$$\text{写操作带来的停顿周期数} = \frac{\text{写操作数目}}{\text{程序}} \times \text{写失效率} \times \text{写失效代价} + \text{写缓冲满时的停顿周期}$$

由于写缓冲停顿主要依赖于写操作的密集度，而不只是它的频度，不可能给出一个简单的计算此类停顿的等式。幸运的是，如果系统中有一个容量合理的写缓冲（例如，四个或者更多字），同时主存接收写请求的速度能够大于程序的平均写速度，写缓冲引起的停顿将会很少，几乎能够忽略。如果系统不能满足这些要求，那么这个设计可能不合理。设计者要么使用更深的写缓冲，要么使用写返回策略。

写返回策略也会额外增加停顿，主要来源于当数据块被替换并需要将其写回到主存时。这部分将在 5.8 节讨论。

在大多数写穿透 cache 的结构中，读和写的失效代价是相同的（都是将数据块从内存取至 cache 所花的时间）。假设写缓冲停顿是可以忽略不计的，就可以使用失效率和失效代价来同时刻画读操作和写操作：

$$\text{等待存储访问的时钟周期数} = \frac{\text{访存操作数目}}{\text{程序}} \times \text{写失效率} \times \text{写失效代价}$$

该公式也可以记作

$$\text{等待存储访问的时钟周期数} = \frac{\text{指令数目}}{\text{程序}} \times \frac{\text{失效次数}}{\text{指令数目}} \times \text{写失效代价}$$

下面使用一个简单的例子来帮助大家理解 cache 性能对处理器性能的影响。

例题 | 计算 cache 的性能

假设指令 cache 的失效率为 2%，数据 cache 的失效率为 4%。如果处理器的 CPI 为 2，没有任何的访存停顿；对于所有的失效，失效代价都为 100 个时钟周期。如果配置了一个从不失效的完美 cache，那么处理器的性能会提高多少？假设 load 和 store 指令占所有指令的 36%。

答案 | 假设指令数目为 I，则指令访存失效的周期数为

$$\text{指令访存失效周期数} = I \times 2\% \times 100 = 2.00 \times I$$

由于 load 和 store 指令占总指令数的 36%，则数据访存失效的周期数为

$$\text{数据访存失效周期数} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

存储访问失效的周期数为 $2.00\,I + 1.44\,I = 3.44\,I$。这意味着平均每条指令等待存储访问的周期数大于 3。相应的，考虑了存储访问停顿的 CPI 为 $2 + 3.44 = 5.44$。由于指令数目或者时钟频率都不变，则 CPU 执行时间的比率为

$$\frac{\text{带有存储访问停顿的 CPU 执行时间}}{\text{具有完美 cache 的 CPU 执行时间}} = \frac{I \times \text{CPI}_{\text{stall}} \times \text{时钟周期}}{I \times \text{CPI}_{\text{perfect}} \times \text{时钟周期}} = \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2}$$

具有完美 cache 的 CPU 性能是原来的 5.44/2= 2.72 倍。

如果处理器运行得更快，但是主存系统却并不是这样，会发生什么呢？用于存储访问停顿的时间占整个执行时间的比例在不断增大。在第 1 章中讲过的 Amdahl 定律提醒我们这个事实。一些简单的例子说明这个问题的严重性。假设在前面的例子中，使用一个改进的流水线，将 CPI 从 2 变为 1，时钟频率不变，则处理器的性能在提升。那么考虑了 cache 失效的

系统将会让 CPI 变为 1 + 3.44 = 4.44。与配置完美 cache 的系统相比，两者的比率为

$$\frac{4.44}{1} = 4.44$$

那么用于存储访问的停顿周期占总周期数的比例将会从

$$\frac{3.44}{5.44} = 63\% \quad 变为 \quad \frac{3.44}{4.44} = 77\%$$

同理，如果不改变存储体系结构，仅提高时钟频率，由于 cache 失效带来的系统性能损失所占比例也会增大。

之前的例子和等式都假设命中时间不是影响 cache 性能的重要因素。很清楚，如果命中时间增加，在存储系统中访问一个字所花的时间也在增加。这可能引起处理器时钟周期的增大。本书提供一些其他实例以了解导致命中时间略微增加的原因，其中的一个例子就是增大 cache 的容量。更大的 cache 自然会有更长的访问时间，这就好像如果你在图书馆的桌子足够大（例如 3 平方米），那么在桌上找到一本所需的书会耗费更长的时间。命中时间的增加可能会为流水线增加一个流水级，这样即使 cache 命中也会花费多个时钟周期。计算加深流水线对性能的影响则更为复杂。在某些情况下，对于大容量 cache 来说，相比命中率的提升，命中时间的增加反而会更占优势，这将会导致处理器性能的下降。

为了说明不论命中还是失效访问数据存储的时间都会影响性能，设计者采用*平均存储访问时间*（Average Memory Access Time，AMAT）作为指标来衡量不同的 cache 设计。平均存储访问时间是考虑了命中、失效以及不同访问频度的影响后的平均访存时间，定义如下：

$$AMAT = 命中时间 + 失效率 \times 失效代价$$

例题 | 计算存储的平均访问时间

时钟周期为 1ns 的处理器，失效代价为 20 个时钟周期，失效率为 5%，cache 访问时间（包括命中判断）为 1 个时钟周期，计算该处理器的平均访存时间 AMAT。假设读写的失效代价相同，并忽略写操作引起的其他停顿。

每条指令的平均访存时间为

$$\begin{aligned} AMAT &= 命中时间 + 失效率 \times 失效代价 \\ &= 1 + 0.05 \times 20 \\ &= 2 \text{ 时钟周期（或 2ns）} \end{aligned}$$

之后的章节将会讨论另一种降低失效率的 cache 组织结构，但该结构有时可能会增加命中时间，具体示例见 5.16 节。

5.4.1　使用更为灵活的替换策略降低 cache 失效率

迄今为止，当将数据块写入 cache 时，采用的是一种最简单的定位策略：一个数据块在 cache 中只有一个对应位置。正如之前提到的，这种策略称为直接映射，因为它将主存中的任意数据块地址直接映射到上层存储的一个准确位置。但是，事实上还有很多存放数据块的策略。直接映射（数据块只能放在唯一的位置）只是其中的一个特例。

另一个特例是，数据块可以存放在 cache 的任意位置，这种策略称为**全相联**，即主存中的某个数据块和 cache 中的任意表项都可能有关联。在全相联 cache 中查找给定的数据块，所有的表项都必

全相联 cache：cache 的一种组织结构，数据块可以存放在 cache 的任意位置。

须进行比对，因为数据块可以存放在任意位置。为让比对过程更实际，每个 cache 表项都有一个比较器可以并行地进行比较。这些比较器显然增加了硬件开销，这使得全相连策略只能用于那些小容量的 cache。

介于直接映射和全相联之间的组织结构称为**组相联**。在一个组相联 cache 中，每一个数据块可以存放的位置数量是固定的。每个数据块有 n 个位置可放的组相联 cache 称为 n 路组相联 cache。在一个 n 路组相联 cache 中，包含有若干组（set），每一组包含有 n 个数据块。主存中的每个数据块通过索引位映射到 cache 中对应的组，数据块可以存放在该组中的任意位置。因此，组相联 cache 将直接映射和全相联结合起来：某个数据块直接映射到某一组，之后组中的所有数据块都需要与之进行命中比对。例如，图 5-14 中假设 cache 中共有 8 个数据块，根据以上三种组织结构，给出数据块 12 在 cache 中的位置。

