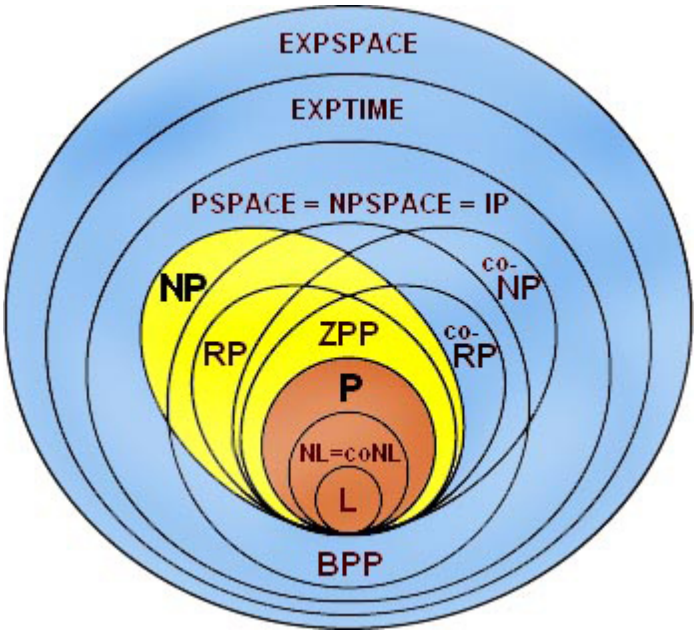
彎曲評論
科技 · 人物 · 潮流

浅谈性能优化方法和技巧

作者：KernelChina
kernelchina@gmail.com
编辑：陈怀临
huailin@gmail.com



1. 前言：

这是一个可以用一本书来讲的话题，用一系列博客来讲，可能会比较单薄一点，这里只捡重要的说，忽略很多细节，当然以后还可以补充和扩展这个话题。
我以前就说过，性能优化有三个层次：

- 系统层次
- 算法层次
- 代码层次

系统层次关注系统的控制流程和数据流程，优化主要考虑如何减少消息传递的个数；如何使系统的负载更加均衡；如何充分利用硬件的性能和设施；如何减少系统额外开销（比如上下文切换等）。
算法层次关注算法的选择（用更高效的算法替换现有算法，而不改变其接口）；现有算法的优化（时间和空间的优化）；并发和锁的优化（增加任务的并行性，减小锁的开销）；数据结构的设计（比如lock-free的数据结构和算法）。
代码层次关注代码优化，主要是cache相关的优化（I-cache, D-cache相关的优化）；代码执行顺序的调整；编译优化选项；语言相关的优化技巧等等。
性能优化需要相关的工具支持，这些工具包括编译器的支持；CPU的支持；以及集成到代码里面的测量工具等等。这些工具主要目的是测量代码的执行时间以及相关的cache miss, cache hit等数据，这些工具可以帮助开发者定位和分析问题。
性能优化和性能设计不同。性能设计贯穿于设计，编码，测试的整个环节，是产品生命周期的第一个阶段；而性能优化，通常是在现有系统和代码基础上所做 的改进，属于产品生命周期的后续几个阶段（假设产品有多个生命周期）。性能优化不是重新设计，性能优化是以现有的产品和代码为基础的，而不是推倒重来。性 能优化的方法和技巧可以指导性能设计，但两者的方法和技巧不能等同。两者关注的对象不同。性能设计是从正向考虑问题：如何设计出高效，高性能的系统；而性 能优化是从反向考虑问题：在出现性能问题时，如何定位和优化性能。性能设计考验的是开发者正向建设的能力，而性能优化考验的是开发者反向修复的能力。两者 可以互补。
后续我会就工具，架构，算法，代码，cache等方面展开讨论这个话题，敬请期待。

2. 代码优化：

代码层次的优化是最直接，也是最简单的，但前提是要对代码很熟悉，对系统很熟悉。很多事情做到后来，都是一句话：无他，但手熟尔^-^。
在展开这个话题之前，有必要先简单介绍一下Cache相关的内容，如果对这部分内容不熟悉，建议先补补课，做性能优化对Cache不了解，基本上就是盲人骑瞎马。
Cache一般来说，需要关心以下几个方面
1）Cache hierarchy
Cache的层次，一般有L1, L2, L3（L是level的意思）的cache。通常来说L1，L2是集成在CPU里面的（可以称之为On-chip cache），而L3是放在CPU外面（可以称之为Off-chip cache）。当然这个不是绝对的，不同CPU的做法可能会不太一样。这里面应该还需要加上 register，虽然register不是cache，但是把数据放到register里面是能够提高性能的。
2）Cache size
Cache的容量决定了有多少代码和数据可以放到Cache里面，有了Cache才有了竞争，才有 了替换，才有了优化的空间。如果一个程序的热点(hotspot)已经完全填充了整个Cache，那 么再从Cache角度考虑优化就是白费力气了，巧妇难为无米之炊。我们优化程序的目标是把 程序尽可能放到Cache里面，但是把程序写到能够占满整个Cache还是有一定难度的，这么大 的一个Code path，相应的代码得有多少，代码逻辑肯定是相当的复杂（基本上是不可能，至少 我没有见过）。
3）Cache line size
CPU从内存load数据是一次一个cache line；往内存里面写也是一次一个cache line，所以一个 cache line里面的数据最好是读写分开，否则就会相互影响。
4）Cache associative
Cache的关联。有全关联(full associative)，内存可以映射到任意一个Cache line；也有N-way 关联，这个就是一个哈希表的结构，N就是冲突链的长度，超过了N，就需要替换。
5）Cache type
有I-cache（指令cache），D-cache（数据cache），TLB（MMU的cache），每一种又有L1， L2等等，有区分指令和数据的cache，也有不区分指令和数据的cache。
更多与cache相关的知识，可以参考这个链接：
http://en.wikipedia.org/wiki/CPU_cache
或者是附件里面的cache.pdf，里面有一个简单的总结。
代码层次的优化，主要是从以下两个角度考虑问题：
1）I-cache相关的优化
例如精简code path，简化调用关系，减少冗余代码等等。尽量减少不必要的调用。但是

