每个程序员都应该了解的 CPU 高速缓存

[编者按：这是Ulrich Drepper写“程序员都该知道存储器”的第二部。那些没有读过*第一部*的读者可能希望从这一部开始。这本书写的非常好，并且感谢Ulrich授权我们出版。

一点说明：书籍出版时可能会有一些印刷错误，如果你发现，并且想让它在后续的出版中更正，请将意见发邮件到lwn@lwn.net ，我们一定会更正，并反馈给Ulrich的文档副本，别的读者就不会受到这些困扰。]

现在的CPU比25年前要精密得多了。在那个年代，CPU的频率与内存总线的频率基本在同一层面上。内存的访问速度仅比寄存器慢那么一点点。但是，这一局面在上世纪90年代被打破了。CPU的频率大大提升，但内存总线的频率与内存芯片的性能却没有得到成比例的提升。并不是因为造不出更快的内存，只是因为太贵了。内存如果要达到目前CPU那样的速度，那么它的造价恐怕要贵上好几个数量级。

如果有两个选项让你选择，一个是速度非常快、但容量很小的内存，一个是速度还算快、但容量很多的内存，如果你的工作集比较大，超过了前一种情况，那么人们总是会选择第二个选项。原因在于辅存(一般为磁盘)的速度。由于工作集超过主存，那么必须用辅存来保存交换出去的那部分数据，而辅存的速度往往要比主存慢上好几个数量级。

好在这问题也并不全然是非甲即乙的选择。在配置大量DRAM的同时，我们还可以配置少量SRAM。将地址空间的某个部分划给SRAM，剩下的部分划给DRAM。一般来说，SRAM可以当作扩展的寄存器来使用。

上面的做法看起来似乎可以，但实际上并不可行。首先，将SRAM内存映射到进程的虚拟地址空间就是个非常复杂的工作，而且，在这种做法中，每个进程都需要管理这个SRAM区内存的分配。每个进程可能有大小完全不同的SRAM区，而组成程序的每个模块也需要索取属于自身的SRAM，更引入了额外的同步需求。简而言之，快速内存带来的好处完全被额外的管理开销给抵消了。 因此，SRAM是作为CPU自动使用和管理的一个资源，而不是由OS或者用户管理的。在这种模式下，SRAM用来复制保存（或者叫缓存）主内存中有可能即将被CPU使用的数据。这意味着，在较短时间内，CPU很有可能重复运行某一段代码，或者重复使用某部分数据。从代码上看，这意味着CPU执行了一个循环，所以相同的代码一次又一次地执行（空间局部性的绝佳例子）。数据访问也相对局限在一个小的区间内。即使程序使用的物理内存不是相连的，在短期内程序仍然很有可能使用同样的数据（时间局部性）。这个在代码上表现为，程序在一个循环体内调用了入口一个位于另外的物理地址的函数。这个函数可能与当前指令的物理位置相距甚远，但是调用的时间差不大。在数据上表现为，程序使用的内存是有限的（相当于工作集的大小）。但是实际上由于RAM的随机访问特性，程序使用的物理内存并不是连续的。正是由于空间局部性和时间局部性的存在，我们才提炼出今天的CPU缓存概念。

我们先用一个简单的计算来展示一下高速缓存的效率。假设，访问主存需要200个周期，而访问高速缓存需要15个周期。如果使用100个数据元素100次，那么在没有高速缓存的情况下，需要2000000个周期，而在有高速缓存、而且所有数据都已被缓存的情况下，只需要168500个周期。节约了91.5%的时间。

用作高速缓存的SRAM容量比主存小得多。以我的经验来说，高速缓存的大小一般是主存的千分之一左右(目前一般是4GB主存、4MB缓存)。这一点本身并不是什么问题。只是，计算机一般都会有比较大的主存，因此工作集的大小总是会大于缓存。特别是那些运行多进程的系统，它的工作集大小是所有进程加上内核的总和。

处理高速缓存大小的限制需要制定一套很好的策略来决定在给定的时间内什么数据应该被缓存。由于不是所有数据的工作集都是在完全相同的时间段内被使用的，我们可以用一些技术手段将需要用到的数据临时替换那些当前并未使用的缓存数据。这种预取将会减少部分访问主存的成本，因为它与程序的执行是异步的。所有的这些技术将会使高速缓存在使用的时候看起来比实际更大。我们将在3.3节讨论这些问题。 我们将在第6章讨论如何让这些技术能很好地帮助程序员，让处理器更高效地工作。

3.1 高速缓存的位置

在深入介绍高速缓存的技术细节之前，有必要说明一下它在现代计算机系统中所处的位置。

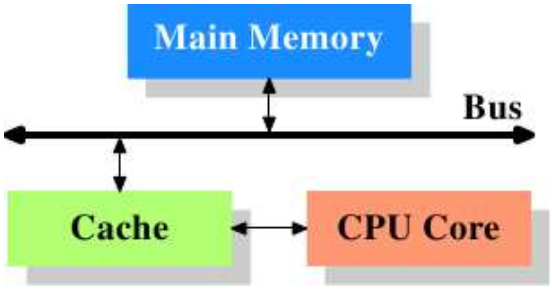


图3.1: 最简单的高速缓存配置图

图3.1展示了最简单的高速缓存配置。早期的一些系统就是类似的架构。在这种架构中，CPU核心不再直连到主存。{在一些更早的系统中，高速缓存像CPU与主存一样连到系统总线上。那种做法更像是一种hack，而不是真正的解决方案。}数据的读取和存储都经过高速缓存。CPU核心与高速缓存之间是一条特殊的快速通道。在简化的表示法中，主存与高速缓存都连到系统总线上，这条总线同时还用于与其它组件通信。我们管这条总线叫“FSB”——就是现在称呼它的术语，参见第2.2节。在这一节里，我们将忽略北桥。

在过去的几十年，经验表明使用了冯诺伊曼结构的 计算机，将用于代码和数据的高速缓存分开是存在巨大优势的。自1993年以来，Intel 并且一直坚持使用独立的代码和数据高速缓存。由于所需的代码和数据的内存区域是几乎相互独立的，这就是为什么独立缓存工作得更完美的原因。近年来，独立缓存的另一个优势慢慢显现出来：常见处理器解码 指令的步骤 是缓慢的，尤其当管线为空的时候，往往会伴随着错误的预测或无法预测的分支的出现， 将高速缓存技术用于 指令 解码可以加快其执行速度。

在高速缓存出现后不久，系统变得更加复杂。高速缓存与主存之间的速度差异进一步拉大，直到加入了另一级缓存。新加入的这一级缓存比第一级缓存更大，但是更慢。由于加大一级缓存的做法从经济上考虑是行不通的，所以有了二级缓存，甚至现在的有些系统拥有三级缓存，如图3.2所示。随着单个CPU中核数的增加，未来甚至可能会出现更多层级的缓存。

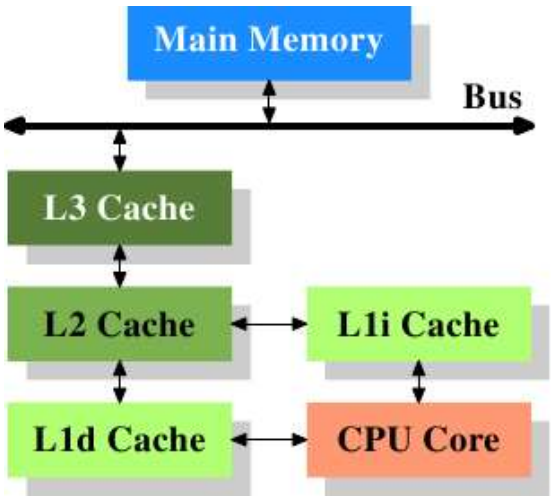


图3.2: 三级缓存的处理器

图3.2展示了三级缓存，并介绍了本文将使用的一些术语。L1d是一级数据缓存，L1i是一级指令缓存，等等。请注意，这只是示意图，真正的数据流并不需要流经上级缓存。CPU的设计者们在设计高速缓存的接口时拥有很大的自由。而程序员是看不到这些设计选项的。

另外，我们有多核CPU，每个核心可以有多个“线程”。核心与线程的不同之处在于，核心拥有独立的硬件资源({早期的多核CPU甚至有独立的二级缓存。})。在不同时使用相同资源(比如，通往外界的连接)的情况下，核心可以完全独立地运行。而线程只是共享资源。Intel的线程只有独立的寄存器，而且还有限制——不是所有寄存器都独立，有些是共享的。综上，现代CPU的结构就像图3.3所示。

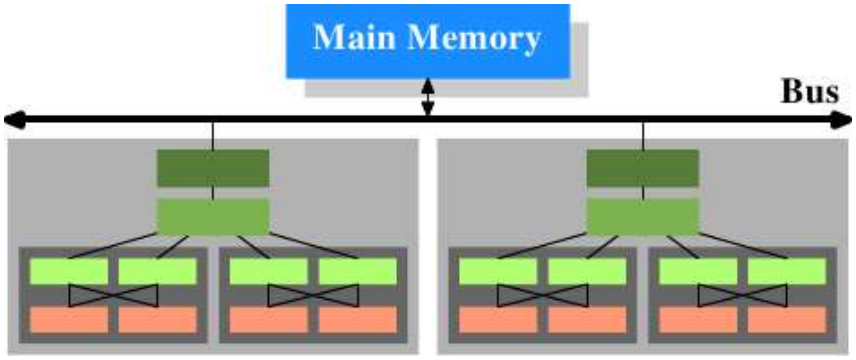


图3.3 多处理器、多核心、多线程

在上图中，有两个处理器，每个处理器有两个核心，每个核心有两个线程。线程们共享一级缓存。核心(以深灰色表示)有独立的一级缓存，同时共享二级缓存。处理器(淡灰色)之间不共享任何缓存。这些信息很重要，特别是在讨论多进程和多线程情况下缓存的影响时尤为重要。

3.2 高级的缓存操作

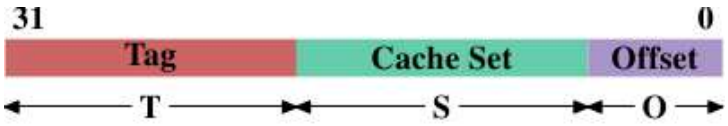
了解成本和节约使用缓存，我们必须结合在第二节中讲到的关于计算机体系结构和RAM技术，以及前一节讲到的缓存描述来探讨。

默认情况下，CPU核心所有的数据的读或写都存储在缓存中。当然，也有内存区域不能被缓存的，但是这种情况只发生在操作系统的实现者对数据考虑的前提下；对程序实现者来说，这是不可见的。这也说明，程序设计者可以故意绕过某些缓存，不过这将是第六节中讨论的内容了。

如果CPU需要访问某个字(word)，先检索缓存。很显然，缓存不可能容纳主存所有内容(否则还需要主存干嘛)。系统用字的内存地址来对缓存条目进行标记。如果需要读写某个地址的字，那么根据标签来检索缓存即可。这里用到的地址可以是虚拟地址，也可以是物理地址，取决于缓存的具体实现。

标签是需要额外空间的，用字作为缓存的粒度显然毫无效率。比如，在x86机器上，32位字的标签可能需要32位，甚至更长。另一方面，由于空间局部性的存在，与当前地址相邻的地址有很大可能会被一起访问。再回忆下2.2.1节——内存模块在传输位于同一行上的多份数据时，由于不需要发送新CAS信号，甚至不需要发送RAS信号，因此可以实现很高的效率。基于以上的原因，缓存条目并不存储单个字，而是存储若干连续字组成的“线”。在早期的缓存中，线长是32字节，现在一般是64字节。对于64位宽的内存总线，每条线需要8次传输。而DDR对于这种传输模式的支持更为高效。

当处理器需要内存中的某块数据时，整条缓存线被装入L1d。缓存线的地址通过对内存地址进行掩码操作生成。对于64字节的缓存线，是将低6位置0。这些被丢弃的位作为线内偏移量。其它的位作为标签，并用于在缓存内定位。在实践中，我们将地址分为三个部分。32位地址的情况如下:



如果缓存线长度为2^O，那么地址的低O位用作线内偏移量。上面的S位选择“缓存集”。后面我们会说明使用缓存集的原因。现在只需要明白一共有2^S个缓存集就够了。剩下的32 - S - O = T位组成标签。它们用来区分别名相同的各条线{有相同S部分的缓存线被称为有相同的别名。}用于定位缓存集的S部分不需要存储，因为属于同一缓存集的所有线的S部分都是相同的。

当某条指令修改内存时，仍然要先装入缓存线，因为任何指令都不可能同时修改整条线(只有一个例外——第6.1节中将会介绍的写合并(write-combine))。因此需要在写操作前先把缓存线装载进来。如果缓存线被写入，但还没有写回主存，那就是所谓的“脏了”。脏了的线一旦写回主存，脏标记即被清除。

为了装入新数据，基本上总是要先在缓存中清理出位置。L1d将内容逐出L1d，推入L2(线长相同)。当然，L2也需要清理位置。于是L2将内容推入L3，最后L3将它推入主存。这种逐出操作一级比一级昂贵。这里所说的是现代AMD和VIA处理器所采用的*独占型缓存(exclusive cache)*。而Intel采用的是*包容型缓存(inclusive cache)*，{并不完全正确，Intel有些缓存是独占型的，还有一些缓存具有独占型缓存的特点。}L1d的每条线同时存在于L2里。对这种缓存，逐出操作就很快了。如果有足够L2，对于相同内容存在不同地方造成内存浪费的缺点可以降到最低，而且在逐出时非常有利。而独占型缓存在装载新数据时只需要操作L1d，不需要碰L2，因此会比较快。

处理器体系结构中定义的作为存储器的模型只要还没有改变，那就允许多CPU按照自己的方式来管理高速缓存。这表示，例如，设计优良的处理器，利用很少或根本没有内存总线活动，并主动写回主内存脏高速缓存行。这种高速缓存架构在如x86和x86-64各种各样的处理器间存在。制造商之间，即使在同一制造商生产的产品中，证明了的内存模型抽象的力量。

在对称多处理器（SMP）架构的系统中，CPU的高速缓存不能独立的工作。在任何时候，所有的处理器都应该拥有相同的内存内容。保证这样的统一的内存视图被称为“高速缓存一致性”。如果在其自己的高速缓存和主内存间，处理器设计简单，它将不会看到在其他处理器上的脏高速缓存行的内容。从一个处理器直接访问另一个处理器的高速缓存这种模型设计代价将是非常昂贵的，它是一个相当大的瓶颈。相反，当另一个处理器要读取或写入到高速缓存线上时，处理器会去检测。

如果CPU检测到一个写访问，而且该CPU的cache中已经缓存了一个cache line的原始副本，那么这个cache line将被标记为无效的cache line。接下来在引用这个cache line之前，需要重新加载该cache line。需要注意的是读访问并不会导致cache line被标记为无效的。

更精确的cache实现需要考虑到其他更多的可能性，比如第二个CPU在读或者写他的cache line时，发现该cache line在第一个CPU的cache中被标记为脏数据了，此时我们就需要做进一步的处理。在这种情况下，主存储器已经失效，第二个CPU需要读取第一个CPU的cache line。通过测试，我们知道在这种情况下第一个CPU会将自己的cache line数据自动发送给第二个CPU。这种操作是绕过主存储器的，但是有时候存储控制器是可以直接将第一个CPU中的cache line数据存储到主存储器中。对第一个CPU的cache的写访问会导致本地cache line的所有拷贝被标记为无效。

随着时间的推移，一大批缓存一致性协议已经建立。其中，最重要的是MESI,我们将在第3.3.4节进行介绍。以上结论可以概括为几个简单的规则:

- 一个脏缓存线不存在于任何其他处理器的缓存之中。
- 同一缓存线中的干净拷贝可以驻留在任意多个其他缓存之中。

如果遵守这些规则,处理器甚至可以在多处理器系统中更加有效的使用它们的缓存。所有的处理器需要做的就是监控其他每一个写访问和比较本地缓存中的地址。在下一节中,我们将介绍更多细节方面的实现,尤其是存储开销方面的细节。