组相联 cache：cache 的一种组织结构，每个数据块在 cache 中存放的位置数量具有固定值（至少为 2）。

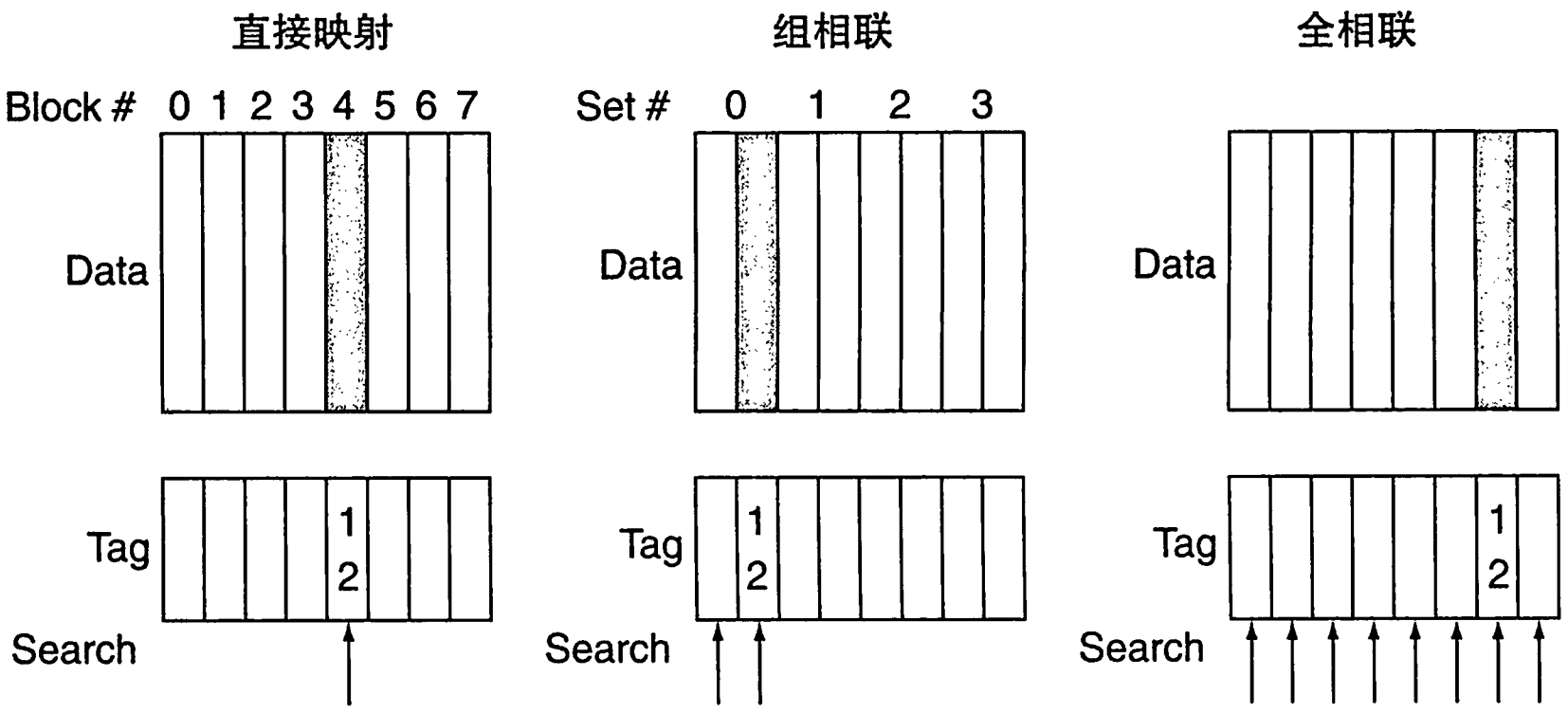

图 5-14　地址为 12 的主存数据块在 cache 中的位置，该 cache 具有 8 个数据块，分别采用直接映射、组相联和全相联策略。在直接映射 cache 中，主存块 12 只对应一个数据块位置，数据块号为（12 mod 8）= 4。在两路组相联 cache 中，共有 4 组，主存块 12 对应的组为（12 mod 4）= 0；主存块可以在这个组中的任意位置。在全相联 cache 中，主存块 12 可以放置在任意块中。

注意，在直接映射 cache 中，主存数据块的位置为

（数据块号）mod（cache 中的数据块数量）

在组相联 cache 中，包含主存块的组号为

（数据块号）mod（cache 中的组数）

由于数据块可以放置在该组内的任意位置，组内所有元素的所有标签都必须被检查。在全相联 cache 中，数据块可以放置在任意位置，cache 中所有数据块的所有标签都要被检查。

可以将所有的 cache 组织结构看作组相联结构的特例。图 5-15 给出 8 个数据块 cache 可能的相联结构。直接映射 cache 就是一路组相联 cache：每个表项放置一个数据块，每一组只有一个元素。而 m 个表项的全相联 cache 则是一个 m 路组相联 cache，只有一个组，组内有 m 个数据块，每个数据块可以驻留在这一组内的任意块内。

