# 每个程序员都应该了解的内存知识

2013/5/1

# 目录

# 内容概要【第一部分】

# 1 简介

早期计算机比现在更为简单。系统的各种组件例如 CPU，内存，大容量存储器和网口，由于被共同开发因而有非常均衡的表现。例如，内存和网口并不比 CPU 在提供数据的时候更（特别的）快。

曾经计算机稳定的基本结构悄然改变，硬件开发人员开始致力于优化单个子系统。于是电脑一些组件的性能大大的落后因而成为了瓶颈。由于开销的原因，大容量存储器和内存子系统相对于其他组件来说改善得更为缓慢。

大容量存储的性能问题往往靠软件来改善：操作系统将常用(且最有可能被用)的数据放在主存中，因为后者的速度要快上几个数量级。或者将缓存加入存储设备中，这样就可以在不修改操作系统的前提下提升性能。{然而，为了在使用缓存时保证数据的完整性，仍然要作出一些修改。}这些内容不在本文的谈论范围之内，就不作赘述了。

而解决内存的瓶颈更为困难，它与大容量存储不同，几乎每种方案都需要对硬件作出修改。目前，这些变更主要有以下这些方式：

- RAM 的硬件设计(速度与并发度)
- 内存控制器的设计
- CPU 缓存
- 设备的直接内存访问(DMA)

本文主要关心的是 CPU 缓存和内存控制器的设计。在讨论这些主题的过程中，我们还会研究 DMA。不过，我们首先会从当今商用硬件的设计谈起。这有助于我们理解目前在使用内存子系统时可能遇到的问题和限制。我们还会详细介绍 RAM 的分类，说明为什么会存在这么多不同类型的内存。

本文不会包括所有内容，也不会包括最终性质的内容。我们的讨论范围仅止于商用硬件，而且只限于其中的小部分。另外，本文中的许多论题，我们只会点到为止，以达到本文目标为标准。对于这些论题，大家可以阅读其它文档，获得更详细的说明。

当本文提到操作系统特定的细节和解决方案时，针对的都是 Linux。无论何时都不会包含别的操作系统的任何信息，作者无意讨论其他操作系统的情况。如果读者认为他/她不得不使用别的操作系统，那么必须去要求供应商提供其操作系统类似于本文的文档。

在开始之前最后的一点说明，本文包含大量出现的术语“经常”和别的类似的限定词。这里讨论的技术在现实中存在于很多不同的实现，所以本文只阐述使用得最广泛最主流的版本。在阐述中很少有地方能用到绝对的限定词。

## 1.1 文档结构

这个文档主要视为软件开发者而写的。本文不会涉及太多硬件细节，所以喜欢硬件的读者也许不会觉得有用。但是在我们讨论一些有用的细节之前，我们先要描述足够多的背景。

在这个基础上，本文的第二部分将描述 RAM（随机寄存器）。懂得这个部分的内容很好，但是此部分的内容并不是懂得其后内容必须部分。我们会在之后引用不少之前的部分，所以心急的读者可以跳过任何章节来读他们认为有用的部分。

第三部分会谈到不少关于 CPU 缓存行为模式的内容。我们会列出一些图标，这样你们不至于觉得太枯燥。第三部分对于理解整个文章非常重要。第四部分将简短的描述虚拟内存是怎么被实现的。这也是你们需要理解全文其他部分的背景知识之一。

第五部分会提到许多关于 Non Uniform Memory Access (NUMA)系统。

第六部分是本文的中心部分。在这个部分里面，我们将回顾其他许多部分中的信息，并且我们将给阅读本文的程序员许多在各种情况下的编程建议。如果你真的很心急，那么你可以直接阅读第六部分，并且我们建议你在必要的时候回到之前的章节回顾一下必要的背景知识。

本文的第七部分将介绍一些能够帮助程序员更好的完成任务的工具。即便在彻底理解了某一项技术的情况下，距离彻底理解在非测试环境下的程序还是很遥远的。我们需要借助一些工具。

第八部分，我们将展望一些在未来我们可能认为好用的科技。

## 1.2 反馈问题

作者会不定期更新本文档。这些更新既包括伴随技术进步而来的更新也包含更改错误。非常欢迎有志于反馈问题的读者发送电子邮件。

## 1.3 致谢

我首先需要感谢 Johnray Fuller 尤其是 Jonathan Corbet，感谢他们将作者的英语转化成为更为规范的形式。Markus Armbruster 提供大量本文中对于问题和缩写有价值的建议。

## 1.4 关于本文

本文题目对 David Goldberg 的经典文献《What Every Computer Scientist Should Know About Floating-Point Arithmetic》[goldberg]表示致敬。Goldberg 的论文虽然不普及，但是对于任何有志于严格编程的人都会是一个先决条件。

# 2 商用硬件现状

鉴于目前专业硬件正在逐渐淡出，理解商用硬件的现状变得十分重要。现如今，人们更多的采用水平扩展，也就是说，用大量小型、互联的商用计算机代替巨大、超快(但超贵)的系统。原因在于，快速而廉价的网络硬件已经崛起。那些大型的专用系统仍然有一席之地，但已被商用硬件后来居上。2007 年，Red Hat 认为，未来构成数据中心的“积木”将会是拥有最多 4 个插槽的计算机，每个插槽插入一个四核 CPU，这些 CPU 都是超线程的。{超线程使单个处理器核心能同时处理两个以上的任务，只需加入一点点额外硬件}。也就是说，这些数据中心中的标准系统拥有最多 64 个虚拟处理器。当然可以支持更大的系统，但人们认为 4 插槽、4 核 CPU 是最佳配置，绝大多数的优化都针对这样的配置。

在不同商用计算机之间，也存在着巨大的差异。不过，我们关注在主要的差异上，可以涵盖到超过 90%以上的硬件。需要注意的是，这些技术上的细节往往日新月异，变化极快，因此大家在阅读的时候也需要注意本文的写作时间。

这么多年来，个人计算机和小型服务器被标准化到了一个芯片组上，它由两部分组成：北桥和南桥，见图 2.1。

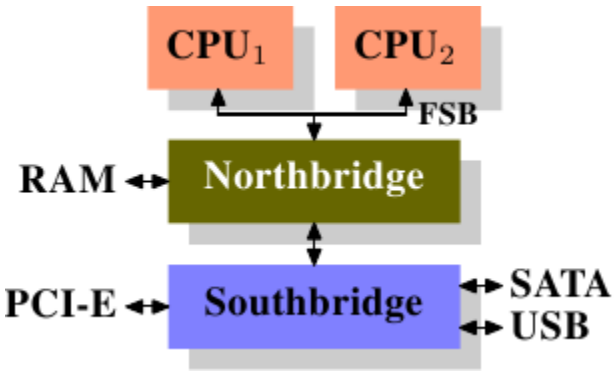



图 2.1 北桥和南桥组成的结构

CPU 通过一条通用总线(前端总线，FSB)连接到北桥。北桥主要包括内存控制器和其它一些组件，内存控制器决定了 RAM 芯片的类型。不同的类型，包括 DRAM、Rambus 和 SDRAM 等等，要求不同的内存控制器。

为了连通其它系统设备，北桥需要与南桥通信。南桥又叫 I/O 桥，通过多条不同总线与设备们通信。目前，比较重要的总线有 PCI、PCI Express、SATA 和 USB 总线，除此以外，南桥还支持 PATA、IEEE 1394、串行口和并行口等。比较老的系统上有连接北桥的 AGP 槽。那是由于南北桥间缺乏高速连接而采取的措施。现在的 PCI-E 都是直接连到南桥的。

这种结构有一些需要注意的地方：

- 从某个 CPU 到另一个 CPU 的数据需要走它与北桥通信的同一条总线。
- 与 RAM 的通信需要经过北桥
- RAM 只有一个端口。{本文不会介绍多端口 RAM，因为商用硬件不采用这种内存，至少程序员无法访问到。这种内存一般在路由器等专用硬件中采用。}

- CPU 与南桥设备间的通信需要经过北桥

在上面这种设计中，瓶颈马上出现了。第一个瓶颈与设备对 RAM 的访问有关。早期，所有设备之间的通信都需要经过 CPU，结果严重影响了整个系统的性能。为了解决这个问题，有些设备加入了直接内存访问(DMA)的能力。DMA 允许设备在北桥的帮助下，无需 CPU 的干涉，直接读写 RAM。到了今天，所有高性能的设备都可以使用 DMA。虽然 DMA 大大降低了 CPU 的负担，却占用了北桥的带宽，与 CPU 形成了争用。

第二个瓶颈来自北桥与 RAM 间的总线。总线的具体情况与内存的类型有关。在早期的系统上，只有一条总线，因此不能实现并行访问。近期的 RAM 需要两条独立总线(或者说通道，DDR2 就是这么叫的，见图 2.8)，可以实现带宽加倍。北桥将内存访问交错地分配到两个通道上。更新的内存技术(如 FB-DRAM)甚至加入了更多的通道。

由于带宽有限，我们需要以一种使延迟最小化的方式来对内存访问进行调度。我们将会看到，处理器的速度比内存要快得多，需要等待内存。如果有多个超线程核心或 CPU 同时访问内存，等待时间则会更长。对于 DMA 也是同样。

除了并发以外，访问模式也会极大地影响内存子系统、特别是多通道内存子系统的性能。关于访问模式，可参见 2.2 节。

在一些比较昂贵的系统上，北桥自己不含内存控制器，而是连接到外部的多个内存控制器上(在下例中，共有 4 个)。

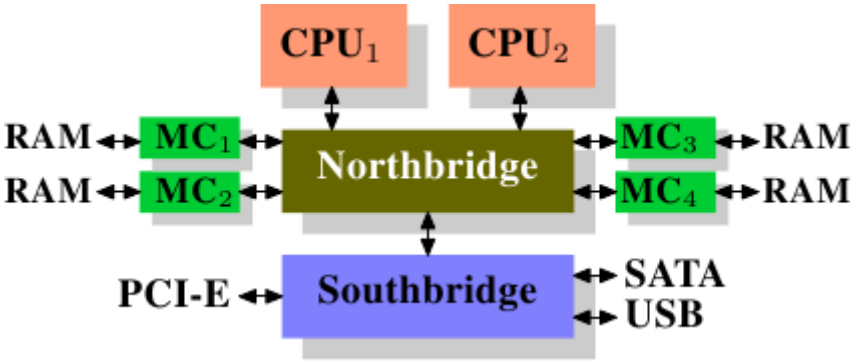



图 2.2 拥有外部控制器的北桥

这种架构的好处在于，多条内存总线的存在，使得总带宽也随之增加了。而且也可以支持更多的内存。通过同时访问不同内存区，还可以降低延时。对于像图 2.2 中这种多处理器直连北桥的设计来说，尤其有效。而这种架构的局限在于北桥的内部带宽，非常巨大(来自 Intel)。{出于完整性的考虑，还需要补充一下，这样的内存控制器布局还可以用于其它用途，比如说「内存 RAID」，它可以与热插拔技术一起使用。}

使用外部内存控制器并不是唯一的办法，另一个最近比较流行的方法是将控制器集成到 CPU 内部，将内存直连到每个 CPU。这种架构的走红归功于基于 AMD Opteron 处理器的 SMP 系统。图 2.3 展示了这种架构。Intel 则会从 Nehalem 处理器开始支持通用系统接口(CSI)，基本上也是类似的思路——集成内存控制器，为每个处理器提供本地内存。

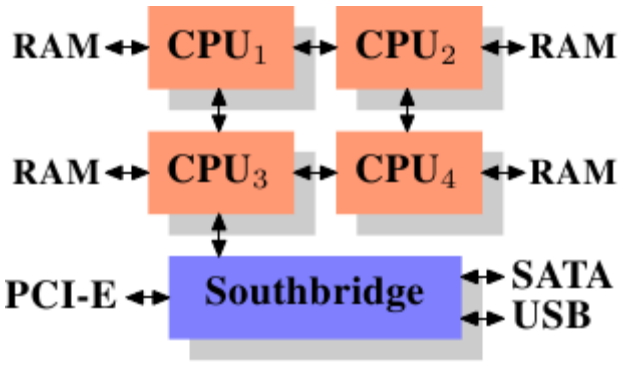



图 2.3 集成的内存控制器

通过采用这样的架构，系统里有几个处理器，就可以有几个内存库(memory bank)。比如，在 4 CPU 的计算机上，不需要一个拥有巨大带宽的复杂北桥，就可以实现 4 倍的内存带宽。另外，将内存控制器集成到 CPU 内部还有其它一些优点，这里就不赘述了。

同样也有缺点。首先，系统仍然要让所有内存能被所有处理器所访问，导致内存不再是统一的资源(NUMA 即得名于此)。处理器能以正常的速度访问本地内存(连接到该处理器的内存)。但它访问其它处理器的内存时，却需要使用处理器之间的互联通道。比如说，CPU 1 如果要访问 CPU 2 的内存，则需要使用它们之间的互联通道。如果它需要访问 CPU 4 的内存，那么需要跨越两条互联通道。

使用互联通道是有代价的。在讨论访问远端内存的代价时，我们用「NUMA 因子」这个词。在图 2.3 中，每个 CPU 有两个层级：相邻的 CPU，以及两个互联通道外的 CPU。在更加复杂的系统中，层级也更多。甚至有些机器有不止一种连接，比如说 IBM 的 x445 和 SGI 的 Altix 系列。CPU 被归入节点，节点内的内存访问时间是一致的，或者只有很小的 NUMA 因子。而在节点之间的连接代价很大，而且有巨大的 NUMA 因子。

目前，已经有商用的 NUMA 计算机，而且它们在未来应该会扮演更加重要的角色。人们预计，从 2008 年底开始，每台 SMP 机器都会使用 NUMA。每个在 NUMA 上运行的程序都应该认识到 NUMA 的代价。在第 5 节中，我们将讨论更多的架构，以及 Linux 内核为这些程序提供的一些技术。

除了本节中所介绍的技术之外，还有其它一些影响 RAM 性能的因素。它们无法被软件所左右，所以没有放在这里。如果大家有兴趣，可以在第 2.1 节中看一下。介绍这些技术，仅仅是因为它们能让我们绘制的 RAM 技术全图更为完整，或者是可能在大家购买计算机时能够提供一些帮助。

以下的两节主要介绍一些入门级的硬件知识，同时讨论内存控制器与 DRAM 芯片间的访问协议。这些知识解释了内存访问的原理，程序员可能会得到一些启发。不过，这部分并不是必读的，心急的读者可以直接跳到第 2.2.5 节。

# 2.1 RAM 类型

这些年来，出现了许多不同类型的 RAM，各有差异，有些甚至有非常巨大的不同。那些很古老的类型已经乏人问津，我们就不仔细研究了。我们主要专注于几类现代 RAM，剖开它们的表面，研究一下内核和应用开发人员们可以看到的一些细节。

第一个有趣的细节是，为什么在同一台机器中有不同的 RAM？或者说得更详细一点，为什么既有静态 RAM(SRAM {SRAM 还可以表示「同步内存」。})，又有动态 RAM(DRAM)。功能相同，前者更快。那么，为什么不全部使用 SRAM？答案是，代价。无论在生产还是在使用上，SRAM 都比 DRAM 要贵得多。生产和使用，这两个代价因子都很重要，后者则是越来越重要。为了理解这一点，我们分别看一下 SRAM 和 DRAM 一个位的存储的实现过程。

在本节的余下部分，我们将讨论 RAM 实现的底层细节。我们将尽量控制细节的层面，比如，在「逻辑的层面」讨论信号，而不是硬件设计师那种层面，因为那毫无必要。

### 2.1.1 静态 RAM

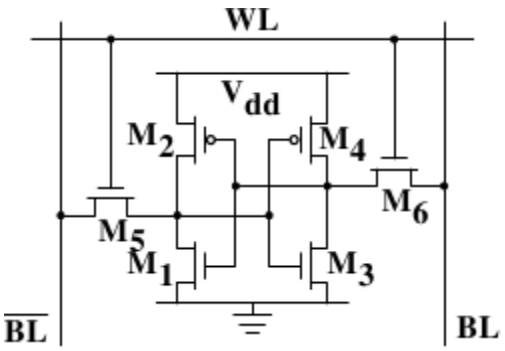



图 2.6 6-T 静态 RAM

图 2.4 展示了 6 晶体管 SRAM 的一个单元。核心是 4 个晶体管 M1-M4，它们组成两个交叉耦合的反相器。它们有两个稳定的状态，分别代表 0 和 1。只要保持 Vdd 有电，状态就是稳定的。

当需要访问单元的状态时，升起字访问线 WL。BL 和 BL 上就可以读取状态。如果需要覆盖状态，先将 BL 和 BL 设置为期望的值，然后升起 WL。由于外部的驱动强于内部的 4 个晶体管，所以旧状态会被覆盖。

更多详情，可以参考[sramwiki]。为了下文的讨论，需要注意以下问题:

一个单元需要 6 个晶体管。也有采用 4 个晶体管的 SRAM，但有缺陷。

维持状态需要恒定的电源。

升起 WL 后立即可以读取状态。信号与其它晶体管控制的信号一样，是直角的(快速在两个状态间变化)。

状态稳定，不需要刷新循环。

SRAM 也有其它形式，不那么费电，但比较慢。由于我们需要的是快速 RAM，因此不在关注范围内。这些较慢的 SRAM 的主要优点在于接口简单，比动态 RAM 更容易使用。

### 2.1.2 动态 RAM

动态 RAM 比静态 RAM 要简单得多。图 2.5 展示了一种普通 DRAM 的结构。它只含有一个晶体管和一个电容器。显然，这种复杂性上的巨大差异意味着功能上的迥异。

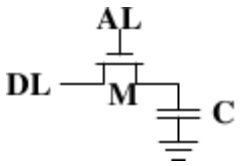



图 2.5 1-T 动态 RAM

动态 RAM 的状态是保持在电容器 C 中。晶体管 M 用来控制访问。如果要读取状态，升起访问线 AL，这时，可能会有电流流到数据线 DL 上，也可能没有，取决于电容器是否有电。如果要写入状态，先设置 DL，然后升起 AL 一段时间，直到电容器充电或放电完毕。

动态 RAM 的设计有几个复杂的地方。由于读取状态时需要对电容器放电，所以这一过程不能无限重复，不得不在某个点上对它重新充电。

更糟糕的是，为了容纳大量单元(现在一般在单个芯片上容纳 10 的 9 次方以上的 RAM 单元)，电容器的容量必须很小(0.000000000000001 法拉以下)。这样，完整充电后大约持有几万个电子。即使电容器的电阻很大(若干兆欧姆)，仍然只需很短的时间就会耗光电荷，称为「泄漏」。