最后，我们至少应该关注高速缓存命中或未命中带来的消耗。下面是英特尔奔腾 M 的数据：

To Where	Cycles
Register	<= 1
L1d	~3
L2	~14
Main Memory	~240

这是在CPU周期中的实际访问时间。有趣的是，对于L2高速缓存的访问时间很大一部分（甚至是大部分）是由线路的延迟引起的。这是一个限制，增加高速缓存的大小变得更糟。只有当减小时（例如，从60纳米的Merom到45纳米Penryn处理器），可以提高这些数据。

表格中的数字看起来很高，但是，幸运的是，整个成本不必须负担每次出现的缓存加载和缓存失效。某些部分的成本可以被隐藏。现在的处理器都使用不同长度的内部管道，在管道内指令被解码，并为准备执行。如果数据要传送到一个寄存器，那么部分的准备工作是从存储器（或高速缓存）加载数据。如果内存加载操作在管道中足够早的进行，它可以与其他操作并行发生，那么加载的全部发销可能会被隐藏。对L1D常常可能如此；某些有长管道的处理器的L2也可以。

提早启动内存的读取有许多障碍。它可能只是简单的不具有足够资源供内存访问，或者地址从另一个指令获取，然后加载的最终地址才变得可用。在这种情况下，加载成本是不能隐藏的（完全的）。

对于写操作，CPU并不需要等待数据被安全地放入内存。只要指令具有类似的效果，就没有什么东西可以阻止CPU走捷径了。它可以早早地执行下一条指令，甚至可以在影子寄存器(shadow register)的帮助下，更改这个写操作将要存储的数据。

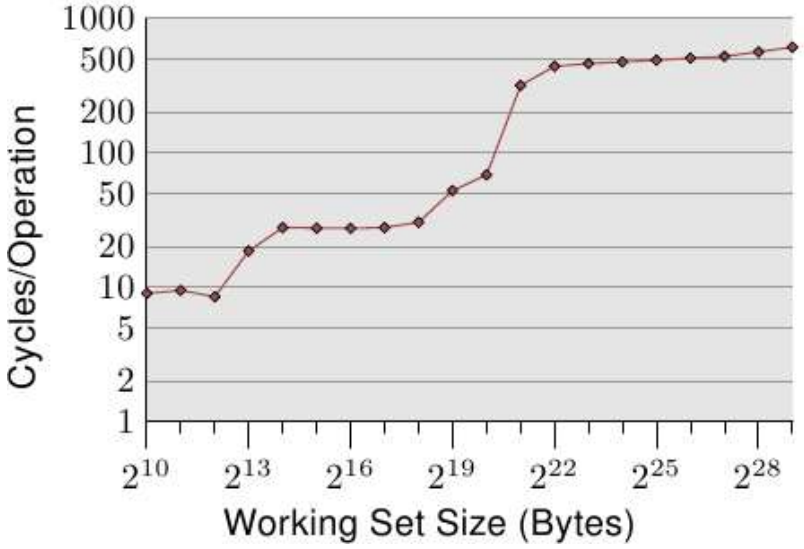


图3.4: 随机写操作的访问时间

图3.4展示了缓存的效果。关于产生图中数据的程序，我们会在稍后讨论。这里大致说下，这个程序是连续随机地访问某块大小可配的内存区域。每个数据项的大小是固定的。数据项的多少取决于选择的工作集大小。Y轴表示处理每个元素平均需要多少个CPU周期，注意它是对数刻度。X轴也是同样，工作集的大小都以2的n次方表示。

图中有三个比较明显的不同阶段。很正常，这个处理器有L1d和L2，没有L3。根据经验可以推测出，L1d有2^{13}字节，而L2有2^{20}字节。因为，如果整个工作集都可以放入L1d，那么只需不到10个周期就可以完成操作。如果工作集超过L1d，处理器不得不从L2获取数据，于是时间飘升到28个周期左右。如果工作集更大，超过了L2，那么时间进一步暴涨到480个周期以上。这时候，许多操作将不得不从主存中获取数据。更糟糕的是，如果修改了数据，还需要将这些脏了的缓存线写回内存。

看了这个图，大家应该会有足够的动力去检查代码、改进缓存的利用方式了吧？这里的性能改善可不只是微不足道的几个百分点，而是几个数量级呀。在第6节中，我们将介绍一些编写高效代码的技巧。而下一节将进一步深入缓存的设计。虽然精彩，但并不是必修课，大家可以选择性地跳过。

3.3 CPU缓存实现的细节

缓存的实现者们都要面对一个问题——主存中每一个单元都可能需被缓存。如果程序的工作集很大，就会有许多内存位置为了缓存而打架。前面我们曾经提过缓存与主存的容量比，1:1000也十分常见。

3.3.1 关联性

我们可以让缓存的每条线能存放任何内存地址的数据。这就是所谓的*全关联缓存(fully associative cache)*。对于这种缓存，处理器为了访问某条线，将不得不检索所有线的标签。而标签则包含了整个地址，而不仅仅只是线内偏移量(也就意味着，图3.2中的S为0)。

高速缓存有类似这样的实现，但是，看看在今天使用的L2的数目，表明这是不切实际的。给定4MB的高速缓存和64B的高速缓存段，高速缓存将有65,536个项。为了达到足够的性能，缓存逻辑必须能够在短短的几个时钟周期内，从所有这些项中，挑一个匹配给定的标签。实现这一点的工作将是巨大的。

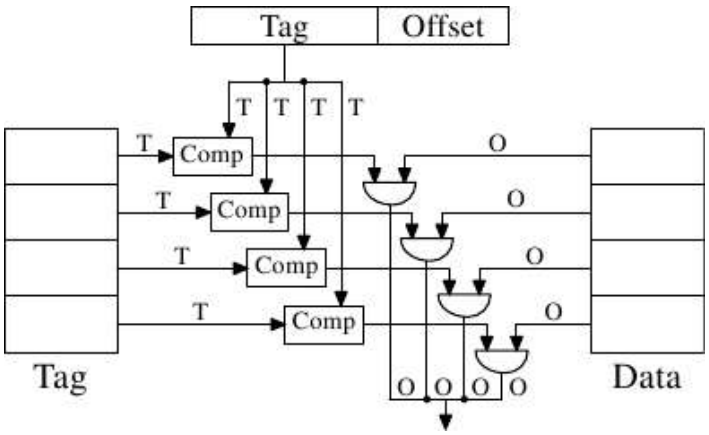


Figure 3.5: 全关联高速缓存原理图

对于每个高速缓存行，比较器是需要比较大标签（注意，S是零）。每个连接旁边的字母表示位的宽度。如果没有给出，它是一个单比特线。每个比较器都要比较两个T-位宽的值。然后，基于该结果，适当的高速缓存行的内容被选中，并使其可用。这需要合并多套O数据线，因为他们是缓存桶（译注：这里类似把O输出接入多选器，所以需要合并）。实现仅仅一个比较器，需要晶体管的数量就非常大，特别是因为它必须非常快。没有迭代比较器是可用的。节省比较器的数目的唯一途径是通过反复比较标签，以减少它们的数目。这是不适合的，出于同样的原因，迭代比较器不可用：它的时间太长。

全关联高速缓存对 小缓存是实用的（例如，在某些Intel处理器的TLB缓存是全关联的），但这些缓存都很小，非常小的。我们正在谈论的最多几十项。

对于L1i，L1d和更高级别的缓存，需要采用不同的方法。可以做的就是是限制搜索。最极端的限制是，每个标签映射到一个明确的缓存条目。计算很简单：给定的4MB/64B缓存有65536项，我们可以使用地址的bit6到bit21（16位）来直接寻址高速缓存的每一个项。地址的低6位作为高速缓存段的索引。

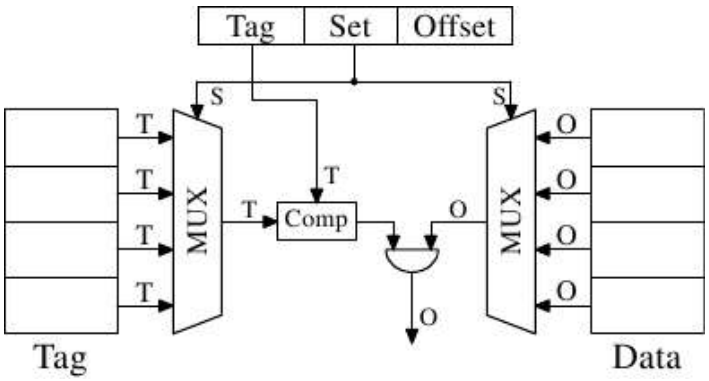


Figure 3.6: Direct-Mapped Cache Schematics

在图3.6中可以看出，这种直接映射的高速缓存，速度快，比较容易实现。它只是需要一个比较器，一个多路复用器（在这个图中有两个，标记和数据是分离的，但是对于设计这不是一个硬性要求），和一些逻辑来选择只是有效的高速缓存行的内容。由于速度的要求，比较器是复杂的，但是现在只需要一个，结果是可以花更多的精力，让其变得快速。这种方法的复杂性在于在多路复用器。一个简单的多路转换器中的晶体管的数量增速是O（log N）的，其中N是高速缓存段的数目。这是可以容忍的，但可能会很慢，在某种情况下，速度可提升，通过增加多路复用器晶体管数量，来并行化的一些工作和自身增速。晶体管的总数只是随着快速增长的高速缓存缓慢的增加，这使得这种解决方案非常有吸引力。但它有一个缺点：只有用于直接映射地址的相关的地址位均匀分布，程序才能很好工作。如果分布的不均匀，而且这是常态，一些缓存项频繁的使用，并因此多次被换出，而另一些则几乎不被使用或一直是空的。

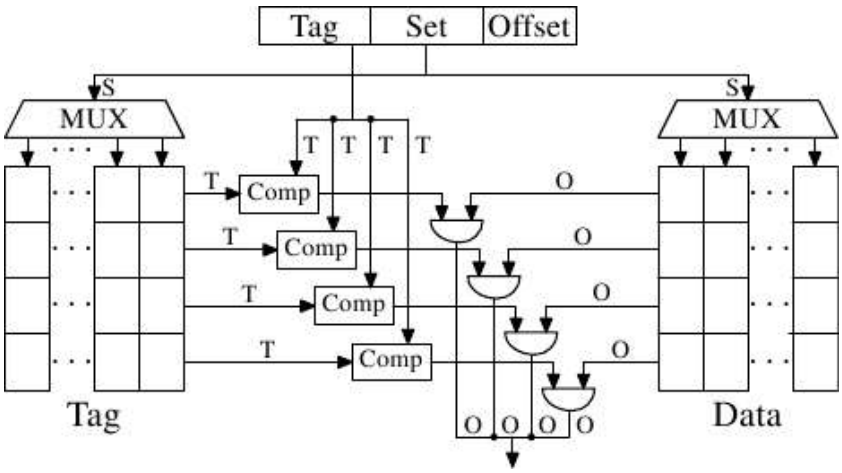


Figure 3.7: 组关联高速缓存原理图

可以通过使高速缓存的组关联来解决此问题。组关联结合高速缓存的全关联和直接映射高速缓存特点，在很大程度上避免那些设计的弱点。图3.7显示了一个组关联高速缓存的设计。标签和数据存储分成不同的组并可以通过地址选择。这类似直接映射高速缓存。但是，小数目的值可以在同一个高速缓存组缓存，而不是一个缓存组只有一个元素，用于在高速缓存中的每个设定值是相同的一组值的缓存。所有组的成员的标签可以并行比较，这类似全关联缓存的功能。

其结果是高速缓存，不容易被不幸或故意选择同属同一组编号的地址所击败，同时高速缓存的大小并不限于由比较器的数目，可以以并行的方式实现。如果高速缓存增长，只（在该图中）增加列的数目，而不增加行数。只有高速缓存之间的关联性增加，行数才会增加。今天，处理器的L2高速缓存或更高的高速缓存，使用的关联性高达16。 L1高速缓存通常使用8。

L2 Cache Size	Associativity							
Direct	2	4	8					
CL=32	CL=64	CL=32	CL=64	CL=32	CL=64	CL=32	CL=64	
512k	27,794,595	20,422,527	25,222,611	18,303,581	24,096,510	17,356,121	23,666,929	17,029,334
1M	19,007,315	13,903,854	16,566,738	12,127,174	15,537,500	11,436,705	15,162,895	11,233,896
2M	12,230,962	8,801,403	9,081,881	6,491,011	7,878,601	5,675,181	7,391,389	5,382,064
4M	7,749,986	5,427,836	4,736,187	3,159,507	3,788,122	2,418,898	3,430,713	2,125,103
8M	4,731,904	3,209,693	2,690,498	1,602,957	2,207,655	1,228,190	2,111,075	1,155,847
16M	2,620,587	1,528,592	1,958,293	1,089,580	1,704,878	883,530	1,671,541	862,324

Table 3.1: 高速缓存大小，关联行，段大小的影响

给定我们4MB/64B高速缓存，8路组关联，相关的缓存留给我们的有8192组，只用标签的13位，就可以寻址缓集。要确定哪些（如果有的话）的缓存组设置中的条目包含寻址的高速缓存行，8个标签都要进行比较。在很短的时间内做出来是可行的。通过一个实验，我们可以看到，这是有意义的。

表3.1显示一个程序在改变缓存大小，缓存段大小和关联集大小，L2高速缓存的缓存失效数量（根据Linux内核相关的方面人的说法，GCC在这种情况下，是他们所有中最重要的标尺）。在7.2节中，我们将介绍工具来模拟此测试要求的高速缓存。

万一这还不是很明显，所有这些值之间的关系是高速缓存的大小为：

cache line size × associativity × number of sets

地址被映射到高速缓存使用

O = log $_2$ cache line size

S = log $_2$ number of sets

在第3.2节中的图显示的方式。

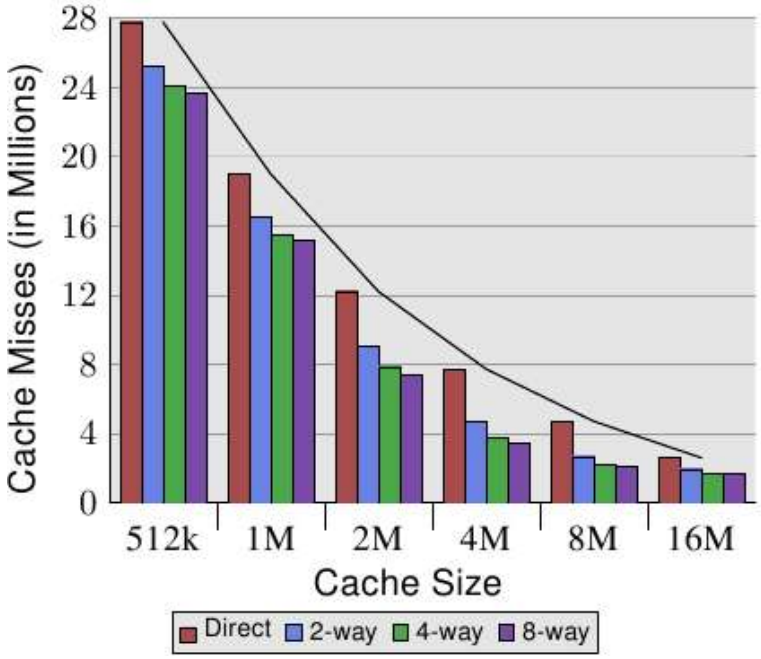


Figure 3.8: 缓存段大小 vs 关联行 (CL=32)

图3.8表中的数据更易于理解。它显示一个固定的32个字节大小的高速缓存行的数据。对于一个给定的高速缓存大小，我们可以看出，关联性，的确可以帮助明显减少高速缓存未命中的数量。对于8MB的缓存，从直接映射到2路组相联，可以减少近44%的高速缓存未命中。组相联高速缓存和直接映射缓存相比，该处理器可以把更多的工作集保持在缓存中。