有用还是无用，是和应用相关的，所以代码层次的优化很多是针对某个应用或者性能指标的优化。有针对性的优化，更容易得到可观的结果。
2）D-cache相关的优化
减少D-cache miss的数量，增加有效的数据访问的数量。这个要比I-cache优化难一些。下面是一个代码优化技巧列表，需要不断地补充，优化和筛选。
1) Code adjacency （把相关代码放在一起），推荐指数：5颗星
把相关代码放在一起有两个涵义，一是相关的源文件要放在一起；二是相关的函数在object文件 里面，也应该是相邻的。这样，在可执行文件被加载到内存里面的时候，函数的位置也是相邻的。 相邻的函数，冲突的几率比较小。而且相关的函数放在一起，也符合模块化编程的要求：那就是 高内聚，低耦合。
如果能够把一个code path上的函数编译到一起（需要编译器支持，把相关函数编译到一起）， 很显然会提高I-cache的命中率，减少冲突。但是一个系统有很多个code path，所以不可能面 面俱到。不同的性能指标，在优化的时候可能是冲突的。所以尽量做对所以case都有效的优化， 虽然做到这一点比较难。
2) Cache line alignment （cache对齐），推荐指数：4颗星
数据跨越两个cache line，就意味着两次load或者两次store。如果数据结构是cache line对齐的， 就有可能减少一次读写。数据结构的首地址cache line对齐，意味着可能有内存浪费（特别是 数组这样连续分配的数据结构），所以需要在空间和时间两方面权衡。
3) Branch prediction （分支预测），推荐指数：3颗星
代码在内存里面是顺序排列的。对于分支程序来说，如果分支语句之后的代码有更大的执行几率， 那么就可以减少跳转，一般CPU都有指令预取功能，这样可以提高指令预取命中的几率。分支预测 用的就是likely/unlikely这样的宏，一般需要编译器的支持，这样做是静态的分支预测。现在也有 很多CPU支持在CPU内部保存执行过的分支指令的结果（分支指令的cache），所以静态的分支预测 就没有太多的意义。如果分支是有意义的，那么说明任何分支都会执行到，所以在特定情况下，静态 分支预测的结果并没有多好，而且likely/unlikely对代码有很大的侵害（影响可读性），所以一般不 推荐使用这个方法。
4) Data prefetch (数据预取），推荐指数：4颗星
指令预取是CPU自动完成的，但是数据预取就是一个有技术含量的工作。数据预取的依据是预取的数据 马上会用到，这个应该符合空间局部性（spatial locality），但是如何知道预取的数据会被用到，这个 要看上下文的关系。一般来说，数据预取在循环里面用的比较多，因为循环是最符合空间局部性的代码。
但是数据预取的代码本身对程序是有侵害的（影响美观和可读性），而且优化效果不一定

很明显（命中 的概率）。数据预取可以填充流水线，避免访问内存的等待，还是有一定的好处的。

5) Memory coloring （内存着色），推荐指数：不推荐

内存着色属于系统层次的优化，在代码优化阶段去考虑内存着色，有点太晚了。所以这个话题可以放到 系统层次优化里面去讨论。

6）Register parameters （寄存器参数），推荐指数：4颗星

寄存器做为速度最快的内存单元，不好好利用实在是浪费。但是，怎么用？一般来说，函数调用的参数 少于某个数，比如3，参数是通过寄存器传递的（这个要看ABI的约定）。所以，写函数的时候，不要 带那么多参数。c语言里还有一个register关键词，不过通常都没什么用处（没试过，不知道效果，不过 可以反汇编看看具体的指令，估计是和编译器相关）。尝试从寄存器里面读取数据，而不是内存。

7) Lazy computation （延迟计算），推荐指数：5颗星

延迟计算的意思是最近用不上的变量，就不要去初始化。通常来说，在函数开始就会初始化很多数据，但是 这些数据在函数执行过程中并没有用到（比如一个分支判断，就退出了函数），那么这些动作就是浪费了。

变量初始化是一个好的编程习惯，但是在性能优化的时候，有可能就是一个多余的动作，需要综合考虑函数 的各个分支，做出决定。

延迟计算也可以是系统层次的优化，比如COW(copy-on-write)就是在fork子进程的时候，并没有复制父 进程所有的页表，而是只复制指令部分。当有写发生的时候，再复制数据部分，这样可以避免不必要的复制， 提供进程创建的速度。