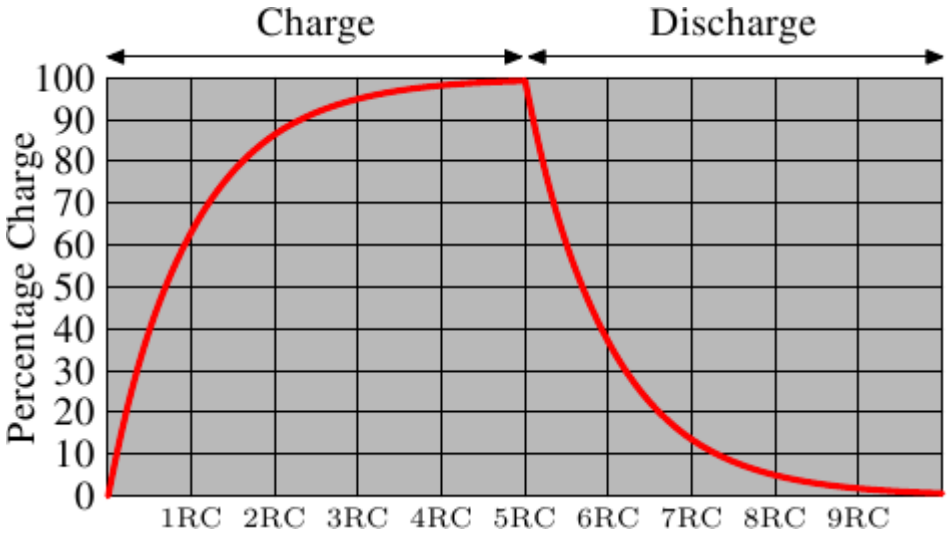
这种泄露就是现在的大部分 DRAM 芯片每隔 64ms 就必须进行一次刷新的原因。在刷新期间，对于该芯片的访问是不可能的，这甚至会造成半数任务的延宕。（相关内容请察看【highperfdram】一章）

这个问题的另一个后果就是无法直接读取芯片单元中的信息，而必须通过信号放大器将 0 和 1 两种信号间的电势差增大。

最后一个问题在于电容器的冲放电是需要时间的，这就导致了信号放大器读取的信号并不是典型的矩形信号。所以当放大器输出信号的时候就需要一个小小的延宕，相关公式如下

$$\begin{aligned} Q_{\text{Charge}}(t) &= Q_0(1 - e^{-\frac{t}{RC}}) \\ Q_{\text{Discharge}}(t) &= Q_0 e^{-\frac{t}{RC}} \end{aligned}$$

这就意味着需要一些时间（时间长短取决于电容 C 和电阻 R）来对电容进行冲放电。另一个负面作用是，信号放大器的输出电流不能立即就作为信号载体使用。图 2.6 显示了冲放电的曲线，x 轴表示的是单位时间下的 R*C




与静态 RAM 可以即刻读取数据不同的是，当要读取动态 RAM 的时候，必须花一点时间来等待电容的冲放电完全。这一点点的时间最终限制了 DRAM 的速度。

当然了，这种读取方式也是有好处的。最大的好处在于缩小了规模。一个动态 RAM 的尺寸是小于静态 RAM 的。这种规模的减小不单单建立在动态 RAM 的简单结构之上，也是由于减少了静态 RAM 的各个单元独立的供电部分。以上也同时导致了动态 RAM 模具的简单化。

综上所述，由于不可思议的成本差异，除了一些特殊的硬件（包括路由器什么的）之外，我们的硬件大多是使用 DRAM 的。这一点深深的影响了咱们这些程序员，后文将会对此进行讨论。在此之前，我们还是先了解下 DRAM 的更多细节。

### 2.1.3 DRAM 访问

一个程序选择了一个内存位置使用到了一个虚拟地址。处理器转换这个到物理地址最后将内存控制选择 RAM 芯片匹配了那个地址。在 RAM 芯片去选择单个内存单元，部分的物理地址以许多地址行的形式被传递。

它单独地去处理来自于内存控制器的内存位置将完全不切实际：4G 的 RAM 将需要 $2^{32}$ 地址行。地址传递 DRAM 芯片的这种方式首先必须被路由器解析。一个路由器的 N 多地址行将有 $2^{N}$ 输出行。这些输出行能被使用到选择内存单元。使用这个直接方法对于小容量芯片不再是个大问题。

但如果许多的单元生成这种方法不在适合。一个 1G 的芯片容量（我反感那些 SI 前缀，对于我一个 *giga-bit 将总是* $2^{30}$ *而不是* $10^{9}$ 字节）将需要 30 地址行和 $2^{30}$ 选 项行。一个路由器的大小及许多的输入行以指数方式递增当速度不被牺牲时。一个 30 地址行路由器需要一大堆芯片的真实身份另外路由器也就复杂起来了。更重要 的是，传递 30 脉冲在地址行同步要比仅仅传递 15 脉冲困难的多。较少列能精确布局相同长度或恰当的时机（现代 DRAM 类型像 DDR3 能自动调整时序但这个 限制能让他什么都能忍受）

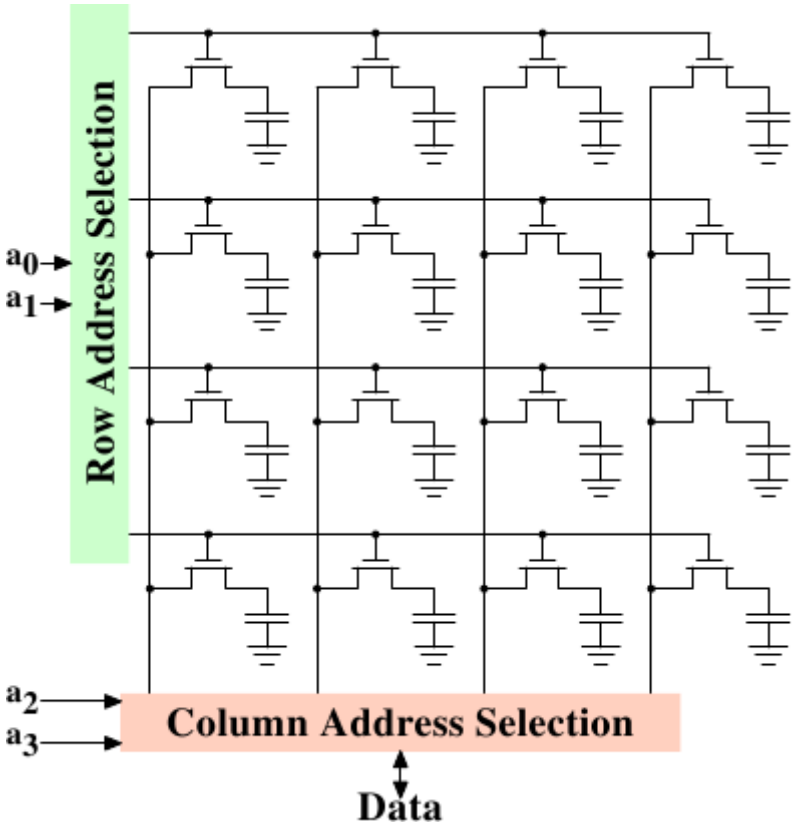



图 2.7 展示了一个很高级别的一个 DRAM 芯片，DRAM 被组织在行和列里。他们能在一行中对奇但 DRAM 芯片需要一个大的路由器。通过阵列方法设计能被 一个路由器和一个半的 multiplexer 获得{多路复用器（multiplexer）和路由器是一样的，这的 multiplexer 需要以路由器身份 工作当写数据时候。那么从现在开始我们开始讨论其区别.}这在所有方面会是一个大的存储。例如地址 linesa$_0$和 a$_1$通过行地址选择路由器来选择整个行的芯片的地址列，当读的时候，所有的芯片目录能使其纵列选择路由器可用，依据地址 linesa$_2$和 a$_3$一个纵列的目录用于数据 DRAM 芯片的接口类型。这发生了许多次在许多 DRAM 芯片产生一个总记录数的字节匹配给一个宽范围的数据总线。

对于写操作，内存单元的数据新值被放到了数据总线，当使用 RAS 和 CAS 方式选中内存单元时，数据是存放在内存单元内的。这是一个相当直观的设计，在现实中——很显然——会复杂得多，对于读，需要规范从发出信号到数据在数据总线上变得可读的时延。电容不会像前面章节里面描述的那样立刻自动放电，从内存单元发出的信号是如此这微弱以至于它需要被放大。对于写，必须规范从数据 RAS 和 CAS 操作完成后到数据成功的被写入内存单元的时延（当然，电容不会立刻自动充电和放电）。这些时间常量对于 DRAM 芯片的性能是至关重要的，我们将在下章讨论它。

另一个关于伸缩性的问题是，用 30 根地址线连接到每一个 RAM 芯片是行不通的。芯片的针脚是非常珍贵的资源，以至数据必须能并行传输就并行传输（比如：64 位为一组）。内存控制器必须有能力解析每一个 RAM 模块（RAM 芯片集合）。如果因为性能的原因要求并发行访问多个 RAM 模块并且每个 RAM 模块需要自己独占的 30 或多个地址线，那么对于 8 个 RAM 模块，仅仅是解析地址，内存控制器就需要 240+之多的针脚。

在很长一段时间里，地址线被复用以解决 DRAM 芯片的这些次要的可扩展性问题。这意味着地址被转换成两部分。第一部分由地址位 a0 和 a1 选择行（如图 2.7）。这个选择保持有

效直到撤销。然后是第二部分，地址位 a2 和 a3 选择列。关键差别在于：只需要两根外部地址线。需要一些很少的线指明 RAS 和 CAS 信号有效，但是把地址线的数目减半所付出的代价更小。可是地址复用也带来自身的一些问题。我们将在 2.2 章中提到。

### 2.1.4 总结

如果这章节的内容有些难以应付，不用担心。纵观这章节的重点有：

- 为什么不是所有的存储器都是 SRAM 的原因
- 存储单元需要单独选择来使用
- 地址线数目直接负责存储控制器，主板，DRAM 模块和 DRAM 芯片的成本
- 在读或写操作结果之前需要占用一段时间是可行的

接下来的章节会涉及更多的有关访问 DRAM 存储器的实际操作的细节。我们不会提到更多有关访问 SRAM 的具体内容，它通常是直接寻址。这里是由于速度和有 限的 SRAM 存储器的尺寸。SRAM 现在应用在 CPU 的高速缓存和芯片，它们的连接件很小而且完全能在 CPU 设计师的掌控之下。我们以后会讨论到 CPU 高 速缓存这个主题，但我们所需要知道的是 SRAM 存储单元是有确定的最大速度，这取决于花在 SRAM 上的艰难的尝试。这速度与 CPU 核心相比略慢一到两个数量级。

## 2.2 DRAM 访问细节

在上文介绍 DRAM 的时候，我们已经看到 DRAM 芯片为了节约资源，对地址进行了复用。而且，访问 DRAM 单元是需要一些时间的，因为电容器的放电并不是 瞬时的。此外，我们还看到，DRAM 需要不停地刷新。在这一节里，我们将把这些因素拼合起来，看看它们是如何决定 DRAM 的访问过程。

我们将主要关注在当前的科技上，不会再去讨论异步 DRAM 以及它的各种变体。如果对它感兴趣,可以去参考[highperfdram]及 [arstechtwo]。我们也不会讨论 Rambus DRAM(RDRAM)，虽然它并不过时，但在系统内存领域应用不广。我们将主要介绍同步 DRAM(SDRAM)及其后继者双倍速 DRAM(DDR)。

同步 DRAM，顾名思义，是参照一个时间源工作的。由内存控制器提供一个时钟，时钟的频率决定了前端总线(FSB)的速度。FSB 是内存控制器提供给 DRAM 芯片的接口。在我写作本文的时候，FSB 已经达到 800MHz、1066MHz，甚至 1333MHz，并且下一代的 1600MHz 也已经宣布。但这并不表示时钟频率有这么高。实际上，目前的总线都是双倍或四倍传输的，每个周期传输 2 次或 4 次数据。报的越高，卖的越好，所以这些厂商们喜欢把四倍传输的 200MHz 总线宣传为“有效的”800MHz 总线。

以今天的 SDRAM 为例，每次数据传输包含 64 位，即 8 字节。所以 FSB 的传输速率应该是有效总线频率乘于 8 字节(对于 4 倍传输 200MHz 总线而言，传输速率为 6.4GB/s)。听起来很高，但要知道这只是峰值速率，实际上无法达到的最高速率。我们将会看到，与 RAM 模块交流的协议有大量时间是处于非工作状态，不进行数据传输。我们必须对这些非工作时间有所了解，并尽量缩短它们，才能获得最佳的性能。

### 2.2.1 读访问协议

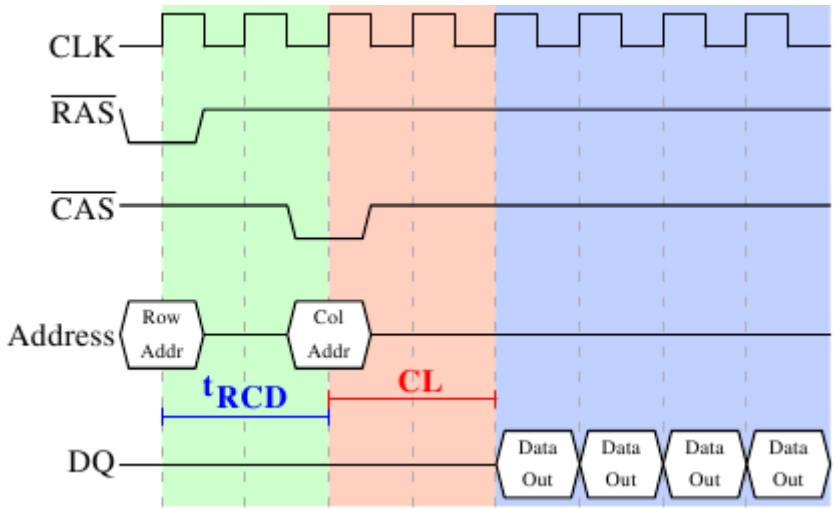



图 2.8: SDRAM 读访问的时序

图 2.8 展示了某个 DRAM 模块一些连接器上的活动，可分为三个阶段，图上以不同颜色表示。按惯例，时间为从左向右流逝。这里忽略了许多细节，我们只关注 时钟频率、RAS 与 CAS 信号、地址总线和数据总线。首先，内存控制器将行地址放在地址总线上，并降低 RAS 信号，读周期开始。所有信号都在时钟 (CLK)的上升沿读取，因此，只要信号在读取的时间点上保持稳定，就算不是标准的方波也没有关系。设置行地址会促使 RAM 芯片锁住指定的行。

CAS 信号在 tRCD(RAS 到 CAS 时延)个时钟周期后发出。内存控制器将列地址放在地址总线上，降低 CAS 线。这里我们可以看到，地址的两个组成部分是怎么通过同一条总线传输的。

至此，寻址结束，是时候传输数据了。但 RAM 芯片任然需要一些准备时间，这个时间称为 CAS 时延(CL)。在图 2.8 中 CL 为 2。这个值可大可小，它取决 于内存控制器、主板和 DRAM 模块的质量。CL 还可能是半周期。假设 CL 为 2.5，那么数据将在蓝色区域内的第一个下降沿准备就绪。

既然数据的传输需要这么多的准备工作，仅仅传输一个字显然是太浪费了。因此，DRAM 模块允许内存控制指定本次传输多少数据。可以是 2、4 或 8 个字。这 样，就可以一次填满高速缓存的整条线，而不需要额外的 RAS/CAS 序列。另外，内存控制器还可以在不重置行选择的前提下发送新的 CAS 信号。这样，读取 或写入连续的地址就可以变得非常快，因为不需要发送 RAS 信号，也不需要把行置为非激活状态(见下文)。是否要将行保持为“打开”状态是内存控制器判断的 事情。让它一直保持打开的话，对真正的应用会有不好的影响(参见[highperfdram])。CAS 信号的发送仅与 RAM 模块的命令速率 (Command Rate)有关(常常记为 T$x$，其中 $x$ 为 1 或 2，高性能的 DRAM 模块一般为 1，表示在每个周期都可以接收新命令)。

在上图中，SDRAM 的每个周期输出一个字的数据。这是第一代的 SDRAM。而 DDR 可以在一个周期中输出两个字。这种做法可以减少传输时间，但无法降低 时延。DDR2 尽管看上去不同，但在本质上也是相同的做法。对于 DDR2，不需要再深入介绍了，我们只需要知道 DDR2 更快、更便宜、更可靠、更节能(参 见[ddrtwo])就足够了。

## 2.2.2 预充电与激活

图 2.8 并不完整，它只画出了访问 DRAM 的完整循环的一部分。在发送 RAS 信号之前，必须先把当前锁住的行置为非激活状态，并对新行进行预充电。在这 里，我们主要讨论由于显式发送指令而触发以上行为的情况。协议本身作了一些改进，在某些情况下是可以省略这个步骤的，但预充电带来的时延还是会影响整个操 作。

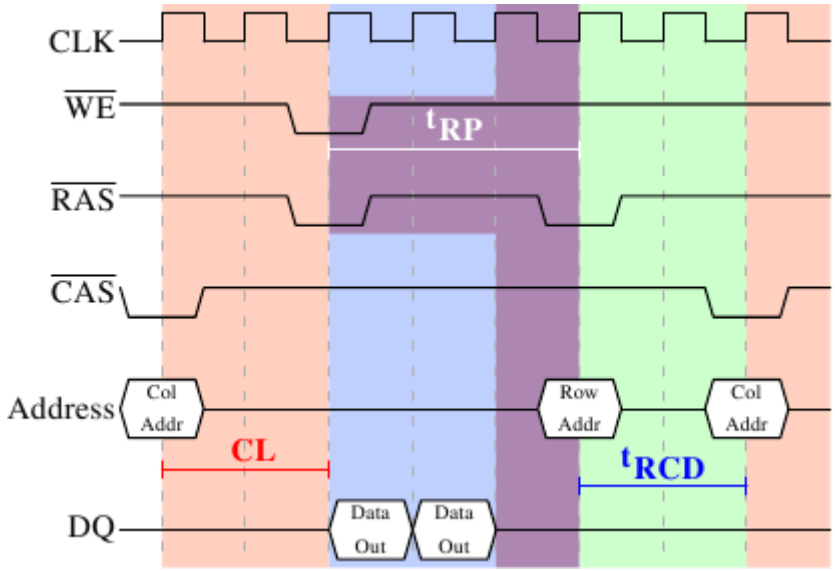



图 2.9: SDRAM 的预充电与激活

图 2.9 显示的是两次 CAS 信号的时序图。第一次的数据在 CL 周期后准备就绪。图中的例子里，是在 SDRAM 上，用两个周期传输了两个字的数据。如果换成 DDR 的话，则可以传输 4 个字。

