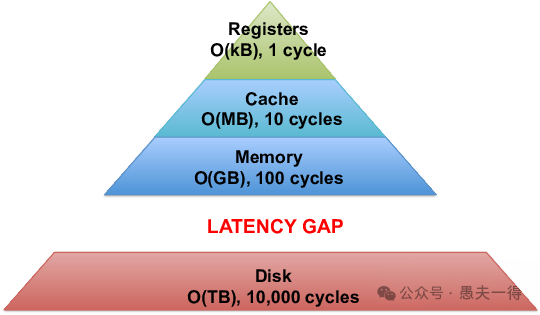
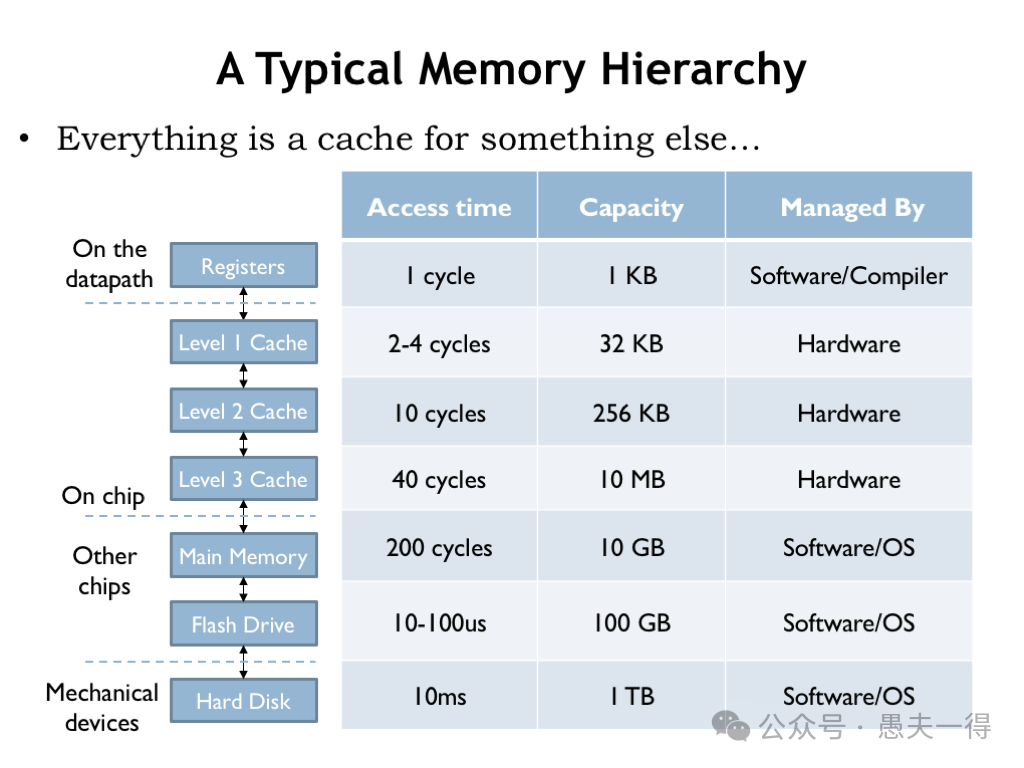
# **低时延交易设计 之 内存带宽**

原创 愚夫 [愚夫一得](javascript:void(0);)

 2025年04月04日 13:47 上海

CPU和主内存（RAM）之间隔着L1~L3的缓存（Caches）。这些缓存，离CPU越近，速度越快，但容量也越小。这里的“快”包含两个意思：

1. 1. **延迟（Latency）**：从发出读写请求到数据实际到达（或写入完成）的等待时间。越低越好。
2. 2. **带宽（Bandwidth）**：单位时间内能传输的数据量。越高越好。
3. 
4. 
5. 上图是经典的访存hierarchy图。可以看到，从寄存器->缓存->内存，每层性能差异都在10倍左右，相当可观。
6. 对很多算法来说，尤其是那些需要处理成块数据的（想想处理行情数据流、更新本地订单簿），内存带宽往往是决定性能的关键因素。而且，相对**延迟**来说，**带宽**更容易测量，所以我们先从**带宽**讲起。

### 测量带宽：一个简单的实验

我们怎么知道自己的代码到底能用多少带宽呢？搞个实验。创建一个数组，然后反复遍历它，做点简单的操作，比如给每个元素加1。

int a[N];*// K次重复是为了让测量时间足够长，减少误差*for (int t = 0; t < K; t++) {    *// 遍历大小为N的数组*    for (int i = 0; i < N; i++) {        a[i]++; *// 读取a[i]，加1，写回a[i]*    }}