在文献中，偶尔可以读到，引入关联性，和加倍高速缓存的大小具有相同的效果。在从4M缓存跃升到8MB缓存的极端的情况下，这是正确的。关联性再提高一倍那就肯定不正确啦。正如我们所看到的数据，后面的收益要小得多。我们不应该完全低估它的效果，虽然。在示例程序中的内存使用的峰值是5.6M。因此，具有8MB缓存不太可能有很多（两个以上）使用相同的高速缓存的组。从较小的缓存的关联性的巨大收益可以看出，较大工作集可以节省更多。

在一般情况下，增加8以上的高速缓存之间的关联性似乎对只有一个单线程工作量影响不大。随着介绍一个使用共享L2的多核处理器，形势发生了变化。现在你基本上有两个程序命中相同的缓存， 实际上导致高速缓存减半（对于四核处理器是1/4）。因此，可以预期，随着核的数目的增加，共享高速缓存的相关性也应增长。一旦这种方法不再可行（16 路组关联性已经很难）处理器设计者不得不开始使用共享的三级高速缓存和更高级别的，而L2高速缓存只被核的一个子集共享。

从图3.8中，我们还可以研究缓存大小对性能的影响。这一数据需要了解工作集的大小才能进行解读。很显然，与主存相同的缓存比小缓存能产生更好的结果，因此，缓存通常是越大越好。

上文已经说过，示例中最大的工作集为5.6M。它并没有给出最佳缓存大小值，但我们可以估算出来。问题主要在于内存的使用并不连续，因此，即使是16M的缓存，在处理5.6M的工作集时也会出现冲突(参见2路集合关联式16MB缓存vs直接映射式缓存的优点)。不管怎样，我们可以有把握地说，在同样5.6M的负载下，缓存从16MB升到32MB基本已没有多少提高的余地。但是，工作集是会变的。如果工作集不断增大，缓存也需要随之增大。在购买计算机时，如果需要选择缓存大小，一定要先衡量工作集的大小。原因可以参见图3.10。

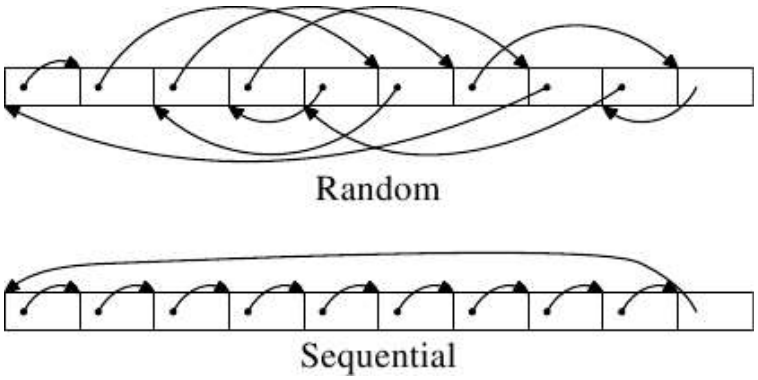


图3.9: 测试的内存分布情况

我们执行两项测试。第一项测试是按顺序地访问所有元素。测试程序循着指针n进行访问，而所有元素是链接在一起的，从而使它们的被访问顺序与在内存中排布的顺序一致，如图3.9的下半部分所示，末尾的元素有一个指向首元素的引用。而第二项测试(见图3.9的上半部分)则是按随机顺序访问所有元素。在上述两个测试中，所有元素都构成一个单向循环链表。

3.3.2 Cache的性能测试

用于测试程序的数据可以模拟一个任意大小的工作集：包括读、写访问，随机、连续访问。在图3.4中我们可以看到，程序为工作集创建了一个与其大小和元素类型相同的数组：

```
struct l {
  struct l *n;
  long int pad[NPAD];
};
```

n字段将所有节点随机得或者顺序的加入到环形链表中，用指针从当前节点进入到下一个节点。pad字段用来存储数据，其可以是任意大小。在一些测试程序中，pad字段是可以修改的, 在其他程序中，pad字段只可以进行读操作。

在性能测试中，我们谈到工作集大小的问题，工作集使用结构体l定义的元素表示的。2^N 字节的工作集包含

2^N/sizeof(struct l)

个元素. 显然sizeof(struct l) 的值取决于NPAD的大小。在32位系统上，NPAD=7意味着数组的每个元素的大小为32字节，在64位系统上，NPAD=7意味着数组的每个元素的大小为64字节。

单线程顺序访问

最简单的情况就是遍历链表中顺序存储的节点。无论是从前向后处理，还是从后向前，对于处理器来说没有什么区别。下面的测试中，我们需要得到处理链表中一个元素所需要的时间，以CPU时钟周期最为计时单元。图3.10显示了测试结构。除非有特殊说明, 所有的测试都是在Pentium 4 64-bit 平台上进行的，因此结构体l中NPAD=0，大小为8字节。

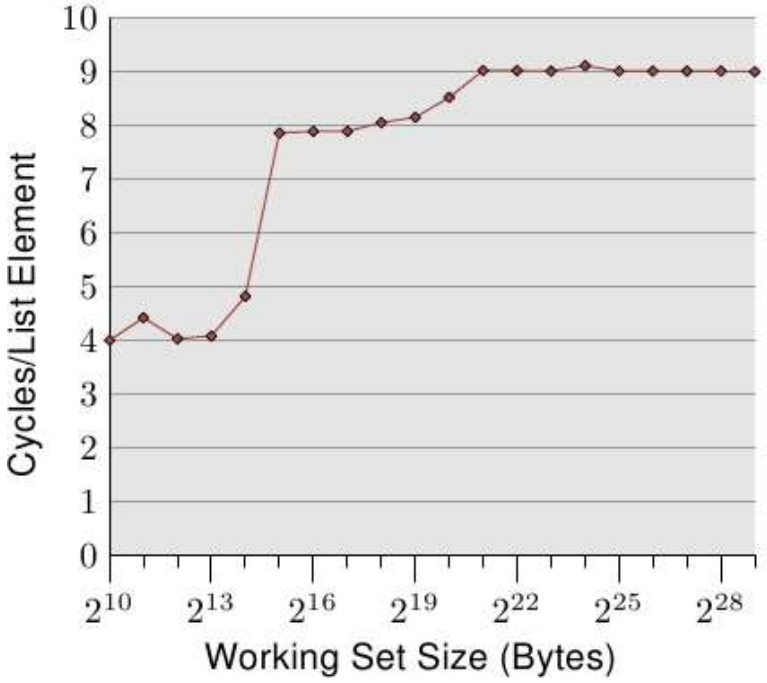


图 3.10: 顺序读访问, NPAD=0

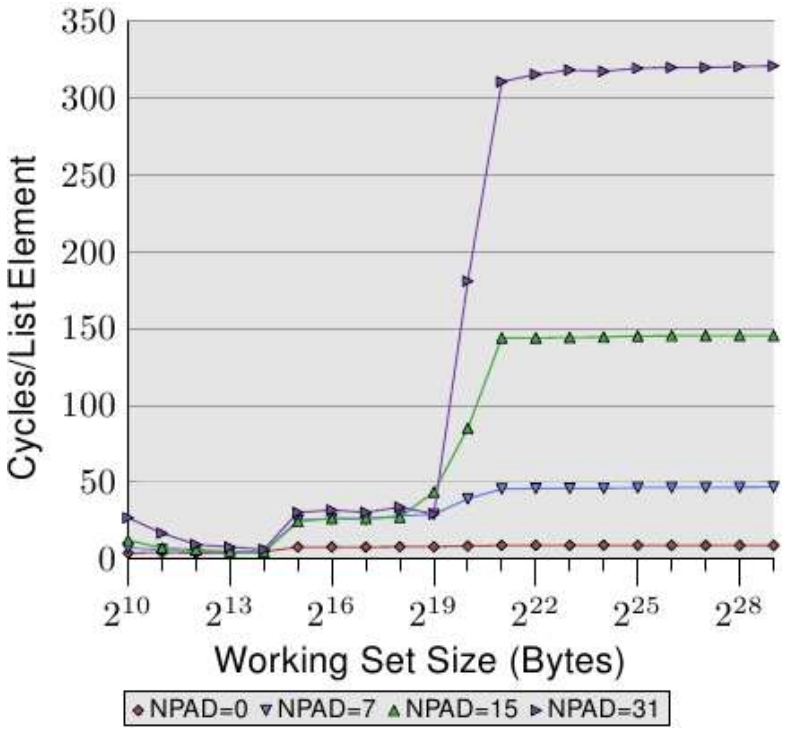


图 3.11: 顺序读多个字节

一开始的两个测试数据收到了噪音的污染。由于它们的工作负荷太小，无法过滤掉系统内其它进程对它们的影响。我们可以认为它们都是4个周期以内的。这样一来，整个图可以划分为比较明显的三个部分:

- 工作集小于2^{14}字节的。
- 工作集从2^{15}字节到2^{20}字节的。
- 工作集大于2^{21}字节的。

这样的结果很容易解释——是因为处理器有16KB的L1d和1MB的L2。而在这三个部分之间，并没有非常锐利的边缘，这是因为系统的其它部分也在使用缓存，我们的测试程序并不能独占缓存的使用。尤其是L2，它是统一式的缓存，处理器的指令也会使用它(注: Intel使用的是包容式缓存)。

测试的实际耗时可能会出乎大家的意料。L1d的部分跟我们预想的差不多，在一台P4上耗时为4个周期左右。但L2的结果则出乎意料。大家可能觉得需要14个周期以上，但实际只用了9个周期。这要归功于处理器先进的处理逻辑，当它使用连续的内存区时，会 *预先读取*下一条缓存线的数据。这样一来，当真正使用下一条线的时候，其实已经早已读完一半了，于是真正的等待耗时会比L2的访问时间少很多。

在工作集超过L2的大小之后，预取的效果更明显了。前面我们说过，主存的访问需要耗时200个周期以上。但在预取的帮助下，实际耗时保持在9个周期左右。200 vs 9，效果非常不错。

我们可以观察到预取的行为，至少可以间接地观察到。图3.11中有4条线，它们表示处理不同大小结构时的耗时情况。随着结构的变大，元素间的距离变大了。图中4条线对应的元素距离分别是0、56、120和248字节。

图中最下面的这一条线来自前一个图，但在这里更像是一条直线。其它三条线的耗时情况比较差。图中这些线也有比较明显的三个阶段，同时，在小工作集的情况下也有比较大的错误(请再次忽略这些错误)。在只使用L1d的阶段，这些线条基本重合。因为这时候还不需要预取，只需要访问L1d就行。

在L2阶段，三条新加的线基本重合，而且耗时比老的那条线高很多，大约在28个周期左右，差不多就是L2的访问时间。这表明，从L2到L1d的预取并没有生效。这是因为，对于最下面的线(NPAD=0)，由于结构小，8次循环后才需要访问一条新缓存线，而上面三条线对应的结构比较大，拿相对最小的NPAD=7来说，光是一次循环就需要访问一条新线，更不用说更大的NPAD=15和31了。而预取逻辑是无法在每个周期装载新线的，因此每次循环都需要从L2读取，我们看到的就是从L2读取的时延。

更有趣的是工作集超过L2容量后的阶段。快看，4条线远远地拉开了。元素的大小变成了主角，左右了性能。处理器应能识别每一步(stride)的大小，不去为NPAD=15和31获取那些实际并不需要的缓存线(参见6.3.1)。元素大小对预取的约束是根源于硬件预取的限制——它无法跨越页边界。如果允许预取器跨越页边界，而下一页不存在或无效，那么OS还得去寻找它。这意味着，程序需要遭遇一次并非由它自己产生的页错误，这是完全不能接受的。在NPAD=7或者更大的时候，由于每个元素都至少需要一条缓存线，预取器已经帮不上忙了，它没有足够的时间去从内存装载数据。 另一个导致慢下来的原因是TLB缓存的未命中。TLB是存储虚实地址映射的缓存，参见第4节。为了保持快速，TLB只有很小的容量。如果有大量页被反复访问，超出了TLB缓存容量，就会导致反复地进行地址翻译，这会耗费大量时间。TLB查找的代价分摊到所有元素上，如果元素越大，那么元素的数量越少，每个元素承担的那一份就越多。

为了观察TLB的性能，我们可以进行另两项测试。第一项：我们还是顺序存储列表中的元素，使NPAD=7，让每个元素占满整个cache line，第二项：我们将列表的每个元素存储在一个单独的页上，忽略每个页没有使用的部分以用来计算工作集的大小。（这样做可能不太一致，因为在前面的测试中，我计算了结构体中每个元素没有使用的部分，从而用来定义NPAD的大小，因此每个元素占满了整个页，这样以来工作集的大小将会有所不同。但是这不是这项测试的重点，预取的低效率多少使其有点不同）。结果表明，第一项测试中，每次列表的迭代都需要一个新的cache line，而且每64个元素就需要一个新的页。第二项测试中，每次迭代都会在一个新的页中加载一个新的cache line。

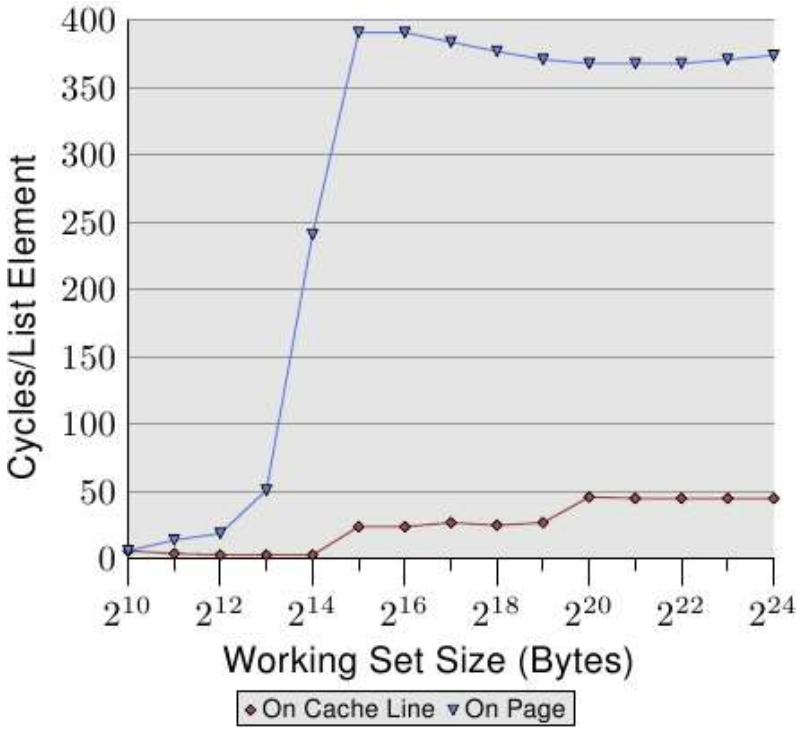


图 3.12: TLB 对顺序读的影响

结果见图3.12。该测试与图3.11是在同一台机器上进行的。基于可用RAM空间的有限性，测试设置容量空间大小为2的24次方字节，这就需要1GB的容量将对象放置在分页上。图3.12中下方的红色曲线正好对应了图3.11中NPAD等于7的曲线。我们看到不同的步长显示了高速缓存L1d和L2的大小。第二条曲线看上去完全不同，其最重要的特点是当工作容量到达2的13次方字节时开始大幅度增长。这就是TLB缓存溢出的时候。我们能计算出一个64字节大小的元素的TLB缓存有64个输入。成本不会受页面错误影响，因为程序锁定了存储器以防止内存被换出。

可以看出，计算物理地址并把它存储在TLB中所花费的周期数量级是非常高的。图3.12的表格显示了一个极端的例子，但从中可以清楚的得到：TLB缓存效率降低的一个重要因素是大型NPAD值的减缓。由于物理地址必须在缓存行能被L2或主存读取之前计算出来，地址转换这个不利因素就增加了内存访问时间。这一点部分解释了为什么NPAD等于31时每个列表元素的总花费比理论上的RAM访问时间要高。

