计算机性能优化

简介

我所了解到的软硬件优化方法有三种：1，通过运行软件 workloads、收集微架构信息，了解系统性能特点并指导软件优化；2，通过分析软件从上层 app 到 runtime 到底层 kernel 的运行时整体行为，做出系统软件优化，包括编译器优化、锁优化等；3，针对软件热点模块，做出代码优化。

微架构性能测试

芯片厂商一般会宣讲自己产品的某些规格和性能指标。但不一定靠谱哦:-)。是可以用一些软件方法来测出其真实性能指标的。

IPI 测试

Cache 性能测试

待整理。

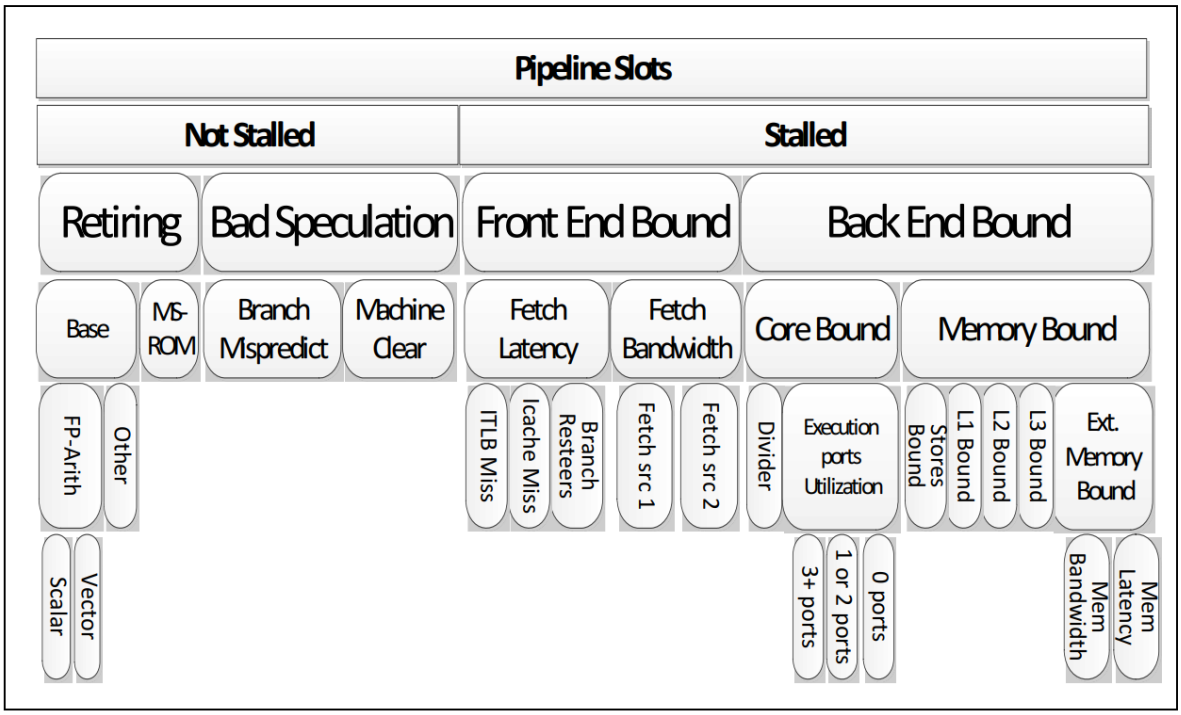
微架构分析与软件优化

这种方法实际工作中会在配置复杂的服务器级别 CPU 场合比较有用。由于 CPU 换代升级的软件版本不匹配，或者 NUMA、CACHE 等误配导致软件性能问题，这时候用这种方法能辅助快速定位。

芯片厂商也会收集微架构信息来分析 CPU 性能。但改动测试一般需要精准的 CPU 模拟

器，在此不讨论。

IA 的 CPU 提供了大量的微架构级别的性能 metrics。要把这些至少几百个 metrics 快速理清楚，一般用 Top-down Microarchitecture Analysis Method (TMAM) (Intel® 64 and IA-32 Architectures Optimization Reference Manual, Appendix B.1). 介绍可参见这篇 https://easyperf.net/blog/2019/02/09/Top-Down-performance-analysis-methodology 。这里重点讲优化案例。