8] Early computation （提前计算），推荐指数：5颗星

有些变量，需要计算一次，多次使用的时候。最好是提前计算一下，保存结果，以后再引用，避免每次都 重新计算一次。函数多了，有时就会忽略这个函数都做了些什么，写程序的人可以不了解，但是优化的时候 不能不了解。能使用常数的地方，尽量使用常数，加减乘除都会消耗CPU的指令，不可不查。

9）Inline or not inline （inline函数），推荐指数：5颗星

Inline or not inline，这是个问题。Inline可以减少函数调用的开销（入栈，出栈的操作），但是inline也 有可能造成大量的重复代码，使得代码的体积变大。Inline对debug也有坏处（汇编和语言对不上）。所以 用这个的时候要谨慎。小的函数（小于10行），可以尝试用inline；调用次数多的或者很长的函数，尽量不 要用inline。

10) Macro or not macro (宏定义或者宏函数），推荐指数：5颗星

Macro和inline带来的好处，坏处是一样的。但我的感觉是，可以用宏定义，不要用宏函数。用宏写函数， 会有很多潜在的危险。宏要简单，精炼，最好是不要用。中看不中用。

11) Allocation on stack（局部变量），推荐指数：5颗星
如果每次都要在栈上分配一个1K大小的变量，这个代价是不是太大了哪？如果这个变量还需要初始化（因 为值是随机的），那是不是更浪费了。全局变量好的一点是不需要反复的重建，销毁；而局部变量就有这个 坏处。所以避免在栈上使用数组等变量。
12) Multiple conditions（多个条件的判断语句），推荐指数：3颗星
多个条件判断时，是一个逐步缩小范围的过程。条件的先后，决定了前面的判断是否多余的。根据code path 的情况和条件分支的几率，调整条件的顺序，可以在一定程度上减少code path的开销。但是这个工作做 起来有点难度，所以通常不推荐使用。
13) Per-cpu data structure (非共享的数据结构），推荐指数：5颗星
Per-cpu data structure 在多核，多CPU或者多线程编程里面一个通用的技巧。使用Per-cpu data structure的目的是避免共享变量的锁，使得每个CPU可以独立访问数据而与其他CPU无关。坏处是会 消耗大量的内存，而且并不是所有的变量都可以per-cpu化。并行是多核编程追求的目标，而串行化 是多核编程里面最大的伤害。有关并行和串行的话题，在系统层次优化里面还会提到。
局部变量肯定是thread local的，所以在多核编程里面，局部变量反而更有好处。
14) 64 bits counter in 32 bits environment（32位环境里的64位counter），推荐指数：5颗星
32位环境里面用64位counter很显然会影响性能，所以除非必要，最好别用。有关counter的优化可以多 说几句。counter是必须的，但是还需要慎重的选择，避免重复的计数。关键路径上的counter可以使用 per-cpu counter，非关键路径(exception path）就可以省一点内存。
15) Reduce call path or call trace（减少函数调用的层次），推荐指数：4颗星
函数越多，有用的事情做的就越少（函数的入栈，出栈等）。所以要减少函数的调用层次。但是不应该 破坏程序的美观和可读性。个人认为好程序的首要标准就是美观和可读性。不好看的程序读起来影响心 情。所以需要权衡利弊，不能一个程序就一个函数。
16) Move exception path out（把exception处理放到另一个函数里面），推荐指数：5颗星
把exception path和critical path放到一起（代码混合在一起），就会影响critical path的cache性能。 而很多时候，exception path都是长篇大论，有点喧宾夺主的感觉。如果能把critical path和 exception path完全分离开，这样对i-cache有很大帮助。
17) Read, write split（读写分离），推荐指数：5颗星
在cache.pdf里面提到了伪共 享(false sharing)，就是说两个无关的变量，一个读，一个写，而这 两个变量在一个cache line里面。那么写会导致cache line失效（通常是在多核编程里面，两个变量 在不同的core上引用）。读写分离是一个很难运用的技巧，特别是