即使是在一个命令速率为 1 的 DRAM 模块上，也无法立即发出预充电命令，而要等数据传输完成。在上图中，即为两个周期。刚好与 CL 相同，但只是巧合而已。 预充电信号并没有专用线，某些实现是用同时降低写使能(WE)线和 RAS 线的方式来触发。这一组合方式本身没有特殊的意义(参见 [micronddr])。

发出预充电信命令后，还需等待 $t_{RP}$(行预充电时间)个周期之后才能使行被选中。在图 2. 中，这个时间(紫色部分)大部分与内存传输的时间(淡蓝色部分)重合。不错。但 $t_{RP}$ 大于传输时间，因此下一个 RAS 信号只能等待一个周期。

如果我们补充完整上图中的时间线，最后会发现下一次数据传输发生在前一次的 5 个周期之后。这意味着，数据总线的 7 个周期中只有 2 个周期才是真正在用的。再用它乘于 FSB 速度，结果就是，800MHz 总线的理论速率 6.4GB/s 降到了 1.8GB/s。真是太糟了。第 6 节将介绍一些技术，可以帮助我们提高总线有效速率。程序员们也需要尽自己的努力。

SDRAM 还有一些定时值，我们并没有谈到。在图 2.9 中，预充电命令仅受制于数据传输时间。除此之外，SDRAM 模块在 RAS 信号之后，需要经过一段时间，才能进行预充电(记为 tRAS)。

它的值很大，一般达到 tRP 的 2 到 3 倍。如果在某个 RAS 信号之后，只有一个 CAS 信号，而且数据只传输很少几个周期，那么就有问题了。假设在图 2.9 中，第一个 CAS 信号是直接跟在一个 RAS 信号后免的，而 tRAS 为 8 个周期。那么预充电命令还需要被推迟一个周期，因为 tRCD、CL 和 tRP 加起来才 7 个周期。

DDR 模块往往用 w-z-y-z-T 来表示。例如，2-3-2-8-T1，意思是：

w 2 CAS 时延(CL)

x 3 RAS-to-CAS 时延(t RCD)

y 2 RAS 预充电时间(t RP)

z 8 激活到预充电时间(t RAS)

T T1 命令速率

当然，除以上的参数外，还有许多其它参数影响命令的发送与处理。但以上 5 个参数已经足以确定模块的性能。

在解读计算机性能参数时，这些信息可能会派上用场。而在购买计算机时，这些信息就更有用了，因为它们与 FSB/SDRAM 速度一起，都是决定计算机速度的关键因素。

喜欢冒险的读者们还可以利用它们来调优系统。有些计算机的 BIOS 可以让你修改这些参数。SDRAM 模块有一些可编程寄存器，可供设置参数。BIOS 一般会挑选最佳值。如果 RAM 模块的质量足够好，我们可以在保持系统稳定的前提下将减小以上某个时延参数。互联网上有大量超频网站提供了相关的文档。不过，这是有风险的，需要大家自己承担，可别怪我没有事先提醒哟。

### 2.2.3 重充电

谈到 DRAM 的访问时，重充电是常常被忽略的一个主题。在 2.1.2 中曾经介绍，DRAM 必须保持刷新。……行在充电时是无法访问的。[highperfdram]的研究发现，“令人吃惊，DRAM 刷新对性能有着巨大的影响”。

根据 JEDEC 规范，DRAM 单元必须保持每 64ms 刷新一次。对于 8192 行的 DRAM，这意味着内存控制器平均每 7.8125μs 就需要发出一个刷新 命令(在实际情况下，由于刷新命令可以纳入队列，因此这个时间间隔可以更大一些)。刷新命令的调度由内存控制器负责。DRAM 模块会记录上一次刷新行的地 址，然后在下次刷新请求时自动对这个地址进行递增。

对于刷新及发出刷新命令的时间点，程序员无法施加影响。但我们在解读性能参数时有必要知道，它也是 DRAM 生命周期的一个部分。如果系统需要读取某个重要的字，而刚好它所在的行正在刷新，那么处理器将会被延迟很长一段时间。刷新的具体耗时取决于 DRAM 模块本身。

## 2.2.4 内存类型

我们有必要花一些时间来了解一下目前流行的内存，以及那些即将流行的内存。首先从SDR(单倍速)SDRAM 开始，因为它们是 DDR(双倍速)SDRAM 的基础。SDR 非常简单，内存单元和数据传输率是相等的。

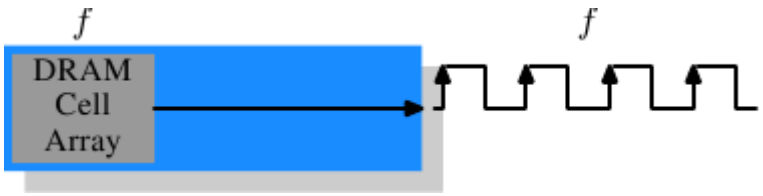
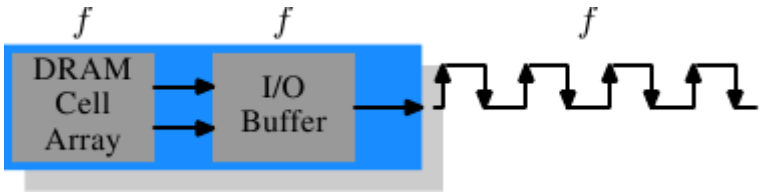



图 2.10: SDR SDRAM 的操作

在图 2.10 中，DRAM 单元阵列能以等同于内存总线的速率输出内容。假设 DRAM 单元阵列工作在 100MHz 上，那么总线的数据传输率可以达到 100Mb/s。所有组件的频率 $f$ 保持相同。由于提高频率会导致耗电量增加，所以提高吞吐量需要付出很高的的代价。如果是很大规模的内存阵列，代价会非常巨大。{*功率 = 动态电容 x 电压*$^2$ *x 频率*}。而且，提高频率还需要在保持系统稳定的情况下提高电压,这更是一个问题。因此,就有了 DDR SDRAM(现在叫 DDR1)，它可以在不提高频率的前提下提高吞吐量。




我们从图 2.11 上可以看出 DDR1 与 SDR 的不同之处，也可以从 DDR1 的名字里猜到那么几分，DDR1 的每个周期可以传输两倍的数据，它的上升沿和下 降沿都传输数据。有时又被称为“双泵(double-pumped)”总线。为了在不提升频率的前提下实现双倍传输，DDR 引入了一个缓冲区。缓冲区的每 条数据线都持有两位。它要求内存单元阵列的数据总线包含两条线。实现的方式很简单，用同一个列地址同时访问两个 DRAM 单元。对单元阵列的修改也很小。

SDR DRAM 是以频率来命名的(例如，对应于 100MHz 的称为 PC100)。为了让 DDR1 听上去更好听，营销人员们不得不想了一种新的命名方案。这种新方案中含有 DDR 模块可支持的传输速率(DDR 拥有 64 位总线):

100MHz x 64 位 x 2 = 1600MB/s

于是，100MHz 频率的 DDR 模块就被称为 PC1600。由于 1600 > 100，营销方面的需求得到了满足，听起来非常棒，但实际上仅仅只是提升了两倍而已。{我接受两倍这个事实，但不喜欢类似的数字膨胀戏法。}

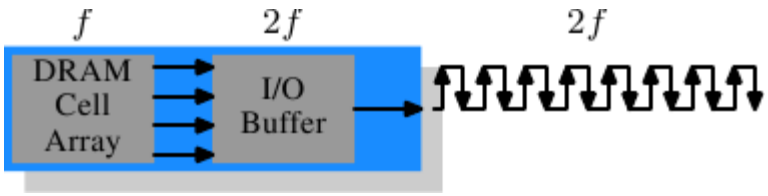



图 2.12：DDR2 SDRAM 的操作

为了更进一步，DDR2 有了更多的创新。在图 2.12 中，最明显的变化是，总线的频率加倍了。频率的加倍意味着带宽的加倍。如果对单元阵列的频率加倍，显 然是不经济的，因此 DDR2 要求 I/O 缓冲区在每个时钟周期读取 4 位。也就是说，DDR2 的变化仅在于使 I/O 缓冲区运行在更高的速度上。这是可行的，而 且耗电也不会显著增加。DDR2 的命名与 DDR1 相仿，只是将因子 2 替换成 4(四泵总线)。图 2.13 显示了目前常用的一些模块的名称。

| 阵列频率 | 总线频率 | 数据率 | 名称(速率) | 名称(FSB) |
|---|---|---|---|---|
| 133MHz | 266MHz | 4,256MB/s | PC2-4200 | DDR2-533 |
| 166MHz | 333MHz | 5,312MB/s | PC2-5300 | DDR2-667 |
| 200MHz | 400MHz | 6,400MB/s | PC2-6400 | DDR2-800 |
| 250MHz | 500MHz | 8,000MB/s | PC2-8000 | DDR2-1000 |
| 266MHz | 533MHz | 8,512MB/s | PC2-8500 | DDR2-1066 |

图 2.13：DDR2 模块名

在命名方面还有一个拧巴的地方。FSB 速度是用有效频率来标记的，即把上升、下降沿均传输数据的因素考虑进去，因此数字被撑大了。所以，拥有 266MHz 总线的 133MHz 模块有着 533MHz 的 FSB“频率”。

DDR3 要求更多的改变(这里指真正的 DDR3，而不是图形卡中假冒的 GDDR3)。电压从 1.8V 下降到 1.5V。由于耗电是与电压的平方成正比，因此 可以节约 30%的电力。加上管芯(die)的缩小和电气方面的其它进展，DDR3 可以在保持相同频率的情况下，降低一半的电力消耗。或者，在保持相同耗电 的情况下，达到更高的频率。又或者，在保持相同热量排放的情况下，实现容量的翻番。

DDR3 模块的单元阵列将运行在内部总线的四分之一速度上，DDR3 的 I/O 缓冲区从 DDR2 的 4 位提升到 8 位。见图 2.14。

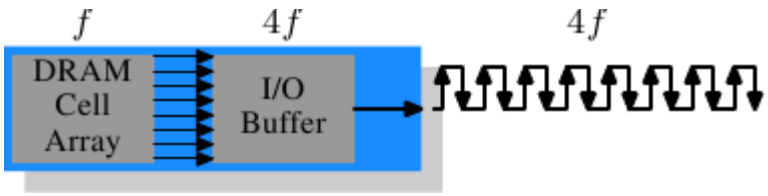



图 2.14：DDR3 SDRAM 的操作

一开始，DDR3 可能会有较高的 CAS 时延，因为 DDR2 的技术相比之下更为成熟。由于这个原因，DDR3 可能只会用于 DDR2 无法达到的高频率下，而且 带宽比时延更重要的场景。此前，已经有讨论指出，1.3V 的 DDR3 可以达到与 DDR2 相同的 CAS 时延。无论如何，更高速度带来的价值都会超过时延增加 带来的影响。

DDR3 可能会有一个问题，即在 1600Mb/s 或更高速率下，每个通道的模块数可能会限制为 1。在早期版本中，这一要求是针对所有频率的。我们希望这个要求可以提高一些，否则系统容量将会受到严重的限制。

图 2.15 显示了我们预计中各 DDR3 模块的名称。JEDEC 目前同意了前四种。由于 Intel 的 45nm 处理器是 1600Mb/s 的 FSB，1866Mb/s 可以用于超频市场。随着 DDR3 的发展，可能会有更多类型加入。

| 阵列频率 | 总线频率 | 数据速率 | 名称(速率) | 名称 (FSB) |
|---|---|---|---|---|
| 100MHz | 400MHz | 6,400MB/s | PC3-6400 | DDR3-800 |
| 133MHz | 533MHz | 8,512MB/s | PC3-8500 | DDR3-1066 |
| 166MHz | 667MHz | 10,667MB/s | PC3-10667 | DDR3-1333 |
| 200MHz | 800MHz | 12,800MB/s | PC3-12800 | DDR3-1600 |
| 233MHz | 933MHz | 14,933MB/s | PC3-14900 | DDR3-1866 |

**图 2.15：DDR3 模块名**

所有的 DDR 内存都有一个问题：不断增加的频率使得建立并行数据总线变得十分困难。一个 DDR2 模块有 240 根引脚。所有到地址和数据引脚的连线必须被布 置得差不多一样长。更大的问题是，如果多于一个 DDR 模块通过菊花链连接在同一个总线上，每个模块所接收到的信号随着模块的增加会变得越来越扭曲。 DDR2 规范允许每条总线（又称通道）连接最多两个模块，DDR3 在高频率下只允许每个通道连接一个模块。每条总线多达 240 根引脚使得单个北桥无法以合 理的方式驱动两个通道。替代方案是增加外部内存控制器（如图 2.2），但这会提高成本。

这意味着商品主板所搭载的 DDR2 或 DDR3 模块数将被限制在最多四条，这严重限制了系统的最大内存容量。即使是老旧的 32 位 IA-32 处理器也可以使用 64GB 内存。即使是家庭对内存的需求也在不断增长，所以，某些事必须开始做了。

一种解法是，在处理器中加入内存控制器，我们在第 2 节中曾经介绍过。AMD 的 Opteron 系列和 Intel 的 CSI 技术就是采用这种方法。只要我们能把 处理器要求的内存连接到处理器上，这种解法就是有效的。如果不能，按照这种思路就会引入 NUMA 架构，当然同时也会引入它的缺点。而在有些情况下，我们需 要其它解法。

Intel 针对大型服务器方面的解法(至少在未来几年)，是被称为全缓冲 DRAM(FB-DRAM)的技术。FB-DRAM 采用与 DDR2 相同的器件，因 此造价低廉。不同之处在于它们与内存控制器的连接方式。FB-DRAM 没有用并行总线，而用了串行总线(Rambus DRAM had this back when,

too, 而 SATA 是 PATA 的继任者，就像 PCI Express 是 PCI/AGP 的继承人一样)。串行总线可以达到更高的频率，串行化的负面影响，甚至可以增加带宽。使用串行总线后

1. 每个通道可以使用更多的模块。
2. 每个北桥/内存控制器可以使用更多的通道。
3. 串行总线是全双工的(两条线)。

FB-DRAM 只有 69 个脚。通过菊花链方式连接多个 FB-DRAM 也很简单。FB-DRAM 规范允许每个通道连接最多 8 个模块。

在对比下双通道北桥的连接性，采用 FB-DRAM 后，北桥可以驱动 6 个通道，而且脚数更少——6x69 对比 2x240。每个通道的布线也更为简单，有助于降低主板的成本。

全双工的并行总线过于昂贵。而换成串行线后，这不再是一个问题，因此串行总线按全双工来设计的，这也意味着，在某些情况下，仅靠这一特性，总线的理论带宽已经翻了一倍。还不止于此。由于 FB-DRAM 控制器可同时连接 6 个通道，因此可以利用它来增加某些小内存系统的带宽。对于一个双通道、4 模块的 DDR2 系统，我们可以用一个普通 FB-DRAM 控制器，用 4 通道来实现相同的容量。串行总线的实际带宽取决于在 FB-DRAM 模块中所使用的 DDR2(或 DDR3)芯片的类型。

我们可以像这样总结这些优势：

DDR2 FB-DRAM

| | DDR2 | FB-DRAM |
|---|---|---|
| 脚 | 240 | 69 |
| 通道 | 2 | 6 |
| 每通道 DIMM 数 | 2 | 8 |
| 最大内存 | 16GB | 192GB |
| 吞吐量 | ~10GB/s | ~40GB/s |

如果在单个通道上使用多个 DIMM，会有一些问题。信号在每个 DIMM 上都会有延迟(尽管很小)，也就是说，延迟是递增的。不过，如果在相同频率和相同容 量上进行比较，FB-DRAM 总是能快过 DDR2 及 DDR3，因为 FB-DRAM 只需要在每个通道上使用一个 DIMM 即可。而如果说到大型内存系统，那么 DDR 更是没有商用组件的解决方案。

## 2.2.5 结论

通过本节，大家应该了解到访问 DRAM 的过程并不是一个快速的过程。至少与处理器的速度相比，或与处理器访问寄存器及缓存的速度相比，DRAM 的访问不算 快。大家还需要记住 CPU 和内存的频率是不同的。Intel Core 2 处理器运行在 2.933GHz，而 1.066GHz FSB 有 11:1 的时钟比率(注: 1.066GHz 的总线为四泵总线)。那么，内存总线上延迟一个周期意味着处理器延迟 11 个周期。绝大多数机器使用的 DRAM 更慢，因此延迟更大。在后续 的章节中，我们需要讨论延迟这个问题时，请把以上的数字记在心里。

前文中读命令的时序图表明，DRAM 模块可以支持高速数据传输。每个完整行可以被毫无延迟地传输。数据总线可以 100%被占。对 DDR 而言，意味着每个周期传输 2 个 64 位字。对于 DDR2-800 模块和双通道而言，意味着 12.8GB/s 的速率。

但是，除非是特殊设计，DRAM 的访问并不总是串行的。访问不连续的内存区意味着需要预充电和 RAS 信号。于是，各种速度开始慢下来，DRAM 模块急需帮助。预充电的时间越短，数据传输所受的惩罚越小。

硬件和软件的预取(参见第 6.3 节)可以在时序中制造更多的重叠区，降低延迟。预取还可以转移内存操作的时间，从而减少争用。我们常常遇到的问题是，在这一轮中生成的数据需要被存储，而下一轮的数据需要被读出来。通过转移读取的时间，读和写就不需要同时发出了。

# 2.3 主存的其它用户

除了 CPU 外，系统中还有其它一些组件也可以访问主存。高性能网卡或大规模存储控制器是无法承受通过 CPU 来传输数据的，它们一般直接对内存进行读写(直 接内存访问，DMA)。在图 2.1 中可以看到，它们可以通过南桥和北桥直接访问内存。另外，其它总线，比如 USB 等也需要 FSB 带宽，即使它们并不使用 DMA，但南桥仍要通过 FSB 连接到北桥。