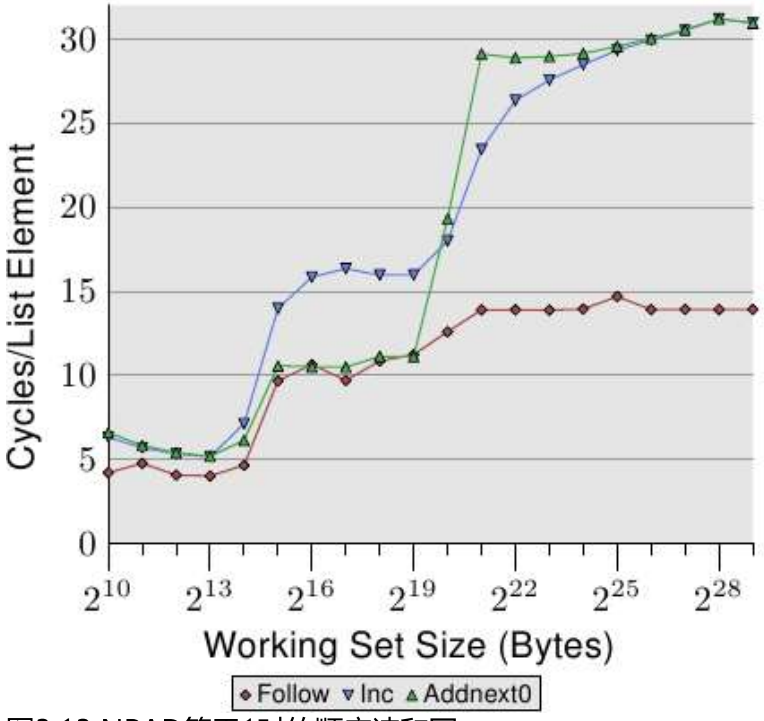


图3.13 NPAD等于1时的顺序读和写

通过查看链表元素被修改时测试数据的运行情况，我们可以窥见一些更详细的预取实现细节。图3.13显示了三条曲线。所有情况下元素宽度都为16个字节。第一条曲线“Follow”是熟悉的链表走线在这里作为基线。第二条曲线，标记为“Inc”，仅仅在当前元素进入下一个前给其增加thepad[0]成员。第三条曲线，标记为"Addnext0"， 取出下一个元素的thepad[0]链表元素并把它添加为当前链表元素的thepad[0]成员。

在没运行时，大家可能会以为"Addnext0"更慢，因为它要做的事情更多——在没进到下个元素之前就需要装载它的值。但实际的运行结果令人惊讶——在某些小工作集下，"Addnext0"比"Inc"更快。这是为什么呢？原因在于，系统一般会对下一个元素进行强制性预取。当程序前进到下个元素时，这个元素其实早已被预取在L1d里。因此，只要工作集比L2小，"Addnext0"的性能基本就能与"Follow"测试媲美。

但是，"Addnext0"比"Inc"更快离开L2，这是因为它需要从主存装载更多的数据。而在工作集达到2^{21}字节时，"Addnext0"的耗时达到了28个周期，是同期"Follow"14周期的两倍。这个两倍也很好解释。"Addnext0"和"Inc"涉及对内存的修改，因此L2的逐出操作不能简单地把数据一扔了事，而必须将它们写入内存。因此FSB的可用带宽变成了一半，传输等量数据的耗时也就变成了原来的两倍。

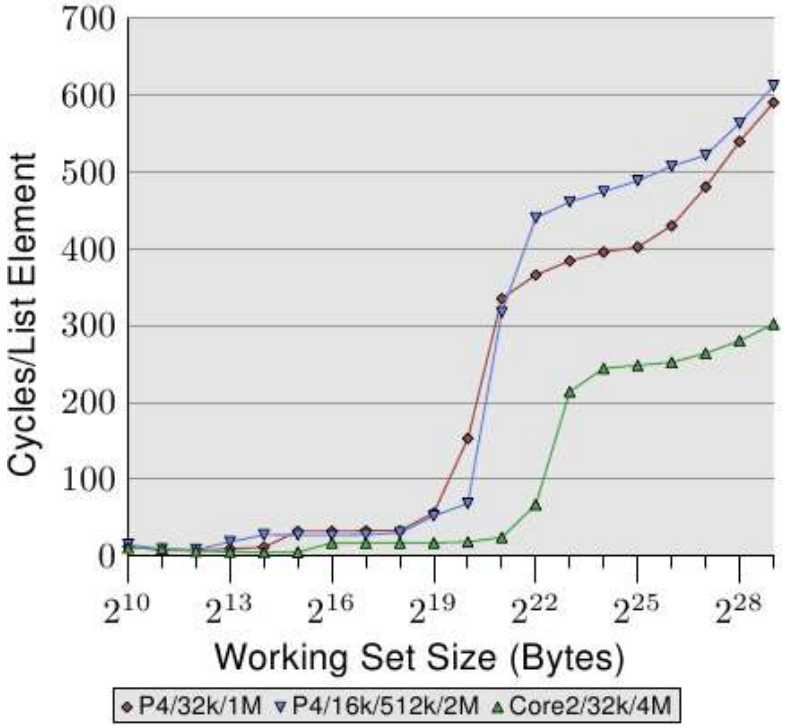


图3.14: 更大L2/L3缓存的优势

决定顺序式缓存处理性能的另一个重要因素是缓存容量。虽然这一点比较明显，但还是值得一说。图3.14展示了128字节长元素的测试结果(64位机，NPAD=15)。这次我们比较三台不同计算机的曲线，两台P4，一台Core 2。两台P4的区别是缓存容量不同，一台是32k的L1d和1M的L2，一台是16K的L1d、512k的L2和2M的L3。Core 2那台则是32k的L1d和4M的L2。

图中最有趣的地方，并不是Core 2如何大胜两台P4，而是工作集开始增长到连末级缓存也放不下、需要主存热情参与之后的部分。

Set Size	Sequential		Random							
L2 Hit	L2 Miss	#Iter	Ratio Miss/Hit	L2 Accesses Per Iter	L2 Hit	L2 Miss	#Iter	Ratio Miss/Hit	L2 Accesses Per Iter	
2^{20}	88,636	843	16,384	0.94%	5.5	30,462	4721	1,024	13.42%	34.4
2^{21}	88,105	1,584	8,192	1.77%	10.9	21,817	15,151	512	40.98%	72.2
2^{22}	88,106	1,600	4,096	1.78%	21.9	22,258	22,285	256	50.03%	174.0
2^{23}	88,104	1,614	2,048	1.80%	43.8	27,521	26,274	128	48.84%	420.3
2^{24}	88,114	1,655	1,024	1.84%	87.7	33,166	29,115	64	46.75%	973.1
2^{25}	88,112	1,730	512	1.93%	175.5	39,858	32,360	32	44.81%	2,256.8
2^{26}	88,112	1,906	256	2.12%	351.6	48,539	38,151	16	44.01%	5,418.1
2^{27}	88,114	2,244	128	2.48%	705.9	62,423	52,049	8	45.47%	14,309.0
2^{28}	88,120	2,939	64	3.23%	1,422.8	81,906	87,167	4	51.56%	42,268.3
2^{29}	88,137	4,318	32	4.67%	2,889.2	119,079	163,398	2	57.84%	141,238.5

表3.2: 顺序访问与随机访问时L2命中与未命中的情况，NPAD=0

与我们预计的相似，最末级缓存越大，曲线停留在L2访问耗时区的时间越长。在220字节的工作集时，第二台P4(更老一些)比第一台P4要快上一倍，这要完全归功于更大的末级缓存。而Core 2拜它巨大的4M L2所赐，表现更为卓越。

对于随机的工作负荷而言，可能没有这么惊人的效果，但是，如果我们能将工作负荷进行一些裁剪，让它匹配末级缓存的容量，就完全可以得到非常大的性能提升。也是由于这个原因，有时候我们需要多花一些钱，买一个拥有更大缓存的处理器。

单线程随机访问模式的测量

前面我们已经看到，处理器能够利用L1d到L2之间的预取消除访问主存、甚至是访问L2的时延。

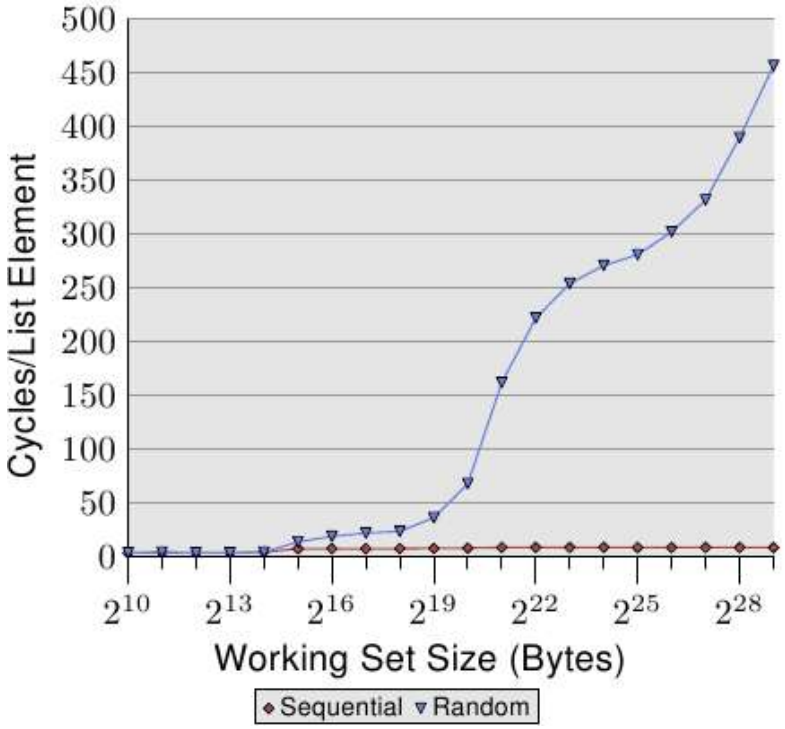


图3.15: 顺序读取vs随机读取，NPAD=0

但是，如果换成随机访问或者不可预测的访问，情况就大不相同了。图3.15比较了顺序读取与随机读取的耗时情况。

换成随机之后，处理器无法再有效地预取数据，只有少数情况下靠运气刚好碰到先后访问的两个元素挨在一起的情形。

图3.15中有两个需要关注的地方。首先，在大的工作集下需要非常多的周期。这台机器访问主存的时间大约为200-300个周期，但图中的耗时甚至超过了450个周期。我们前面已经观察到过类似现象(对比图3.11)。这说明，处理器的自动预取在这里起到了反效果。

其次，代表随机访问的曲线在各个阶段不像顺序访问那样保持平坦，而是不断攀升。为了解释这个问题，我们测量了程序在不同工作集下对L2的访问情况。结果如图3.16和表3.2。

从图中可以看出，当工作集大小超过L2时，未命中率(L2未命中次数/L2访问次数)开始上升。整条曲线的走向与图3.15有些相似: 先急速爬升，随后缓缓下滑，最后再度爬升。它与耗时图有紧密的关联。L2未命中率会一直爬升到100%为止。只要工作集足够大(并且内存也足够大)，就可以将缓存线位于L2内或处于装载过程中的可能性降到非常低。

缓存未命中率的攀升已经可以解释一部分的开销。除此以外，还有一个因素。观察表3.2的L2/#Iter列，可以看到每个循环对L2的使用次数在增长。由于工作集每次为上一次的两倍，如果没有缓存的话，内存的访问次数也将是上一次的两倍。在按顺序访问时，由于缓存的帮助及完美的预见性，对L2使用的增长比较平缓，完全取决于工作集的增长速度。

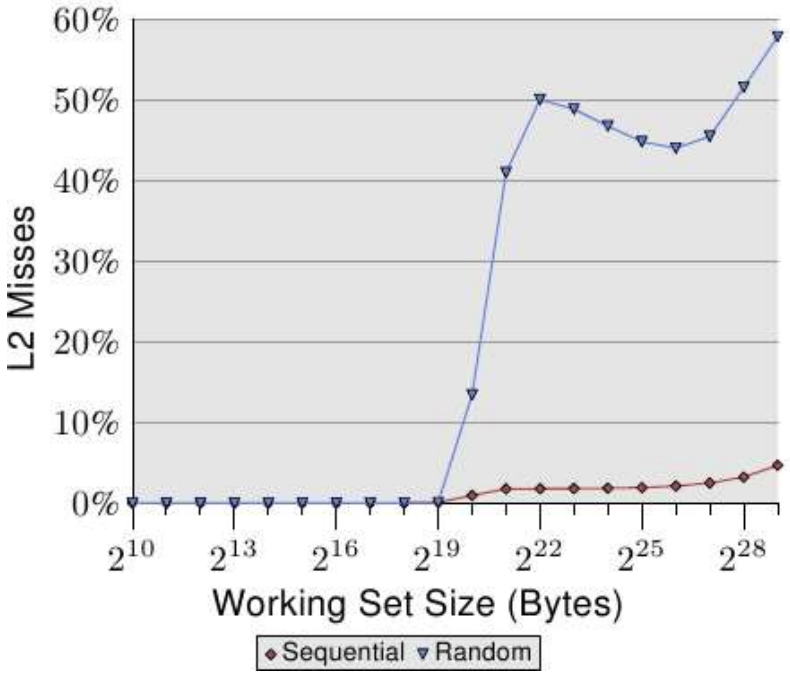


图3.16: L2d未命中率

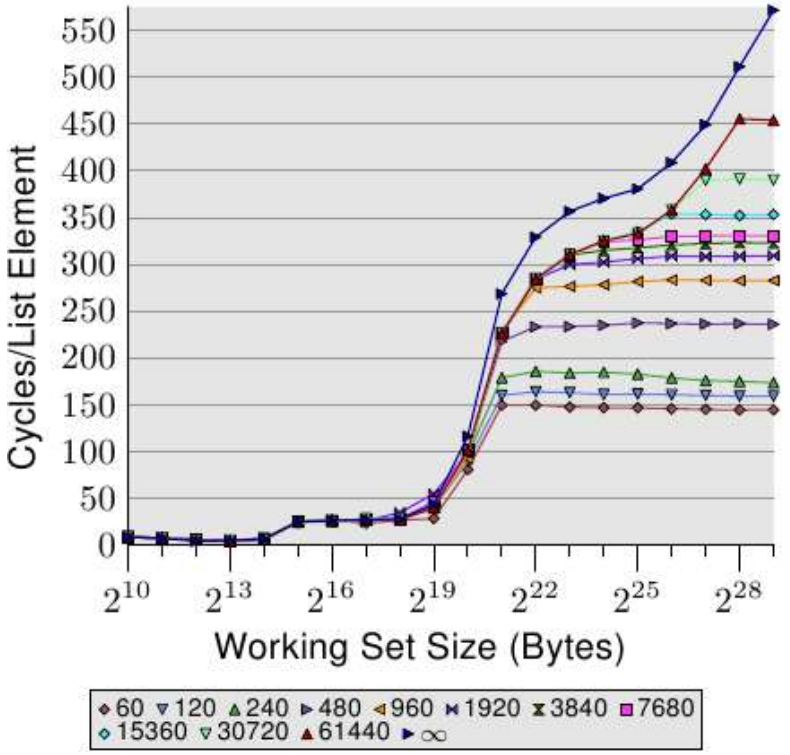


图3.17: 页意义上(Page-Wise)的随机化，NPAD=7

而换成随机访问后，单位耗时的增长超过了工作集的增长，根源是TLB未命中率的上升。图3.17描绘的是NPAD=7时随机访问的耗时情况。这一次，我们修改了随机访问的方式。正常情况下是把整个列表作为一个块进行随机(以∞表示)，而其它11条线则是在小一些的块里进行随机。例如，标签为'60'的线表示以60页(245760字节)为单位进行随机。先遍历完这个块里的所有元素，再访问另一个块。这样一来，可以保证任意时刻使用的TLB条目数都是有限的。 NPAD=7对应于64字节，正好等于缓存线的长度。由于元素顺序随机，硬件预取不可能有任何效果，特别是在元素较多的情况下。这意味着，分块随机时的L2未命中率与整个列表随机时的未命中率没有本质的差别。随着块的增大，曲线逐渐逼近整个列表随机对应的曲线。这说明，在这个测试里，性能受到TLB命中率的影响很大，如果我们能提高TLB命中率，就能大幅度地提升性能(在后面的一个例子里，性能提升了38%之多)。