在code很复杂的情况下。需要 不断地调试，是个力气活（如果有工具帮助会好一点，比如cache miss时触发cpu的execption处理 之类的）。
18) Reduce duplicated code（减少冗余代码），推荐指数：5颗星
代码里面的冗余代码和死代码(dead code)很多。减少冗余代码就是减小浪费。但冗余代码有时 又是必不可少（copy-paste太多，尾大不掉，不好改了），但是对critical path，花一些功夫还 是必要的。
19) Use compiler optimization options （使用编译器的优化选项），推荐指数：4颗星
使用编译器选项来优化代码，这个应该从一开始就进行。写编译器的人更懂CPU，所以可以放心 地使用。编译器优化有不同的目标，有优化空间的，有优化时间的，看需求使用。
20) Know your code path （了解所有的执行路径，并优化关键路径），推荐指数：5颗星
代码的执行路径和静态代码不同，它是一个动态的执行过程，不同的输入，走过的路径不同。 我们应该能区分出主要路径和次要路径，关注和优化主要路径。要了解执行路径的执行流程， 有多少个锁，多少个原子操作，有多少同步消息，有多少内存拷贝等等。这是性能优化里面 必不可少，也是唯一正确的途径，优化的过程，也是学习，整理知识的过程，虽然有时很无聊， 但有时也很有趣。
代码优化有时与编程规则是冲突的，比如直接访问成员变量，还是通过接口来访问。编程规则上肯定是说要通过接口来访问，但直接访问效率更高。还有就是 许多ASSERT之类的代码，加的多了，也影响性能，但是不加又会给debug带来麻烦。所以需要权衡。代码层次的优化是基本功课，但是指望代码层次的优 化来解决所有问题，无疑是缘木求鱼。从系统层次和算法层次考虑问题，可能效果会更好。
代码层次的优化需要相关工具的配合，没有工具，将会事倍功半。所以在优化之前，先把工具准备好。有关工具的话题，会在另一篇文章里面讲。
还有什么，需要好好想想。这些优化技巧都是与c语言相关的。对于其他语言不一定适用。每个语言都有一些与性能相关的编码规范和约定俗成，遵守就可以了。有很多Effective, Exceptional 系列的书籍，可以看看。
代码相关的优化，着力点还是在代码上，多看，多想，就会有收获。
参考资料：
1）http://en.wikipedia.org/wiki/CPU_cache
2) Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)
3) More Effective C++: 35 New Ways to Improve Your Programs and Designs
4) Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library
5) Effective Java (2nd Edition)

6) Effective C# (Covers C# 4.0): 50 Specific Ways to Improve Your C# (2nd Edition) (Effective Software Development Series)
7) **The Elements of Cache Programming Style**

3. 工具优化：

“工欲善其事，必先利其器”（孔子），虽然“思想比工具更重要”（弯曲网友），但是，如果没有工具支持，性能优化就会非常累。思想不好掌握，但是使用工具还是比较好学习的，有了工具支持，可以让初级开发者更容易入门。
性能优化用到的工具，需要考虑哪些方面的问题？
1）使用工具是否需要重新编译代码？
一般来说，性能优化工具基本上都需要重新编译代码。因为在生产环境里面使用的image，应该是已经优化过的image。不应该在用户环境里面去调试性能 问题。但Build-in的工具有一个好处就是性能测试所用的image和性能调试所用的image是相同的，这样可以避免重新编译所带来的误差。
2）工具本身对测量结果的影响
如果是Build-in的工具，需要减小工具对性能的影响，启用工具和不启用工具对性能的影响应该在一定范围之内，比如5%，否则不清楚是工具本身影响性能还是被测量的代码性能下降。
如果是需要重新编译使用的工具，这里的测试是一个相对值，不能做为性能指标的依据。因为编译会修改代码的位置，也可能会往代码里面加一个测量函数，它生成的image和性能测试的image不一样。
在这里要列出几个我用过的Linux工具，其他系统应该也有对应的工具，读者可以自己搜索。
性能测试工具一般分这么几种
1）收集CPU的performance counter。CPU里面有很多performance counter，打开之后，会记录CPU某些事件的数量，比如cache miss, 指令数，指令时间等等。这些counter需要编程才能使用。测量哪一段代码完全由自己掌握。
2）利用编译器的功能，在函数入口和出口自动加回调函数，在回调函数里面，记录入口和出口的时间。收集这些信息，可以得到函数的调用流程和每个函数所花费的时间。
3）自己在代码里面加入时间测量点，测量某段代码执行的时间。这种工具看起来和#1的作用差不多，但是由于performance counter编程有很多限制，所以这种工具有时还是有用处的。
在Linux里面，我们经常会用到

1）Oprofile

Oprofile已经加入了linux的内核代码库，所以不需要打patch，但是还需要重新编译内核才可以使用。这是使用最广泛的linux工具，网上有很多使用指南，读者可以自己搜索参考。

http://oprofile.sourceforge.net/news/

http://people.redhat.com/wcohen/Oprofile.pdf

2) KFT and Gprof

KFT是kernel的一个patch，只对kernel有效；Gprof是gcc里面的一个工具，只对用户空间的程序有效。这两个工具都需要重新 编译代码，它们都用到了gcc里面的finstrument-functions选项。编译时会在函数入口，出口加回调函数，而且inline函数也会改 成非inline的。它的工作原理可以参考：