DMA 当然有很大的优点，但也意味着 FSB 带宽会有更多的竞争。在有大量 DMA 流量的情况下，CPU 在访问内存时必然会有更大的延迟。我们可以用一些硬件 来解决这个问题。例如，通过图 2.3 中的架构，我们可以挑选不受 DMA 影响的节点，让它们的内存为我们的计算服务。还可以在每个节点上连接一个南桥，将 FSB 的负荷均匀地分担到每个节点上。除此以外，还有许多其它方法。我们将在第 6 节中介绍一些技术和编程接口，它们能够帮助我们通过软件的方式改善这个问题。

最后，还需要提一下某些廉价系统，它们的图形系统没有专用的显存，而是采用主存的一部分作为显存。由于对显存的访问非常频繁(例如，对于 1024x768、16bpp、60Hz 的显示设置来说，需要 95MB/s 的数据速率)，而主存并不像显卡上的显存，并没有两个端口，因此这种配置会对系 统性能、尤其是时延造成一定的影响。如果大家对系统性能要求比较高，最好不要采用这种配置。这种系统带来的问题超过了本身的价值。人们在购买它们时已经做 好了性能不佳的心理准备。

继续阅读：

# 每个程序员都应该了解的 CPU 高速缓存【第二部分】

# 3. 高速缓存

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。

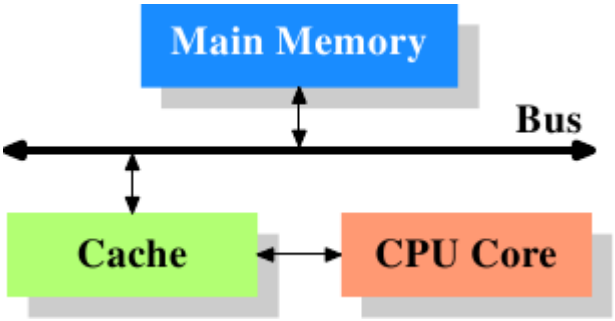



**图 3.1: 最简单的高速缓存配置图**

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。

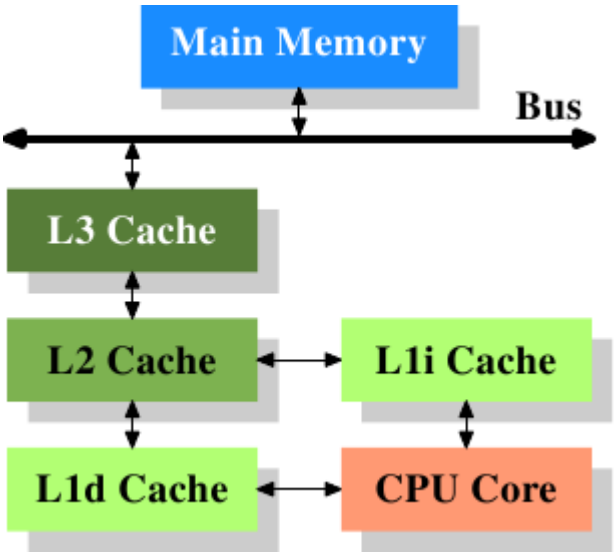



图 3.2: 三级缓存的处理器

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。

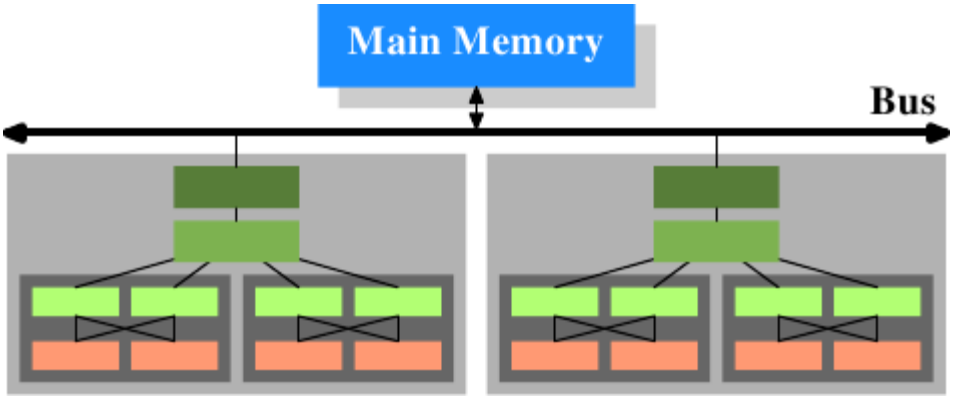



图 3.3 多处理器、多核心、多线程

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

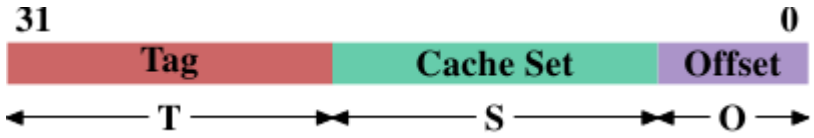
了解成本和节约使用缓存，我们必须结合在第二节中讲到的关于计算机体系结构和 RAM 技术，以及前一节讲到的缓存描述来探讨。

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。

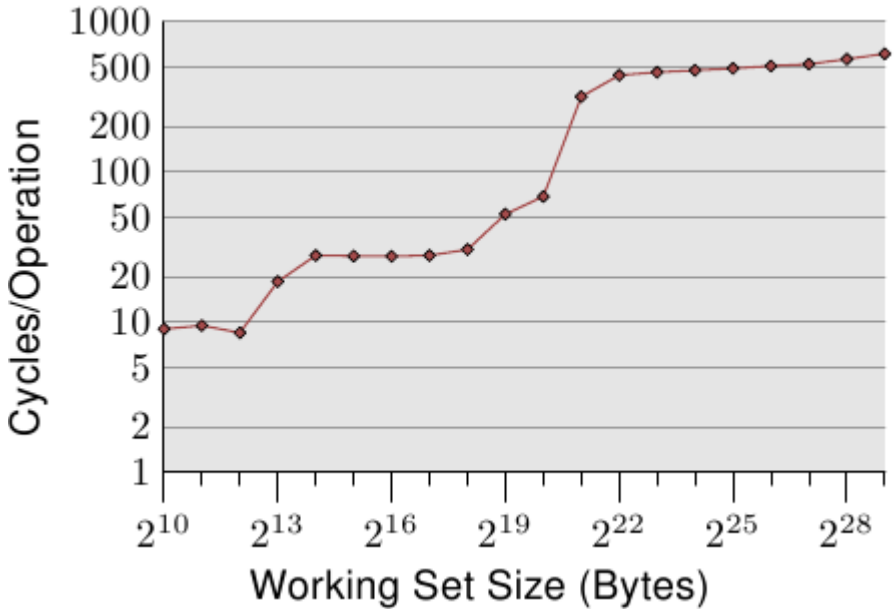



**图 3.4：随机写操作的访问时间**

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。

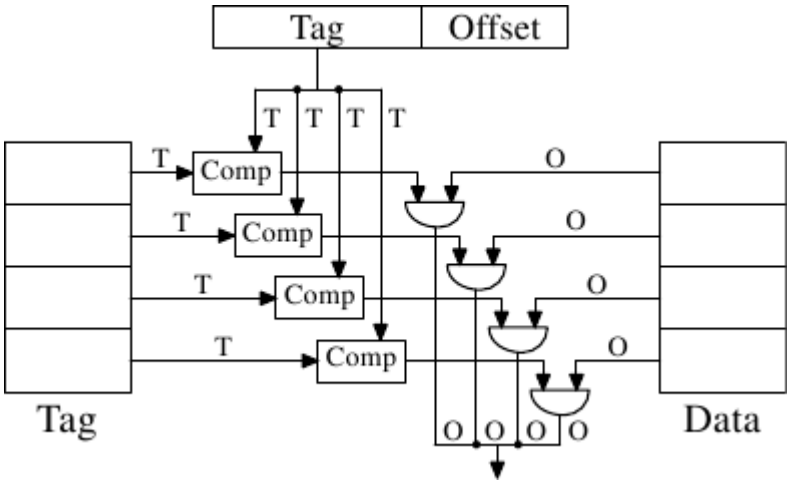



Figure 3.5: 全关联高速缓存原理图

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。

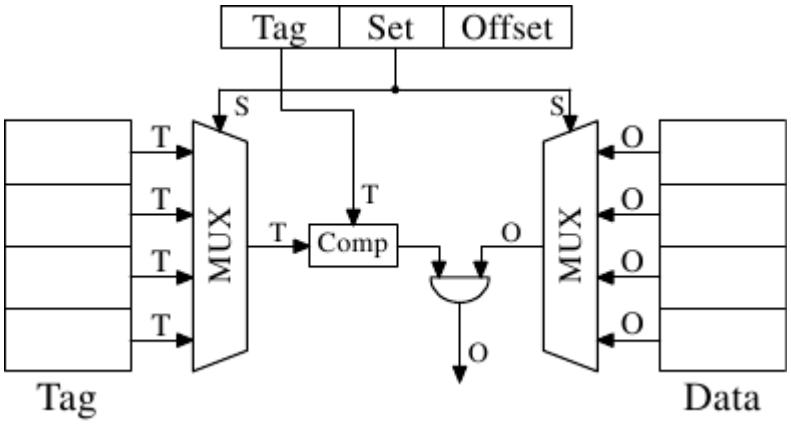



Figure 3.6: Direct-Mapped Cache Schematics

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。

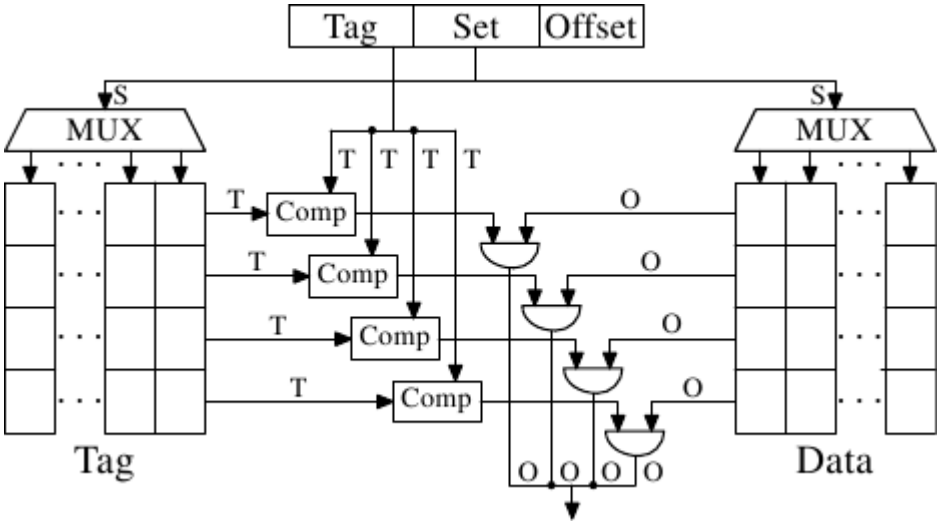



Figure 3.7: 组关联高速缓存原理图

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：

$$\text{cache line size} \times \text{associativity} \times \text{number of sets}$$

地址被映射到高速缓存使用

$$O = \log_2 \text{cache line size}$$
$$S = \log_2 \text{number of sets}$$

在第 3.2 节中的图显示的方式。

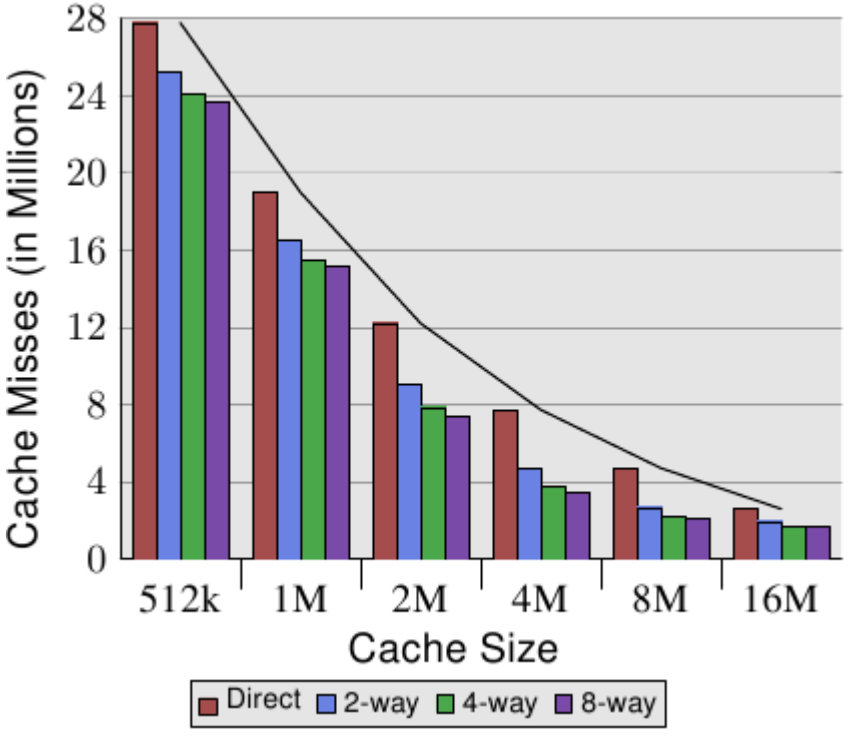


Figure 3.8: 缓存段大小 vs 关联行 (CL=32)