3.3.3 写入时的行为

在我们开始研究多个线程或进程同时使用相同内存之前，先来看一下缓存实现的一些细节。我们要求缓存是一致的，而且这种一致性必须对用户级代码完全透明。而内核代码则有所不同，它有时候需要对缓存进行转储

(flush)。

这意味着，如果对缓存线进行了修改，那么在这个时间点之后，系统的结果应该是与没有缓存的情况下是相同的，即主存的对应位置也已经被修改的状态。这种要求可以通过两种方式或策略实现：

- 写通(write-through)
- 写回(write-back)

写通比较简单。当修改缓存线时，处理器立即将它写入主存。这样可以保证主存与缓存的内容永远保持一致。当缓存线被替代时，只需要简单地将它丢弃即可。这种策略很简单，但是速度比较慢。如果某个程序反复修改一个本地变量，可能导致FSB上产生大量数据流，而不管这个变量是不是有人在用，或者是不是短期变量。

写回比较复杂。当修改缓存线时，处理器不再马上将它写入主存，而是打上已弄脏(dirty)的标记。当以后某个时间点缓存线被丢弃时，这个已弄脏标记会通知处理器把数据写回到主存中，而不是简单地扔掉。

写回有时候会有非常不错的性能，因此较好的系统大多采用这种方式。采用写回时，处理器们甚至可以利用FSB的空闲容量来存储缓存线。这样一来，当需要缓存空间时，处理器只需清除脏标记，丢弃缓存线即可。

但写回也有一个很大的问题。当有多个处理器(或核心、超线程)访问同一块内存时，必须确保它们在任何时候看到的都是相同的内容。如果缓存线在其中一个处理器上弄脏了(修改了，但还没写回主存)，而第二个处理器刚好要读取同一个内存地址，那么这个读操作不能去读主存，而需要读第一个处理器的缓存线。在下一节中，我们将研究如何实现这种需求。

在此之前，还有其它两种缓存策略需要提一下:

- 写入合并
- 不可缓存

这两种策略用于真实内存不支持的特殊地址区，内核为地址区设置这些策略(x86处理器利用内存类型范围寄存器MTRR)，余下的部分自动进行。MTRR还可用于写通和写回策略的选择。

写入合并是一种有限的缓存优化策略，更多地用于显卡等设备之上的内存。由于设备的传输开销比本地内存要高的多，因此避免进行过多的传输显得尤为重要。如果仅仅因为修改了缓存线上的一个字，就传输整条线，而下个操作刚好是修改线上的下一个字，那么这次传输就过于浪费了。而这恰恰对于显卡来说是比较常见的情形——屏幕上水平邻接的像素往往在内存中也是靠在一起的。顾名思义，写入合并是在写出缓存线前，先将多个写入访问合并起来。在理想的情况下，缓存线被逐字逐字地修改，只有当写入最后一个字时，才将整条线写入内存，从而极大地加速内存的访问。

最后来讲一下不可缓存的内存。一般指的是不被RAM支持的内存位置，它可以是硬编码的特殊地址，承担CPU以外的某些功能。对于商用硬件来说，比较常见的是映射到外部卡或设备的地址。在嵌入式主板上，有时也有类似的地址，用来开关LED。对这些地址进行缓存显然没有什么意义。比如上述的LED，一般是用来调试或报告状态，显然应该尽快点亮或关闭。而对于那些PCI卡上的内存，由于不需要CPU的干涉即可更改，也不该缓存。

3.3.4 多处理器支持

在上节中我们已经指出当多处理器开始发挥作用的时候所遇到的问题。甚至对于那些不共享的高速级别的缓存（至少在L1d级别）的多核处理器也有问题。

直接提供从一个处理器到另一处理器的高速访问，这是完全不切实际的。从一开始，连接速度根本就不够快。实际的选择是，在其需要的情况下，转移到其他处理器。需要注意的是，这同样应用在相同处理器上无需共享的高速缓存。

现在的问题是，当该高速缓存线转移的时候会发生什么？这个问题回答起来相当容易：当一个处理器需要在另一个处理器的高速缓存中读或者写的脏的高速缓存线的时候。但怎样处理器怎样确定在另一个处理器的缓存中的高速缓存线是脏的？假设它仅仅是因为一个高速缓存线被另一个处理器加载将是次优的（最好的）。通常情况下，大多数的内存访问是只读的访问和产生高速缓存线，并不脏。在高速缓存线上处理器频繁的操作（当然，否则为什么我们有这样的文件呢？），也就意味着每一次写访问后，都要广播关于高速缓存线的改变将变得不切实际。

多年来，人们开发除了MESI缓存一致性协议(MESI=Modified, Exclusive, Shared, Invalid，变更的、独占的、共享的、无效的)。协议的名称来自协议中缓存线可以进入的四种状态:

- **变更的**: 本地处理器修改了缓存线。同时暗示，它是所有缓存中唯一的拷贝。
- **独占的**: 缓存线没有被修改，而且没有被装入其它处理器缓存。
- **共享的**: 缓存线没有被修改，但可能已被装入其它处理器缓存。
- **无效的**: 缓存线无效，即，未被使用。

MESI协议开发了很多年，最初的版本比较简单，但是效率也比较差。现在的版本通过以上4个状态可以有效地实现写回式缓存，同时支持不同处理器对只读数据的并发访问。

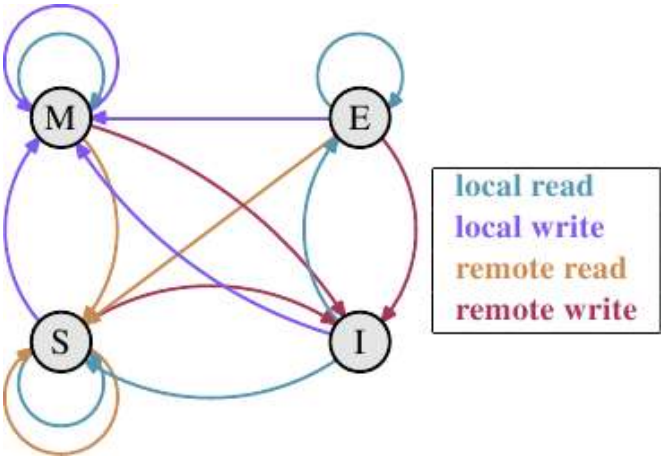


图3.18: MESI协议的状态跃迁图

在协议中，通过处理器监听其它处理器的活动，不需太多努力即可实现状态变更。处理器将操作发布在外部引脚上，使外部可以了解到处理过程。目标的缓存线地址则可以在地址总线上看到。在下文讲述状态时，我们将介绍总线参与的时机。

一开始，所有缓存线都是空的，缓存为无效(Invalid)状态。当有数据装进缓存供写入时，缓存变为变更(Modified)状态。如果有数据装进缓存供读取，那么新状态取决于其它处理器是否已经状态了同一条缓存线。如果是，那么新状态变成共享(Shared)状态，否则变成独占(Exclusive)状态。

如果本地处理器对某条Modified缓存线进行读写，那么直接使用缓存内容，状态保持不变。如果另一个处理器希望读它，那么第一个处理器将内容发给第一个处理器，然后可以将缓存状态置为Shared。而发给第二个处理器的数据由内存控制器接收，并放入内存中。如果这一步没有发生，就不能将这条线置为Shared。如果第二个处理器希望的是写，那么第一个处理器将内容发给它后，将缓存置为Invalid。这就是臭名昭著的"请求所有权(Request For Ownership,RFO)"操作。在末级缓存执行RFO操作的代价比较高。如果是写通式缓存，还要加上将内容写入上一层缓存或主存的时间，进一步提升了代价。 对于Shared缓存线，本地处理器的读取操作并不需要修改状态，而且可以直接从缓存满足。而本地处理器的写入操作则需要将状态置为Modified，而且需要将缓存线在其它处理器的所有拷贝置为Invalid。因此，这个写入操作需要通过RFO消息发通知其它处理器。如果第二个处理器请求读取，无事发生。因为主存已经包含了当前数据，而且状态已经为Shared。如果第二个处理器需要写入，则将缓存线置为Invalid。不需要总线操作。

Exclusive状态与Shared状态很像，只有一个不同之处: 在Exclusive状态时，本地写入操作不需要在总线上声明，因为本地的缓存是系统中唯一的拷贝。这是一个巨大的优势，所以处理器会尽量将缓存线保留在Exclusive状态，而不是Shared状态。只有在信息不可用时，才退而求其次选择shared。放弃Exclusive不会引起任何功能缺失，但会导致性能下降，因为E→M要远远快于S→M。

从以上的说明中应该已经可以看出，在多处理器环境下，哪一步的代价比较大了。填充缓存的代价当然还是很高，但我们还需要留意RFO消息。一旦涉及RFO，操作就快不起来了。

RFO在两种情况下是必需的:

- 线程从一个处理器迁移到另一个处理器，需要将所有缓存线移到新处理器。
- 某条缓存线确实需要被两个处理器使用。{*对于同一处理器的两个核心，也有同样的情况，只是代价稍低。RFO消息可能会被发送多次。*}

多线程或多进程的程序总是需要同步，而这种同步依赖内存来实现。因此，有些RFO消息是合理的，但仍然需要尽量降低发送频率。除此以外，还有其它来源的RFO。在第6节中，我们将解释这些场景。缓存一致性协议的消息必须发给系统中所有处理器。只有当协议确定已经给过所有处理器响应机会之后，才能进行状态跃迁。也就是说，协议的速度取决于最长响应时间。{*这也是现在能看到三插槽AMD Opteron系统的原因。这类系统只*

有三个超级链路(hyperlink)，其中一个连接南桥，每个处理器之间都只有一跳的距离。}总线上可能会发生冲突，NUMA系统的延时很大，突发的流量会拖慢通信。这些都是让我们避免无谓流量的充足理由。

此外，关于多处理器还有一个问题。虽然它的影响与具体机器密切相关，但根源是唯一的——FSB是共享的。在大多数情况下，所有处理器通过唯一的总线连接到内存控制器(参见图2.1)。如果一个处理器就能占满总线(十分常见)，那么共享总线的两个或四个处理器显然只会得到更有限的带宽。

即使每个处理器有自己连接内存控制器的总线，如图2.2，但还需要通往内存模块的总线。一般情况下，这种总线只有一条。退一步说，即使像图2.2那样不止一条，对同一个内存模块的并发访问也会限制它的带宽。

对于每个处理器拥有本地内存的AMD模型来说，也是同样的问题。的确，所有处理器可以非常快速地同时访问它们自己的内存。但是，多线程呢？多进程呢？它们仍然需要通过访问同一块内存来进行同步。

对同步来说，有限的带宽严重地制约着并发度。程序需要更加谨慎的设计，将不同处理器访问同一块内存的机会降到最低。以下的测试展示了这一点，还展示了与多线程代码相关的其它效果。

多线程测量

为了帮助大家理解问题的严重性，我们来看一些曲线图，主角也是前文的那个程序。只不过这一次，我们运行多个线程，并测量这些线程中最快那个的运行时间。也就是说，等它们全部运行完是需要更长时间的。我们用的机器有4个处理器，而测试是做多跑4个线程。所有处理器共享同一条通往内存控制器的总线，另外，通往内存模块的总线也只有一条。

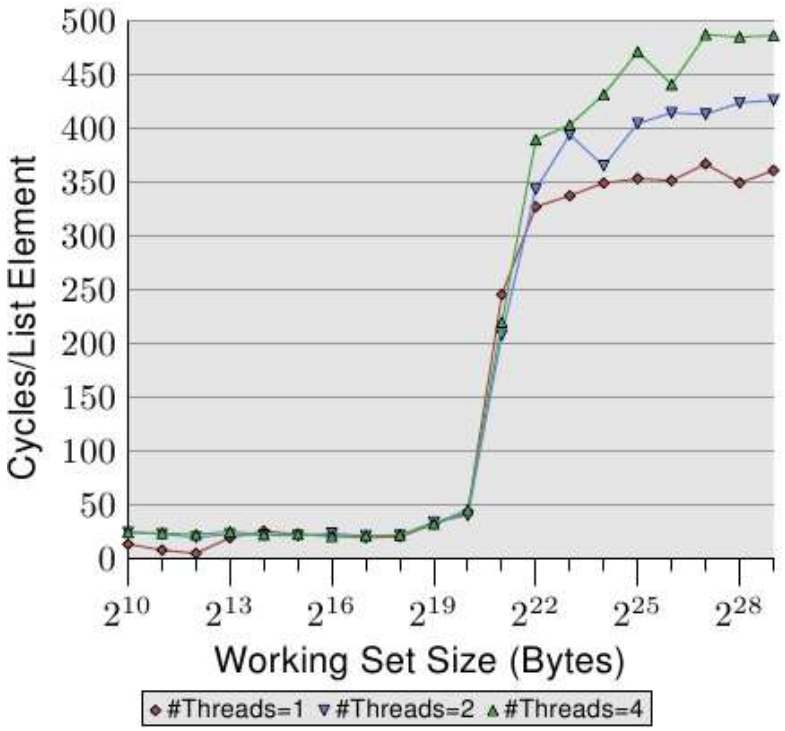


图3.19: 顺序读操作，多线程

图3.19展示了顺序读访问时的性能，元素为128字节长(64位计算机，NPAD=15)。对于单线程的曲线，我们预计是与图3.11相似，只不过是换了一台机器，所以实际的数字会有些小差别。

更重要的部分当然是多线程的环节。由于是只读，不会去修改内存，不会尝试同步。但即使不需要RFO，而且所有缓存线都可共享，性能仍然分别下降了18%(双线程)和34%(四线程)。由于不需要在处理器之间传输缓存，因此这里的性能下降完全由以下两个瓶颈之一或同时引起: 一是从处理器到内存控制器的共享总线，二是从内存控制器到内存模块的共享总线。当工作集超过L3后，三种情况下都要预取新元素，而即使是双线程，可用的带宽也无法满足线性扩展(无惩罚)。

当加入修改之后，场面更加难看了。图3.20展示了顺序递增测试的结果。

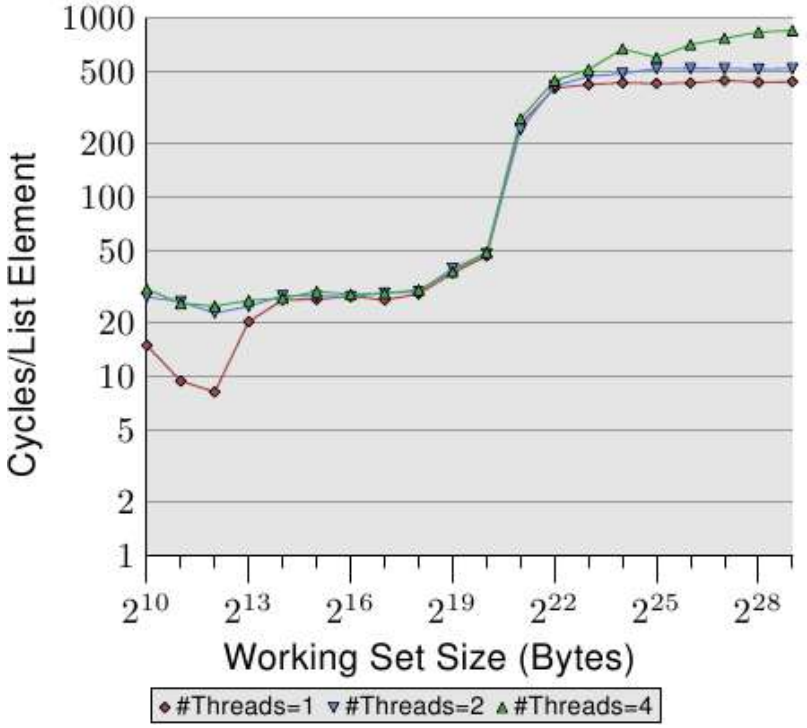


图3.20: 顺序递增，多线程

图中Y轴采用的是对数刻度，不要被看起来很小的差值欺骗了。现在，双线程的性能惩罚仍然是18%，但四线程的惩罚飙升到了93%！原因在于，采用四线程时，预取的流量与写回的流量加在一起，占满了整个总线。