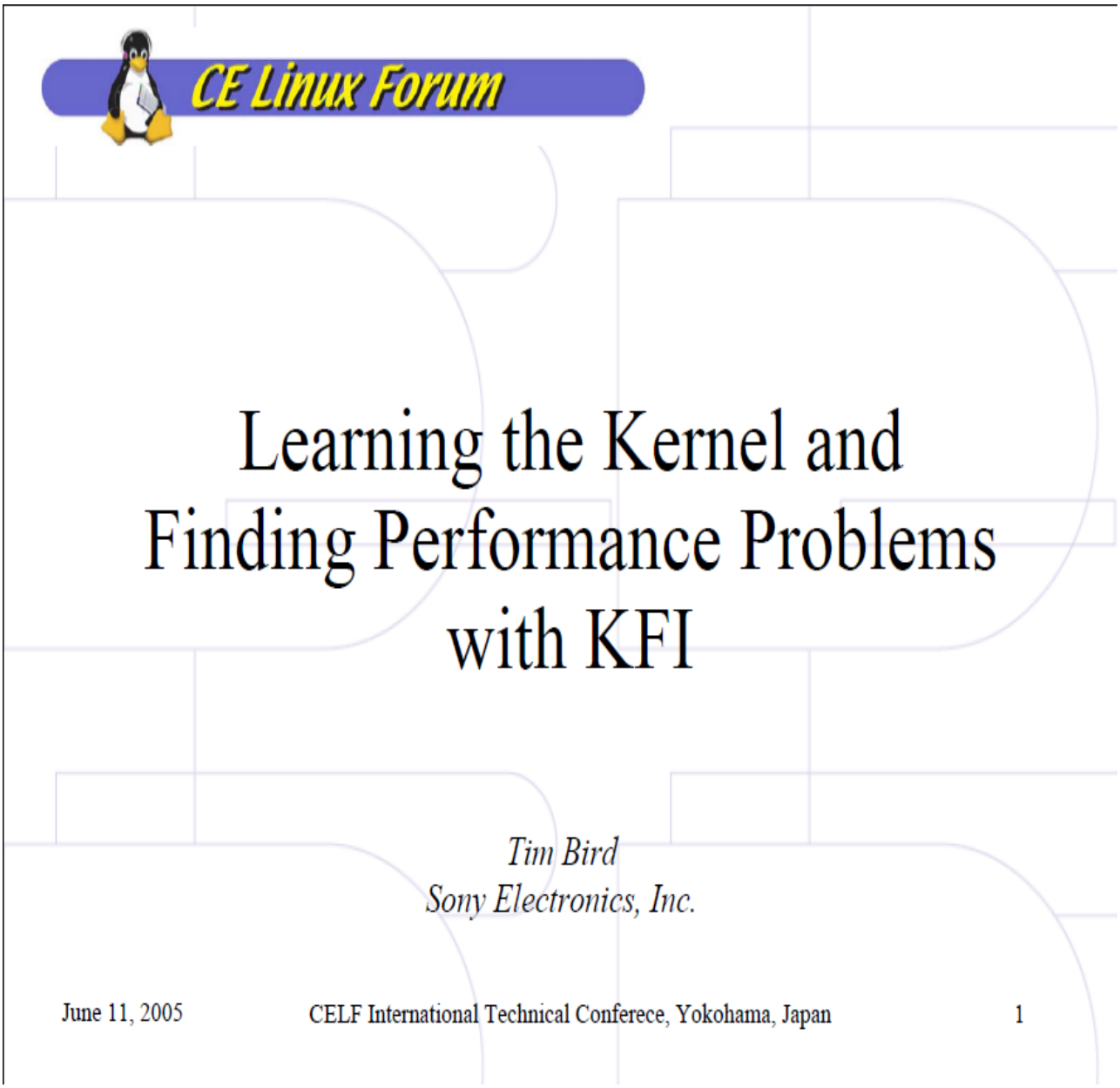
http://blog.linux.org.tw/~jserv/archives/001870.html

http://blog.linux.org.tw/~jserv/archives/001723.html

http://elinux.org/Kernel_Function_Trace

http://www.network-theory.co.uk/docs/gccintro/gccintro_80.html

个人认为这是一个非常有用的工具，对读代码也有帮助，是居家旅行的必备。这里还有一个slide比较各种工具的，可以看看。



3) Performance counter

http://anton.ozlabs.org/blog/2009/09/04/using-performance-counters-for-linux/

Linux performance counter，用于收集CPU的performance counter，已经加入了内核代码库。通常来说，performance counter的粒度太大，基本没有什么用处，因为没法定位问题出在哪里；如果粒度太小，就需要手工编程，不能靠加几个检查点就可以了。所以还是要结合上 面两个工具一起用才有好的效果。

工具解决哪些问题？

1）帮助建立基线。没有基线，就没办法做性能优化。性能优化是个迭代的过程，指望一次搞定是不现实的。

2）帮助定位问题。这里有两个涵义：一是性能问题出现在什么地方，是由哪一段代码引

起的；二是性能问题的原因，cache miss，TLB miss还是其他。
3）帮助验证优化方案。优化的结果应该能在工具里面体现出来，而不是靠蒙。
就这些了，还有什么补充？
参考资料
1）http://software.intel.com/en-us/articles/intel-microarchitecture-codename-nehalem-performance-monitoring-unit-programming-guide/
2) http://www.celinuxforum.org/CelfPubWiki/KernelInstrumentation
3) Continuous Profiling: Where Have All the Cycles Gone?