提高相联度的好处是通常可以降低失效率，正如下面的例子所示。主要的问题在于可能会增加命中时间，之后将进行详细讨论。

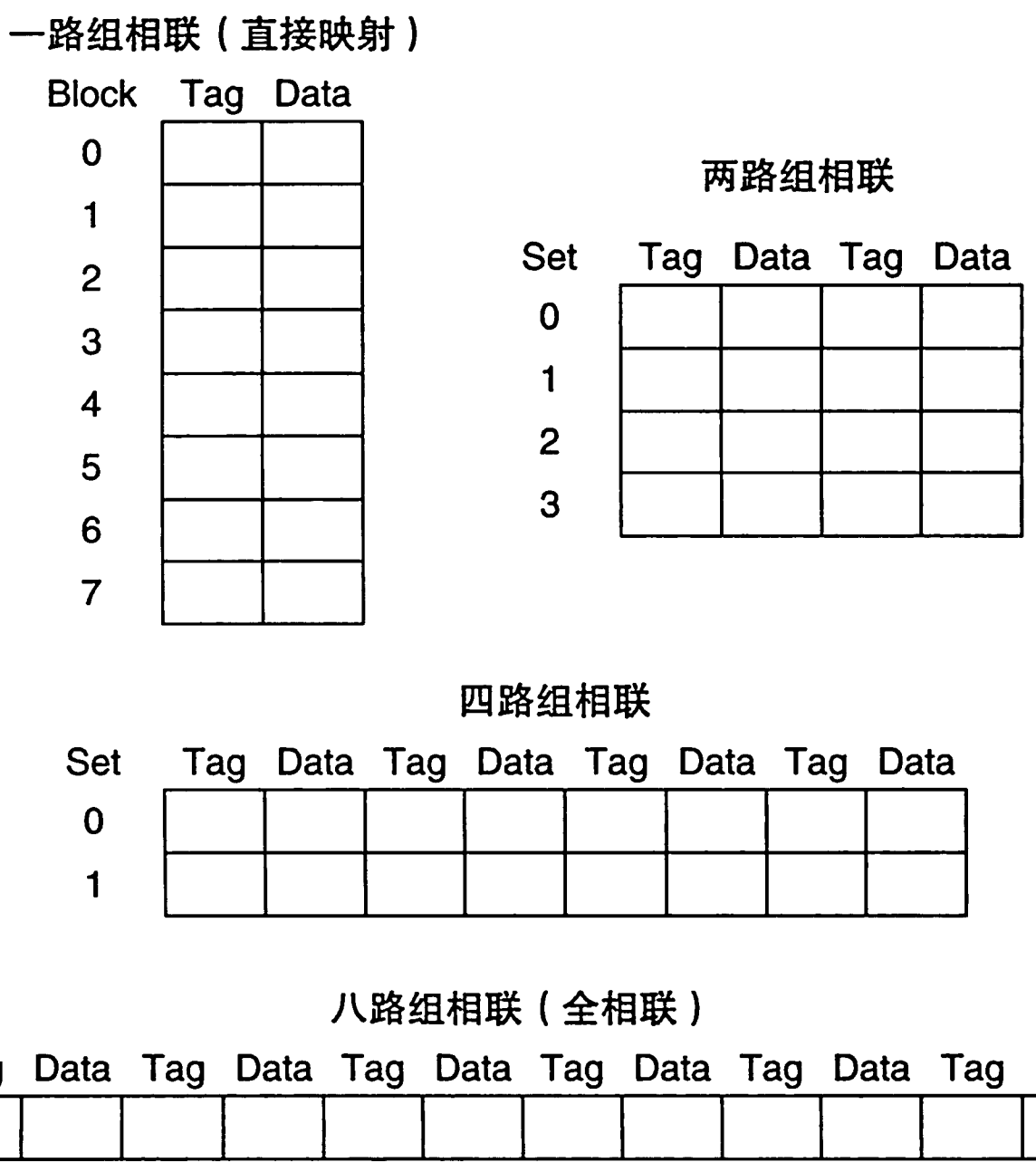

图 5-15 8 个数据块的 cache 被配置成直接映射、两路组相联、四路组相联和全相联。cache 的容量（以块为单位）等于组数乘以相联度。因此，对于固定大小的 cache 来说，提高相联度相当于降低组数，即增加每一组中的数据块数。对于 8 个数据块来说，一个八路组相联的 cache 就相当于全相联 cache

例题 | cache 中的失效和相联度

假设有三个小 cache，每一个 cache 包含 4 个数据块，数据块大小为 1 个字。第一个 cache 是直接映射，第二 cache 是两路组相联，第三个 cache 是直接映射。给定如下数据块地址序列，对于每一种 cache 组织结构，计算失效次数。序列为：0，8，0，6，8。

答案 | 直接映射最简单。首先，检查一下每个数据块地址会被映射到哪个 cache 块上：

数据块地址	cache数据块
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

每一次访问后填写 cache 内容，如果表项为空，表示该数据块为无效（invalid），带颜色标注的是在最近访问过程中新分配的 cache 表项，未带颜色标注的是 cache 中的旧表项。

访问的主存数据块地址	命中或失效	访问后cache数据块的内容			
		0	1	2	3
0	失效	Memory[0]			
8	失效	Memory[8]			
0	失效	Memory[0]			
6	失效	**Memory[0]**		Memory[6]	
8	失效	Memory[8]		**Memory[6]**	

在这 5 次访问中，直接映射 cache 产生了 5 次失效。

组相联 cache 中有两个组（标号 0 和 1），每组有两个数据块。首先判断每个数据块地址对应哪一组。

数据块地址	cache组号
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

由于在发生失效时需要在每一组中选择替换的表项，因此需要一种替换规则。组相联 cache 通常替换组内最近最少使用的数据块，也就是说，替换掉在过去最长时间未被使用的数据块（后续将详细讨论其他替换规则）。使用这样的替换规则，在上述访存序列完成后，组相联 cache 的内容如下所示：

访问的主存数据块地址	命中或失效	访问后cache数据块的内容			
		组0	组0	组	组
0	失效	Memory[0]			
8	失效	Memory[0]	Memory[8]		
0	命中	Memory[0]	Memory[8]		
6	失效	Memory[0]	Memory[6]		
8	失效	Memory[8]	Memory[6]		

注意，当访问数据块 6 时，它替换了数据块 8 的内容。这是因为，相比于数据块 0，数据块 8 最近最少被访问。使用两路组相连 cache，共产生 4 次失效。相比直接映射 cache，减少了一次失效。

全相联 cache 仅有一个组，组内有 4 个数据块，任意一个内存数据块都可以放置在 cache 的任意位置。全相联 cache 的性能最优，只有三次失效：

访问的主存数据块地址	命中或失效	访问后cache数据块的内容			
		数据块0	数据块1	数据块2	数据块3
0	失效	Memory[0]			
8	失效	Memory[0]	Memory[8]		
0	命中	Memory[0]	Memory[8]		
6	失效	Memory[0]	Memory[8]	Memory[6]	
8	命中	Memory[0]	Memory[8]	Memory[6]	

针对这个访问序列，三次失效是最优的，这是因为有三个不同的数据块地址需要被访问。注意，如果 cache 中有 8 个数据块，对于两路组相联 cache 来说不会发生替换（请自行检查）。和使用全相联 cache 相比，发生失效的次数是相同的。类似地，如果 cache 中有 16 个数据块，所有三个类型的 cache 的失效次数是相同的。上述例子说明，在考虑 cache 性能时，容量和相联度并不是相互独立的。

失效率的降低和相联度是什么关系？对于一个 64KiB 的数据 cache，每个数据块大小为 16 个字，图 5-16 中给出了随着相联度增加性能的变化情况（从直接映射到八路组相联）。从直接映射到两路组相联，失效率减少了大约 15%，但是，随着相联度的增加，失效率几乎没有变化。

相联度	数据失效率
1	10.3%
2	8.6%
4	8.3%
8	8.1%

图 5-16 针对结构类似 Intrinsity FastMATH 处理器中的数据 cache，使用 SPEC CPU2000 标准测试程序，测试获得的数据 cache 失效率。其中，相联度从直接映射变化到八路组相联。这些结果是 2003 年 Hennessy 和 Patterson 使用 SPEC CPU2000 中的 10 个测试程序获得的测试结果

5.4.2 在 cache 中查找数据块

来考虑一下如何在组相联 cache 中找到所需的数据块。正如在直接映射 cache 中，组相联 cache 中的每一个数据块都包括一个确定其地址的标签。在同一组内的每个 cache 块的标签都需要比较，判断是否与处理器访问的数据块地址匹配。图 5-17 中给出了地址的组成。索引位用来选择访问数据所在的组，该组内所有数据块的标签都需要比较。考虑到数据访问的速度，被选中组内的所有数据块的标签是并行比较的。串行比较策略将大大增加组相联 cache 的命中时间，这在全相联 cache 中也是一样的。

图 5-17 组相联或直接映射 cache 中地址的三个组成部分。索引位用来选择组，标签位用来在所在组内比较并选出数据块，block offset 用来在数据块内找到所需的数据

如果 cache 容量保持相同，增加相联度可以增加每组内数据块的数量，这也增加了需要并行比较的数据块数量：相联度以 2 的幂递增，每组内的数据块数量将会翻倍，组数将会减半。相应的，相联度以 2 的幂递增，索引的位长将减少 1，标签的位长将增加 1。在全相联 cache 中仅包含一个组，所有的数据块都必须并行比较。因此，其地址中没有索引，除块内偏移（block offset）以外，整个地址都需要进行比较。换句话说，需要与整个 cache 进行比较，无须先进行任何索引。

在直接映射 cache 中，只需要一个比较器，因为每组内只有一个数据块，可以通过简单的索引访问 cache。图 5-18 中给出，在四路组相联 cache 中，需要 4 个比较器，还需要一个 4 选 1 的选择器，用来在该组内四个数据块中进行选择。cache 访问过程包括了相应组的索引和组内的标签比较。组相联 cache 的开销在于需要更多的比较器，以及由于比较和选择带来的延迟。

任何一个存储层次，在直接映射、组相联或全相联映射结构之间进行选择，都需要在失效代价与相联度实现两方面进行权衡，既需要考虑时间，也需要考虑额外的硬件成本。

详细阐述 按内容寻址存储器（Content Addressable Memory，CAM）是一种将比较和存储集成在单一设备上的电路结构。与 RAM 提供地址读取数据的工作方式不同，CAM 用户传送数据至设备，CAM 查找是否存在该内容副本，并返回对应存储行的索引。相比于使用 SRAM 和比较器等硬件，使用 CAM 意味着 cache 设计者可以实现更高的相联度。2013 年，CAM 更大的容量和功耗使得两路和四路组相联结构一般采用标准 SRAM 和比较器实现，而八路或更多路组相联的结构则由 CAM 实现。

