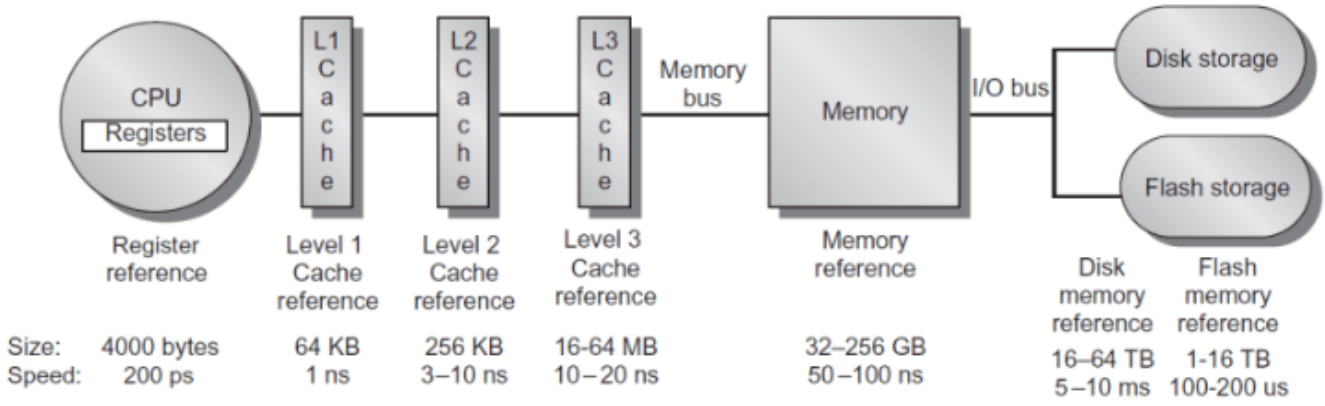
编译器优化那些事儿（7）：Cache优化

引言

软件开发人员往往期望计算机硬件拥有无限容量、零访问延迟、无限带宽以及便宜的内存，但是现实却是内存容量越大，相应的访问时间越长；内存访问速度越快，价格也更贵；带宽越大，价格越贵。为了解决大容量、高速度、低成本之间的矛盾，基于程序访问的局部性原理，将更常用数据放在小容量的高速存储器中，多种速度不同的存储器分层级联，协调工作。

图1 memory hierarchy for sever[1]



现代计算机的存储层次可以分几层。如图1所示，位于处理器内部的是寄存器；稍远一点的是一级Cache，一级Cache一般能够保存64k字节，访问它大约需要1ns，同时一级Cache通常划分为指令Cache(处理器从指令Cache中取要执行的指令)和数据Cache(处理器从数据Cache中存/取指令的操作数)；然后是二级Cache，通常既保存指令又保存数据，容量大约256k，访问它大约需要3-10ns；然后是三级Cache，容量大约16-64MB，访问它大约需要10-20ns；再接着是主存、硬盘等。注意，CPU和Cache是以word传输的，Cache到主存以块(一般64byte)传输的。

前文提到了程序的局部性原理，一般指的是时间局部性(在一定时间内，程序可能会多次访问同一内存空间)和空间局部性(在一定时间内，程序可能会访问附近的内存空间)，高速缓存(Cache)的效率取决于程序的空间和时间的局部性性质。比如一个程序重复地执行一个循环，在理想情况下，循环的第一个迭代将代码取至高速缓存中，后续的迭代直接从高速缓存中取数据，而不需要重新从主存装载。因此，为了使程序获得更好的性能，应尽可能让数据访问发生在高速缓存中。但是如果数据访问在高速缓存时发生了冲突，也可能会导致性能下降。