4. 系统优化：

从系统层次去优化系统往往有比较明显的效果。但是，在优化之前，我们先要问一问，能否通过扩展系统来达到提高性能的目的，比如:

- Scale up: 用更强的硬件替代当前的硬件
- Scale out: 用更多的部件来增强系统的性能

使用更强的硬件当然和优化没有半点关系，但是如果这是一个可以接受的方案，为什么不用这个简单易行的方案哪？替换硬件的风险要比改架构，改代码的风险小多了，何乐而不为？
Scale out的方案就有一点麻烦。它要求系统本身是支持scale out，或者把系统优化成可以支持scale out。不管是哪一种选择，都不是一个简单的选择。设计一个可以scale out的系统已经超出了本文所要关注的范围，但是，scale out应该是系统优化的一个重要方向。
下面会讨论一些常见的系统优化的方法，如果还有其他没有提到的，也欢迎读者指出来。
1) Cache

- Cache是什么？Cache保存了已经执行过的结果。
- Cache为什么有效？一是可以避免计算的开销（比如SQL查询的开销）；二是离计算单元更近，所以访 问更快（比如CPU cache）。
- Cache的难点在哪里？一是快速匹配，这涉及到匹配算法选择（一般用哈希表），Cache容量（哈希表的容量影响查找速度）；二是替换策略（一般使用LRU或者随机替换等等）。
- Cache在哪些情况下有效？毫无疑问，时间局部性，也就是当前的结果后面会用到，如果没有时间局部性，Cache就不能提高性能，反而对性能和系统架构有害处。所以在系统设计之初，最好是审视一下数据流程，再决定是否引入Cache层。

2) Lazy computing
Lazy computing（延迟计算），简而言之，就是不要做额外的事情，特别是无用

的事情。最常见的一个例子就是COW(copy on write)，可以参考这个链接http://en.wikipedia.org/wiki/Copy-on-write。

- COW是什么？写时复制。也就是说fork进程时，子进程和父进程共享相同的代码段和数据段，如果没有写的动作发生，就不要为子进程分配新的数据段（通常在fork之后，会有exec，用新的代码段和数据段替换原来的代码段和数据段，所以复制父进程的数据段是没有用的）。
- COW为什么有效？一是可以节省复制内存的时间，二是可以节省内存分配的时间（到真正需要时再分配，虽然时间不会减少，但是CPU的使用更加均匀，避免抖动）。
- COW的难点在哪里？一是引用计数，多个指针指向同一块内存，如果没有引用计数，内存无法释放；二是 如何知道哪块内存是可以共享的？（在fork的例子里面，父进程，子进程的关系非常明确，但是在有些应用里面，需要查找能够共享的内存，查找需要花时间）

Lazy computing在哪些情况下有效？目前能想到的只有内存复制。用时分配内存算不算哪？用时分配内存不能节省时间，但是可以节省空间。静态内存对时间性能有好处；动态内存对空间性能有好处。就看目标是优化哪个性能了。

3) Read ahead

Read ahead（预读），也可以称之为pre-fetch（预取）。就是要提前准备所需要的数据，避免使用时的等待。

- Read ahead是什么？可以参考http://en.wikipedia.org/wiki/Readahead，这个是讲文件预读的。CPU里面也有pre-fetch（CPU预取需要仔细安排，最好是能够填充流水线，所以需要多次尝试才有结果）。
- Read ahead为什么有效？Read ahead可以减少等待内存的时间。其实相当于把多个读的动作集合成一个。这个和网络里面的buffering或者sliding window有异曲同工之妙。停-等协议是最简单的，但是效率也最低。
- Read ahead的难点在哪里？预读多少才合适？预读窗口的大小需要根据负载，文件使用的多少等因素动态调整。预测的成功与否关系的性能。所以这并不是一个简单的优化方法。
- Read ahead在哪些情况下有效？毫无疑问，空间局部性。没有空间局部性，read ahead就失去了用武之地。用错了，反而会降低性能。

4) Hardware assist

Hardware assist（硬件辅助），顾名思义，就是用硬件实现某些功能。常见的，比如加密，解密；正则表达式或者DFA engine，或者规则查找，分类，压缩，解压缩等等。逻辑简单，功能确定，CPU intensive的工作可以考虑用硬件来代替。

- Hardware assist为什么有效？协处理器可以减轻CPU的工作，而且速度比CPU做要快（这个要看情况，并不是任何情况下都成立）。Hardware assist和Hardware centric的设计完全不同，不能混为一谈。在Hardware assist的设计里面，主要工作还是由软件完成；而hardware centric就是基于ASIC的设计方案，大部分工作是有硬件来完成。
- Hardware assist的难点在哪里？一是采用同步还是异步的方式与硬件交互（通常是异步）；二是如何使硬件满负荷工作，同时又避免缓冲区溢出或丢弃（这个要安排好硬件和软件的节奏，使之协调工作）；还有就是硬件 访问内存的开销（尽量硬件本身所带的内存，如果有的话）。

5) Asynchronous

Asynchronous（异步）。同步，异步涉及到消息传递。一般来说，同步比较简单，性能稍低；而异步比较复杂，但是性能较高。

- Asynchronous是什么？异步的含义就是请求和应答分离，请求和应答可以由不同的进程或线程完成。比如在 TCP协议的实现里面，如果滑动窗口是1，那么每次只能发送一个字节，然后等待应答；如果增加滑动窗口，那么一次可以发送多个字节，而无需等待前一个字节 的应答。这样可以提高性能。
- Asyncrhonous为什么有效？异步消除了等待的时间，可以更有效利用带宽。
- Asynchronous的难点是什么？一是如何实现分布式的状态机？由于请求和应答双方是独立的，所以要避免状态之间有依赖关系，在无法消除状 态之间的依赖关系时，必须使用同步消息（比如三次握手）；二是应答来了之后， 如果激活原来的执行过程，使之能够继续执行。
- Asynchronous在哪些情况下有效？很明显，状态之间不能有依赖关系，同时需要足够的带宽（或者窗口）。

