# **低时延交易设计 之 内存时延**

原创 愚夫 [愚夫一得](javascript:void(0);)

 2025年05月01日 10:02 上海

在追求极致速度时，常常有一个潜伏的性能杀手，它不像**内存带宽**那样显眼，却可能在关键时刻拖慢你的整个系统——那就是**内存时延**（Memory Latency）。

你可能会问，上篇文章讲了《[内存带宽](https://mp.weixin.qq.com/s?__biz=MzkxNDMyNzQxMw==&mid=2247485504&idx=1&sn=e5db2fef372bb7d9176b06f5f6df40be&scene=21#wechat_redirect)》，那内存时延和带宽之间啥区别？**内存时延不就等于内存带宽的倒数**么？在这里，你可能忽视了一个问题，那就是内存存取并非是单线程处理的，而是一个并行的过程，同时也夹杂着预取、乱序、流水线等技术运用。因此，内存时延 ≠ 1/内存带宽 。这个知识点与交易系统订单的时延和吞吐类似，详见《[一文讲透交易性能指标](https://mp.weixin.qq.com/s?__biz=MzkxNDMyNzQxMw==&mid=2247484251&idx=1&sn=cd47a94871afdc3d40ce36fe2ca612b3&scene=21#wechat_redirect)》一文。

内存带宽（Bandwidth）的测量和优化相对直接：扔给系统一大堆独立的读写请求，聪明的内存控制器和CPU调度器会自动排兵布阵，重排、合并请求，尽可能把数据通路塞满，把带宽用到极致。

但时延不同。时延是关于“等待”的艺术。从你发出请求到数据真正到手，中间需要多少时间？这在低时延交易里至关重要。一次缓存未命中（Cache Miss）导致的几十甚至几百纳秒的等待，足以让你错失一个转瞬即逝的套利机会。

### 如何揪出真正的内存时延？

要测量纯粹的时延，我们得让CPU那些“小聪明”——比如预取（Prefetching）、乱序执行（Out-of-Order Execution）——失灵。CPU很擅长预测，尤其是当你按“**顺序遍历**”时（比如 for(i=0; i<N; ++i) process(arr[i]);），它会提前把 arr[i+1], arr[i+2] 等数据加载到缓存里，等你用到时，数据已经恭候多时，时延自然低。

为了打破这种预测，我们需要设计一个实验，让CPU完全猜不到下一步需要哪个内存地址。一种经典方法是“**随机遍历**”（Pointer Chasing）。

想象一下，你有一个数组（或链表），里面的元素存储着下一个要访问元素的*索引*或*地址*。就像“城市定向赛”中，你只有到达一个任务点时，才知道下一个任务点在何处。

我们构造的两种遍历算法如下图：

随机遍历的代码如下：

int p[N], q[N]; // p是原始排列，q是我们要追逐的单循环排列// 1. 生成一个随机排列 pstd::iota(p, p + N, 0);  // p = {0, 1, 2, ..., N-1}std::random\_shuffle(p, p + N);  // 打乱 p// 2. 构造一个单循环排列 q//    目标是让 q[q[...q[k]...]] 恰好遍历所有 N 个元素才回到 k//    原始代码通过 p 来构造 q，确保了这一点int k = p[N - 1];for (int i = 0; i < N; i++)    k = q[k] = p[i]; // q[上一个k] = 当前p[i], 新k = 当前p[i]// 3. 反复追逐指针 N 次，构成一次完整遍历//    (外层循环 K 是为了多次测量取平均)for (int t = 0; t < K; t++) {    for (int i = 0; i < N; i++) {        k = q[k]; // 核心：下一次访问地址 k 完全依赖于本次访存 q[k] 的结果    }}

这段代码的核心在于 k = q[k]。在CPU执行这条指令之前，它完全不知道下一个 k 的值是多少，必须**等待** q[k] 的值从内存（或缓存）中取回来才能继续。这就强制暴露了真实的内存访问时延。

我们来看一下顺序遍历（蓝色线）和随机遍历（红色线）量化的吞吐差异是多少？

可见，性能差得不是一点半点，都达几个数量级的差距！原因有2：

1. 1. **流水线停滞（Pipeline Stall）**: CPU内部的指令流水线被卡住了。后续依赖 k 值的指令，只能眼巴巴地等着数据从内存系统慢悠悠地返回，造成巨大的“交通堵塞”。
2. 2. **SIMD 失效**: 现代CPU喜欢用单指令多数据（SIMD）指令并行处理数据块。但这种随机跳转访问模式，数据不连续，SIMD完全派不上用场。

这种糟糕的性能模式就是所谓的“随机遍历”（Pointer Chasing）。它不仅存在于这种刻意设计的测试中，在实际应用中也很常见，尤其是在那些喜欢用大量堆分配对象和指针的高级语言（如C++，Java对象引用）里。

