳入大页

尽量让你系统的热点数据集的大小能够尽可能地被 L1 dTLB 容纳，那么内存地址转换的延迟将降到最低。

因此，尽可能设计紧凑的核心数据结构，利用好空间局部性，永远是正道。反之，如果你访问的数据集超过了 L1 TLB 的容量，就会频繁发生 TLB Miss，性能也会随之下降。

# 大页的量化效果

本文开头的程序，使用了大页之后，效果立竿见影。 趋势图如下：

对比 4K pages 和 2M pages 两种情况下的性能曲线。2M pages 曲线几乎是一条直线，性能稳定在高位。

# 大页内存的权衡

虽然大页内存有很多优点，但并非所有场景都适用，也并非总是“免费的午餐”。使用大页内存也需要考虑一些权衡和注意事项：

* • **内存浪费**: 如果应用程序实际使用的内存大小不是大页尺寸的整数倍，可能会造成一定的内存浪费。
* • **内存碎片**: 虽然大页内存本身可以减少页表碎片，但在某些情况下，不合理的大页分配策略反而可能导致物理内存碎片问题。
* • **启动时间**: 预留大量大页内存可能会增加系统启动时间。当然，大多数情况下这都不是回事。

# **结语**

总而言之，如果你的程序里，稀疏读取内存的情况比较多，用上大页通常是没错的，能提升性能，而且几乎不会有坏处。但要用好大页内存，确实也要下些细心的功夫。

点击**关注**，共同进步

卡通画

AI 生成的内容可能不正确。

**愚夫一得**

某交易所技术人。在持续学习中和你聊聊交易、研发、运维、数字化、管理那些事。

135篇原创内容

公众号

#### 引用链接

*[1]* https://en.algorithmica.org/hpc/cpu-cache/paging/

*[2]* https://www.hudsonrivertrading.com/hrtbeat/low-latency-optimization-part-1/

*[3]* https://www.hudsonrivertrading.com/hrtbeat/low-latency-optimization-part-2/