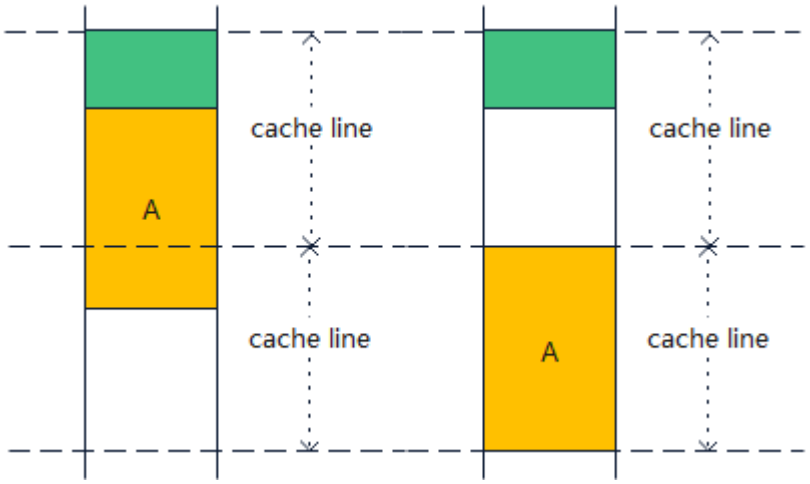
篇幅原因，本文重点讨论编译器在Cache优化中可以做哪些工作，如果读者对其他内存层次优化感兴趣，欢迎留言。下面将介绍几种通过优化Cache使用提高程序性能的方法。

对齐和布局

现代编译器可以通过调整代码和数据的布局方式，提高Cache命中率，进而提升程序性能。本节主要讨论数据和指令的对齐、代码布局对程序性能的影响，大部分处理器中Cache到主存是以Cache line(一般为64Byte，也有地方称Cache块，本文统一使用Cache line)传输的，CPU从内存加载数据是一次一个Cache line，CPU往内存写数据也是一次一个Cache line。假设处理器首次访问数据对象A，其大小刚好为64Byte，如果数据对象A首地址并没有进行对齐，即数据对象A占用两个不同Cache line的一部分，此时处理器访问该数据对象时需要两次内存访问，效率低。但是如果数据对象A进行了内存对齐，即刚好在一个Cache line中，那么处理器访问该数据时只需要一次内存访问，效率会高很多。编译器可以通过合理安排数据对象，避免不必要地将它们跨越在多个Cache line中，尽量使得同一对象集中在一个Cache中，进而有效地使用Cache来提高程序的性能。通过顺序分配对象，即如果下一个对象不能放入当前Cache line的剩余部分，则跳过这些剩余的部分，从下一个Cache line的开始处分配对象，或者将大小(size)相同的对象分配在同一个存储区，所有对象都对齐在size的倍数边界上等方式达到上述目的。

Cache line对齐可能会导致存储资源的浪费，如图2所示，但是执行速度可能会因此得到改善。对齐不仅仅可以作用于全局静态数据，也可以作用于堆上分配的数据。对于全局数据，编译器可以通过汇编语言的对齐指令命令来通知链接器。对于堆上分配的数据，将对象放置在Cache line的边界或者最小化对象跨Cache line的次数的工作不是由编译器来完成的，而是由runtime中的存储分配器来完成的[2]。

图2 因块对齐可能会浪费存储空间



前文提到了数据对象对齐，可以提高程序性能。指令Cache的对齐，也可以提高程序性能。同时，代码布局也会影响程序的性能，将频繁执行的基本块的首地址对齐在Cache line的大小倍数边界上能增加在指令Cache中同时容纳的基本块数目，将不频繁执行的指令和频繁指令的指令放到不同的Cache line中，通过优化代码布局来提升程序性能。

利用硬件辅助

Cache预取是将内存中的指令和数据提前存放至Cache中，达到加快处理器执行速度的目的。Cache预取可以通过硬件或者软件实现，硬件预取是通过处理器中专门的硬件单元实现的，该单元通过跟踪内存访问指令数据地址的变化规律来预测将会被访问到的内存地址，并提前从主存中读取这些数据到Cache；软件预取是在程序中显示地插入预取指令，以非阻塞的方式让处理器从内存中读取指定地址数据至Cache。由于硬件预取器通常无法正常动态关闭，因此大部

