本周主要对两篇文献进行学习，一篇是《多核实时操作系统关键技术研究》中基于测量方法的硬件结构判定章节，另一篇是Cache Pirating \_ Measuring The Curse of the Shared Cache。另外熟悉了Cache结构信息在Linux系统中的存储以及Perf的使用，并运行了第一篇文献中关于Cache结构判定的相关代码。

**基于测量方法的硬件结构判定**

本文主要以具体的测量方法对Intel Core i7-3770处理器的L3 Cache以及内存进行判定。测量方法是指通过构造一定规律的测试集，利用Hardware Performance Counter观察目标机器在该测试集下产生硬件行为来分析其硬件结构。

在Cache地址映射函数的判定中，本文构造了这样的访问序列：以Cache组为单位进行顺序访问，每次组访问从可访问至该组的最小地址开始，以一路大小为步长进行N次顺序访问。通常来说间隔为一路大小的地址会映射到Cache一组的不同路上，并当N大于实际Cache路数时，顺序访问行为会产生连续的不命中。

本文利用开销较小的RDTSC指令得到各组访问所需的平均周期数，以组号与平均周期为横纵坐标绘制曲线图，通过分析不同N下访问各组所需的周期数来判断硬件结构，发现在N趋向64（实际路数）时有部分组仍能保持较低的访问周期，最终得出了该Cache并非以类LRU策略为替换策略的结论。

最后，针对序列的具体实现，本文分析了多个影响因素，并提出了相应的解决方案。对于乱序执行带来内存访问乱序影响，本文采用p = \*(void\*\*)p的指针访问形式制造数据相关以解决，针对虚实地址转换带来的地址不连续问题，则采用申请hugepage的方式解决，另外，为降低访问延迟，所有的访问采用读的方式执行。

在Cache组竞争替换策略的判定中，本文在上述访问序列的基础上额外增加了一组逆序的组访问序列。利用两组访问序列访问专用组的先后次序不同，且序列针对新缓存数据优先与已缓存数据优先这两类策略有不同的命中情况来获取专用组的位置。

在内存带宽的测定中，本文主要依赖Little定律：

bandwidth = transfer\_size × MLP ÷ latency

其中，bandwidth为带宽，transfer\_size为每次访问传输的数据大小，MLP为发送至内存的访问请求数，latency为完成一个访问的平均时间。利用上述定律，可以通过测定一段时间内的L3 Cache Miss次数估算程序所获得的带宽。

Further Thinking：

1. 对于内存并行度测试，hugepage还有屏蔽组位置的作用。真实硬件环境为2048组、块大小64B的L3 Cache，即物理地址的0-5位用于映射块内地址，6-16位用于映射组地址，hugepage以4MB为单位分配内存，可以保证低22位物理地址全0，因此hugepage起始地址所在块不会因每次运行导致的起始地址不同而映射在不同Cache组，这使得策略绑定能正常进行。
2. 多核环境存在以下几个原因使得RDTSC指令准确率下降：不能保证同一块主板上每个核的 TSC 是同步的；CPU 的时钟频率可能变化；乱序执行导致 RDTSC 测得的周期数不准(https://blog.csdn.net/solstice/article/details/5196544)。因此采用一些API较合适，如windows下的QueryPerformanceFrequency、QueryPerformanceCounter等。
3. 夹在时间记录函数之间的代码应格外注意栈内存访问，可采用register关键字缓存局部变量至寄存器，改善因局部变量访问带来的影响。

Problems：

1. 命中提升(Hit-Promotion)策略难以理解。
2. 内存参数测定实验中对并行度这一参数的操作不太理解，比如为什么调整每一个thread上发出的内存访问可以控制并行度等。

**Cache Pirating \_ Measuring The Curse of the Shared Cache**

本文提出一种多核环境下利用Pirate进程窃取Target进程所占Cache空间的方法以较小代价精确测定其CPI、内存带宽需求、Cache失效率与Cache命中率关于可用Cache大小的曲线。其具体做法是，使Pirate与Target同时运行于不同的核心，Pirate以特定访问序列频繁访问主存，以抢占Target的可用Cache空间，最后针对不同的可用Cache空间，利用硬件计数器(Hardware Performance Counter)完成Target各类性能指标的测定。

对于上述方法的具体实现，本文提出了如下几个问题：

1. 如何保证Pirate确实缩小了Target的可用Cache空间。
2. 如何尽可能多地占用Cache。
3. 如何消除替换策略对性能测定的影响。
4. 如何在只运行一次Target进程的情况下测定不同可用Cache空间下的性能参数。

对于问题1，本文通过测定Pirate的Fetch Ratio以保证。其原理是若Pirate的Fetch Ratio趋向于0(实验中采用小于某个经验值)，则可认为Pirate的工作集完全被置于Cache中，进而证明Target的可用Cache空间确实被缩小了。

对于问题2，本文提出了多线程的方法。通过绑定Pirate线程至不同的核心提升抢占速率，以尽可能保证Pirate的工作集完全置于Cache空间。另外，为保证Target的运行效率，不能盲目地提升线程数，本文通过Target的CPI是否显著变化来决定线程数。

对于问题3，本文先分析了Cache，发现其替换策略为类时钟置换算法。理想状态是Pirate占用Cache空间后，剩余Cache空间的替换行为不影响被Pirate占用的Cache块。本文先以最坏情况(影响被Pirate占用的Cache块)模拟该过程，最后将模拟结果于理想结果相比较，发现偏差可以忽略不计。

对于问题4，本文采取动态改变Pirate工作集的方法解决。具体做法是每次在原先工作集上固定增量，并在每次增量后给予一定的预热时间，使Pirate充分占据Cache空间，并利用问题1所述方法判定占据是否充分。

Further thinking:

1. 根据本文所述，在每一次改变Pirate工作集后都需要一定时间预热，其大小应当也可由Pirate Fetch Ratio决定。另由于Target进程在不同时期会呈现不同的性能曲线，一次测定中多次预热时间的总和在理论上应保持为一个相对时期来说较小的数值，否则测定结果可能会受较大影响。