现在，我们改变数组的大小 N，同时调整重复次数 K，保证总的访问次数 (N \* K) 大致不变。然后我们测量跑完这些操作总共花了多少时间，换算成“每秒完成多少次a[i]++操作”。把这个速度画成图，横轴是数组大小 N（对数坐标），纵轴是速度。

你会看到一张很有意思的图：

* *曲线表示*性能在特定数组大小（对应L1, L2, L3缓存大小）处出现明显下降。在32K、512K、4M（这些是示例值，具体看CPU）附近有“悬崖”。

### 缓存的断崖

这张图清楚地告诉我们：

* • **当数组很小，能完全塞进最快的L1缓存时**：性能最好，这时瓶颈往往是CPU核心本身的计算能力，而不是访存。图上可能接近一个理论峰值（比如这款CPU的16 GFLOPS，虽然这个单位可能不是关键，关键是相对值）。
* • **当数组变大，L1装不下了，但还能装进L2/L3时**：性能会掉一个台阶。比如，刚超出L1时，速度可能降到12-13 GFLOPS。
* • **当数组再大，连最大的L3缓存也装不下，必须频繁访问RAM时**：性能会急剧下降！可能只有原来的几分之一，比如掉到2 GFLOPS。

**💡低延迟交易启示：**

1. 1. **核心数据结构必须小！** 一种技巧是将订单等核心数据结构拆解为常用和不常用两部分。这样，L1或至少L2缓存里可以塞进尽可能多的数据。一旦数据“掉出”缓存，访问速度就是断崖式下跌。这是算法和数据结构设计时必须时刻牢记的第一原则。知道你目标机器的L1/L2/L3大小至关重要。当然，对于行情等一过性数据，并不需要拆分结构。对于订单等需重复处理的数据才需要拆分。
2. 2. **理解性能曲线不是平滑的。** 它更像阶梯。优化时，目标是把工作集（Working Set）推到更高一级的“平台”上。

### 频率的“陷阱”：Turbo Boost并不总是朋友

CPU缓存通常和CPU核心在同一个芯片上，它们的性能（带宽、延迟）会随着CPU主频变化。但RAM内存频率是独立的，基本不受CPU主频影响。

如果你开了Turbo Boost（CPU自动超频），事情就复杂了。CPU频率动态变化，会导致缓存性能也跟着变，而RAM性能不变。这使得性能测试结果不稳定，难以比较不同算法实现的优劣。

上图展示两条线： 一条是固定频率（如2GHz），另一条是开启Turbo Boost（如动态到4.1GHz）。高频时，缓存内的性能更高，但一旦访问RAM，两条线的差距就大大缩小了。

**💡低延迟交易启示：**

* • **锁死频率！** 可以看到固定频率的抖动明显要小于开启Turbo Boost。在生产环境中，为了获得可预测的、稳定的低延迟，通常建议关闭Turbo Boost，将CPU频率固定在一个较高的值。牺牲一点峰值性能，换取确定性。
* • **在目标环境测试。** Benchmark必须在你最终部署的机器和配置（包括BIOS设置）下进行，否则结果可能毫无意义。

### 读和写：方向不同，带宽也不同

我们之前的 a[i]++ 操作，其实做了三件事：读取 a[i] 的旧值，加1，写回新值。这是一个读写结合（Read-Modify-Write）的操作。但在实际应用中，很多时候我们只需要单向操作：

* • **纯读**：比如，根据行情数据计算信号，只需要读。

*// 累加数组元素，只需要读* long long s = 0; for (int i = 0; i < N; i++) {     s += a[i]; *// 只读a[i]* }

• **纯写**：比如，清零一大块内存，或者记录日志。

*// 将数组清零，只需要写* for (int i = 0; i < N; i++) {     a[i] = 0; *// 只写a[i]* }

* 这两种单向操作的带宽需求和读写结合是不同的。我们来测一下：
* • *展示三条线：* Read-Modify-Write (increment)，Read-Only (sum)，Write-Only (memset)。
* • *观察：* 在缓存内，Read-Only和Write-Only通常比Read-Modify-Write要快。

为什么？因为读和写可能要竞争使用缓存、内存总线等CPU资源。特别是访问RAM时，内存控制器需要在读模式和写模式间切换，对于读写结合的操作，这会浪费一半的带宽。而在缓存内部（如L2），虽然也有影响，但可能没那么严重，因为瓶颈可能不在总线本身。

**有趣的反常现象：memset 的“慢”**

看上图，当数组大到需要访问RAM时，奇怪的事情发生了：Write-Only (memset) 的性能居然和Read-Modify-Write差不多慢，反而比Read-Only慢很多！按理说，纯写应该更快才对啊？