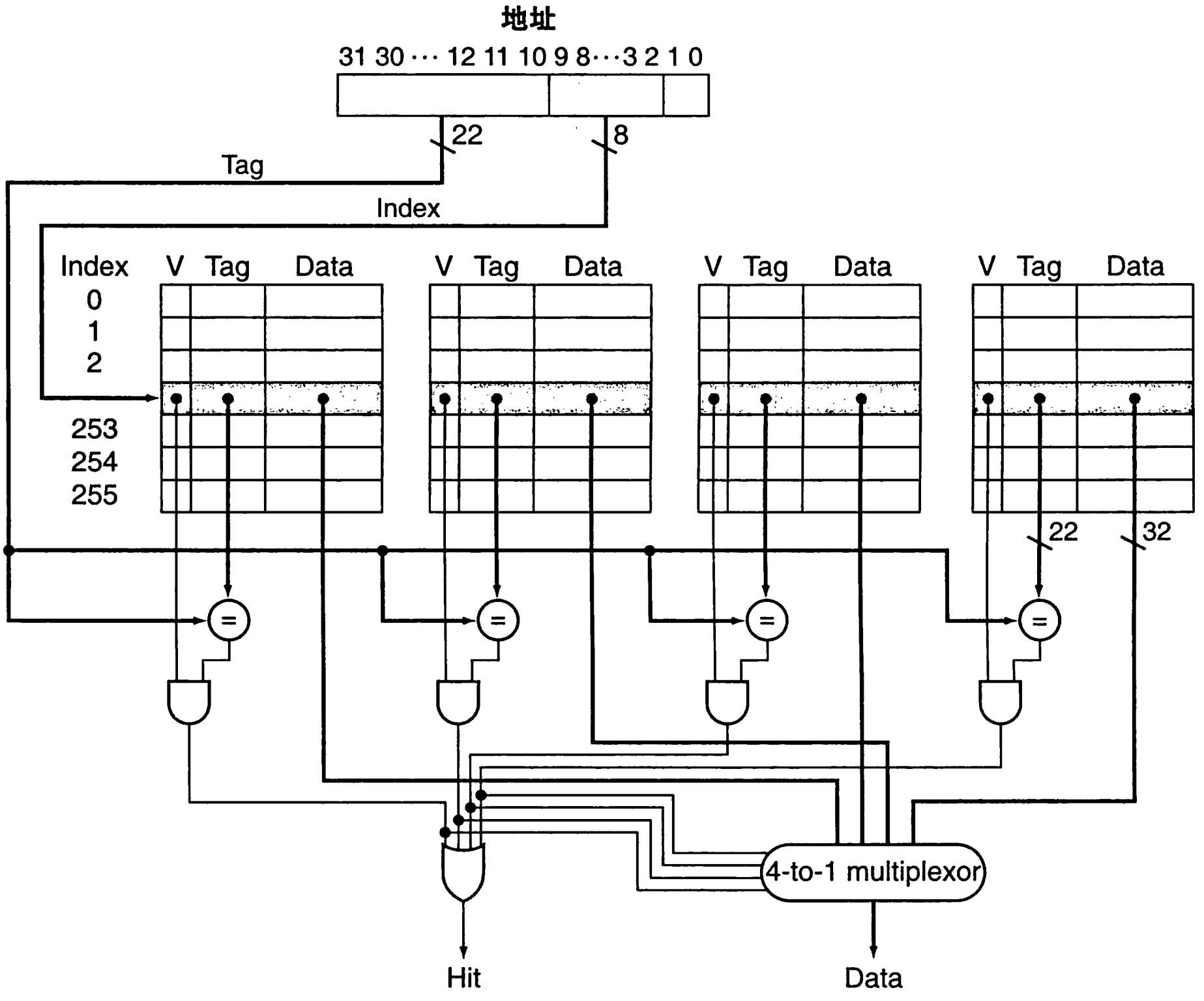

图 5-18　四路组相联 cache 的实现需要四个比较器和一个 4 选 1 选择器。比较器用来判断相应组内（如果有）的哪个数据块与标签匹配。比较器的输出作为多选器的选择信号，用来在相应组内的四个 cache 块中选择数据。在一些实现中，cache 的数据 RAM 部分的输出使能信号可以用来作为组内数据块选择信号。这些输出使能信号来自于比较器，可以用来驱动匹配的数据输出。这种结构中不需要使用多选器

5.4.3　选择替换的数据块

当直接映射 cache 中出现失效，所需的数据块将直接调入对应的唯一位置，之前在该位置的数据块将被替换。在组相联 cache 中出现失效，需要选择所需数据块的存放位置以及被替换的数据块。在全相联 cache 中出现失效，所有数据块都可以被替换。在组相联 cache 中，需要在对应组内选择数据块。

最常用的策略是**最近最少使用**（Least Recently Used，LRU），之前的例子中已经提到。在该策略中，被替换的数据块应该是最长时间未被使用的。5.4.1 节的组相联示例中使用的就是 LRU，因此数据块 Memory（0）被替换为 Memory（6）。

最近最少使用：一种替换策略。该策略中，最长时间未被使用的数据块将被替换。

LRU 替换策略可以通过跟踪某个数据块相对于同组内其他数据块的使用时间来实现。对于两路组相联 cache 来说，对两个数据块的使用情况进行跟踪，可以在每组内为其单独保留 1 位，该位表示哪一项被访问过。当相联度变大时，LRU 的实现将变得困难。在 5.8 节中将讨论另一种替换策略。

例题 | 标签的大小和组相联度

提高相联度需要更多的比较器，每一个 cache 块对应的标签位也越多。假设一个包含

4096 个数据块的 cache，单个数据块容量为 4 字，地址长度为 64 位。按照直接映射、两路组相联、四路组相联和全相联结构，分别计算组数和标签总容量。

答案 | 由于每个数据块的大小为 16（$=2^4$）字节，64 位地址中有 64−4 = 60 位用来表示索引和标签。直接映射 cache 结构中，组数和数据块数目相同，因此索引长度为 12 位，因为 $\log_2(4096)=12$。所以，标签总容量为（60−12）× 4096 = 48 × 4096 = 197Kb。

相联度每增加 1 倍，cache 的组数将减少 1/2。同样，索引的长度每减少 1 位，标签的长度就增加 1 位。因此，对于两路组相联 cache，共有 2048 组，标签总容量为（60−11）× 2 × 2048 = 98 × 2048 = 401 kb。对于四路组相联 cache，组数为 1024，标签总容量为（60−10）× 4 × 1024 = 100 × 1024 = 205 kb。

对于全相联 cache，仅有一组，包含了 4096 个数据块，标签就是 60 位，因此标签总容量为 60 × 4096 × 1 = 246 kb。

5.4.4　使用多级 cache 减少失效代价

所有的现代计算机都使用 cache。为了减小现代处理器高速时钟频率与访问 DRAM 所需的不断增长的延迟之间的差距，大多数微处理器支持不同层级的缓存。二级 cache 一般与一级 cache 封装在同一颗芯片中，一级 cache 失效后就会访问下一级缓存。如果二级 cache 中有所需数据，一级 cache 的失效代价就是二级 cache 的访问时间，这和主存的访问时间相比要少得多。如果一级或二级 cache 中都没有包含所需数据，就需要访问主存，那么失效代价则会增大。

使用二级 cache 对处理器性能提升有多大的影响？下面的例子给出说明。

例题 | 多级 cache 的性能

假设如果所有的存储访问在一级 cache 命中，则基准处理器的 CPI 为 1.0，时钟频率为 4GHz。假设主存的访问时间为 100ns，包括所有的失效处理过程。一级 cache 中每条指令的平均失效率为 2%。