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。

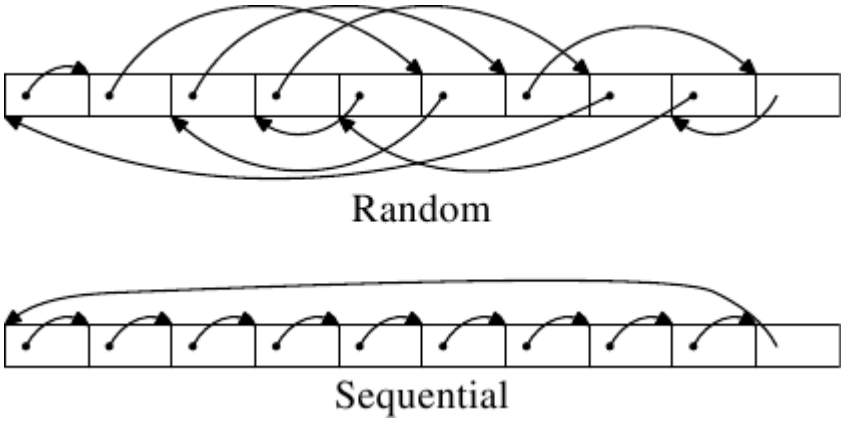



图 3.9: 测试的内存分布情况

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。

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)。

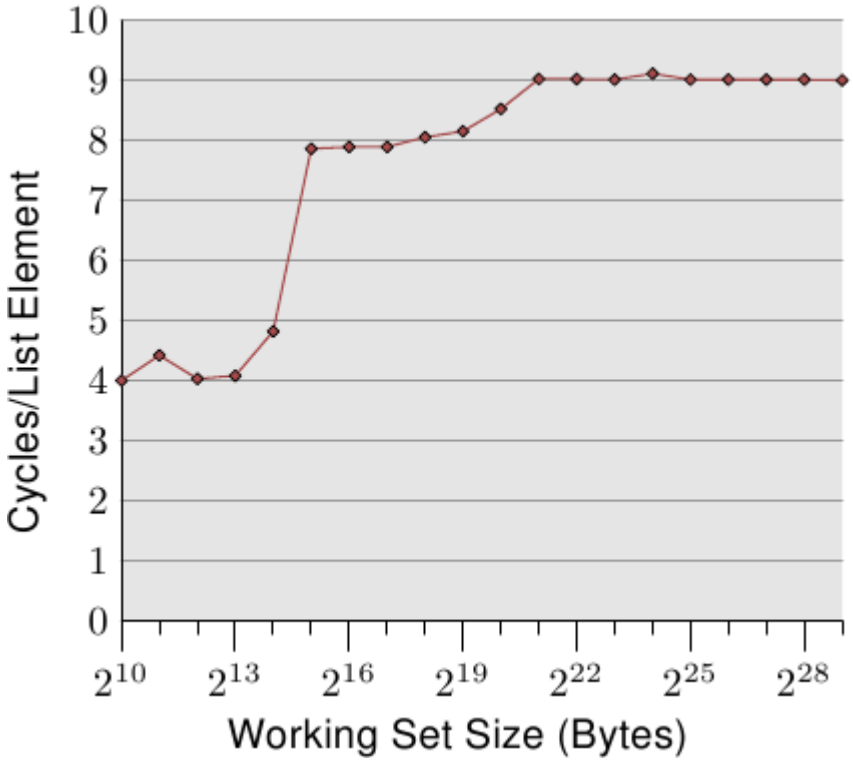



**图 3.10：顺序读访问，NPAD=0**

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。

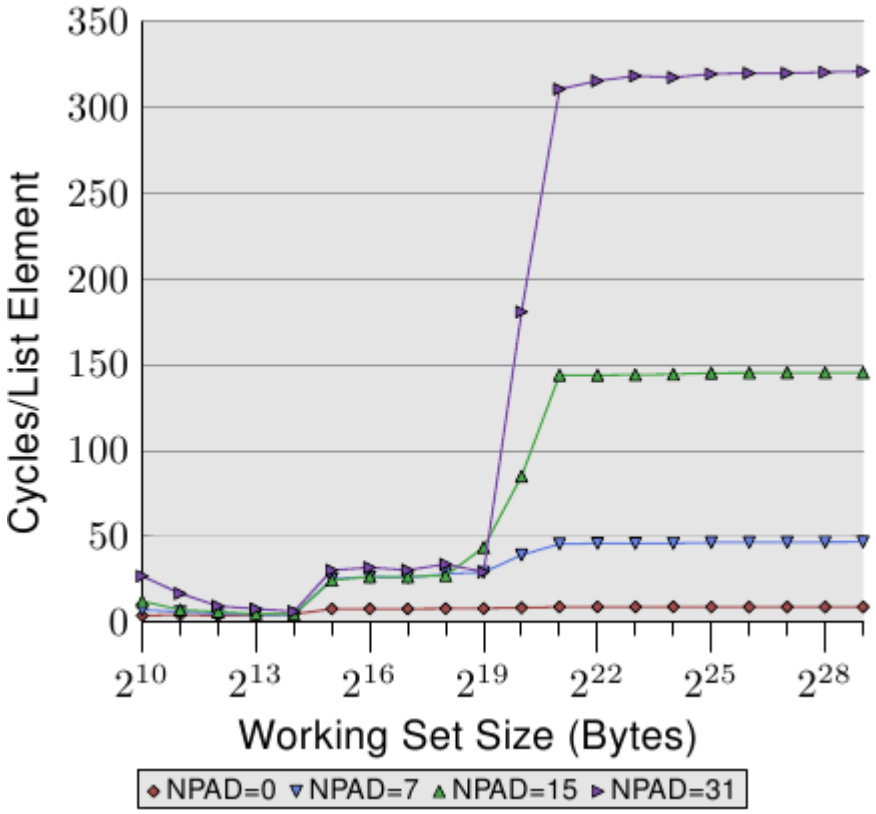



**图 3.11：顺序读多个字节**

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。

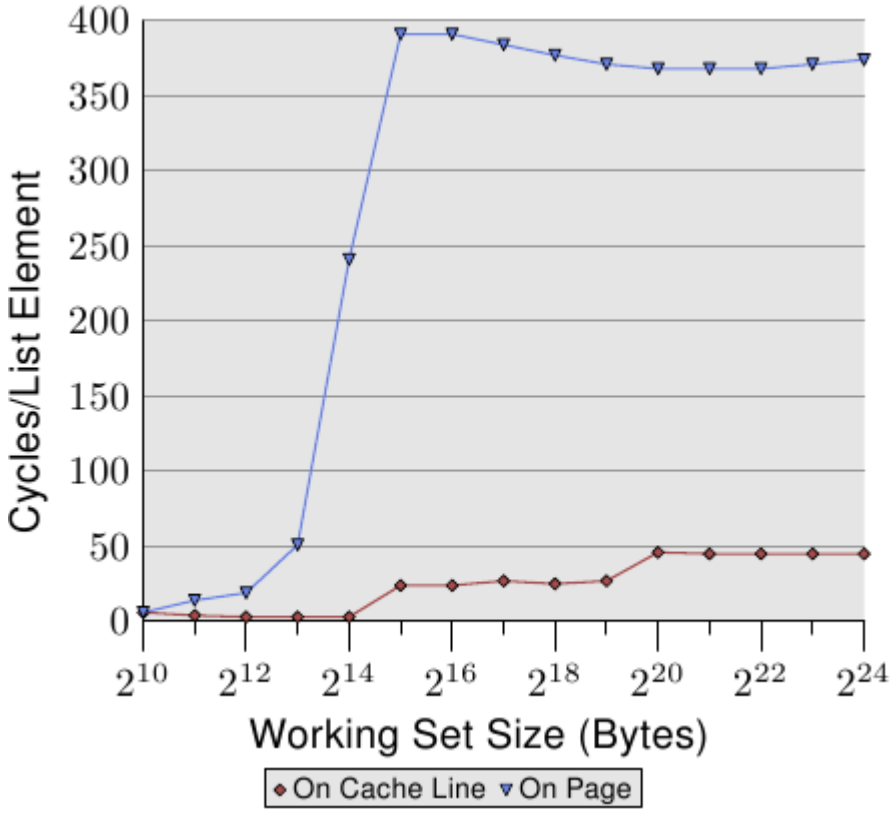



**图 3.12：TLB 对顺序读的影响**

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。

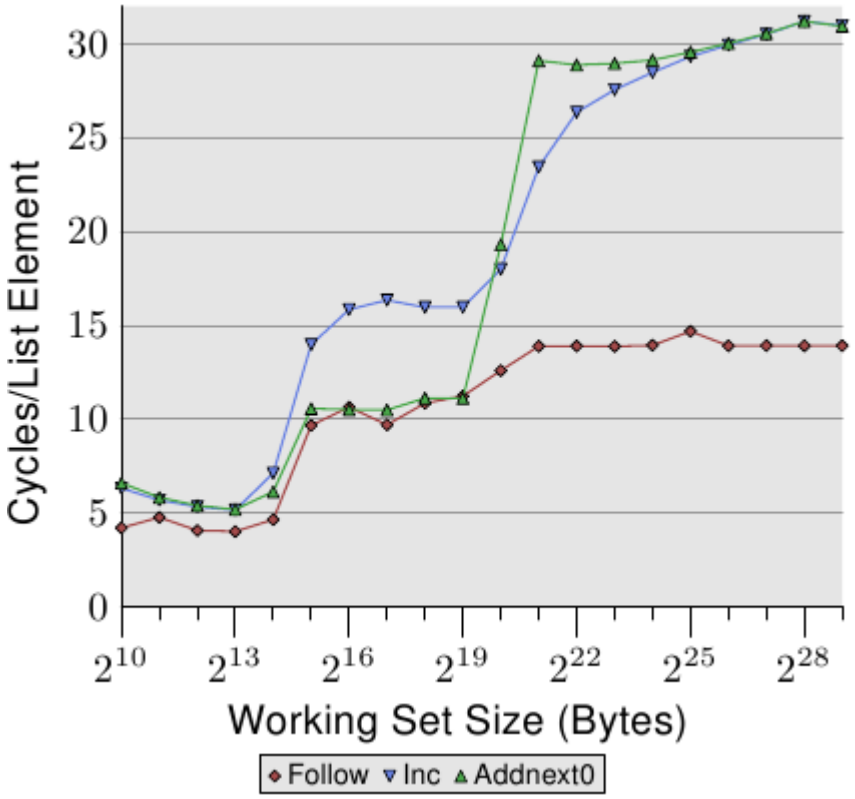



图 3.13 NPAD 等于 1 时的顺序读和写

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。

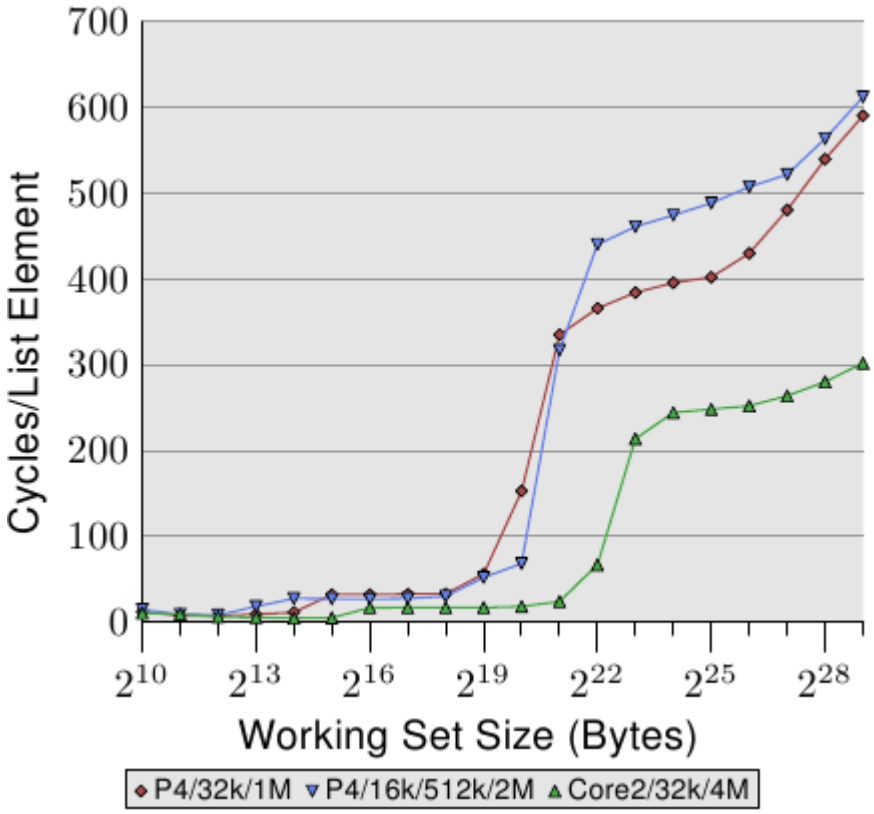



图 3.14：更大 L2/L3 缓存的优势

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。

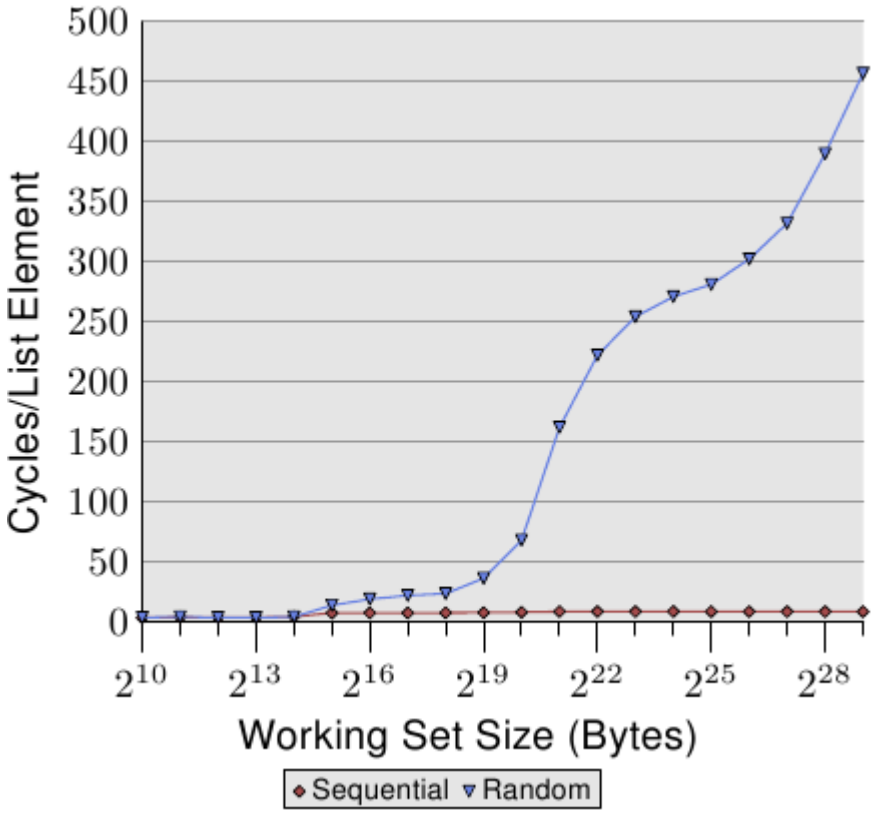



**图 3.15：顺序读取 vs 随机读取，NPAD=0**

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。