SPECCPU 性能分析

优化案例待补充。

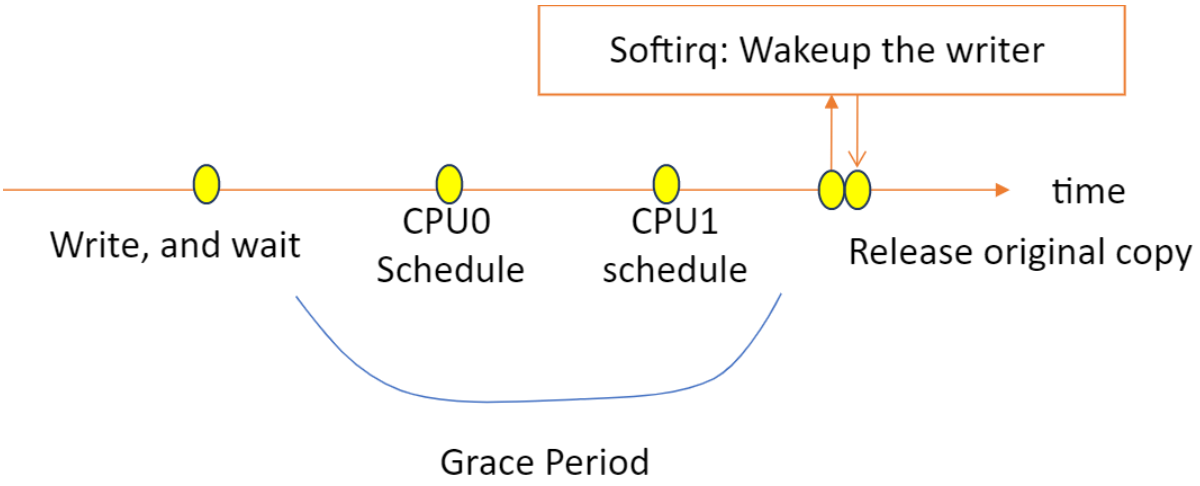
系统软件优化

做这块需要熟悉 Linux Kernel, system software。Language Runtime 级别，需要对 Java/JS 等 runtime 语言的执行机制有了解。

Linux 系统提供了 ftrace/perf 等工具来了解系统的运行时状态，可以帮助理解多进程调

度行为、资源利用情况等。也可以自己开发定制，详见我的另一篇关于定制的工具的 blog。

case1. RCU 锁优化

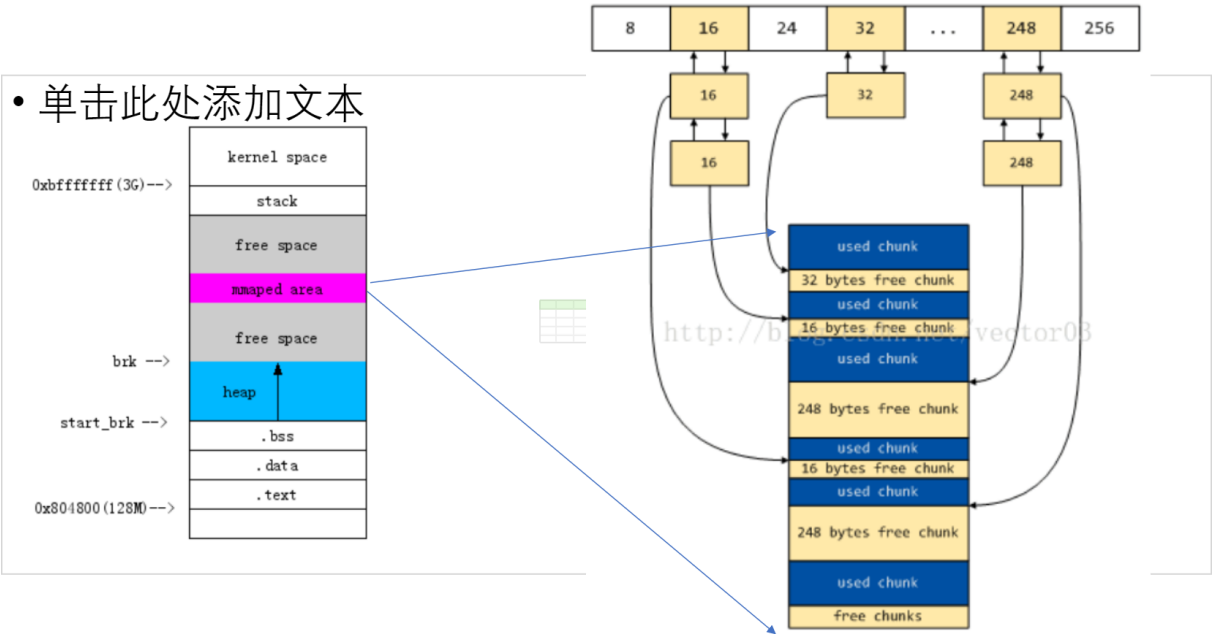


RCU 锁需要所有 CPU 经过静默期后，才会唤醒 writer，在多核系统静默期可能较长。早期 Android 在多核 CPU 上会有严重卡顿，就是 Kernel 的 RCU 机制导致。这种机制在有些情况下是可以优化掉的，比如写者不频繁的情况。

优化方法：Move the “release operation” from the writer thread to the softirq. So the writer thread returns immediately after writing and there is no need to wait for the grace period.