如果增加一个二级 cache，不论失效或命中其访问时间都为 5ns，其容量足够大，可以将失效率降低到 0.5%。问（增加了二级 cache 的）处理器增速多少？

答案 | 主存的失效代价为

$$\frac{100\text{ns}}{0.25\dfrac{\text{ns}}{\text{时钟周期}}}=400\ \text{时钟周期}$$

带有一级 cache 的处理器 CPI 公式如下：

$$\text{实际 CPI} = \text{基准 CPI} + \text{平均每条指令产生的存储访问周期数}$$

对于上述带有一级 cache 的处理器来说：

$$\text{实际 CPI} = 1.0 + \text{平均每条指令产生的存储访问周期数} = 1.0 + 2\% \times 400 = 9$$

对于带有二级 cache 的处理器，一级 cache 的失效可以通过访问二级 cache 或主存来处理。二级 cache 的失效代价为：

$$\frac{5\text{ns}}{0.25\dfrac{\text{ns}}{\text{时钟周期}}}=20\ \text{时钟周期}$$

如果在二级 cache 中命中，那么这就是全部的失效代价。如果在二级 cache 中失效，那就需要访问主存，则完整的失效代价为二级 cache 的访问时间与主存访问时间之和。

因此，对于带有二级 cache 的处理器来说，实际 CPI 等于基准 CPI 与各级 cache 的访问时间之和。

实际 CPI = 1 + 平均每条指令的一级 cache 访问周期数 + 平均每条指令的二级 cache 访问周期数 = 1 + 2% × 20 + 0.5% × 400 = 1 + 0.4 + 2.0 = 3.4

因此，增加了二级 cache 的处理器增速。

$$\frac{9.0}{3.4}=2.6$$

或者换一种思路，还可以计算访问新增加的存储系统花费的时钟周期数。先计算二级 cache 命中的平均访问周期数 ((2%−0.5%) × 20 = 0.3)。然后计算主存访问请求的平均周期数，既需要包括访问二级 cache 的开销，也要包括访问主存的开销，即 0.5% ×（20+400）=2.1。三者相加，1.0 + 0.3 + 2.1 = 3.4，与上文相同。

对于一级和二级 cache，设计考虑是明显不同的。这是因为相比于单独一个 cache 来说，其他 cache 层次的存在会改变其最佳策略的选择。特别地，两级 cache 结构允许其一级 cache 关注命中时间最小化以提高工作频率或关注流水级数的减少，同时其二级 cache 关注失效率来降低长访存延迟带来的失效代价。

两级 cache 结构的这些影响可以通过与单个 cache 的最优设计进行比较来获得。与单个 cache 相比，**多级 cache**（multilevel cache）中的一级 cache 通常都较小。而且，一级 cache 会使用较小的数据块容量，以配合较小的 cache 容量，降低失效代价。相对的，二级 cache 会比单个 cache 容量大很多，因为二级 cache 的访问时间没有那么关键。随着容量的增大，相比于单个 cache，二级 cache 会使用更大的数据块。由于更关注降低失效率，二级 cache 也会使用比一级 cache 更高的相联度。

多级 cache：一种有多级 cache 的存储结构，区别于只有一个 cache 和主存的结构。

理解程序性能 我们用尽一切办法去分析排序算法，如冒泡排序、快速排序、基数排序等，希望找到更好的算法。图 5-19a 给出使用基数排序与快速排序进行搜索时平均每项的执行指令数目。正如预想，对于大型数组来说，在操作数量上基数排序比快速排序有算法上的优势。图 5-19b 给出不同排序项数下平均每项的所用时间（时钟周期数），而不是执行指令数。可以看到，图中各曲线的轨迹与图 5-19a 中相似。但是对于基数排序来说，随着待排序的数据增多，轨迹发生了变化。这里面发生了什么？图 5-19c 中查看了不同排序项数下平均每项的 cache 失效情况：对于每一个待排序的数据项，快速排序一直保持较少的失效次数。

遗憾的是，标准的算法分析通常都会忽略存储层次的影响。由于时钟频率提高和摩尔定律的发展，体系结构设计者不断从指令流中挖掘潜在的性能，充分利用存储层次对于高性能处理器显得尤为关键。正如在概述里提到的，理解层次化存储的行为对于理解当今处理器的程序性能至关重要。

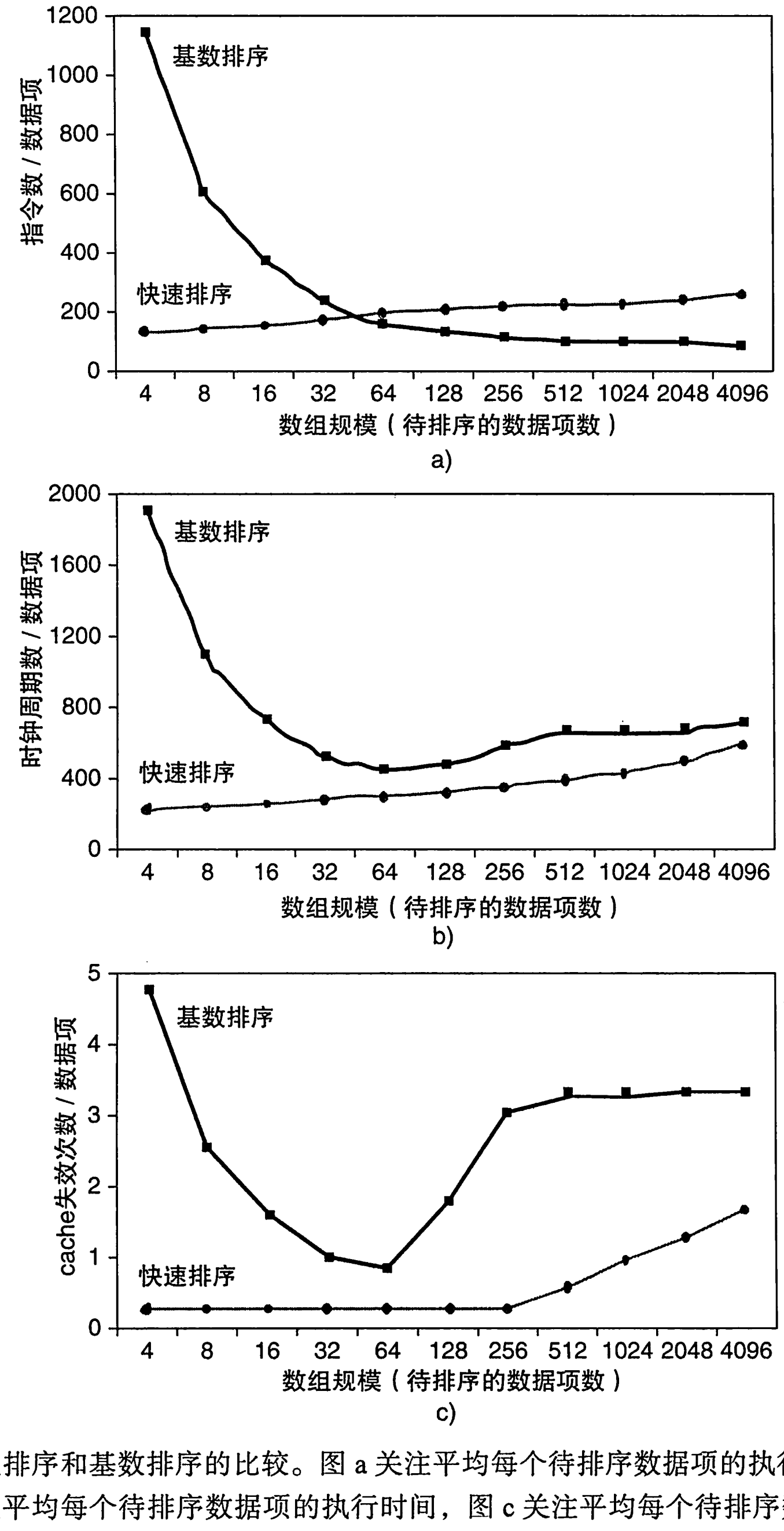

图 5-19　快速排序和基数排序的比较。图 a 关注平均每个待排序数据项的执行指令数，图 b 关注平均每个待排序数据项的执行时间，图 c 关注平均每个待排序数据项的 cache 失效次数。这些数据来自于 LaMarca 和 Ladner 1996 年的论文。由于以上结果，新的基数排序算法将存储层次考虑进去，重新获得了算法上的优势（具体见 5.15 节）。cache 优化的基础在于，当数据未被替换出 cache 之前，程序不断重复利用数据块中的所有数据（即程序的局部性）