6) Polling

Polling（轮询）。Polling是网络设备里面常用的一个技术，比如Linux的NAPI或者epoll。与之对应的是中断，或者是事件。

- Polling为什么有效？Polling避免了状态切换的开销，所以有更高的性能。
- Polling的难点是什么？如果系统里面有多种任务，如何在polling的时候，保证其他任务的执行时间？Polling 通常意味着独占，此时系统无法响应其他事件，可能会造成严重后果。
- Polling在哪些情况下有效？凡是能用事件或中断的地方都能用polling替代，是否合理，需要结合系统的数据流程来决定。

7) Static memory pool

Static memory pool（静态内存）。如前所述，静态内存有更好的性能，但是适应性较

差（特别是系统里面有多个 任务的时候），而且会有浪费（提前分配，还没用到就分配了）。

- Static memory pool为什么有效？它可以使内存管理更加简单，避免分配和是否内存的开销，并且有利于调试内存问题。
- Static memory pool的难点在哪里？分配多大的内存？如何避免浪费？如何实现O(1)的分配和释放？如何初始化内存？
- Static memory pool在哪些情况下有效？一是固定大小的内存需求（通常与系统的capacity有关），内存对象的大小一致，并且要求快速的分配和释放。

系统层次的优化应该还有很多方法，能想起来的就这么多了（这部分比较难，酝酿了很久，才想起来这么一点东西^-^），读者如果有更好的方法，可以一起讨论。性能优化是关注实践的工作，任何纸上谈兵都是瞎扯，与读者共勉。

参考资料：

1：http://en.wikipedia.org/wiki/Copy_on_write

2：http://en.wikipedia.org/wiki/Readahead

3：http://en.wikipedia.org/wiki/Sliding_window

4：http://en.wikipedia.org/wiki/Asynchronous_I/O

5：http://en.wikipedia.org/wiki/Coprocessor

6：http://en.wikipedia.org/wiki/Polling_(computer_science)

7：http://en.wikipedia.org/wiki/Static_memory_allocation

8：http://en.wikipedia.org/wiki/Program_optimization

9: http://en.wikipedia.org/wiki/Scale_out

5. 算法优化

算法的种类和实现浩如烟海，但是在这篇文章里面，不讨论单核，单线程的算法，而是讨论多核，多线程的算法；不讨论所有的算法类型（这个不是本文作者 能力范围之内的事），而是讨论在多核网络设备里面常见的算法，以及可能的优化途径，这些途径有些经过了验证，有些还是处于想法阶段，暂时没有实现数据的支 持。

多核算法优化的目标无非两种：lock-free和lock-less。

lock-free是完全无锁的设计，有两种实现方式：

- Per-cpu data，顾名思义，每个核或者线程都有自己私有的数据结构（这里的私有和thread local data是有区别的，这里的私有是逻辑上私有，并不意味着别的线程无法访问这些数据；而thread local data是线程私有的数据结构，别的线程是无法访问的。当然，不管是逻辑上私有，还是物理上私有，把共享数据转化成线

程私有数据，就可以避免锁，避免竞 争）。全局变量是共享的，而局部变量是私有的，所以多使用局部变量，同样可以达到无锁的目的。

- CAS based，CAS是compare and swap，这是一个原子操作（spinlock的实现同样需要compare and swap，但区别是spinlock只有两个状态LOCKED和UNLOCKED，而CAS的变量可以有多个状态）；其次，CAS的实现必须由硬件来保障（原子操作），CAS一次可以操作32 bits，也有MCAS，一次可以 比较修改一块内存。基于CAS实现的数据结构没有一个统一，一致的实现方法，所以有时不如lock based的算法那么简单，直接，针对不同的数据结构，有不同的CAS实现方法，读者可以自己搜索。

lock-less的目的是减少锁的争用（contention），而不是减少锁。这个和锁的粒度（granularity）相关，锁的粒度越小，等待的时间就越短，并发的时间就越长。

锁的争用，需要考虑不同线程在获取锁后，会执行哪些不同的动作。以session pool的分配释放为例：假设多个线程都会访问同一个session pool，分配或者释放session。session pool是个tailq，分配在head上进行；而释放在tail上进行。

如果多个线程同时访问session pool，需要一个spinlock来保护这个session pool。那么分配和释放两个不同的动作，相互之间就会有争用，而且多个线程上的分配，或者释放本身也会有争用。

现在我们可以考虑分配用一个锁，释放用一个锁，生成一个双端队列，这样可以减少分配和释放之间的争用。

http://www.parallellabs.com/2010/10/25/practical-concurrent-queue-algorithm/ （参考这篇文章）。

也可以考虑用两个pool，分配一个pool，释放一个pool，在分配pool用完之后，交换两个pool的指针（这时要考虑两个pool都是空的情况，这里只是减少了分配和释放的争用，但不能完全消除这种争用）。