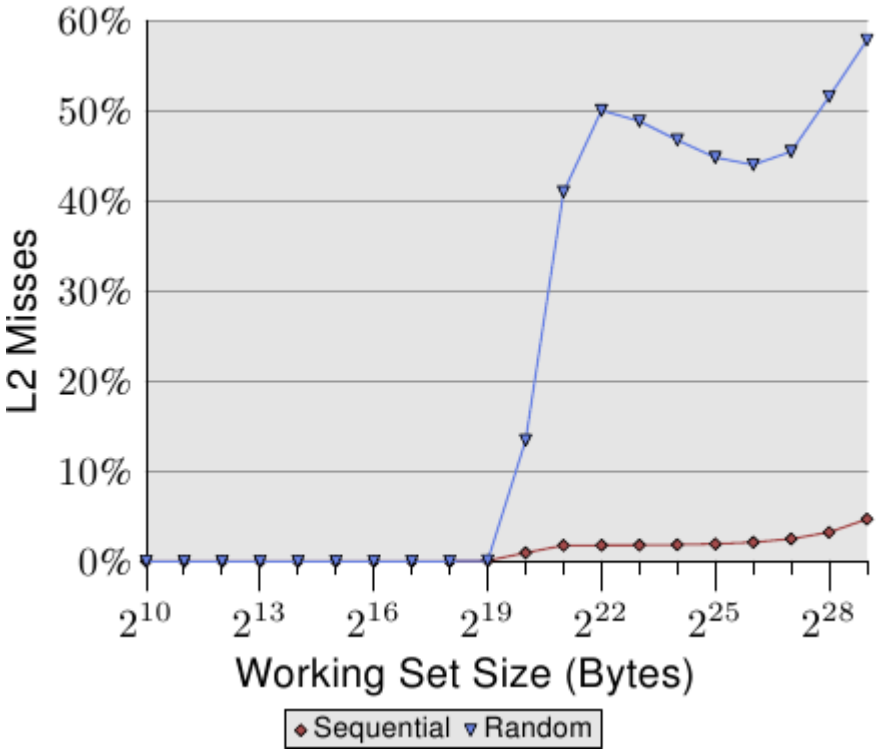



图 3.16: L2d 未命中率

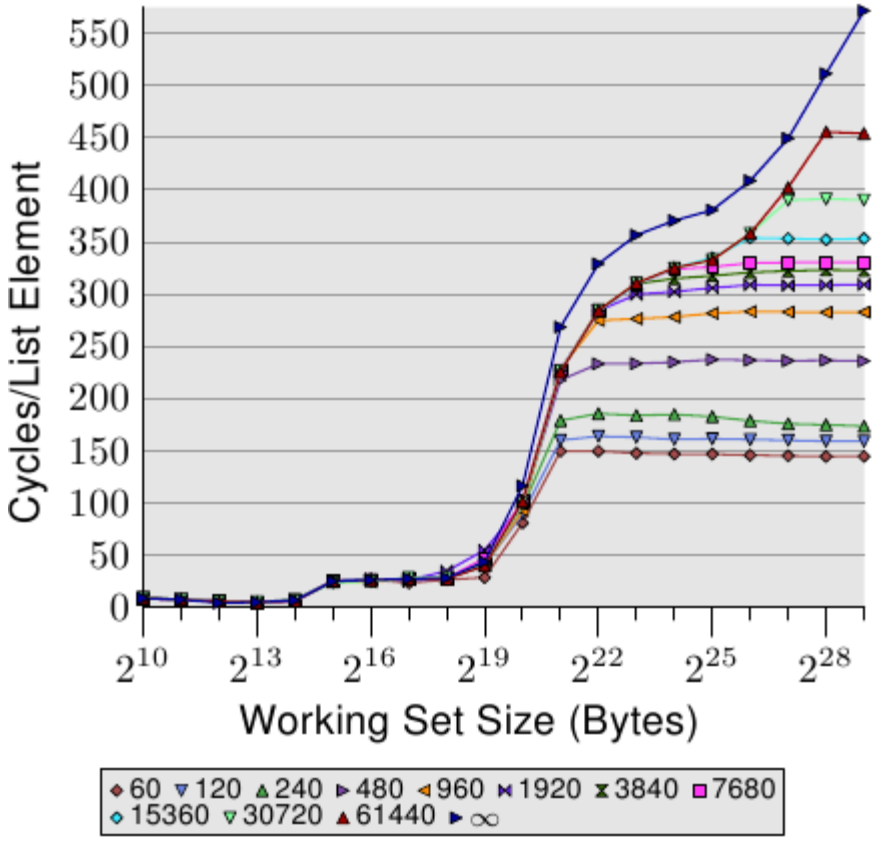



图 3.17: 页意义上(Page-Wise)的随机化，NPAD=7

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。

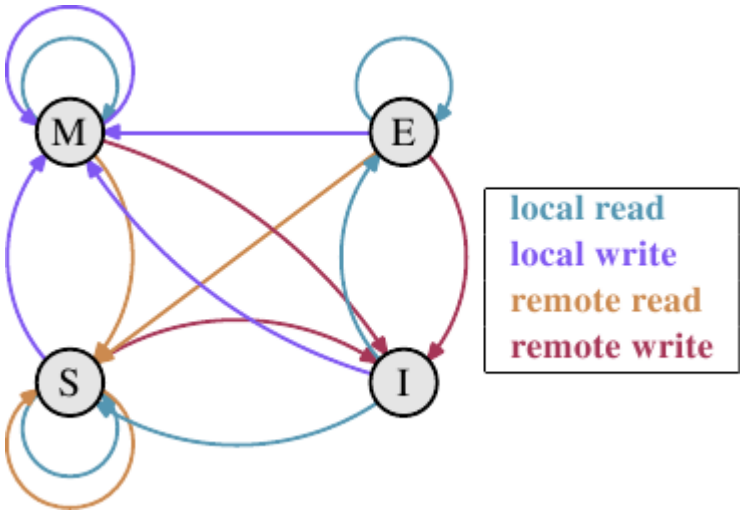



图 3.18：MESI 协议的状态跃迁图

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。

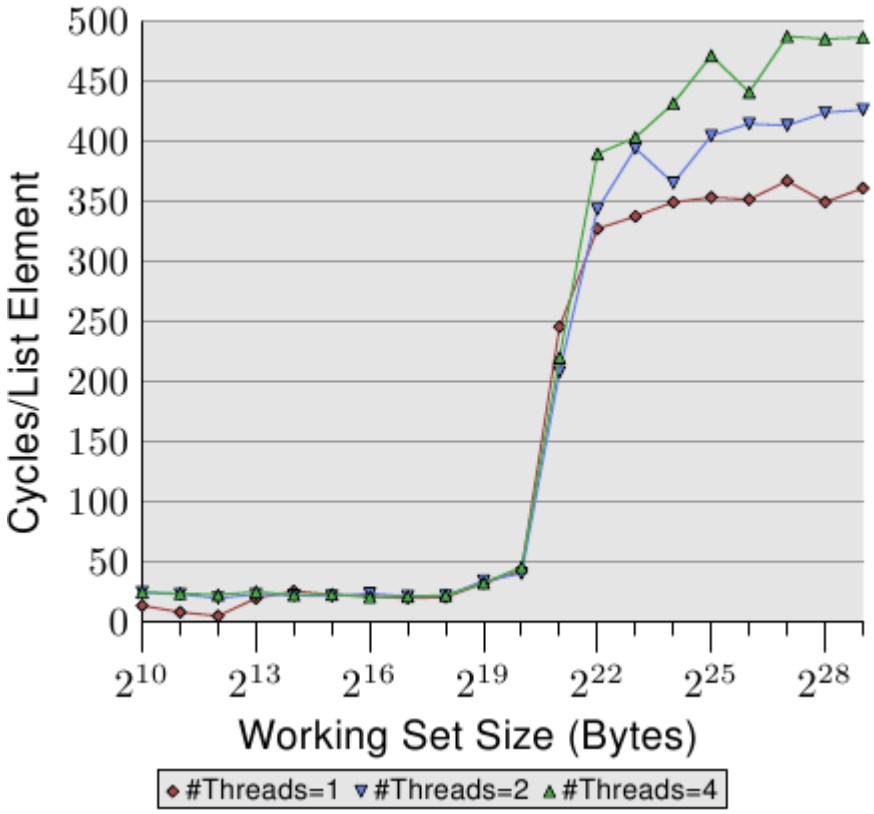



**图 3.19：顺序读操作，多线程**

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。

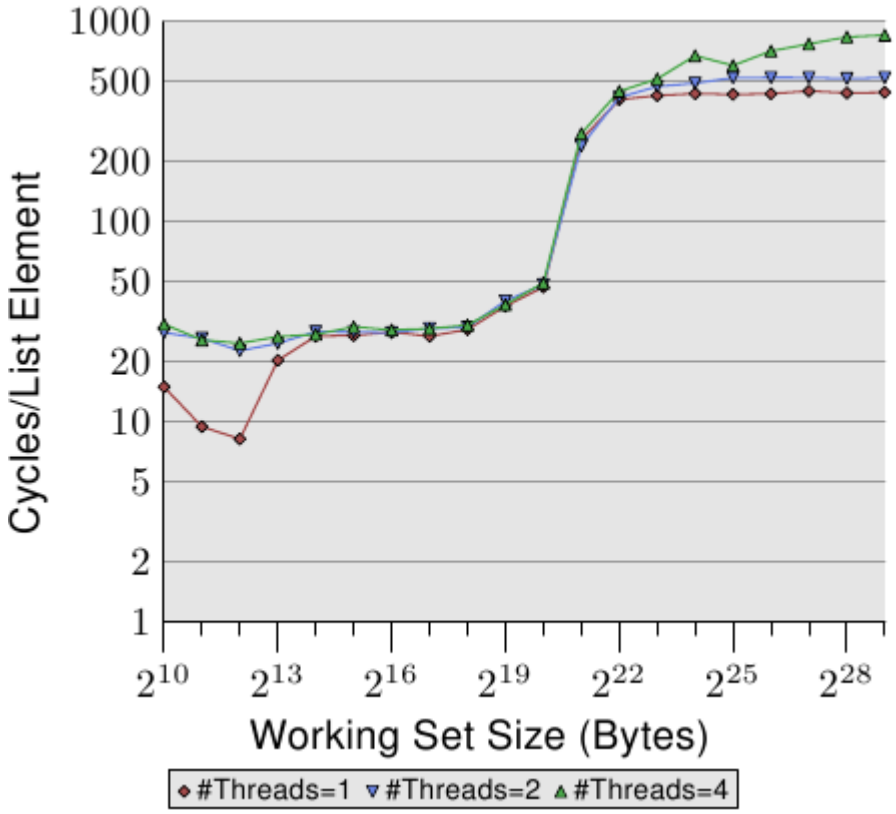



**图 3.20：顺序递增，多线程**

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。

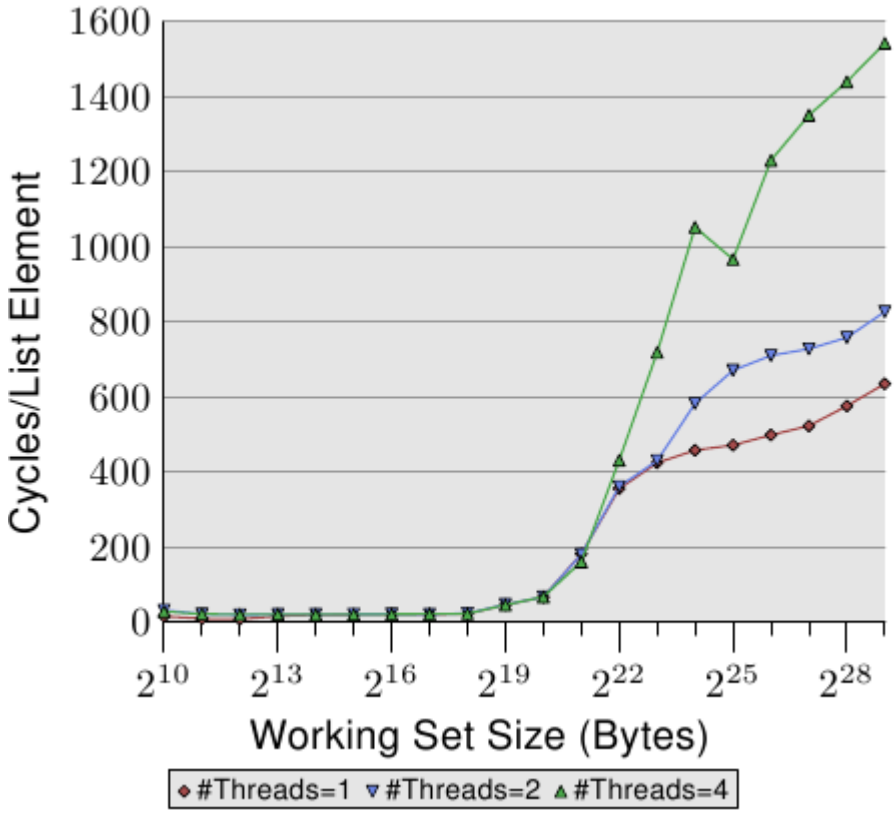



图 3.21：随机的 Addnextlast，多线程

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。

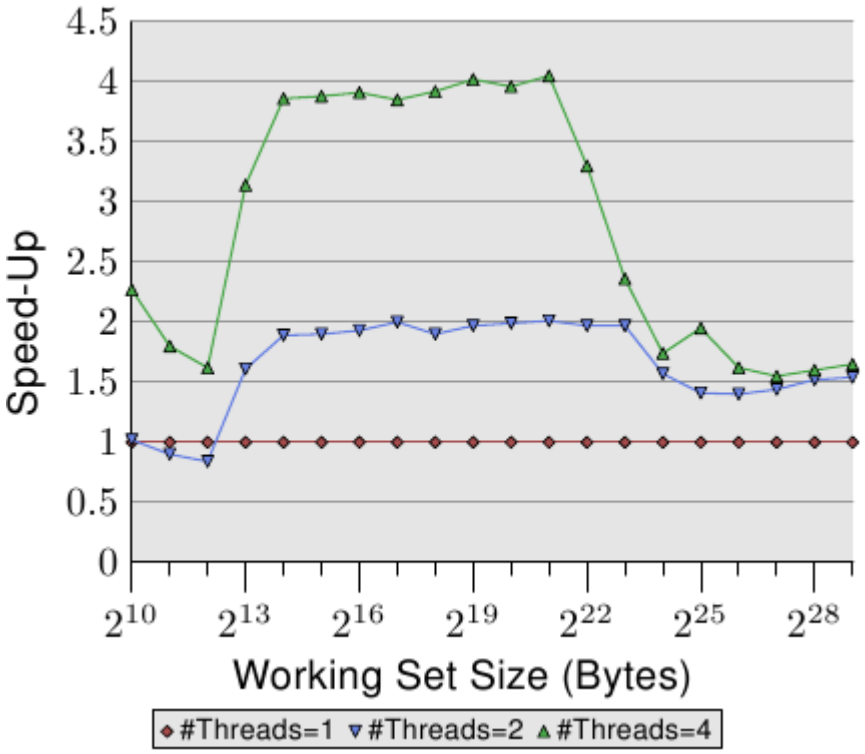



**图 3.22：通过并行化实现的加速因子**

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

Fmem = N 中访问内存的比例

Ghit = 命中缓存的比例

Tproc = 每条指令所用的周期数

Tcache = 缓存命中所用的周期数

Tmiss = 缓冲未命中所用的周期数

Texe = 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.