我们用对数刻度来展示L1d范围的结果。可以发现，当超过一个线程后，L1d就无力了。单线程时，仅当工作集超过L1d时访问时间才会超过20个周期，而多线程时，即使在很小的工作集情况下，访问时间也达到了那个水平。

这里并没有揭示问题的另一方面，主要是用这个程序很难进行测量。问题是这样的，我们的测试程序修改了内存，所以本应看到RFO的影响，但在结果中，我们并没有在L2阶段看到更大的开销。原因在于，要看到RFO的影响，程序必须使用大量内存，而且所有线程必须同时访问同一块内存。如果没有大量的同步，这是很难实现的，而如果加入同步，则会占满执行时间。

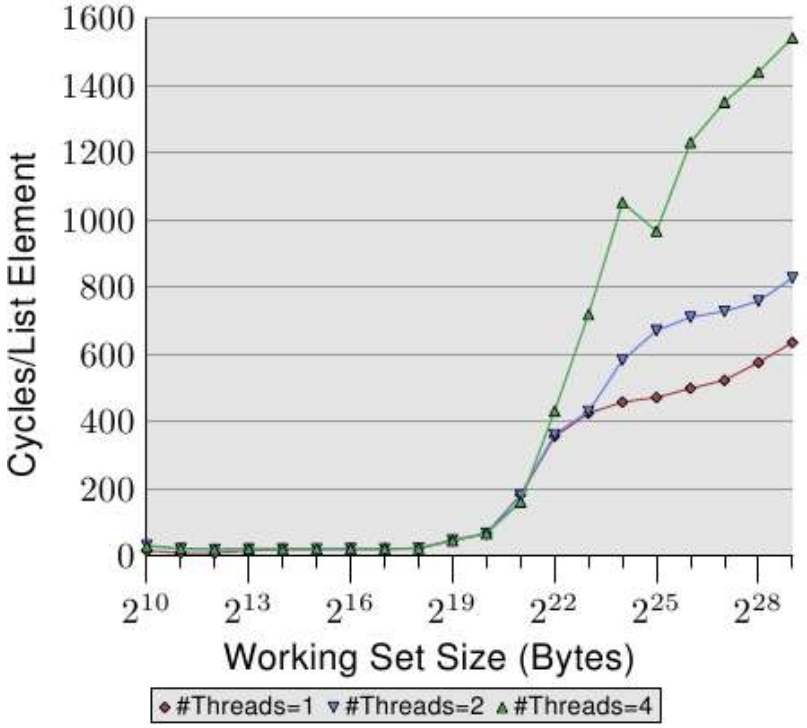


图3.21: 随机的Addnextlast，多线程

最后，在图3.21中，我们展示了随机访问的Addnextlast测试的结果。这里主要是为了让大家感受一下这些巨大到爆的数字。极端情况下，甚至用了1500个周期才处理完一个元素。如果加入更多线程，真是不可想象哪。我

们把多线程的效能总结了一下:

#Threads	Seq Read	Seq Inc	Rand Add
2	1.69	1.69	1.54
4	2.98	2.07	1.65

表3.3: 多线程的效能

这个表展示了图3.21中多线程运行大工作集时的效能。表中的数字表示测试程序在使用多线程处理大工作集时可能达到的最大加速因子。双线程和四线程的理论最大加速因子分别是2和4。从表中数据来看，双线程的结果还能接受，但四线程的结果表明，扩展到双线程以上是没有什么意义的，带来的收益可以忽略不计。只要我们把图3.21换个方式呈现，就可以很容易看清这一点。

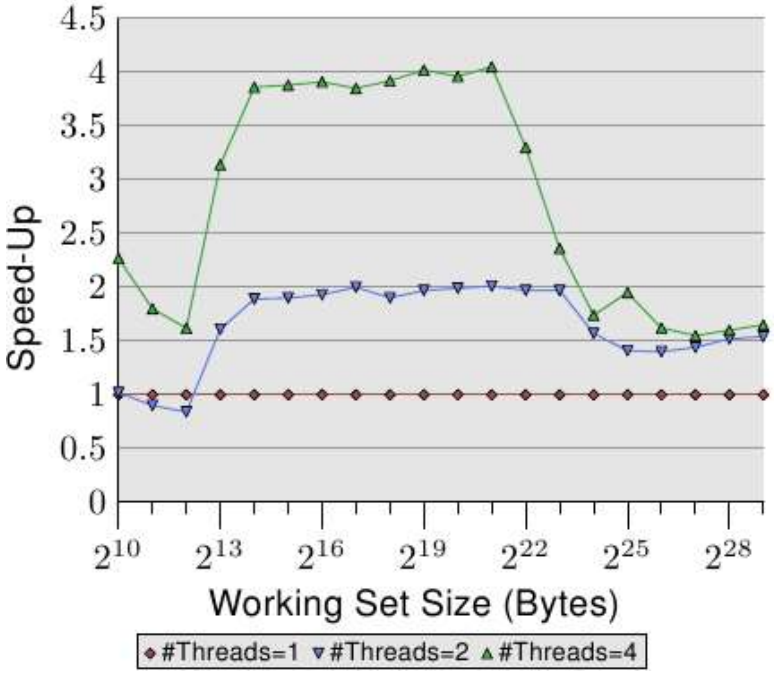


图3.22: 通过并行化实现的加速因子

图3.22中的曲线展示了加速因子，即多线程相对于单线程所能获取的性能加成值。测量值的精确度有限，因此我们需要忽略比较小的那些数字。可以看到，在L2与L3范围内，多线程基本可以做到线性加速，双线程和四线程分别达到了2和4的加速因子。但是，一旦工作集的大小超出L3，曲线就崩塌了，双线程和四线程降到了基本相同的数值(参见表3.3中第4列)。也是部分由于这个原因，我们很少看到4CPU以上的主板共享同一个内存控制器。如果需要配置更多处理器，我们只能选择其它的实现方式(参见第5节)。

可惜，上图中的数据并不是普遍情况。在某些情况下，即使工作集能够放入末级缓存，也无法实现线性加速。实际上，这反而是正常的，因为普通的线程都有一定的耦合关系，不会像我们的测试程序这样完全独立。而反过来说，即使是很大的工作集，即使是两个以上的线程，也是可以通过并行化受益的，但是需要程序员的聪明才智。我们会在第6节进行一些介绍。

特例: 超线程

由CPU实现的超线程(有时又叫对称多线程，SMT)是一种比较特殊的情况，每个线程并不能真正并发地运行。它们共享着除寄存器外的绝大多数处理资源。每个核心和CPU仍然是并行工作的，但核心上的线程则受到这个限制。理论上，每个核心可以有大量线程，不过到目前为止，Intel的CPU最多只有两个线程。CPU负责对各线程进行时分复用，但这种复用本身并没有多少厉害。它真正的优势在于，CPU可以在当前运行的超线程发生延迟时，调度另一个线程。这种延迟一般由内存访问引起。

如果两个线程运行在一个超线程核心上，那么只有当两个线程*合起来的*运行时间少于单线程运行时间时，效率才会比较高。我们可以将通常先后发生的内存访问叠合在一起，以实现这个目标。有一个简单的计算公式，可以帮助我们计算如果需要某个加速因子，最少需要多少的缓存命中率。

程序的执行时间可以通过一个只有一级缓存的简单模型来进行估算(参见[htimpact]):

$$T_{exe} = N[(1-F_{mem})T_{proc} + F_{mem}(G_{hit}T_{cache} + (1-G_{hit})T_{miss})]$$

各变量的含义如下:

N = 指令数

F_{mem} = N中访问内存的比例

G_{hit} = 命中缓存的比例

T_{proc} = 每条指令所用的周期数

T_{cache} = 缓存命中所用的周期数

T_{miss} = 缓冲未命中所用的周期数

T_{exe} = 程序的执行时间

为了让任何判读使用双线程，两个线程之中任一线程的执行时间最多为单线程指令的一半。两者都有一个唯一的变量缓存命中数。 如果我们要解决最小缓存命中率相等的问题需要使我们获得的线程的执行率不少于50%或更多，如图 3.23.

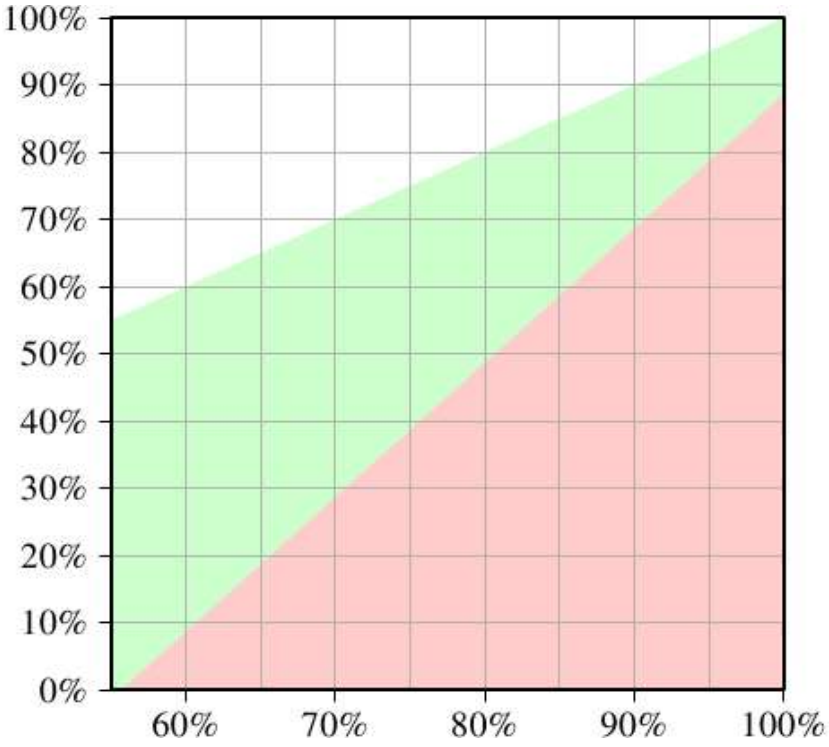


图 3.23: 最小缓存命中率-加速

X轴表示单线程指令的缓存命中率G_{hit}，Y轴表示多线程指令所需的缓存命中率。这个值永远不能高于单线程命中率，否则，单线程指令也会使用改良的指令。为了使单线程的命中率在低于55%的所有情况下优于使用多线程，cup要或多或少的足够空闲因为缓存丢失会运行另外一个超线程。

绿色区域是我们的目标。如果线程的速度没有慢过50%，而每个线程的工作量只有原来的一半，那么它们合起来的耗时应该会少于单线程的耗时。对我们用的示例系统来说(使用超线程的P4机器)，如果单线程代码的命中率为60%，那么多线程代码至少要达到10%才能获得收益。这个要求一般来说还是可以做到的。但是，如果单线程代码的命中率达到了95%，那么多线程代码要做到80%才行。这就很难了。而且，这里还涉及到超线程，在两个超线程的情况下，每个超线程只能分到一半的有效缓存。因为所有超线程是使用同一个缓存来装载数据的，如果两个超线程的工作集没有重叠，那么原始的95%也会被打对折——47%，远低于80%。

因此，超线程只在某些情况下才比较有用。单线程代码的缓存命中率必须低到一定程度，从而使缓存容量变小时新的命中率仍能满足要求。只有在这种情况下，超线程才是有意义的。在实践中，采用超线程能否获得更快的结果，取决于处理器能否有效地将每个进程的等待时间与其它进程的执行时间重叠在一起。并行化也需要一定的开销，需要加到总的运行时间里，这个开销往往是不能忽略的。

在6.3.4节中，我们会介绍一种技术，它将多个线程通过公用缓存紧密地耦合起来。这种技术适用于许多场合，前提是程序员们乐意花费时间和精力扩展自己的代码。

如果两个超线程执行完全不同的代码(两个线程就像被当成两个处理器，分别执行不同进程)，那么缓存容量就真的会降为一半，导致缓冲未命中率大为攀升，这一点应该是很清楚的。这样的调度机制是很有问题的，除非你的缓存足够大。所以，除非程序的工作集设计得比较合理，能够确实从超线程获益，否则还是建议在BIOS中把超线程功能关掉。{我们可能会因为另一个原因 *开启 超线程，那就是调试，因为SMT在查找并行代码的问题方面真的非常好用。*}

3.3.5 其它细节

我们已经介绍了地址的组成，即标签、集合索引和偏移三个部分。那么，实际会用到什么样的地址呢？目前，处理器一般都向进程提供虚拟地址空间，意味着我们有两种不同的地址: 虚拟地址和物理地址。

虚拟地址有个问题——并不唯一。随着时间的变化，虚拟地址可以变化，指向不同的物理地址。同一个地址在不同的进程里也可以表示不同的物理地址。那么，是不是用物理地址会比较好呢?