分情况下软件预取和硬件预取是并存的，软件预取必须尽力配合硬件预取以取得更优的效果。本文假设硬件预取器被关闭后，讨论如何利用软件预取达到性能提升的效果。

预取指令prefech(x)只是一种提示，告知硬件开始将地址x中的数据从主存中读取到Cache中。它并不会引起处理停顿，但若硬件发现会产生异常，则会忽略这个预取操作。如果prefech(x)成功，则意味着下一次取x将命中Cache；不成功的预取操作可能会导致下次读取时发生Cache miss，但不会影响程序的正确性[2]。

数据预取是如何改成程序性能的呢？如下一段程序：

```
1. double a[n];
2. for (int i = 0; i < 100; i++)
3.  a[i] = 0;
```

假设一个Cache line可以存放两个double元素，当第一次访问a[0]时，由于a[0]不在Cache中，会发生一次Cache miss，需要从主存中将其加载至Cache中，由于一个Cache line可以存放两个double元素，当访问a[1]时则不会发生Cache miss。依次类推，访问a[2]时会发生Cache miss，访问a[3]时不会发生Cache miss，我们很容易得到程序总共发生了50次Cache miss。

我们可以通过软件预取等相关优化，降低Cache miss次数，提高程序性能。首先介绍一个公式[3]：

$$iterationsAhead = L/S$$

上述公式中L是memory latency，S是执行一次循环迭代最短的时间。iterationAhead表示的是循环需要经过执行几次迭代，预取的数据才会到达Cache。假设我们的硬件架构计算出来的iterationAhead=6，那么原程序可以优化成如下程序：

```
1. double a[n];
2. for (int i = 0; i < 12; i+=2)   //prologue
3.  prefetch(&a[i]);
4. for (int i = 0; i < 88; i+=2) { // steady state
5.  prefetch(&a[i+12]);
6.  a[i] = 0;
7.  a[i+1] = 0;
8. }
9. for (int i = 88; i < 100; i++) //epilogue
10.  a[i] = 0;
```

由于我们的硬件架构需要循环执行6次后，预取的数据才会到达Cache。一个Cache line可以存放两个double元素，为了避免浪费prefetch指令，所以prologue和steady state循环都展开了，即执行prefetch(&a[0])后会将a[0]、a[1]从主存加载至Cache中，下次执行预取时就无需再次将a[1]从主存加载至Cache了。prologue循环先执行数组a的前12个元素的预取指令，等到执行steady state循环时，当i = 0时，a[0]和a[1]已经被加载至Cache中，就不会发生Cache miss了。依次类推，经过上述优化后，在不改变语义的基础上，通过使用预取指令，程序的Cache miss次数从50下降至0，程序的性能将会得到很大提升。

注意，预取并不能减少从主存储器取数据到高速缓存的延迟，只是通过预取与计算重叠而隐藏这种延迟。总之，当处理器有预取指令或者有能够用作预取的非阻塞的读取指令时，对于处理器不能动态重排指令或者动态重排缓冲区小于我们希望隐藏的具体Cache延迟，并且所考虑的数据大于Cache或者是不能够判断数据是否已在Cache中，预取是适用的。预取也不是万能，不

当的预取可能会导致高速缓存冲突，程序性能降低。我们应该首先利用数据重用来减少延迟，然后才考虑预取。

除了软件预取外，ARMv8还提供了Non-temporal的Load/Store指令，可以提高Cache的利用率。对于一些数据，如果只是访问一次，无需占用Cache，可以使用这个指令进行访问，从而保护Cache中关键数据不被替换，比如memcpy大数据的场景下，使用该指令对于其关键业务而言，是有一定的收益的。

循环变换

重用Cache中的数据是最基本的高效使用Cache方法。对于多层嵌套循环，可以通过交换两个嵌套的循环(loop interchange)、逆转循环迭代执行的顺序(loop reversal)、将两个循环体合并成一个循环体(loop fusion)、循环拆分(loop distribution)、循环分块(loop tiling)、loop unroll and jam等循环变换操作。选择适当的循环变换方式，既能保持程序的语义，又能改善程序性能。我们做这些循环变换的主要目的是为了实现寄存器、数据高速缓存以及其他存储层次使用方面的优化。