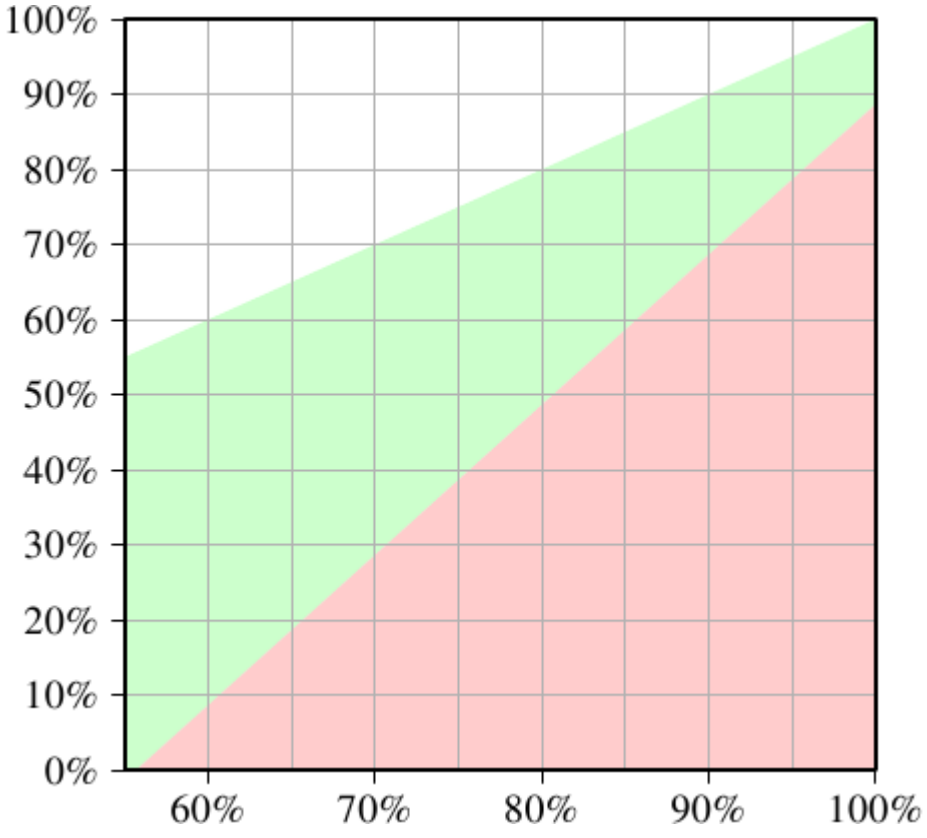


图 3.23：最小缓存命中率-加速

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。

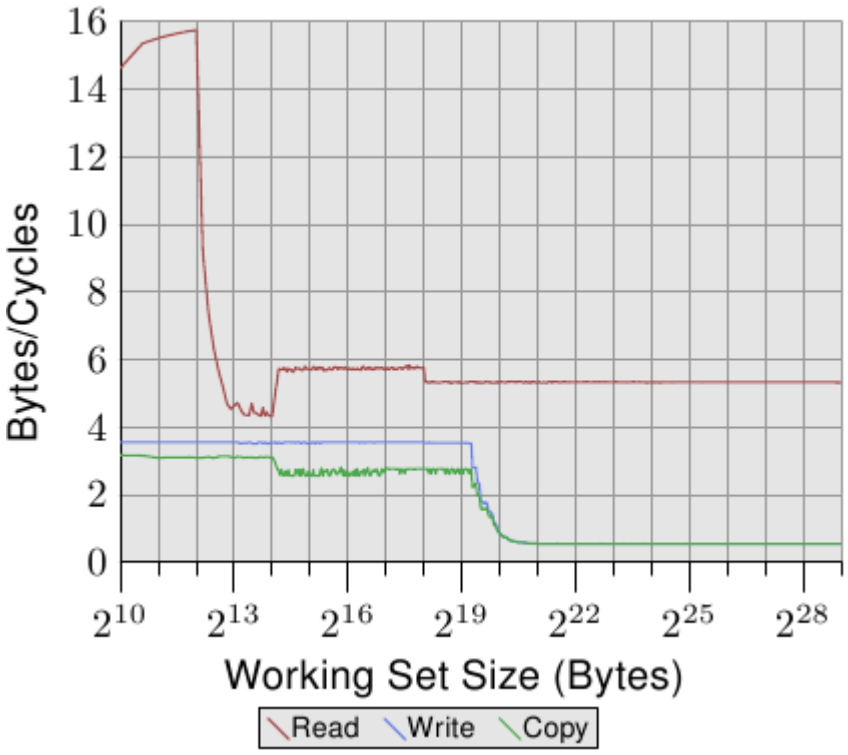



**图 3.24: P4 的带宽**

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。

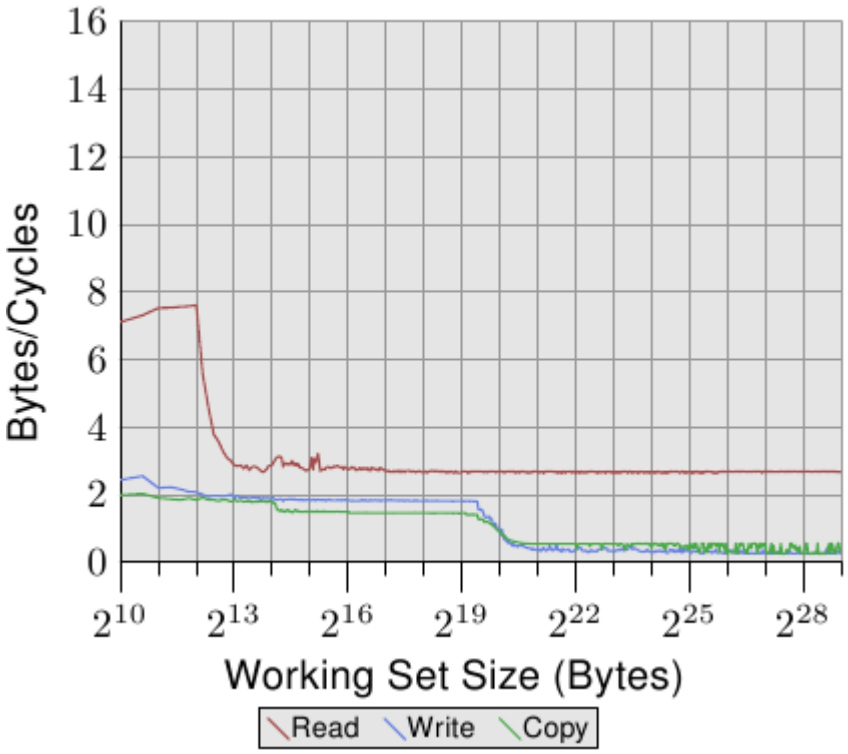



**图 3.25：P4 开启两个超线程时的带宽表现**

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。

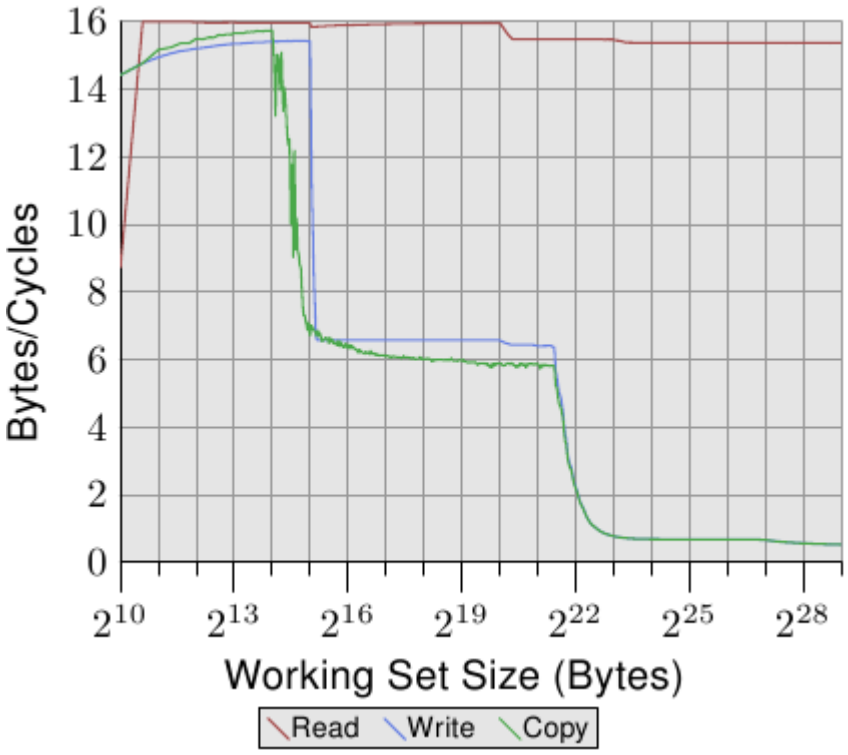



**图 3.26：Core 2 的带宽表现**

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。

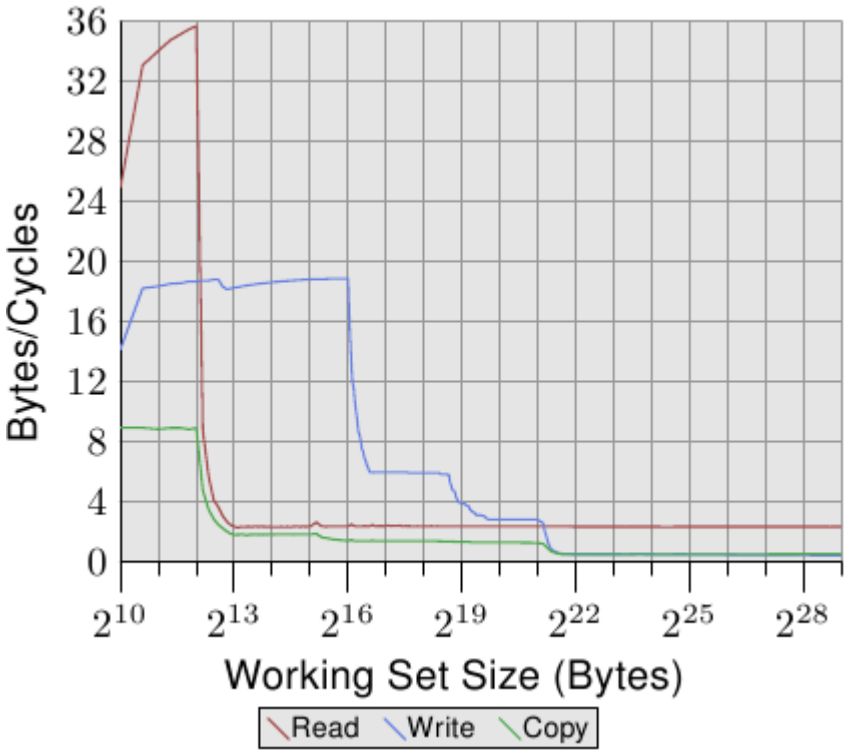



**图 3.28：AMD 家族 10h Opteron 的带宽表现**

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。

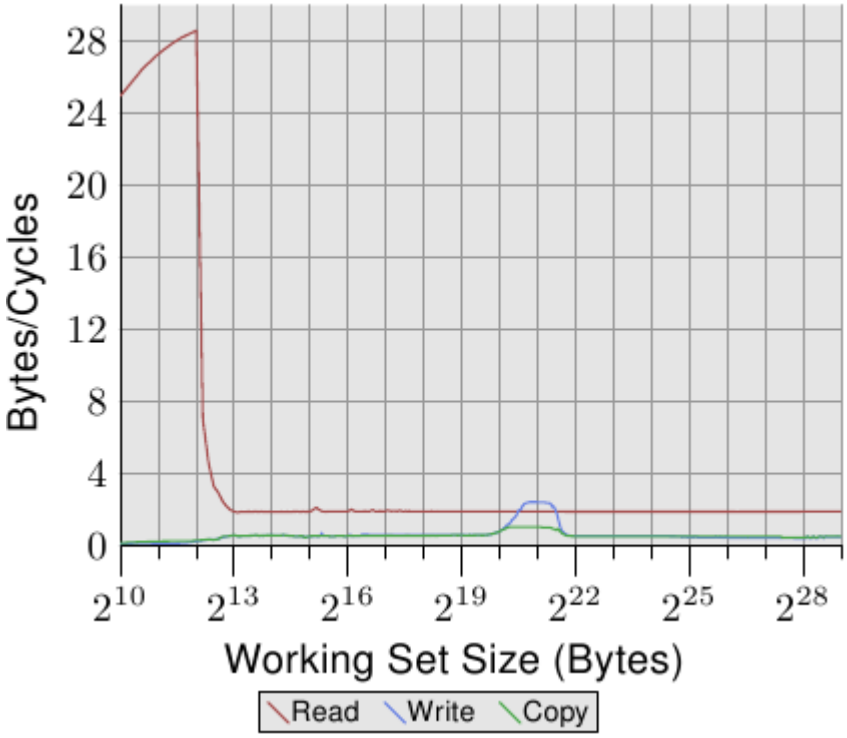



图 3.29: AMD Fam 10h 在双线程时的带宽表现

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。

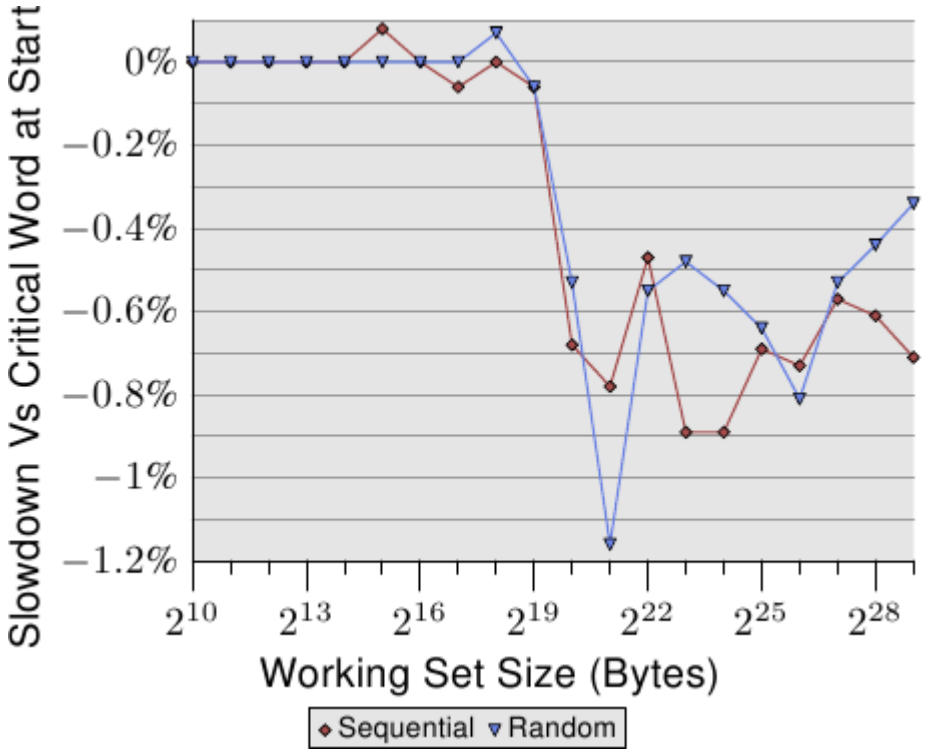



**图 3.30：关键字位于缓存线尾时的表现**

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。

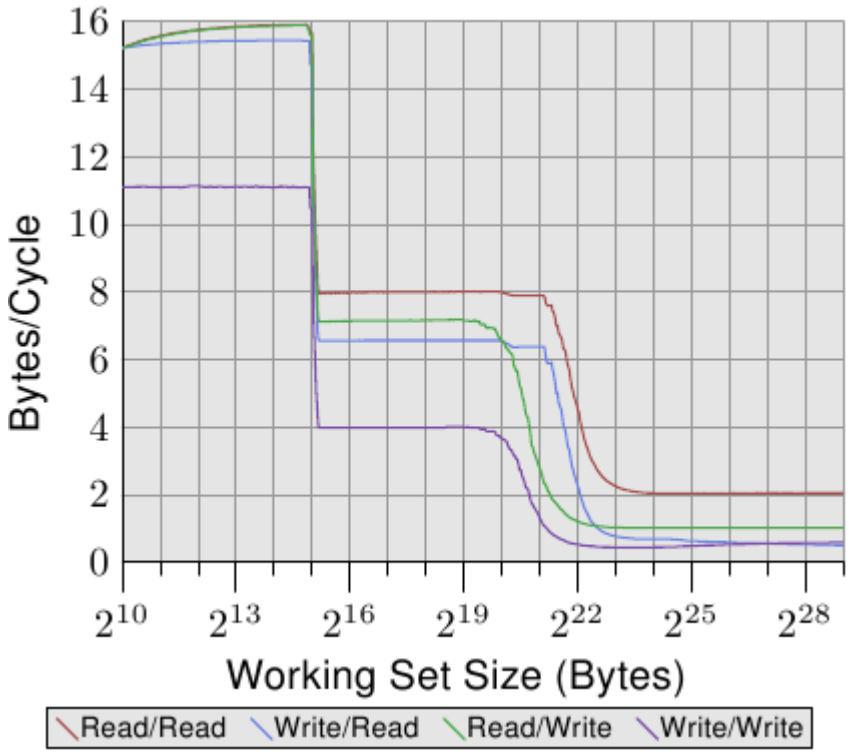



**图 3.31：两个进程的带宽表现**

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%)。

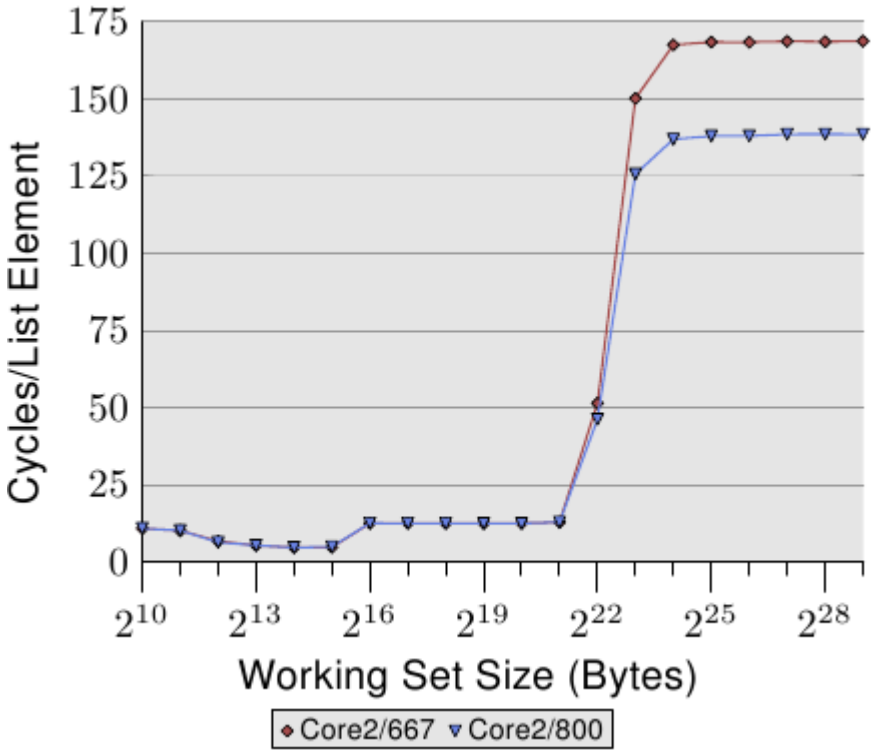



**图 3.32: FSB 速度的影响**

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。

# 每个程序员都应该了解的“虚拟内存”知识 【第三部分】

# 4 虚拟内存

处理器的虚拟内存子系统为每个进程实现了虚拟地址空间。这让每个进程认为它在系统中是独立的。虚拟内存的优点列表别的地方描述的非常详细，所以这里就不重复了。本节集中在虚拟内存的实际的实现细节，和相关的成本。

虚拟地址空间是由 CPU 的内存管理单元(MMU)实现的。OS 必须填充页表数据结构，但大多数 CPU 自己做了剩下的工作。这事实上是一个相当复杂的机制；最好的理解它的方法是引入数据结构来描述虚拟地址空间。

由 MMU 进行地址翻译的输入地址是虚拟地址。通常对它的值很少有限制 — 假设还有一点的话。 虚拟地址在 32 位系统中是 32 位的数值，在 64 位系统中是 64 位的数值。在一些系统，例如 x86 和 x86-64，使用的地址实际上包含了另一个层次的间接寻址：这些结构使用分段，这些分段只是简单的给每个逻辑地址加上位移。我们可以忽略这一部分的地址产生，它不重要,不是程序员非常关心的内存处理性能方面的东西。{x86 的分段限制是与性能相关的，但那是另一回事了}

# 4.1 最简单的地址转换

有趣的地方在于由虚拟地址到物理地址的转换。MMU 可以在逐页的基础上重新映射地址。就像地址缓存排列的时候，虚拟地址被分割为不同的部分。这些部分被用来做多个表的索引，而这些表是被用来创建最终物理地址用的。最简单的模型是只有一级表。

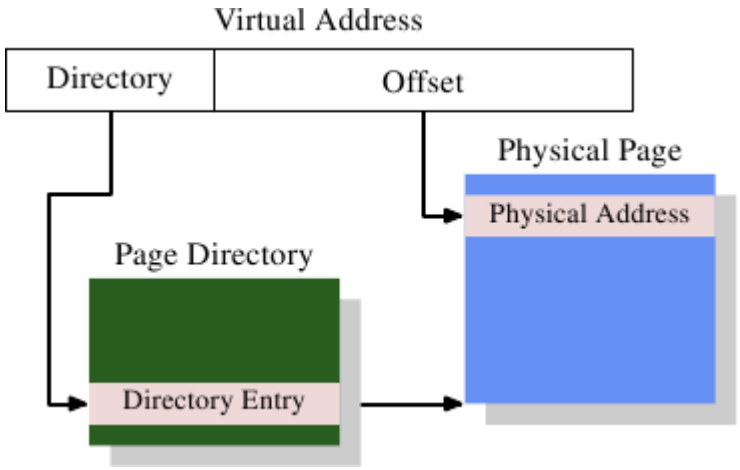



Figure 4.1: 1-Level Address Translation

图 4.1 显示了虚拟地址的不同部分是如何使用的。高字节部分是用来选择一个页目录的条目；那个目录中的每个地址可以被 OS 分别设置。页目录条目决定了物理内存页的 地址；页面中可以有不止一个条目指向同样的物理地址。完整的内存物理地址是由页目录获得的页地址和虚拟地址低字节部分合并起来决定的。页目录条目还包含一 些附加的页面信息，如访问权限。

页目录的数据结构存储在内存中。OS 必须分配连续的物理内存，并将这个地址范围的基地址存入一个特殊的寄存器。然后虚拟地址的适当的位被用来作为页目录的索引，这个页目录事实上是目录条目的列表。

作为一个具体的例子，这是 x86 机器 4MB 分页设计。虚拟地址的位移部分是 22 位大小，足以定位一个 4M 页内的每一个字节。虚拟地址中剩下的 10 位指定页目录中 1024 个条目的一个。每个条目包括一个 10 位的 4M 页内的基地址，它与位移结合起来形成了一个完整的 32 位地址。

# 4.2 多级页表

4MB 的页不是规范，它们会浪费很多内存，因为 OS 需要执行的许多操作需要内存页的队列。对于 4kB 的页（32 位机器的规范，甚至通常是 64 位机器的规范），虚拟地址的位移部分只有 12 位大小。这留下了 20 位作为页目录的指针。具有 $2^{20}$ 个条目的表是不实际的。即使每个条目只要 4 比特，这个表也要 4MB 大小。由于每个进程可能具有其唯一的页目录，因为这些页目录许多系统中物理内存被绑定起来。

解决办法是用多级页表。然后这些就能表示一个稀疏的大的页目录，目录中一些实际不用的区域不需要分配内存。因此这种表示更紧凑，使它可能为内存中的很多进程使用页表而并不太影响性能。

今天最复杂的页表结构由四级构成。图 4.2 显示了这样一个实现的原理图。

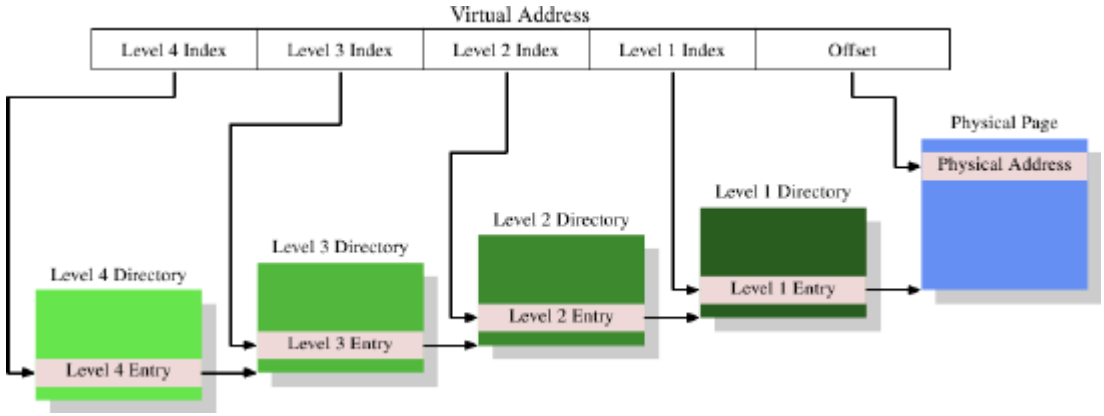



Figure 4.2: 4-Level Address Translation

在这个例子中，虚拟地址被至少分为五个部分。其中四个部分是不同的目录的索引。被引用的第 4 级目录使用 CPU 中一个特殊目的的寄存器。第 4 级到第 2 级目录 的内容是对次低一级目录的引用。如果一个目录条目标识为空，显然就是不需要指向任何低一级的目录。这样页表树就能稀疏和紧凑。正如图 4.1，第 1 级目录的 条目是一部分物理地址，加上像访问权限的辅助数据。

为了决定相对于虚拟地址的物理地址，处理器先决定最高级目录的地址。这个地址一般保存在一个寄存器。然后 CPU 取出虚拟地址中相对于这个目录的索引部分，并用那个索引选择合适的条目。这个条目是下一级目录的地址，它由虚拟地址的下一部分索引。处理器继续直到它到达第 1 级目录，那里那个目录条目的值就是物理地址的高字节部分。物理地址在加上虚拟地址中的页面位移之后就完整了。这个过程称为页面树遍历。一些处理器（像 x86 和 x86-64）在硬件中执行这个操作，其他的需要 OS 的协助。

系统中运行的每个进程可能需要自己的页表树。有部分共享树的可能，但是这相当例外。因此如果页表树需要的内存尽可能小的话将对性能与可扩展性有利。理想的情况是将使用的内存紧靠着放在虚拟地址空间；但实际使用的物理地址不影响。一个小程序可能只需要第 2，3，4 级的一个目录和少许第 1 级目录就能应付过去。在一个采用 4kB 页面和每个目录 512 条目的 x86-64 机器上，这允许用 4 级目录对 2MB 定位（每一级一个）。1GB 连续的内存可以被第 2 到第 4 级的一个目录和第 1 级的 512 个目录定位。

但是，假设所有内存可以被连续分配是太简单了。由于复杂的原因，大多数情况下，一个进程的栈与堆的区域是被分配在地址空间中非常相反的两端。这样使得任一个区域可以根据需要尽可能的增长。这意味着最有可能需要两个第 2 级目录和相应的更多的低一级的目录。

但即使这也不常常匹配现在的实际。由于安全的原因，一个可运行的（代码，数据，堆，栈，动态共享对象，aka 共享库）不同的部分被映射到随机的地址[未选中的]。随机化延伸到不同部分的相对位置；那意味着一个进程使用的不同的内存范围，遍布于虚拟地址空间。通过对随机的地址位数采用一些限定，范围可以被限制，但在大多数情况下，这当然不会让一个进程只用一到两个第 2 和第 3 级目录运行。

如果性能真的远比安全重要，随机化可以被关闭。OS 然后通常是在虚拟内存中至少连续的装载所有的动态共享对象(DSO)。

# 4.3 优化页表访问

页表的所有数据结构都保存在主存中；在那里 OS 建造和更新这些表。当一个进程创建或者一个页表变化，CPU 将被通知。页表被用来解决每个虚拟地址到物理地 址的转换，用上面描述的页表遍历方式。更多有关于此：至少每一级有一个目录被用于处理虚拟地址的过程。这需要至多四次内存访问（对一个运行中的进程的单次 访问来说），这很慢。有可能像普通数据一样处理这些目录表条目，并将他们缓存在 L1d,L2 等等，但这仍然非常慢。

从虚拟内存的早期阶段开始，CPU 的设计者采用了一种不同的优化。简单的计算显示，只有将目录表条目保存在 L1d 和更高级的缓存，才会导致可怕的性能问题。每个绝对地址的计算，都需要相对于页表深度的大量的 L1d 访问。这些访问不能并行，因为它们依赖于前面查询的结果。在一个四级页表的机器上，这种单线性将 至少至少需要 12 次循环。再加上 L1d 的非命中的可能性，结果是指令流水线没有什么能隐藏的。额外的 L1d 访问也消耗了珍贵的缓存带宽。