问题是，处理器指令用的虚拟地址，而且需要在内存管理单元(MMU)的协助下将它们翻译成物理地址。这并不是一个很小的操作。在执行指令的管线(pipeline)中，物理地址只能在很后面的阶段才能得到。这意味着，缓存逻辑需要在很短的时间里判断地址是否已被缓存过。而如果可以使用虚拟地址，缓存查找操作就可以更早地发生，一旦命中，就可以马上使用内存的内容。结果就是，使用虚拟内存后，可以让管线把更多内存访问的开销隐藏起来。

处理器的设计人员们现在使用虚拟地址来标记第一级缓存。这些缓存很小，很容易被清空。在进程页表树发生变更的情况下，至少是需要清空部分缓存的。如果处理器拥有指定变更地址范围的指令，那么可以避免缓存的完全刷新。由于一级缓存L1i及L1d的时延都很小(~3周期)，基本上必须使用虚拟地址。

对于更大的缓存，包括L2和L3等，则需要以物理地址作为标签。因为这些缓存的时延比较大，虚拟到物理地址的映射可以在允许的时间里完成，而且由于主存时延的存在，重新填充这些缓存会消耗比较长的时间，刷新的代价比较昂贵。

一般来说，我们并不需要了解这些缓存处理地址的细节。我们不能更改它们，而那些可能影响性能的因素，要么是应该避免的，要么是有很高代价的。填满缓存是不好的行为，缓存线都落入同一个集合，也会让缓存早早地出问题。对于后一个问题，可以通过缓存虚拟地址来避免，但作为一个用户级程序，是不可能避免缓存物理地址的。我们唯一可以做的，是尽最大努力不要在同一个进程里用多个虚拟地址映射同一个物理地址。

另一个细节对程序员们来说比较乏味，那就是缓存的替换策略。大多数缓存会优先逐出最近最少使用(Least Recently Used,LRU)的元素。这往往是一个效果比较好的策略。在关联性很大的情况下(随着以后核心数的增加，关联性势必会变得越来越大)，维护LRU列表变得越来越昂贵，于是我们开始看到其它的一些策略。

在缓存的替换策略方面，程序员可以做的事情不多。如果缓存使用物理地址作为标签，我们是无法找出虚拟地址与缓存集之间关联的。有可能会出现这样的情形: 所有逻辑页中的缓存线都映射到同一个缓存集，而其它大部分缓存却空闲着。即使有这种情况，也只能依靠OS进行合理安排，避免频繁出现。

虚拟化的出现使得这一切变得更加复杂。现在不仅操作系统可以控制物理内存的分配。虚拟机监视器（VMM，也称为 hypervisor）也负责分配内存。

对程序员来说，最好 a) 完全使用逻辑内存页面 b) 在有意义的情况下，使用尽可能大的页面大小来分散物理地址。更大的页面大小也有其他好处，不过这是另一个话题（见第4节）。

3.4 指令缓存

其实，不光处理器使用的数据被缓存，它们执行的指令也是被缓存的。只不过，指令缓存的问题相对来说要少得多，因为:

- 执行的代码量取决于代码大小。而代码大小通常取决于问题复杂度。问题复杂度则是固定的。
- 程序的数据处理逻辑是程序员设计的，而程序的指令却是编译器生成的。编译器的作者知道如何生成优良的代码。
- 程序的流向比数据访问模式更容易预测。现如今的CPU很擅长模式检测，对预取很有利。
- 代码永远都有良好的时间局部性和空间局部性。

有一些准则是需要程序员们遵守的，但大都是关于如何使用工具的，我们会在第6节介绍它们。而在这里我们只介绍一下指令缓存的技术细节。

随着CPU的核心频率大幅上升，缓存与核心的速度差越拉越大，CPU的处理开始管线化。也就是说，指令的执行分成若干阶段。首先，对指令进行解码，随后，准备参数，最后，执行它。这样的管线可以很长(例如，Intel的Netburst架构超过了20个阶段)。在管线很长的情况下，一旦发生延误(即指令流中断)，需要很长时间才能恢复速度。管线延误发生在这样的情况下: 下一条指令未能正确预测，或者装载下一条指令耗时过长(例如，需要从内存读取时)。

为了解决这个问题，CPU的设计人员们在分支预测上投入大量时间和芯片资产(chip real estate)，以降低管线延误的出现频率。

在CISC处理器上，指令的解码阶段也需要一些时间。x86及x86-64处理器尤为严重。近年来，这些处理器不再将指令的原始字节序列存入L1i，而是缓存解码后的版本。这样的L1i被叫做“追踪缓存(trace cache)”。追踪缓存可以在命中的情况下让处理器跳过管线最初的几个阶段，在管线发生延误时尤其有用。

前面说过，L2以上的缓存是统一缓存，既保存代码，也保存数据。显然，这里保存的代码是原始字节序列，而不是解码后的形式。

在提高性能方面，与指令缓存相关的只有很少的几条准则:

1. 生成尽量少的代码。也有一些例外，如出于管线化的目的需要更多的代码，或使用小代码会带来过高的额外开销。
2. 尽量帮助处理器作出良好的预取决策，可以通过代码布局或显式预取来实现。

这些准则一般会由编译器的代码生成阶段强制执行。至于程序员可以参与的部分，我们会在第6节介绍。

3.4.1 自修改的代码

在计算机的早期岁月里，内存十分昂贵。人们想尽千方百计，只为了尽量压缩程序容量，给数据多留一些空间。其中，有一种方法是修改程序自身，称为自修改代码(SMC)。现在，有时候我们还能看到它，一般是出于提高性能的目的，也有的是为了攻击安全漏洞。

一般情况下，应该避免SMC。虽然一般情况下没有问题，但有时会由于执行错误而出现性能问题。显然，发生改变的代码是无法放入追踪缓存(追踪缓存放的是解码后的指令)的。即使没有使用追踪缓存(代码还没被执行或有段时间没执行)，处理器也可能会遇到问题。如果某个进入管线的指令发生了变化，处理器只能扔掉目前的成果，重新开始。在某些情况下，甚至需要丢弃处理器的大部分状态。

最后，由于处理器认为代码页是不可修改的(这是出于简单化的考虑，而且在99.9999999%情况下确实是正确的)，L1i用到并不是MESI协议，而是一种简化后的SI协议。这样一来，如果万一检测到修改的情况，就需要作出大量悲观的假设。

因此，对于SMC，强烈建议能不用就不用。现在内存已经不再是一种那么稀缺的资源了。最好是写多个函数，而不要根据需要把一个函数改来改去。也许有一天可以把SMC变成可选项，我们就能通过这种方式检测入侵代码。如果一定要用SMC，应该让写操作越过缓存，以免由于L1i需要L1d里的数据而产生问题。更多细节，请参见6.1节。

在Linux上，判断程序是否包含SMC是很容易的。利用正常工具链(toolchain)构建的程序代码都是写保护(write-protected)的。程序员需要在链接时施展某些关键的魔术才能生成可写的代码页。现代的Intel x86和x86-64处理器都有统计SMC使用情况的专用计数器。通过这些计数器，我们可以很容易判断程序是否包含SMC，即使它被准许运行。

3.5 缓存未命中的因素

我们已经看过内存访问没有命中缓存时，那陡然猛涨的高昂代价。但是有时候，这种情况又是无法避免的，因此我们需要对真正的代价有所认识，并学习如何缓解这种局面。

3.5.1 缓存与内存带宽

为了更好地理解处理器的能力，我们测量了各种理想环境下能够达到的带宽值。由于不同处理器的版本差别很大，所以这个测试比较有趣，也因为如此，这一节都快被测试数据灌满了。我们使用了x86和x86-64处理器的

SSE指令来装载和存储数据，每次16字节。工作集则与其它测试一样，从1kB增加到512MB，测量的具体对象是每个周期所处理的字节数。

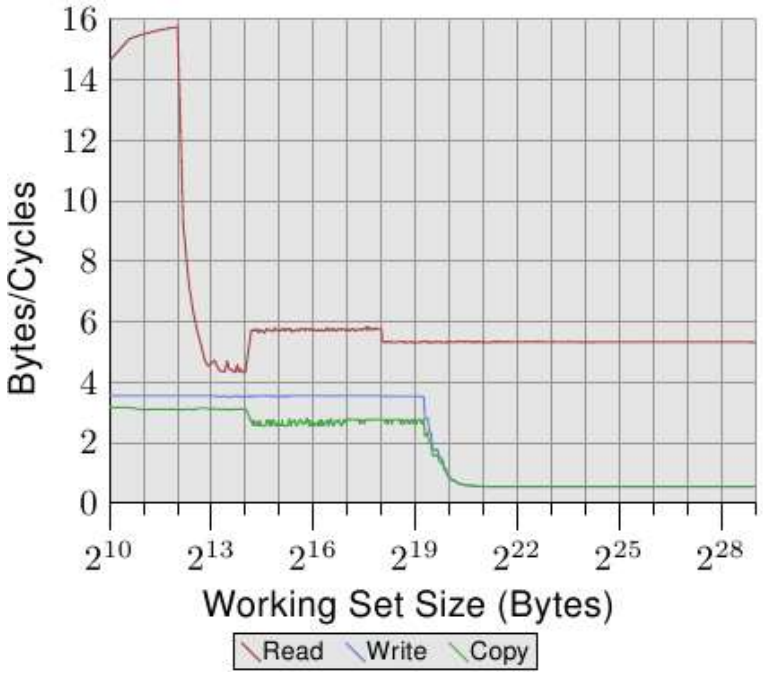


图3.24: P4的带宽

图3.24展示了一颗64位Intel Netburst处理器的性能图表。当工作集能够完全放入L1d时，处理器的每个周期可以读取完整的16字节数据，即每个周期执行一条装载指令(moveaps指令，每次移动16字节的数据)。测试程序并不对数据进行任何处理，只是测试读取指令本身。当工作集增大，无法再完全放入L1d时，性能开始急剧下降，跌至每周期6字节。在2^{18}工作集处出现的台阶是由于DTLB缓存耗尽，因此需要对每个新页施加额外处理。由于这里的读取是按顺序的，预取机制可以完美地工作，而FSB能以5.3字节/周期的速度传输内容。但预取的数据并不进入L1d。当然，真实世界的程序永远无法达到以上的数字，但我们可以将它们看作一系列实际上的极限值。

更令人惊讶的是写操作和复制操作的性能。即使是在很小的工作集下，写操作也始终无法达到4字节/周期的速度。这意味着，Intel为Netburst处理器的L1d选择了写通(write-through)模式，所以写入性能受到L2速度的限制。同时，这也意味着，复制测试的性能不会比写入测试差太多(复制测试是将某块内存的数据拷贝到另一块不重叠的内存区)，因为读操作很快，可以与写操作实现部分重叠。最值得关注的地方是，两个操作在工作集无法完全放入L2后出现了严重的性能滑坡，降到了0.5字节/周期！比读操作慢了10倍！显然，如果要提高程序性能，优化这两个操作更为重要。

再来看图3.25，它来自同一颗处理器，只是运行双线程，每个线程分别运行在处理器的一个超线程上。

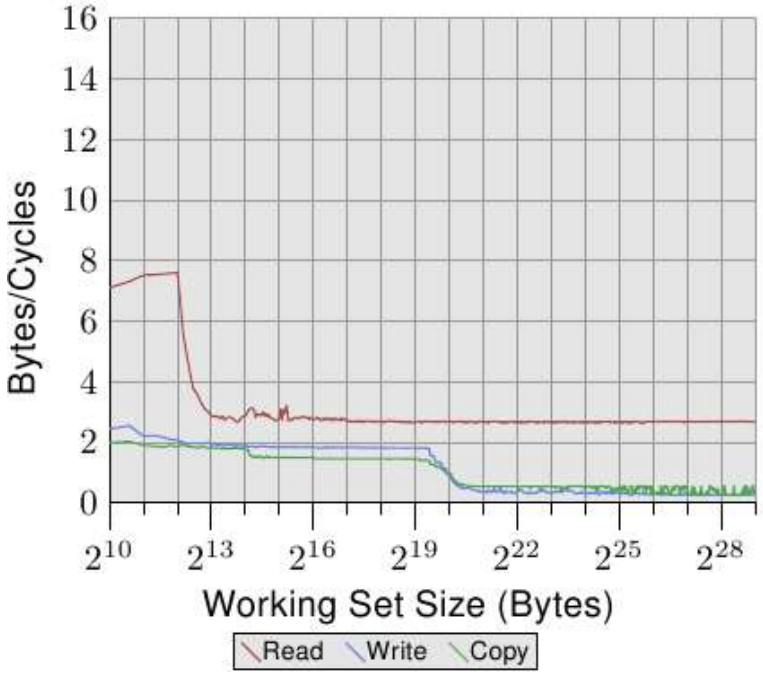


图3.25: P4开启两个超线程时的带宽表现

图3.25采用了与图3.24相同的刻度，以方便比较两者的差异。图3.25中的曲线抖动更多，是由于采用双线程的缘故。结果正如我们预期，由于超线程共享着几乎所有资源(仅除寄存器外)，所以每个超线程只能得到一半的缓存和带宽。所以，即使每个线程都要花上许多时间等待内存，从而把执行时间让给另一个线程，也是无济于事——因为另一个线程也同样需要等待。这里恰恰展示了使用超线程时可能出现的最坏情况。

图3.26: Core 2的带宽表现

再来看Core 2处理器的情况。看看图3.26和图3.27，再对比下P4的图3.24和3.25，可以看出不小的差异。Core 2是一颗双核处理器，有着共享的L2，容量是P4 L2的4倍。但更大的L2只能解释写操作的性能下降出现较晚的现象。

当然还有更大的不同。可以看到，读操作的性能在整个工作集范围内一直稳定在16字节/周期左右，在2^{20}处的下降同样是由于DTLB的耗尽引起。能够达到这么高的数字，不但表明处理器能够预取数据，并且按时完成传输，而且还意味着，预取的数据是被装入L1d的。

写/复制操作的性能与P4相比，也有很大差异。处理器没有采用写通策略，写入的数据留在L1d中，只在必要时才逐出。这使得写操作的速度可以逼近16字节/周期。一旦工作集超过L1d，性能即飞速下降。由于Core 2读操作的性能非常好，所以两者的差值显得特别大。当工作集超过L2时，两者的差值甚至超过20倍！但这并不表示Core 2的性能不好，相反，Core 2永远都比Netburst强。

图3.27: Core 2运行双线程时的带宽表现

在图3.27中，启动双线程，各自运行在Core 2的一个核心上。它们访问相同的内存，但不需要完美同步。从结果上看，读操作的性能与单线程并无区别，只是多了一些多线程情况下常见的抖动。

有趣的地方来了——当工作集小于L1d时，写操作与复制操作的性能很差，就好像数据需要从内存读取一样。两个线程彼此竞争着同一个内存位置，于是不得不频频发送RFO消息。问题的根源在于，虽然两个核心共享着L2，但无法以L2的速度处理RFO请求。而当工作集超过L1d后，性能出现了迅猛提升。这是因为，由于L1d容量不足，于是将被修改的条目刷新到共享的L2。由于L1d的未命中可以由L2满足，只有那些尚未刷新的数据才需要RFO，所以出现了这样的现象。这也是这些工作集情况下速度下降一半的原因。这种渐进式的行为也与我们期待的一致: 由于每个核心共享着同一条FSB，每个核心只能得到一半的FSB带宽，因此对于较大的工作集来说，每个线程的性能大致相当于单线程时的一半。

由于同一个厂商的不同处理器之间都存在着巨大差异，我们没有理由不去研究一下其它厂商处理器的性能。图3.28展示了AMD家族10h Opteron处理器的性能。这颗处理器有64kB的L1d、512kB的L2和2MB的L3，其中L3缓存由所有核心所共享。

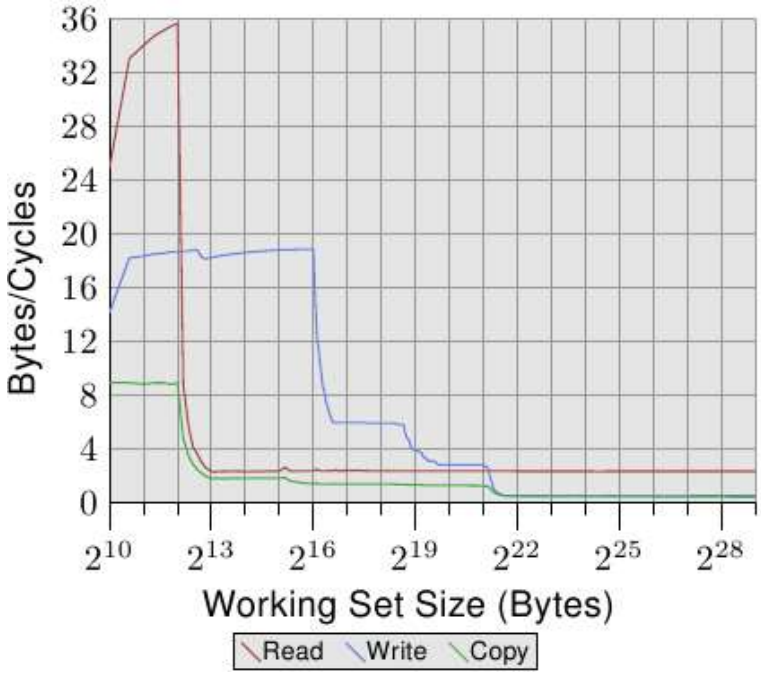


图3.28: AMD家族10h Opteron的带宽表现

大家首先应该会注意到，在L1d缓存足够的情况下，这个处理器每个周期能处理两条指令。读操作的性能超过了32字节/周期，写操作也达到了18.7字节/周期。但是，不久，读操作的曲线就急速下降，跌到2.3字节/周期，非常差。处理器在这个测试中并没有预取数据，或者说，没有有效地预取数据。

另一方面，写操作的曲线随几级缓存的容量而流转。在L1d阶段达到最高性能，随后在L2阶段下降到6字节/周期，在L3阶段进一步下降到2.8字节/周期，最后，在工作集超过L3后，降到0.5字节/周期。它在L1d阶段超过了Core 2，在L2阶段基本相当(Core 2的L2更大一些)，在L3及主存阶段比Core 2慢。