### **💡低延迟交易启示**

缓存！缓存！缓存！低时延交易系统设计的核心就是和缓存打交道。

1. 1. **关键路径慎用指针**（尤其是节点分散的复杂树结构）。随机访问的特性天然会拖慢订单簿更新、行情计算等关键路径。设计时应极力追求数据的连续性，比如使用数组，或者通过数据冗余（空间换时间）将经常访问的内容集中（如：将保证金率直接复制在持仓记录上），避免不必要的指针间接引用，哪怕牺牲一点点开发的便利性，牺牲一点点内存空间。
2. 2. **简化核心数据**: 你的热点数据——比如当前最优买卖价、你自己的挂单状态、关键计算参数——必须尽可能放在最快的缓存里（L1/L2）。这意味着要严格控制核心数据结构的大小。
3. 3. **简化指令**。CPU 不仅有数据缓存（d-cache），还有指令缓存（i-cache）。因此，也需要简化指令缓存占用和函数嵌套。如：使用内联（inline）减少函数调用开销。避免虚函数（virtual functions）等多态机制，因为它们涉及间接跳转，对 i-cache 和分支预测器不友好。
4. 4. **数据局部性 (Data Locality)**: 不仅要让数据小到能进缓存，还要让相互关联的数据在内存中靠得近。处理一个订单更新时，所需的所有信息（持仓、价格、数量、时间戳、订单ID等）最好能一次性加载到缓存中。

### 主频scaling law

我们知道，CPU主频越高，通常计算越快。那么内存访问时延呢？直觉的思考应该是：

* • 在CPU缓存（L1/L2/L3）的时延：与CPU时钟频率成比例。
* • 在主内存（RAM）的时延：由RAM芯片自身规格和内存控制器决定，与CPU主频关系不大。如DDR5必然快于DDR4

下面的图展示了在两种不同CPU频率（比如基础频率2GHz vs睿频4.1GHz）下，“**随机遍历**”测试的时延表现：如果我们画出相对加速比（高频时的速度 / 低频时的速度），如下图：理想情况下，我们期望在缓存区看到接近 4.1/2.0 ≈ 2倍的加速，在RAM区看到接近1倍的加速。但实际情况是：

* • 缓存区的加速比可能略低于2倍，这可能受到其他瓶颈或固定开销的影响。
* • RAM区的加速比略高于1倍。即使最终要访问RAM，CPU还是得先检查各级缓存（看看数据是否碰巧在缓存里）。这个“检查”动作本身是受CPU频率影响的。频率越高，检查越快，所以即使最终访问RAM的时延本身不变，整个“检查+访问RAM”的总时间在高频下还是会短那么一点点

当然，还有其他因素会轻微影响内存时延，比如虚拟内存（地址翻译TLB的开销）、RAM自身的时序参数（CAS Latency等）。

### 💡低延迟交易启示 :

主频不是万能药，尤其是在内存访问成为瓶颈时。

1. 1. **平衡是关键**: 选择CPU时，不能只看最高频率。缓存大小、速度、核心数、以及内存通道数和支持的内存频率同样重要。对于内存密集型应用，一个拥有更快更大缓存和更强内存子系统的CPU，即使主频稍低，可能性能更优。
2. 2. **NUMA 架构**: 在多路服务器上，内存访问时延还和NUMA（Non-Uniform Memory Access）有关。访问连接到本地CPU插槽的内存，比访问连接到远程CPU插槽的内存要快得多。低时延系统必须做NUMA优化，确保关键线程及其使用的内存都绑定在同一个NUMA节点上。

### 结语

内存时延，不像CPU频率那样光鲜亮丽，也不像网络带宽那样容易量化，但它深刻影响着系统的响应速度。理解内存层级、缓存行为、以及如何设计出“缓存友好”的数据结构和算法，是构建顶尖低时延交易系统的基石。

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

#### 低延时交易设计  系列文章

《[低时延交易设计 之 无分支编程](https://mp.weixin.qq.com/s?__biz=MzkxNDMyNzQxMw==&mid=2247485314&idx=1&sn=573cb8e6976e4367a488c18de0627971&scene=21#wechat_redirect)》《[低时延交易设计 之 大页内存](https://mp.weixin.qq.com/s?__biz=MzkxNDMyNzQxMw==&mid=2247485334&idx=1&sn=69032d602f0d0f383b6d2ab5830d27ab&scene=21#wechat_redirect)》《[低时延交易设计 之 内存带宽](https://mp.weixin.qq.com/s?__biz=MzkxNDMyNzQxMw==&mid=2247485504&idx=1&sn=e5db2fef372bb7d9176b06f5f6df40be&scene=21#wechat_redirect)》

#### 参考文档

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