所以，替代于只是缓存目录表条目，物理页地址的完整的计算结果被缓存了。因为同样的原因，代码和数据缓存也工作起来，这样的地址计算结果的缓存是高效的。由于虚拟地址的页面位移部分在物理页地址的计算中不起任何作用，只有虚拟地址的剩余部分被用作缓存的标签。根据页面大小这意味着成百上千的指令或数据对象共享同一个标签，因此也共享同一个物理地址前缀。

保存计算数值的缓存叫做旁路转换缓存(TLB)。因为它必须非常的快，通常这是一个小的缓存。现代 CPU 像其它缓存一样，提供了多级 TLB 缓存；越高级的缓存越大越慢。小号的 L1 级 TLB 通常被用来做全相联映像缓存，采用 LRU 回收策略。最近这种缓存大小变大了，而且在处理器中变得集相联。其结果之一就是，当一个新的条目必须被添加的时候，可能不是最久的条目正如上面提到的，用来访问 TLB 的标签是虚拟地址的一个部分。如果标签在缓存中有匹配，最终的物理地址将被计算出来，通过将来自虚拟地址的页面位移地址加到缓存值的方式。这是一个非常快的过程；也必须这样，因为每条使用绝对地址的指令都需要物理地址，还有在一些情况下，因为使用物理地址作为关键字的 L2 查找。如果 TLB 查询未命中，处理器就必须执行一次页表遍历；这可能代价非常大。

通过软件或硬件预取代码或数据，会在地址位于另一页面时，暗中预取 TLB 的条目。硬件预取不可能允许这样，因为硬件会初始化非法的页面表遍历。因此程序员不能依赖硬件预取机制来预取 TLB 条目。它必须使用预取指令明确的完成。就像数据和指令缓存，TLB 可以表现为多个等级。正如数据缓存,TLB 通常表现为两种形式:指令 TLB(ITLB)和数据 TLB(DTLB)。高级的 TLB 像 L2TLB 通常是统一的，就像其他的缓存情形一样。被回收于替换了。

### 4.3.1 使用 TLB 的注意事项

TLB 是以处理器为核心的全局资源。所有运行于处理器的线程与进程使用同一个 TLB。由于虚拟到物理地址的转换依赖于安装的是哪一种页表树，如果页表变化 了，CPU 不能盲目的重复使用缓存的条目。每个进程有一个不同的页表树（不算在同一个进程中的线程），内核与内存管理器 VMM(管理程序)也一样，如果存 在的话。也有可能一个进程的地址空间布局发生变化。有两种解决这个问题的办法：

- 当页表树变化时 TLB 刷新。
- TLB 条目的标签附加扩展并唯一标识其涉及的页表树

第一种情况，只要执行一个上下文切换 TLB 就被刷新。因为大多数 OS 中，从一个线程/进程到另一个的切换需要执行一些核心代码，TLB 刷新被限制进入或离 开核心地址空间。在虚拟化的系统中，当内核必须调用内存管理器 VMM 和返回的时候，这也会发生。如果内核和/或内存管理器没有使用虚拟地址，或者当进程或 内核调用系统/内存管理器时，能重复使用同一个虚拟地址，TLB 必须被刷新。当离开内核或内存管理器时，处理器继续执行一个不同的进程或内核。

刷新 TLB 高效但昂贵。例如，当执行一个系统调用，触及的内核代码可能仅限于几千条指令，或许少许新页面（或一个大的页面，像某些结构的 Linux 的就是这样）。这个工作将替换触及页面的所有 TLB 条目。对 Intel 带 128ITLB 和 256DTLB 条目的 Core2 架构，完全的刷新意味着多于 100 和 200 条目（分别的）将被不必要的刷新。当系统调用返回同一个进程，所有那些被刷新的 TLB 条目可能被再次用到，但它们没有了。内核或内存管理器常用的代码也一样。每条进入内核的条目上，TLB 必须擦去再装，即使内核与内存管理器的页表通常不会改变。因此理论上说，TLB 条目可以被保持一个很长时间。这也解释了为什么现在处理器中的 TLB 缓存都不大：程序很有可能不会执行时间长到装满所有这些条目。

当然事实逃脱不了 CPU 的结构。对缓存刷新优化的一个可能的方法是单独的使 TLB 条目失效。例如，如果内核代码与数据落于一个特定的地址范围，只有落入这个地址范围的页面必须被清除出 TLB。这只需要比较标签，因此不是很昂贵。在部分地址空间改变的场合，例如对去除内存页的一次调用，这个方法也是有用的。

更好的解决方法是为 TLB 访问扩展标签。如果除了虚拟地址的一部分之外，一个唯一的对应每个页表树的标识（如一个进程的地址空间）被添加，TLB 将根本不需要完全刷新。内核，内存管理程序，和独立的进程都可以有唯一的标识。这种场景唯一的问题在于，TLB 标签可以获得的位数异常有限，但是地址空间的位数却不是。这意味着一些标识的再利用是有必要的。这种情况发生时 TLB 必须部分刷新（如果可能的话）。所有带有再利用标识的条目必须被刷新，但是希望这是一个非常小的集合。

当多个进程运行在系统中时，这种扩展的 TLB 标签具有一般优势。如果每个可运行进程对内存的使用（因此 TLB 条目的使用）做限制，进程最近使用的 TLB 条目,当其再次列入计划时，有很大机会仍然在 TLB。但还有两个额外的优势：

1. 特殊的地址空间，像内核和内存管理器使用的那些，经常仅仅进入一小段时间；之后控制经常返回初始化此次调用的地址空间。没有标签，就有两次 TLB 刷新操 作。有标签，调用地址空间缓存的转换地址将被保存，而且由于内核与内存管理器地址空间根本不会经常改变 TLB 条目，系统调用之前的地址转换等等可以仍然使 用。
2. 当同一个进程的两个线程之间切换时，TLB 刷新根本就不需要。虽然没有扩展 TLB 标签时，进入内核的条目会破坏第一个线程的 TLB 的条目。

有些处理器在一些时候实现了这些扩展标签。AMD 给帕西菲卡（Pacifica）虚拟化扩展引入了一个 1 位的扩展标签。在虚拟化的上下文中，这个 1 位的地址空间 ID（ASID）被用来从客户域区别出内存管理程序的地址空间。这使得 OS 能够避免在每次进入内存管理程序的时候（例如为了处理一个页面错误）刷新客户的 TLB 条目，或者当控制回到客户时刷新内存管理程序的 TLB 条目。这个架构未来会允许使用更多的位。其它主流处理器很可能会随之适应并支持这个功能。

### 4.3.2 影响 TLB 性能

有一些因素会影响 TLB 性能。第一个是页面的大小。显然页面越大，装进去的指令或数据对象就越多。所以较大的页面大小减少了所需的地址转换总次数，即需要更少的 TLB 缓存条目。大多数架构允许使用多个不同的页面尺寸；一些尺寸可以并存使用。例如，x86/x86-64 处理器有一个普通的 4kB 的页面尺寸，但它们也可以分别用 4MB 和 2MB 页面。IA-64 和 PowerPC 允许如 64kB 的尺寸作为基本的页面尺寸。

然而，大页面尺寸的使用也随之带来了一些问题。用作大页面的内存范围必须是在物理内存中连续的。如果物理内存管理的单元大小升至虚拟内存页面的大小，浪费的内存数量将会增长。各种内存操作（如加载可执行文件）需要页面边界对齐。这意味着平均每次映射浪费了物理内存中页面大小的一半。这种浪费很容易累加；因此它给物理内存分配的合理单元大小划定了一个上限。

在 x86-64 结构中增加单元大小到 2MB 来适应大页面当然是不实际的。这是一个太大的尺寸。但这转而意味着每个大页面必须由许多小一些的页面组成。这些小页面必须在物理内存中连续。以 4kB 单元页面大小分配 2MB 连续的物理内存具有挑战性。它需要找到有 512 个连续页面的空闲区域。在系统运行一段时间并且物理内存开始碎片化以后，这可能极为困难（或者不可能）

因此在 Linux 中有必要在系统启动的时候，用特别的 Huge TLBfs 文件系统，预分配这些大页面。一个固定数目的物理页面被保留，以单独用作大的虚拟页面。这使可能不会经常用到的资源捆绑留下来。它也是一个有限的池；增大它一般意味着要重启系统。尽管如此，大页面是进入某些局面的方法，在这些局面中性能具有保险性，资源丰富，而且麻烦的安装不会成为大的妨碍。数据库服务器就是一个例子。

增大最小的虚拟页面大小（正如选择大页面的相反面）也有它的问题。内存映射操作（例如加载应用）必须确认这些页面大小。不可能有更小的映射。对大多数架构 来说，一个可执行程序的各个部分位置有一个固定的关系。如果页面大小增加到超过了可执行程序或DSO(Dynamic Shared Object)创建时考虑的大小，加载操作将无法执行。脑海里记得这个限制很重要。图 4.3 显示了一个 ELF 二进制的对齐需求是如何决定的。它编码在 ELF 程序头部。

```
$ eu-readelf -l /bin/ls
Program Headers:
Type   Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
...
LOAD 0x000000 0x0000000000400000 0x0000000000400000 0x0132ac 0x0132ac R E 0x200000
LOAD 0x0132b0 0x00000000006132b0 0x00000000006132b0 0x001a71 0x001a71 RW  0x200000
...
```

**Figure 4.3: ELF 程序头表明了对齐需求**

在这个例子中，一个 x86-64 二进制，它的值为 0x200000 = 2,097,152 = 2MB，符合处理器支持的最大页面尺寸。

使用较大内存尺寸有第二个影响：页表树的级数减少了。由于虚拟地址相对于页面位移的部分增加了，需要用来在页目录中使用的位，就没有剩下许多了。这意味着当一个 TLB 未命中时，需要做的工作数量减少了。

超出使用大页面大小，它有可能减少移动数据时需要同时使用的 TLB 条目数目，减少到数页。这与一些上面我们谈论的缓存使用的优化机制类似。只有现在对齐需求是巨大的。考虑到 TLB 条目数目如此小，这可能是一个重要的优化。

# 4.4 虚拟化的影响

OS 映像的虚拟化将变得越来越流行；这意味着另一个层次的内存处理被加入了想象。进程（基本的隔间）或者 OS 容器的虚拟化，因为只涉及一个 OS 而没有落入 此分类。类似 Xen 或 KVM 的技术使 OS 映像能够独立运行 — 有或者没有处理器的协助。这些情形下，有一个单独的软件直接控制物理内存的访问。

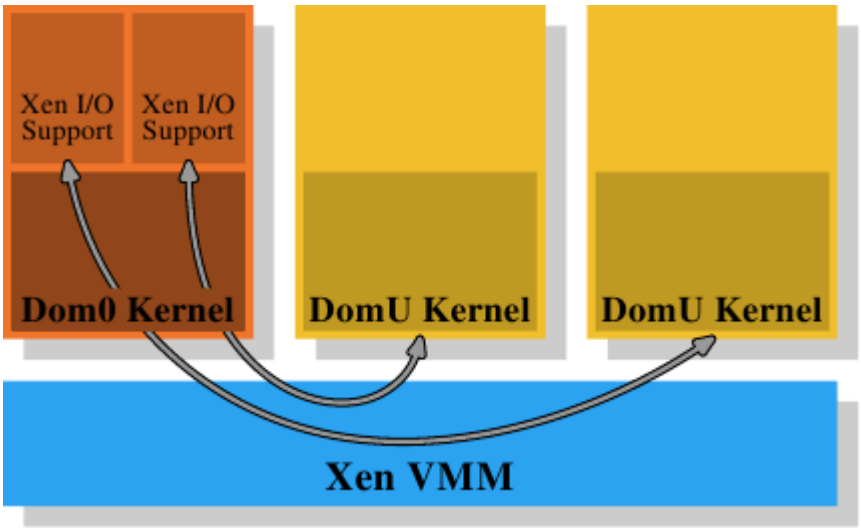



图 4.4: Xen 虚拟化模型

对 Xen 来说（见图 4.4），Xen VMM(Xen 内存管理程序)就是那个软件。但是，VMM 没有自己实现许多硬件的控制，不像其他早先的系统（包括 Xen VMM 的第一个版本）的 VMM，内存以外的硬件和处理器由享有特权的 Dom0 域控制。现在，这基本上与没有特权的 DomU 内核一样，就内存处理方面而言， 它们没有什么不同。这里重要的是，VMM 自己分发物理内存给 Dom0 和 DomU 内核，然后就像他们是直接运行在一个处理器上一样，实现通常的内存处理。

为了实现完成虚拟化所需的各个域之间的分隔，Dom0 和 DomU 内核中的内存处理不具有无限制的物理内存访问权限。VMM 不是通过分发独立的物理页并让客户 OS 处理地址的方式来分发内存；这不能提供对错误或欺诈客户域的防范。替代的，VMM 为每一个客户域创建它自己的页表树，并且用这些数据结构分发内存。好处是对页表树管理信息的访问能得到控制。如果代码没有合适的特权，它不能做任何事。

在虚拟化的 Xen 支持中，这种访问控制已被开发，不管使用的是参数的或硬件的（又名全）虚拟化。客户域以意图上与参数的和硬件的虚拟化极为相似的方法，给每个进程创建它们的页表树。每当客户 OS 修改了 VMM 调用的页表，VMM 就会用客户域中更新的信息去更新自己的影子页表。这些是实际由硬件使用的页表。显然这个过程非常昂贵：每次对页表树的修改都需要 VMM 的一次调用。而没有虚拟化时内存映射的改变也不便宜，它们现在变得甚至更昂贵。

考虑到从客户 OS 的变化到 VMM 以及返回，其本身已经相当昂贵，额外的代价可能真的很大。这就是为什么处理器开始具有避免创建影子页表的额外功能。这样很好不仅是因为速度的问题,而且它减少了 VMM 消耗的内存。Intel 有扩展页表(EPTs),AMD 称之为嵌套页表(NPTs)。基本上两种技术都具有客户 OS 的页表,来产生虚拟的物理地址。然后通过每个域一个 EPT/NPT 树的方式，这些地址会被进一步转换为真实的物理地址。这使得可以用几乎非虚拟化情境的速度进行内存处理，因为大多数用来内存处理的 VMM 条目被移走了。它也减少了 VMM 使用的内存，因为现在一个域（相对于进程）只有一个页表树需要维护。

额外的地址转换步骤的结果也存储于 TLB。那意味着 TLB 不存储虚拟物理地址，而替代以完整的查询结果。已经解释过 AMD 的帕西菲卡扩展为了避免 TLB 刷新而给每个条目引入 ASID。ASID 的位数在最初版本的处理器扩展中是一位；这正好足够区分 VMM 和客户 OS。Intel 有服

务同一个目的的虚拟处理器 ID(VPIDs)，它们只有更多位。但对每个客户域 VPID 是固定的，因此它不能标记单独的进程，也不能避免 TLB 在那个级别刷新。

对虚拟 OS，每个地址空间的修改需要的工作量是一个问题。但是还有另一个内在的基于 VMM 虚拟化的问题：没有什么办法处理两层的内存。但内存处理很难（特 别是考虑到像 NUMA 一样的复杂性，见第 5 部分）。Xen 方法使用一个单独的 VMM，这使最佳的（或最好的）处理变得困难，因为所有内存管理实现的复杂 性，包括像发现内存范围之类“琐碎的”事情，必须被复制于 VMM。OS 有完全成熟的与最佳的实现；人们确实想避免复制它们。

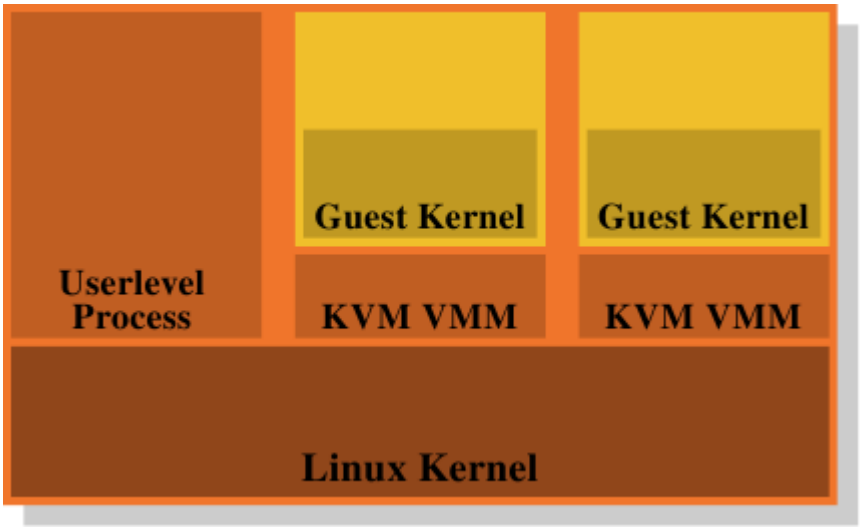



**图 4.5：KVM 虚拟化模型**

这就是为什么对 VMM/Dom0 模型的分析是这么有吸引力的一个选择。图 4.5 显示了 KVM 的 Linux 内核扩展如何尝试解决这个问题的。并没有直接运行在硬件之上且管理所有客户的单独的 VMM，替代的，一个普通的 Linux 内核接管了这个功能。这意味着 Linux 内核中完整且复杂的内存管理功能，被用来管理系统的内存。客户域运行于普通的用户级进程，创建者称其为“客户模式”。虚拟化的功能，参数的或全虚拟化的，被另一个用户级进程 KVM VMM 控制。这也就是另一个进程用特别的内核实现的 KVM 设备，去恰巧控制一个客户域。

这个模型相较 Xen 独立的 VMM 模型好处在于，即使客户 OS 使用时，仍然有两个内存处理程序在工作，只需要在 Linux 内核里有一个实现。不需要像 Xen VMM 那样从另一段代码复制同样的功能。这带来更少的工作，更少的 bug，或许还有更少的两个内存处理程序接触产生的摩擦，因为一个 Linux 客户的内存处理程序与运行于裸硬件之上的 Linux 内核的外部内存处理程序，做出了相同的假设。

总的来说，程序员必须清醒认识到，采用虚拟化时，内存操作的代价比没有虚拟化要高很多。任何减少这个工作的优化，将在虚拟化环境付出更多。随着时间的过去，处理器的设计者将通过像 EPT 和 NPT 技术越来越减少这个差距，但它永远都不会完全消失。