复制的性能既无法超越读操作的性能，也无法超越写操作的性能。因此，它的曲线先是被读性能压制，随后又被写性能压制。

图3.29显示的是Opteron处理器在多线程时的性能表现。

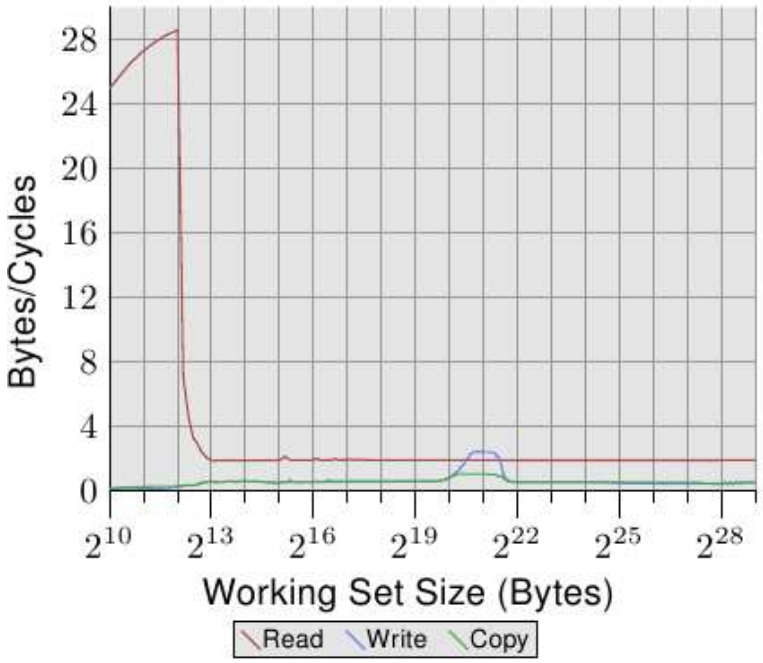


图3.29: AMD Fam 10h在双线程时的带宽表现

读操作的性能没有受到很大的影响。每个线程的L1d和L2表现与单线程下相仿，L3的预取也依然表现不佳。两个线程并没有过渡争抢L3。问题比较大的是写操作的性能。两个线程共享的所有数据都需要经过L3，而这种共享看起来却效率很差。即使是在L3足够容纳整个工作集的情况下，所需要的开销仍然远高于L3的访问时间。再来看图3.27，可以发现，在一定的工作集范围内，Core 2处理器能以共享的L2缓存的速度进行处理。而Opteron处理器只能在很小的一个范围内实现相似的性能，而且，它仅仅只能达到L3的速度，无法与Core 2的L2相比。

3.5.2 关键字加载

内存以比缓存线还小的块从主存储器向缓存传送。如今64位可一次性传送，缓存线的大小为64或128比特。这意味着每个缓存线需要8或16次传送。

DRAM芯片可以以触发模式传送这些64位的块。这使得不需要内存控制器的进一步指令和可能伴随的延迟，就可以将缓存线充满。如果处理器预取了缓存，这有可能是最好的操作方式。

如果程序在访问数据或指令缓存时没有命中(这可能是强制性未命中或容量性未命中，前者是由于数据第一次被使用，后者是由于容量限制而将缓存线逐出)，情况就不一样了。程序需要的并不总是缓存线中的第一个字，而数据块的到达是有先后顺序的，即使是在突发模式和双倍传输率下，也会有明显的时间差，一半在4个CPU周期以上。举例来说，如果程序需要缓存线中的第8个字，那么在首字抵达后它还需要额外等待30个周期以上。

当然，这样的等待并不是必需的。事实上，内存控制器可以按不同顺序去请求缓存线中的字。当处理器告诉它，程序需要缓存中具体某个字，即「关键字(critical word)」时，内存控制器就会先请求这个字。一旦请求的字抵达，虽然缓存线的剩余部分还在传输中，缓存的状态还没有达成一致，但程序已经可以继续运行。这种技术叫做关键字优先及较早重启(Critical Word First & Early Restart)。

现在的处理器都已经实现了这一技术，但有时无法运用。比如，预取操作的时候，并不知道哪个是关键字。如果在预取的中途请求某条缓存线，处理器只能等待，并不能更改请求的顺序。

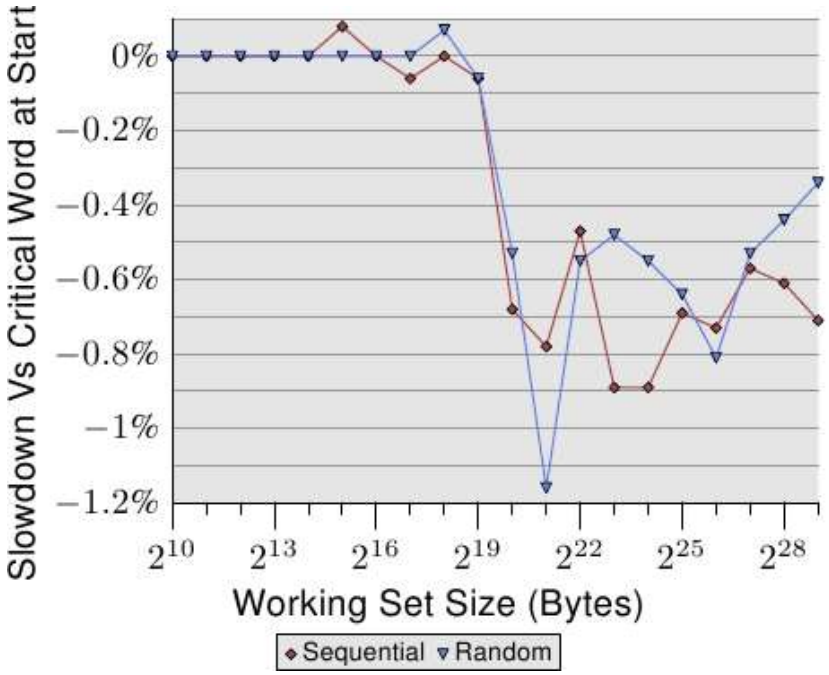


图3.30: 关键字位于缓存线尾时的表现

在关键字优先技术生效的情况下，关键字的位置也会影响结果。图3.30展示了下一个测试的结果，图中表示的是关键字分别在线首和线尾时的性能对比情况。元素大小为64字节，等于缓存线的长度。图中的噪声比较多，但仍然可以看出，当工作集超过L2后，关键字处于线尾情况下的性能要比线首情况下低0.7%左右。而顺序访问时受到的影响更大一些。这与我们前面提到的预取下条线时可能遇到的问题是相符的。

3.5.3 缓存设定

缓存放置的位置与超线程，内核和处理器之间的关系，不在程序员的控制范围之内。但是程序员可以决定线程执行的位置，接着高速缓存与使用的CPU的关系将变得非常重要。

这里我们将不会深入（探讨）什么时候选择什么样的内核以运行线程的细节。我们仅仅描述了在设置关联线程的时候，程序员需要考虑的系统结构的细节。

超线程，通过定义，共享除去寄存器集以外的所有数据。包括 L1 缓存。这里没有什么可以多说的。多核处理器的独立核心带来了一些乐趣。每个核心都至少拥有自己的 L1 缓存。除此之外，下面列出了一些不同的特性：

- 早期多核心处理器有独立的 L2 缓存且没有更高层级的缓存。
- 之后英特尔的双核心处理器模型拥有共享的L2 缓存。对四核处理器，则分对拥有独立的L2 缓存，且没有更高层级的缓存。
- AMD 家族的 10h 处理器有独立的 L2 缓存以及一个统一的L3 缓存。

关于各种处理器模型的优点，已经在它们各自的宣传手册里写得够多了。在每个核心的工作集互不重叠的情况下，独立的L2拥有一定的优势，单线程的程序可以表现优良。考虑到目前实际环境中仍然存在大量类似的情况，这种方法的表现并不会太差。不过，不管怎样，我们总会遇到工作集重叠的情况。如果每个缓存都保存着某些通用运行库的常用部分，那么很显然是一种浪费。

如果像Intel的双核处理器那样，共享除L1外的所有缓存，则会有一个很大的优点。如果两个核心的工作集重叠的部分较多，那么综合起来的可用缓存容量会变大，从而允许容纳更大的工作集而不导致性能的下降。如果两者的工作集并不重叠，那么则是由Intel的高级智能缓存管理(Advanced Smart Cache management)发挥功用，防止其中一个核心垄断整个缓存。

即使每个核心只使用一半的缓存，也会有一些摩擦。缓存需要不断衡量每个核心的用量，在进行逐出操作时可能会作出一些比较差的决定。我们来看另一个测试程序的结果。

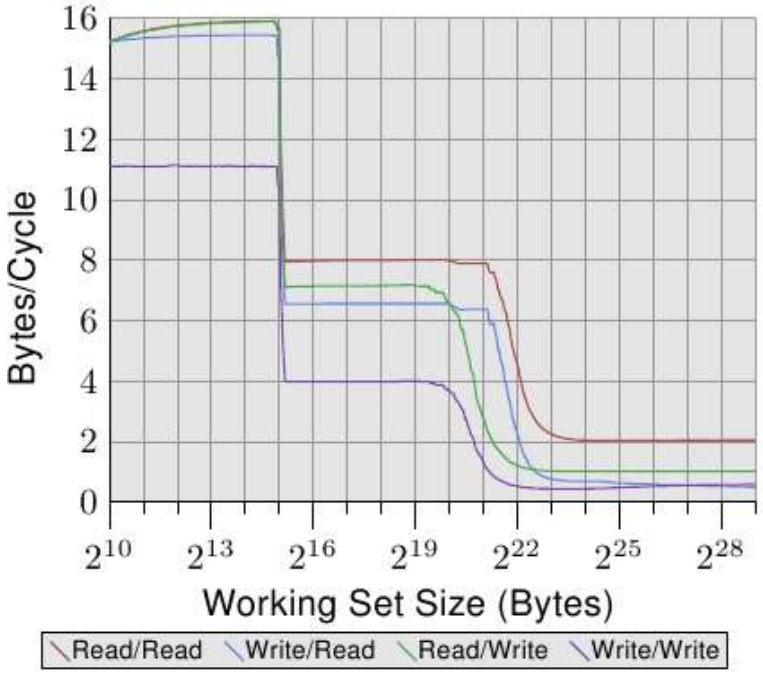


图3.31: 两个进程的带宽表现

这次，测试程序两个进程，第一个进程不断用SSE指令读/写2MB的内存数据块，选择2MB，是因为它正好是Core 2处理器L2缓存的一半，第二个进程则是读/写大小变化的内存区域，我们把这两个进程分别固定在处理器的两个核心上。图中显示的是每个周期读/写的字节数，共有4条曲线，分别表示不同的读写搭配情况。例如，标记为读/写(read/write)的曲线代表的是后台进程进行写操作(固定2MB工作集)，而被测量进程进行读操作(工作集从小到大)。

图中最有趣的是220到223之间的部分。如果两个核心的L2是完全独立的，那么所有4种情况下的性能下降均应发生在221到222之间，也就是L2缓存耗尽的时候。但从图上来看，实际情况并不是这样，特别是背景进程进行写操作时尤为明显。当工作集达到1MB(220)时，性能即出现恶化，两个进程并没有共享内存，因此并不会产生RFO消息。所以，完全是缓存逐出操作引起的问题。目前这种智能的缓存处理机制有一个问题，每个核心能实际用到的缓存更接近1MB，而不是理论上的2MB。如果未来的处理器仍然保留这种多核共享缓存模式的话，我们唯有希望厂商会把这个问题解决掉。

推出拥有双L2缓存的4核处理器仅仅只是一种临时措施，是开发更高级缓存之前的替代方案。与独立插槽及双核处理器相比，这种设计并没有带来多少性能提升。两个核心是通过同一条总线(被外界看作FSB)进行通信，并没有什么特别快的数据交换通道。

未来，针对多核处理器的缓存将会包含更多层次。AMD的10h家族是一个开始，至于会不会有更低级共享缓存的出现，还需要我们拭目以待。我们有必要引入更多级别的缓存，因为频繁使用的高速缓存不可能被许多核心共用，否则会对性能造成很大的影响。我们也需要更大的高关联性缓存，它们的数量、容量和关联性都应该随着共享核心数的增长而增长。巨大的L3和适度的L2应该是一种比较合理的选择。L3虽然速度较慢，但也较少使用。

对于程序员来说，不同的缓存设计就意味着调度决策时的复杂性。为了达到最高的性能，我们必须掌握工作负载的情况，必须了解机器架构的细节。好在我们在判断机器架构时还是有一些支援力量的，我们会在后面的章节介绍这些接口。

3.5.4 FSB的影响

FSB在性能中扮演了核心角色。缓存数据的存取速度受制于内存通道的速度。我们做一个测试，在两台机器上分别跑同一个程序，这两台机器除了内存模块的速度有所差异，其它完全相同。图3.32展示了Addnext0测试(将下一个元素的pad[0]加到当前元素的pad[0]上)在这两台机器上的结果(NPAD=7，64位机器)。两台机器都采用Core 2处理器，一台使用667MHz的DDR2内存，另一台使用800MHz的DDR2内存(比前一台增长20%)。

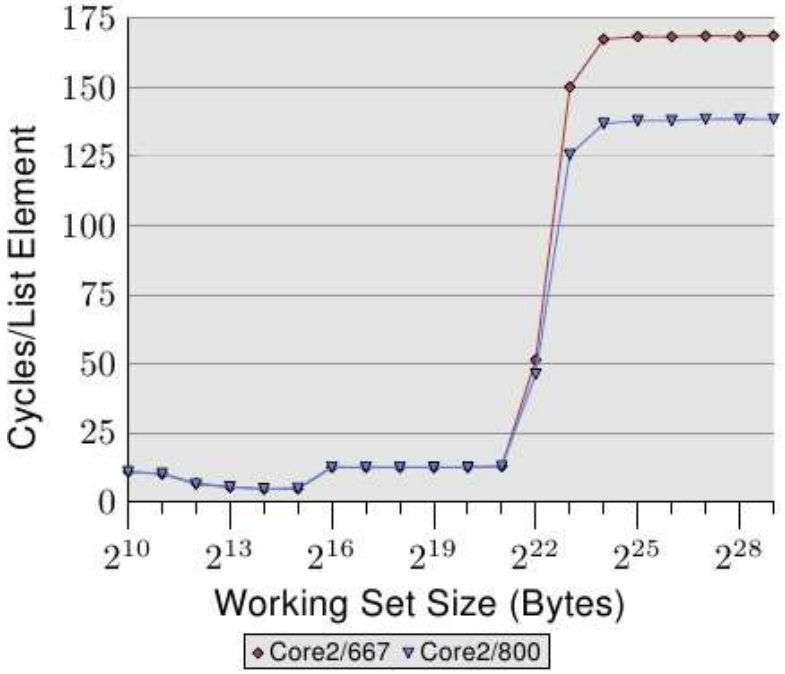


图3.32: FSB速度的影响

图上的数字表明，当工作集大到对FSB造成压力的程度时，高速FSB确实会带来巨大的优势。在我们的测试中，性能的提升达到了18.5%，接近理论上的极限。而当工作集比较小，可以完全纳入缓存时，FSB的作用并不大。当然，这里我们只测试了一个程序的情况，在实际环境中，系统往往运行多个进程，工作集是很容易超过缓存容量的。

如今，一些英特尔的处理器，支持前端总线(FSB)的速度高达1,333 MHz，这意味着速度有另外60%的提升。将来还会出现更高的速度。速度是很重要的，工作集会更大，快速的RAM和高FSB速度的内存肯定是值得投资的。我们必须小心使用它，因为即使处理器可以支持更高的前端总线速度，但是主板的北桥芯片可能不会。使用时，检查它的规范是至关重要的。

本文地址：https://www.oschina.net/translate/what-every-programmer-should-know-about-cpu-cache-part2
原文地址：http://lwn.net/Articles/252125/