原因在于CPU的“小聪明”。当你写入一块内存时，CPU通常会假设你“马上”就可能要读它，所以它在写的同时，还会把刚写入的数据加载到缓存里（这本质上触发了一次隐式的读）。这个“优化”在很多场景下是好的，但对于只需要纯粹、高速写入的场景（比如疯狂写日志），它反而占用了额外的读带宽，导致总的写带宽上不去。

**💡低延迟交易启示：**

* • **区分读写模式。** 你的代码到底是读密集、写密集，还是读写混合？这会影响它的实际内存带宽表现。
* • **警惕memset。** 虽然它是标准库函数，编译器也可能自动生成，但在对RAM进行大块写操作时，它的性能可能不如预期，因为它可能触发了我们不想要的隐式读取。

### 绕过缓存：Non-Temporal Access的威力

既然CPU的默认写操作（以及memset）在写RAM时会自作主张地把数据读回缓存，浪费带宽，我们能不能告诉它：“嘿，我这次写的数据，只是写出去存着，我（短期内）不会再读它了，你不用费心把它弄回缓存了”？

可以！这就是 **Non-Temporal（非临时）内存访问**。它允许我们执行“流式”写操作（Streaming Stores），数据直接写入内存，尽量不干扰缓存。

我们需要直接使用CPU提供的特殊指令（SIMD指令集里通常有），而不是依赖memset或普通的赋值语句。

#include <immintrin.h> *// 包含SIMD intrinsics头文件// 假设我们要用AVX2指令清零一个数组 (每次处理256位/32字节)*  
const \_\_m256i zeros = \_mm256\_set1\_epi32(0); *// 全零向量*for (int i = 0; i + 7 < N; i += 8) { *// 每次处理8个int (32字节)*    *// 注意这里的函数名：\_mm256\_stream\_si256*    \_mm256\_stream\_si256((\_\_m256i\*)&a[i], zeros);}*// 处理末尾不足8个的部分 (这里省略)*

关键在于 \_mm256\_stream\_si256 这个intrinsic函数。它告诉CPU执行一次Non-Temporal写操作。

效果如何？

* • *增加一条线：* Non-temporal write。
* • *观察：* 在缓存内，Non-temporal write可能比memset稍慢（因为它牺牲了缓存局部性）。但一旦数组大小超出缓存，需要访问RAM时，Non-temporal write的性能**远超**memset和Read-Modify-Write，甚至可能比Read-Only还要快！性能提升可能是2倍甚至更多。
* • 也存在Non-Temporal读。 类似地，如果你需要读取一大块数据，但读完就扔(**如行情数据**)，不希望它污染掉缓存里其他更有用的热数据，也可以用Non-Temporal读（如 \_mm256\_stream\_load\_si256）。比如，做一次性的大数据校验或拷贝。

为什么Non-Temporal写RAM这么快？

1. 1. **避免了隐式读回。** 这是最主要的原因，直接省掉了一半的总线流量和操作。
2. 2. **内存控制器无需切换模式。** 一直写就行了。
3. 3. **指令序列更简单。** CPU可以更有效地安排和挂起（pending）内存操作。
4. 4. **“Fire and Forget”。** 对于Non-temporal写，内存控制器发出写指令和数据后，任务基本就结束了。而读操作需要发出地址，然后一直等着数据回来，这个等待时间（Latency）会限制住你能同时发出的请求数量，从而影响实际带宽。

**💡低延迟交易启示：**

* • **用Non-Temporal写大数据块。** 当你需要高速地向内存（而不是缓存）写入大量数据，并且短期内不会读它们时（典型的如：交易流水数据、把结果数据流式传输到内存供下游线程/设备使用、大块内存初始化），Non-Temporal写是你的大杀器。而对于行情等一次性处理数据，也可以用Non-Temporal读来避免缓存污染。
* • **权衡利弊。** Non-Temporal牺牲了缓存局部性。如果你写完之后马上又要读，那它反而会变慢，因为数据不在缓存里了。要根据具体场景判断。

# 总结

内存带宽是低延迟系统设计中一个极其关键的战场。CPU再快，数据跟不上也是白搭。总结如下：

* • **理解缓存层级。** 让核心数据和代码留在高速缓存里，可以将不同使用频率数据分开存放。
* • **追求确定性。** 固定CPU频率，关闭可能导致性能抖动的特性。
* • **分析读写模式。** 知道你的操作是读、写还是读写混合，它们对带宽的影响不同。

每一纳秒的节省，都可能是在激烈市场竞争中胜出的关键。所以，花点时间搞懂你的数据是怎样在CPU和内存之间奔跑的吧。

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。

136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)》

#### 参考文档

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