5.4.5　通过分块进行软件优化

由于存储层次对程序性能具有非常重要的影响，许多软件优化技术通过重用 cache 中的数据来大幅度提高处理器性能，通过改善程序的时间局部性来降低失效率。

处理数组时，如果能将数组元素按照访问顺序存放在存储器中，则能够获得性能上的好处。但是，假设同时处理多个数组，一些数组按行访问，一些数组按列访问。按行存储（称为行优先）或者按列存储（称为列优先）数组都不能解决问题，这是因为在程序的每个循环体中行访问和列访问同时会被使用到。

因而，分块算法针对子矩阵（submatrice）或者数据块来进行操作，并不针对数组中完整的一行或一列进行操作。它的目标是，在替换之前对已在 cache 中的数据进行尽可能多的访问，这就是说，提高程序的时间局部性以减少 cache 失效。

例如，在 DGEMM（图 3-22 中第 4 ～ 9 行）中的内层循环中，

```
for (int j = 0; j < n; ++j)
    {
     double cij = C[i+j*n]; /* cij = C[i][j] */
     for( int k = 0; k < n; k++ )
      cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
     C[i+j*n] = cij; /* C[i][j] = cij */
     }
}
```

首先读入数组 B 的所有 N × N 个元素，重复地读入数组 A 中某一行中的 N 个元素，最后将结果写入 C 数组某一行中对应的 N 个元素（注释中的内容让矩阵的行和列更易于识别）。图 5-20 给出了对这三个数组的访问快照。深色阴影部分表示最近被访问过，浅色阴影部分表示较早被访问过，白色表示还未被访问。

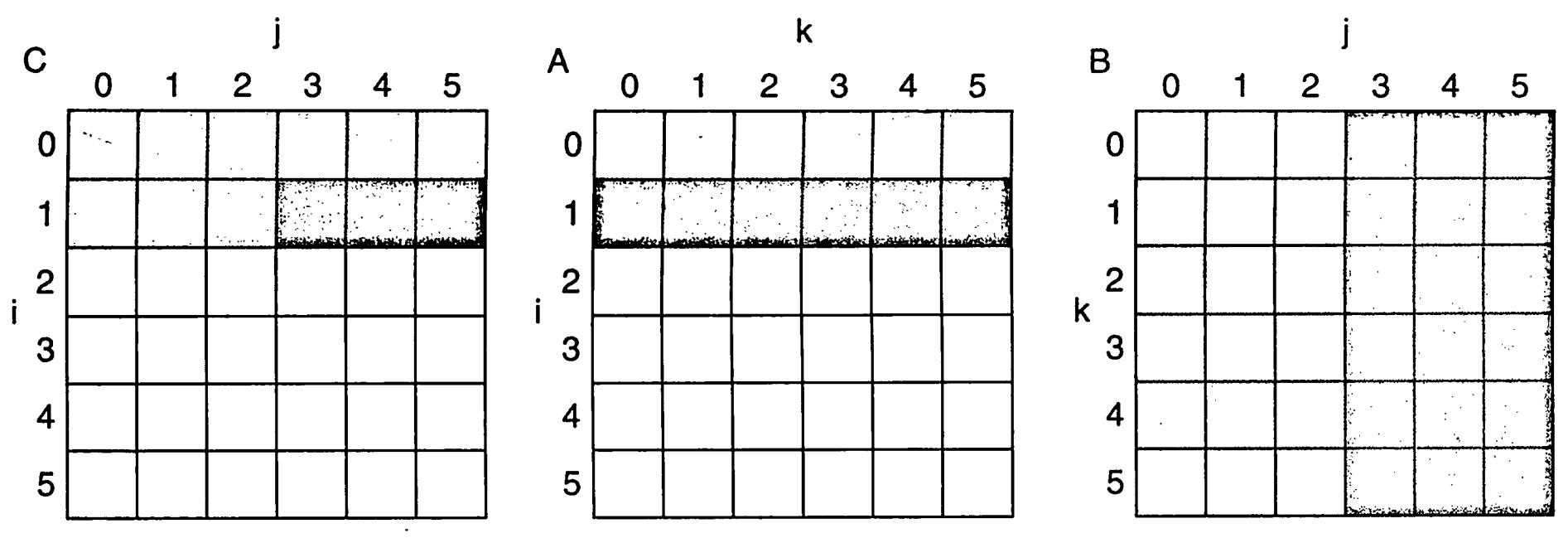

图 5-20　三个数组 C、A 和 B 的快照（N=6，i=1）。采用不同的阴影对各个数组元素的访问时机进行表示：白色表示尚未被访问过，浅色阴影表示较早被访问过，深色阴影表示最近被访问过。与图 5-22 相比，数组 A 和 B 的元素被重复读入以计算数组 C 的新元素。变量 i、j 和 k 用来进行数组访问，对应行或者列的变化

容量失效的次数明显与 N 和 cache 的容量有关。如果 cache 中可以存入三个 N × N 的矩阵，假设没有其他冲突，那么一切完美。在第 3、4 章中的 DGEMM 例子中，特意将矩阵大小设为 32 × 32，那么情况正好如此。每个矩阵有 32 × 32 = 1024 个元素，每个元素的大小为 8 字节，三个矩阵的大小就为 24KiB，这可以很容易存入 Intel Core i7（微架构为 Sandy Bridge）的容量为 32KiB 的数据 cache 中。

如果 cache 能够存入一个 N × N 的矩阵和一行 N 个元素，至少数组 A 中的第 i 行数据和数组 B 可以一直保留在 cache 中。如果容量更小，则数组 B 和数组 C 都可能发生失效。最坏情况下，N^3 个操作需要访问 $2N^3 + N^2$ 个存储数据字。

为保证需要访问的数组元素都尽可能在 cache 中，原始代码需要改写为基于子矩

阵的计算方式。这样，我们需要调用图 4-78 中的 DGEMM 版本，该版本就是在大小为 BLOCKSIZE × BLOCKSIZE 的矩阵上重复计算。BLOCKSIZE 也被称为块参数。

图 5-21 中给出 DGEMM 的分块版本。函数 do_block 改写自图 3-22 中的 DGEMM，增加了 3 个新参数 si、sj 和 sk 用来描述数组 A、B 和 C 的子矩阵的起始点。函数 do_block 的两个内部循环以 BLOCKSIZE 为步长进行计算，并不是按照数组 B 和 C 的全长进行计算。gcc 编译器通过函数内联（function inlining）消除了函数调用的所有开销。也就是说，在程序中直接插入函数代码，以避免通常的参数传递和现场保存操作。

```
1  #define BLOCKSIZE 32
2  void do_block (int n, int si, int sj, int sk, double *A, double
3  *B, double *C)
4  {
5     for (int i = si; i < si+BLOCKSIZE; ++i)
6        for (int j = sj; j < sj+BLOCKSIZE; ++j)
7           {
8              double cij = C[i+j*n];/* cij = C[i][j] */
9              for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11             C[i+j*n] = cij;/* C[i][j] = cij */
12          }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16    for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17       for ( int si = 0; si < n; si += BLOCKSIZE )
18          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19             do_block(n, si, sj, sk, A, B, C);
20 }
```