不管是lock-based还是CAS-based (lock-free)的数据结构，都需要一个状态机。不同状态下，做不同的事，而增加锁的粒度，也就是增加了状态机的数量（不是状态的数量），减小状态保护的范围。这个需要在实践中体会。

参考资料：

1：http://en.wikipedia.org/wiki/Lock-free_and_wait-free_algorithms

2：http://yongsun.me/2010/01/%E4%BD%BF%E7%94%A8cas%E5%AE%9E%E7%8E%B0lock-free%E7%9A%84%E4%B8%80%E4%B8%AA%E7%B1%BB%E6%AF%94/

3：http://www.cppblog.com/johndragon/archive/2010/01/08/105207.html

4：http://kb.cnblogs.com/page/45904/

5：http://www.parallellabs.com/2010/10/25/practical-concurrent-queue-algorithm/

6.总结

性能优化这个话题可大可小。从大的方面来说，在系统设计之初，需要考虑硬件的选择，操作系统的选择，基础软件平台的选择；从小到方面来说，每个子系 统的设计，算法选择，代码怎么写，如何使用编译器的选项，如何发挥硬件的最大性能等等。本系列关注的是在给定硬件平台，给定操作系统和基础软件平台的情况 下，如何优化代码，简而言之，就是关注小的方面。

在给定硬件平台的情况下，如何发挥硬件的最大效能？前提是需要对硬件平台很熟悉。如何分离硬件相关代码和硬件无关的代码是一个很重要的技能。针对某一硬件平台优化固然是好，但是如果代码都是硬件相关的，就失去了可移植性，软件的性价比不一定高。

性能优化只是系统的一个方面，它会和系统的其他要求有冲突，比如

- 可读性：性能优化不能影响可读性，看起来不怎么漂亮的代码，没有人愿意维护
- 模块化：性能优化往往需要打破模块的边界，想想这是否值得
- 可移植：隔离硬件相关的代码，尽量使用统一的API
- 可维护：许多性能优化的技巧，会导致后来维护代码的人崩溃

需要在性能优化和上述的几个要求之间做出tradeoff，不能一意孤行。

性能优化的目标是什么？不外乎两个：

- 时间性能：减小系统执行的时间
- 空间性能：减小系统占用的空间

那么我们优化的策略，首先就是对系统进行分解，测量：

- 分解：系统包含的子系统，执行路径，函数，指令等等
- 测量：每个子系统，执行路径，函数，指令所花费的时间和空间

然后，选取执行次数最多，消耗时间最长的函数进行优化（这里需要指出的是，消耗时间最长的函数并不一定就是优化的对象，这个需要放到某个执行路径上去考察，优化最常使用的执行路径）。

对于单核和多核的优化，有什么不同？以cache miss为例，在单核上

- I-cache miss的原因是什么？一是代码路径太长，以至于超出了cache的容量，cache需要替换；二是 多个执行路径之间相互交替，cache需要替换；三是I-cache, D-cache互相踩。（对于i-cache的优化，基本上没有什么可遵循的规则。最有效的还是：一，减少无用的代码；二，减少冗余的代码；三，减少函数

调用的开销；4，快速路径短小精干，相关代码相邻；5，相关代码放到一个段里面等等）

- D-cache miss的原因是什么？一是数据结构太大，超出了cache的容量（大数组，长链表都会有这个问题，比较之下，还是数组更让人放心一点）；二是多个数据结 构之间相互踩（问题是一个执行路径，需要访问那么多数据吗？如果是，这个路径是不是太复杂了一点）；三是D-cache和I-cache的冲突（这个不好 说，系统里面会出现这种模式的cache miss吗？）

在多核上，cache miss会有什么新的特点？

- I-cache miss：多核有独立的L1 cache，共享的L2/L3 cache。如果多条执行路径同时进行，cache的冲突几率就会增大。但是这是没办法的事情，总不能什么事都不做吧。所以，对i-cache的优化，关 注快速路径就可以了，不要把精力都花费在这个上面
- D-cache miss：锁，原子操作，伪共享等都会引起多核上的D-cache miss。这是多核优化时需要关注的重点。优化的策略也很直接，就是尽量减少锁，原子操作，伪共享等。多核的所有冲突都是由共享引起的，所以要区分出哪些是必须共享的，哪些是可以per thread的。并行优化与应用有关，需要注意的是，优化是否对所有用例都有效？做不好，可能顾此失彼。

性能优化必备技能

- 熟悉系统执行路径：可以通过读代码或者使用profiling工具学习代码，要思考执行路径上不合理的地方，看看哪里可以减少，哪里可以合并。熟悉系统执行路径是性能优化的基础。
- 熟悉测量工具：顺手的工具必不可少
- 常用的代码优化技巧和策略：针对不同的语言，不同的平台，使用与之相应的技巧和策略

学习性能优化是一个不断积累的过程，在这个过程中，总结和学习自己用的顺手的工具和技巧，不断尝试，不断思考，就会有收获（感谢teltalk.org读者的评论，弯曲评论的精华在评论，思想碰撞才有火花）。

参考资料：

1: http://en.wikipedia.org/wiki/Program_optimization
2: http://www.agner.org/optimize/
3: Code Optimization – Effective Memory Usage
4: Hacker's Delight