优化效果：应用启动时间从 1s 减少到 0.36s.

case2 Java GC top-trimming



做这个系统优化，需要了解 Java 语言的对象内存机制和 libc 的内存分配管理机制。早期 Android 版本的 bionic libc 没有 top-trimming 机制，很容易导致内存碎片并报告 out-of-memory。

libc 级别的碎片处理，实现 top-trimming 即可。在内存卡释放的时候检查一下是否可以和邻块合并。

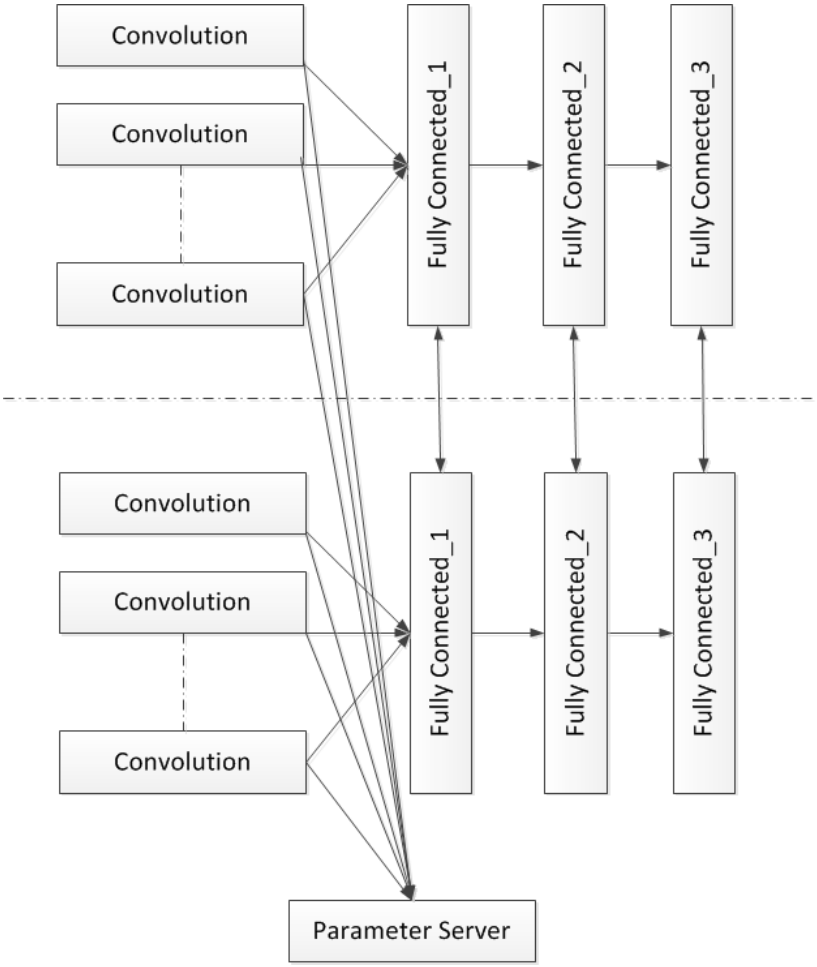
case3. Java 小对象导致的 cache false-sharing

很多时候 Java 里的对象是很小的，比如 50%的对象小于 32byte。这样容易导致多个 thread 的 java 对象被放在了同一条 cache line 里。当一个 thread 修改了其 object 后，整条 cache line 都失效了，另一个 thread 即使没有修改其 object，也需要重新从 memory load，是很低效的。

解决方法是 Java 虚拟机在 heap 里给 thread 划分一些 thread local 区域，分配对象的时候，如果是小对象，就分配到自己的 thread local 区域里，一般就不会出现到一条 cache line 里了。

case4. CAFFE AI training scalability on multiple nodes

这个优化参见论文　。主要是卷积神经网络结构里，卷积层和全连接层的运算特点很不一样。把他们放在不同的计算节点上，可以实现。但是需要做好流水线工作量调度，否则会有工作量不均衡导致的节点等待情况。



像这种优化，是为了性能而去改 framework 设计架构。一般软件设计不建议这样做，除非性能极端的重要。这种不通用的性能优化做法会影响系统的后续演进的灵活性。

hotspot 优化

对特别热点的模块，考虑算法实现或者代码优化。

算法实现需要应用层的知识，比如 JPEG 的编解码等。这块是发论文的高产地，一种新解法，省了几条乘法指令，就能发论文。工作中关注这些论文算法即可。

代码实现，需要对指令在 CPU 的性能比较熟悉。比如 IA 上 64 位除和 32 位除，性能有想象不到的差别。具体代码优化技巧实在是很成熟了。感兴趣的同学可参考《深入理解计算机系统》。