图 5-21　DGEMM（见图 3-22）的 cache 分块版本。假设数组 C 初始化为 0。函数 do_block 以第 3 章中的 DGEMM 为基础，增加了新参数来描述 BLOCKSIZE 大小的子矩阵的起始位置。gcc 优化通过内联 do_block 函数的方式消除了函数调用的开销

图 5-22 给出使用分块思想对三个数组进行访问的示例。仅对于容量失效来说，需要访问的内存数据字总数为 $2N^3/\text{BLOCKSIZE} + N^2$。（相比未分块前）这个数据得到了改善，原因在于参数 BLOCKSIZE。由此可见，分块思想挖掘出程序的时间局部性和空间局部性，比如数组 A 的访问得益于空间局部性，而数组 B 的访问得益于时间局部性。

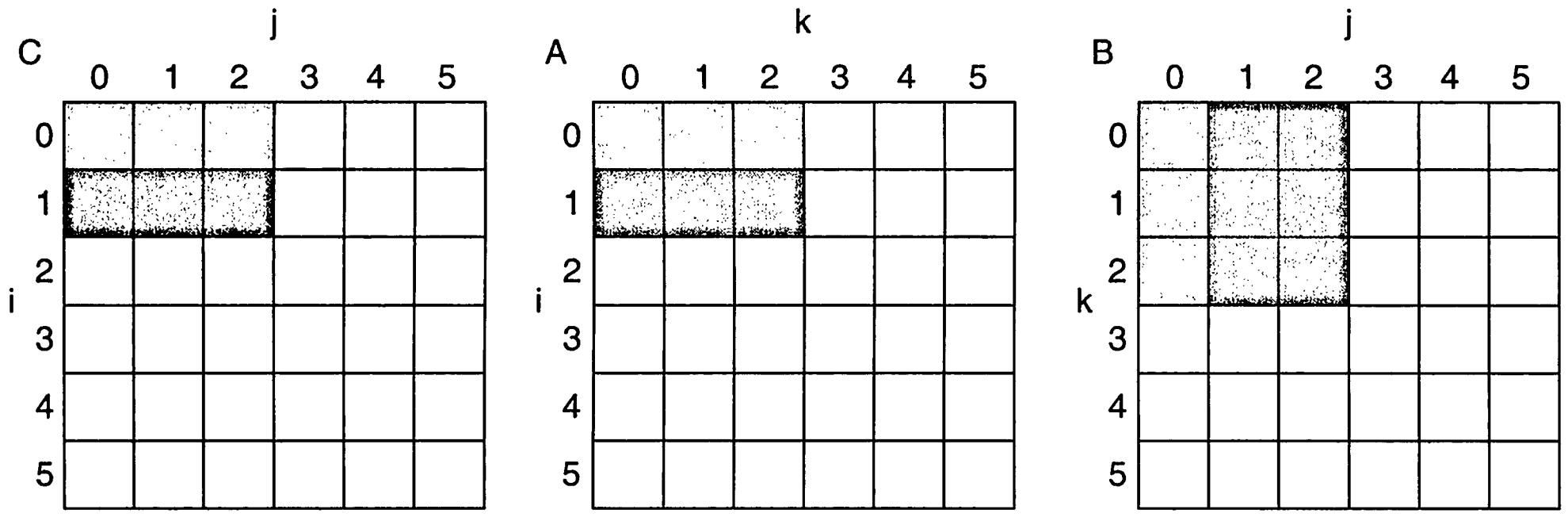

图 5-22　数组 C、A 和 B 的访问（BLOCKSIZE =3）。注意，与图 5-20 相比，访问的元素数量变少

虽然目标是减少 cache 失效次数，不过分块思想也可以用来协助寄存器分配。通过采用规模较小的数据块，可以把数据块保存在寄存器中，这样可以降低程序访问存储的次数，提高程序的性能。

随着矩阵规模的增大，三个矩阵不能完全放入 cache 中，图 5-23 中给出了采用 cache 分块对未优化 DGEMM 性能的影响。矩阵规模最大时，未优化程序的性能折半。采用 cache 分块的版本，即使矩阵规模达到 960 × 960，是 32 × 32 矩阵规模（第 3 章和第 4 章）的 900 倍，性能也仅仅降低了不到 10%。

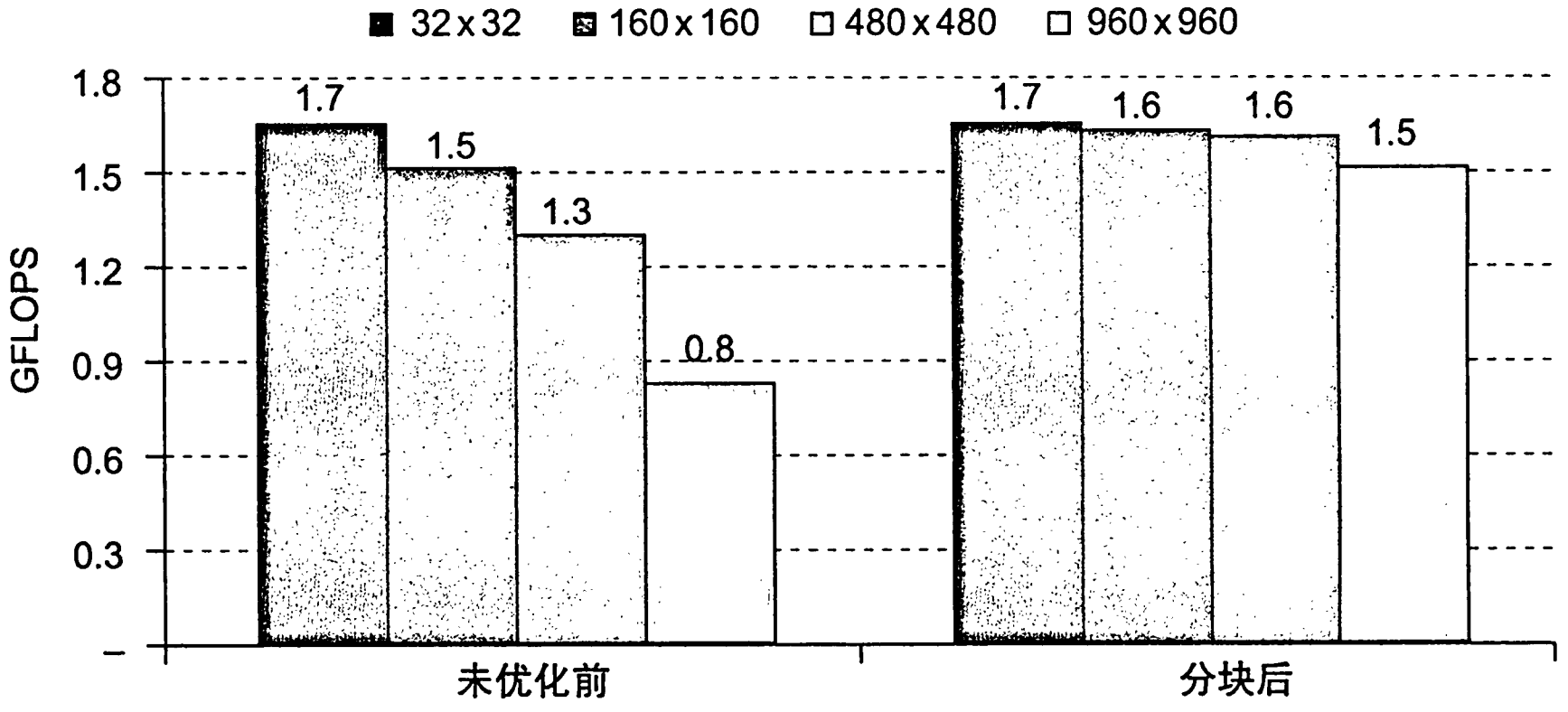

图 5-23　未优化 DGEMM（图 3-22）与 cache 分块 DGEMM（图 5-21）的性能比较，矩阵维度从 32 × 32（所有三个矩阵都可放入 cache）变至 960 × 960