篇幅受限，本节仅讨论循环分块(loop tiling)如何改善程序性能，若对loop interchange感兴趣，请点击查阅。下面这个简单的循环：

```
1. for(int i = 0; i < m; i++) {
2.  for(int j = 0; j < n; j++) {
3.   x = x+a[i]+c*b[j];
4.  }
5. }
```

我们假设数组a、b都是超大数组，m、n相等且都很大，程序不会出现数组越界访问情况发生。那么如果b[j]在j层循环中跨度太大时，那么被下次i层循环重用时数据已经被清出高速缓存。即程序访问b[n-1]时，b[0]、b[1]已经被清出缓存，此时需要重新从主存中将数据加载至缓存中，程序性能会大幅下降。

我们如何通过降低Cache miss次数提升程序的性能呢？通过对循环做loop tiling可以符合我们的期望，即通过循环重排，使得数据分成一个一个tile，让每一个tile的数据都可以在Cache中被hint[4]。从内层循环开始tiling，假设tile的大小为t，t远小于m、n，t的取值使得b[t-1]被访问时b[0]依然在Cache中，将会大幅地减少Cache miss次数。假设n-1恰好被t整除，此时b数组的访问顺序如下所示：

```
 1. i=1; b[0]、b[1]、b[2]...b[t-1]
 2. i=2; b[0]、b[1]、b[2]...b[t-1]
 3. ...
 4. i=n; b[0]、b[1]、b[2]...b[t-1]
 5. ...
 6. ...
 7. ...
 8. i=1; b[n-t]、b[n-t-1]、b[n-t-2]...b[n-1]
 9. i=2; b[n-t]、b[n-t-1]、b[n-t-2]...b[n-1]
10. ...
11. i=n; b[n-t]、b[n-t-1]、b[n-t-2]...b[n-1]
```

经过loop tiling后循环变换成：

```
1. for(int j = 0; j < n; j+=t) {
```

```
2.  for(int i = 0; i < m; i++) {
3.   for(int jj = j; jj < min(j+t, n); jj++) {
4.    x = x+a[i]+c*b[jj];
5.   }
6.  }
7. }
```

假设每个Cache line能够容纳X个数组元素，loop tiling前a的Cache miss次数为m/X，b的Cache miss次数是m*n/X，总的Cache miss次数为m*(n+1)/x。loop tiling后a的Cache miss次数为(n/t)*(m/X)，b的Cache miss次数为(t/X)*(n/t)=n/X，总的Cache miss次数为n*(m+t)/xt。此时，由于n与m相等，那么loop tiling后Cache miss大约可以降低t倍[4]。

前文讨论了loop tiling在小用例上如何提升程序性能，总之针对不同的循环场景，选择合适的循环交换方法，既能保证程序语义正确， 又能获得改善程序性能的机会。

小结

汝之蜜糖，彼之砒霜。针对不同的硬件，我们需要结合具体的硬件架构，利用性能分析工具，通过分析报告和程序，从系统层次和算法层次思考问题，往往会有意想不到的收获。本文简单地介绍了内存层次优化相关的几种方法，结合一些小例子深入浅出地讲解了一些内存层次优化相关的知识。纸上得来终觉浅，绝知此事要躬行，更多性能优化相关的知识需要我们从实践中慢慢摸索。

参考

1. John L. Hennessy, David A. Patterson. 计算机体系结构：量化研究方法(第6版). 贾洪峰,译

2. Andrew W.Apple, with Jens Palsberg. Modern Compiler Implenentation in C

3. http://www.cs.cmu.edu/afs/cs/academic/class/15745-s19/www/lectures/L20-Global-Scheduling.pdf

4. https://zhuanlan.zhihu.com/p/292539074

往期推荐：

编译器优化那些事儿(1)：SLP矢量化介绍

编译器优化那些事儿(2)：常量传播

编译器优化那些事儿(3)：Lazy Code Motion

编译器优化那些事儿(4)：归纳变量

编译器优化那些事儿(5)：寄存器分配

编译器优化那些事儿(6)：别名分析概述