详细阐述　多级 cache 使得很多问题变得复杂。首先，产生了很多不同类型的失效和对应的失效率。在 5.4.4 节的例子中，介绍了一级 cache 失效率和全局失效率（global miss rate），即在所有 cache 层次中都失效的存储访问所占的比例。还有二级 cache 失效率，即所有在二级 cache 上失效的访问数除以二级 cache 的所有访问数。这种失效率称为二级 cache 的局部失效率（local miss rate）。由于一级 cache 对存储访问进行了过滤，特别是那些时间和空间局部性较好的访问，因此二级 cache 的局部失效率比全局失效率要高。考虑 5.4.4 节的例子，二级 cache 的局部失效率为 0.5% / 2% = 25%。幸运的是，全局失效率表示的是必须访问主存的频度。

全局失效率：对于多级 cache，在所有 cache 层次上都失效的访问数目所占比例。

局部失效率：对于多级 cache，在某一 cache 层次上失效的访问数目所占比例。

详细阐述　对于乱序处理器（具体见第 4 章），性能更为复杂，因为处理器会在 cache 失效时继续执行指令。因此，采用平均每条指令失效次数来刻画 cache 性能，而非指令失效率和数据失效率。公式如下：

$$\frac{\text{存储停顿周期数}}{\text{指令}} = \frac{\text{失效次数}}{\text{指令}} \times (\text{所有失效代价} - \text{重叠的失效代价})$$

由于并没有通用的公式来计算重叠的失效代价，所以乱序处理器存储层次的性能评测不可避免地需要对处理器和存储层次进行模拟。只有通过观察每次失效后处理器的执行过程，才能确定等待数据时处理器是暂停还是另找事情来做。基本原则是：对于一级 cache 失效但二级 cache 命中的访问，处理器通常会隐藏失效代价，但是不会为二级 cache 隐藏失效代价。

详细阐述 算法的性能挑战在于，对于相同架构的不同实现，层次化存储在cache容量、相联度、基本块大小和cache数量等方面都存在变化。为适应这样的变化，最近一些算法库可对算法进行参数化处理，在程序运行时搜索参数空间，针对特定处理器找到最佳的参数组合。这种方法称为自动调优（autotuning）。

自我检测 关于多级cache的设计，下面哪句话通常是正确的？

1. 一级cache更关心命中时间，二级cache更关心失效率。
2. 一级cache更关心失效率，二级cache更关心命中时间。

5.4.6 总结

本节集中讨论了四个主题：cache性能，使用组相联结构来降低失效率，使用多级cache结构降低失效代价，软件优化以改善cache有效性。

存储系统对于程序执行时间有显著影响。处理器由于访存导致的暂停时间受到失效率和失效代价的影响。后续在5.8节中将会提到，cache的设计挑战在于如何减少这些影响，同时还不会显著影响存储层次的其他关键因素。

为降低失效率，采用了相联存储结构。这样的方案通过在cache内部更为灵活地放置数据块来降低cache的失效率。全相联结构允许任意放置数据块，但若需要查找数据则需要对cache中的每一个数据块进行搜索。这样高的硬件成本使得大容量的全相联cache不符合实际。组相联cache是一个比较实际的可选方案，因为只需要对索引选中的那一组中的数据块进行比对即可。（相比全相联cache）组相联cache虽然有更高的失效率，但访问速度更快。如何确定相联度以产生最佳性能，这和制造工艺以及实现细节都有密切联系。

可以将多级cache看作一种降低失效代价的技术，因为它允许一级cache的失效访问去访问容量更大的二级cache。设计者发现，受限的芯片面积和更高的时钟频率设计目标阻碍了一级cache的容量扩大，他们只能在二级cache上做文章了。通常，二级cache的容量会是一级cache的10倍或更大，可以处理很多一级cache中失效的访问。这时，失效代价就变为二级cache的访问时间（通常小于10个处理器时钟周期）与主存的访问时间（通常大于100个处理器时钟周期）的折中。关于相联度，设计者可以在二级cache的容量和访问时间之间做权衡，这和许多方面的具体实现密切相关。

最后，如果层次化存储对性能有重要影响，则需要考虑如何改变算法来改善cache的行为。当处理大型数组时，可以考虑分块这个重要的技术。

5.5 可靠的存储器层次

本章前面几节隐含了一个前提，即存储层次不会失效。如果这个前提不成立，只追求速度而没有可靠性是毫无吸引力的。正如第1章所述，增加可靠性的最好方法是冗余。本节将首先回顾与可靠性有关的术语并定义其他术语和度量，然后展开讲述如何采用冗余技术构造可靠的存储器。

5.5.1 失效的定义

假设有某种服务的需求，用户可以看到一个系统在两种分别有需求的服务的状态之间交替：

1. 服务完成：交付的服务与需求相符。

2. 服务中断：交付的服务与需求不同。

失效导致状态 1 到状态 2 的转换，而从状态 2 到状态 1 的转换过程被称为恢复。失效可能是永久性的或间歇性的，间歇性失效更为复杂。因为当系统在两种状态之间摇摆时，诊断更加困难。而永久性失效更容易诊断。

这里引出两个相关术语：可靠性和可用性。

可靠性是一个系统能够持续提供用户需求的服务的度量，即从参考时刻到失效的时间间隔。因此，平均无故障时间（MTTF）是可靠性的度量方法。与之相关的一个术语是年度失效率（AFR），它是指给定 MTTF 一年内预期的器件失效百分比。当 MTTF 变大时，可能会产生误导性的结果，而 AFR 会带来更直观的结果。

例题 | 磁盘的 MTTF 和 ARF

当今一些磁盘声称其 MTTF 为 1 000 000 小时，约等于 1 000 000 /（365 × 24）= 114 年，这意味着这些磁盘几乎从不失效。运行 Internet 服务（如搜索）的仓储级计算机可能有 50 000 台服务器。假定每台服务器有两块磁盘。使用 AFR 来计算每年有多少块磁盘失效。

答案 | 一年有 365 × 24 = 8760 小时。1 000 000 小时的 MTTF 意味着 AFR 为 8760 / 1 000 000 = 0.876%。磁盘总数为 100 000，因此每年将有 876 块磁盘失效，即平均每天有超过两块的磁盘失效！

服务中断使用平均修复时间（MTTR）来衡量。平均失效间隔时间（MTBF）=MTTF + MTTR。尽管 MTBF 被广泛使用，MTTF 却更加合适。然后，可用性是指系统正常工作时间在连续两次服务中断间隔时间中所占的比例：

$$\text{可用性} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

请注意，可靠性和可用性是可量化的，它们不仅仅是可信性的同义词。降低 MTTR 可以提高 MTTF 进而提高可用性。例如，用于故障检测、诊断和修复的工具可减少修复失效的时间，从而提高可用性。

我们希望系统有很高的可用性。一种简写是“每年可用性中 9 的数量”。例如，一个很好的网络服务可提供 4 或 5 个 9 的可用性。一年有 365 × 24 × 60 = 526 000 分钟，简化表示如下：

1 个 9：90%　⇒　36.5 天的维修时间 / 年

2 个 9：99%　⇒　3.65 天的维修时间 / 年

3 个 9：99.9%　⇒　526 分钟的维修时间 / 年

4 个 9：99.99%　⇒　52.6 分钟的维修时间 / 年

5 个 9：99.999%　⇒　5.26 分钟的维修时间 / 年

依此类推。

为了提高 MTTF，可以提高器件的质量，也可以设计能够在器件出现故障的情况下继续运行的系统。因此，由于器件的失效可能不会导致系统的失效，需要根据上下文对失效进行定义。为了明确二者的区别，用术语故障来表示器件的失效。以下是提高 MTTF 的三种