写请求存在额外的复杂情况：必须检查 TLB 中的写访问位。该位防止程序向只具有读权限的页执行写操作。如果程序试图写入，并且写访问位是关闭的，则会产生例外。写访问位是保护机制的一部分，我们将在稍后讨论。

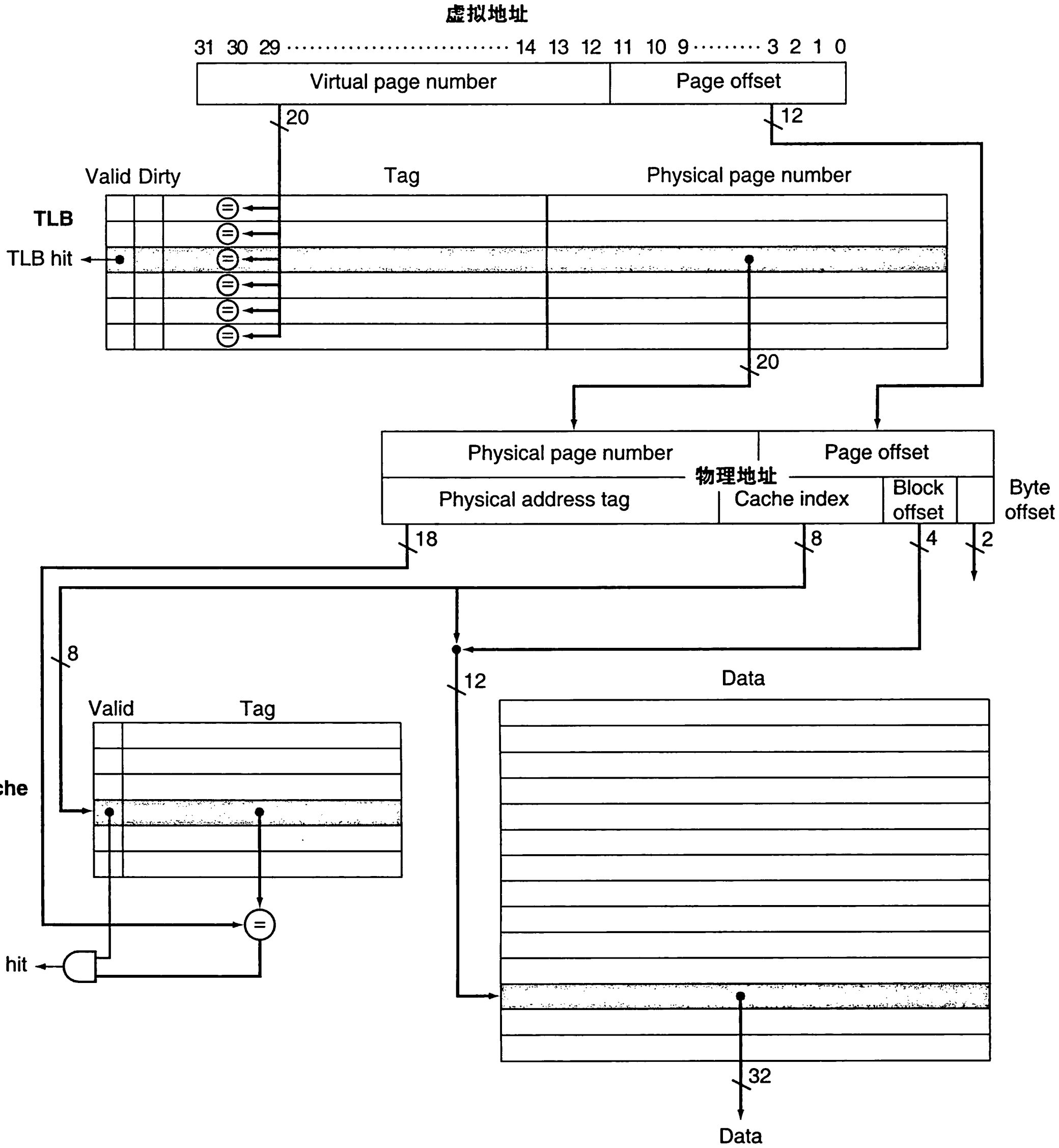

图 5-31 Intrinsity FastMATH 中的 TLB 和 cache 实现从虚拟地址到数据项的转换过程。该图显示了 TLB 和数据 cache 的结构，假设页大小为 4KiB。请注意，此计算机的地址仅为 32 位。本图主要介绍读操作，图 5-32 描述了如何处理写操作。请注意，与图 5-12 不同，标签和数据 RAM 是分开的。用 cache 索引和块偏移来寻址长而窄的数据 RAM，无须使用 16:1 的多路选择器也能选出块中所需的字。当 cache 采用直接映射的方式时，TLB 是全相联的。实现全相联的 TLB 要求将虚拟页号与每个 TLB 标签进行比较，因为需要的项可能在 TLB 中的任何位置。（见 5.4.2 节“详细阐述”中的内容可寻址存储器。）如果匹配表项的有效位有效，那么 TLB 命中，物理页号与页偏移中的位共同形成访问 cache 的索引

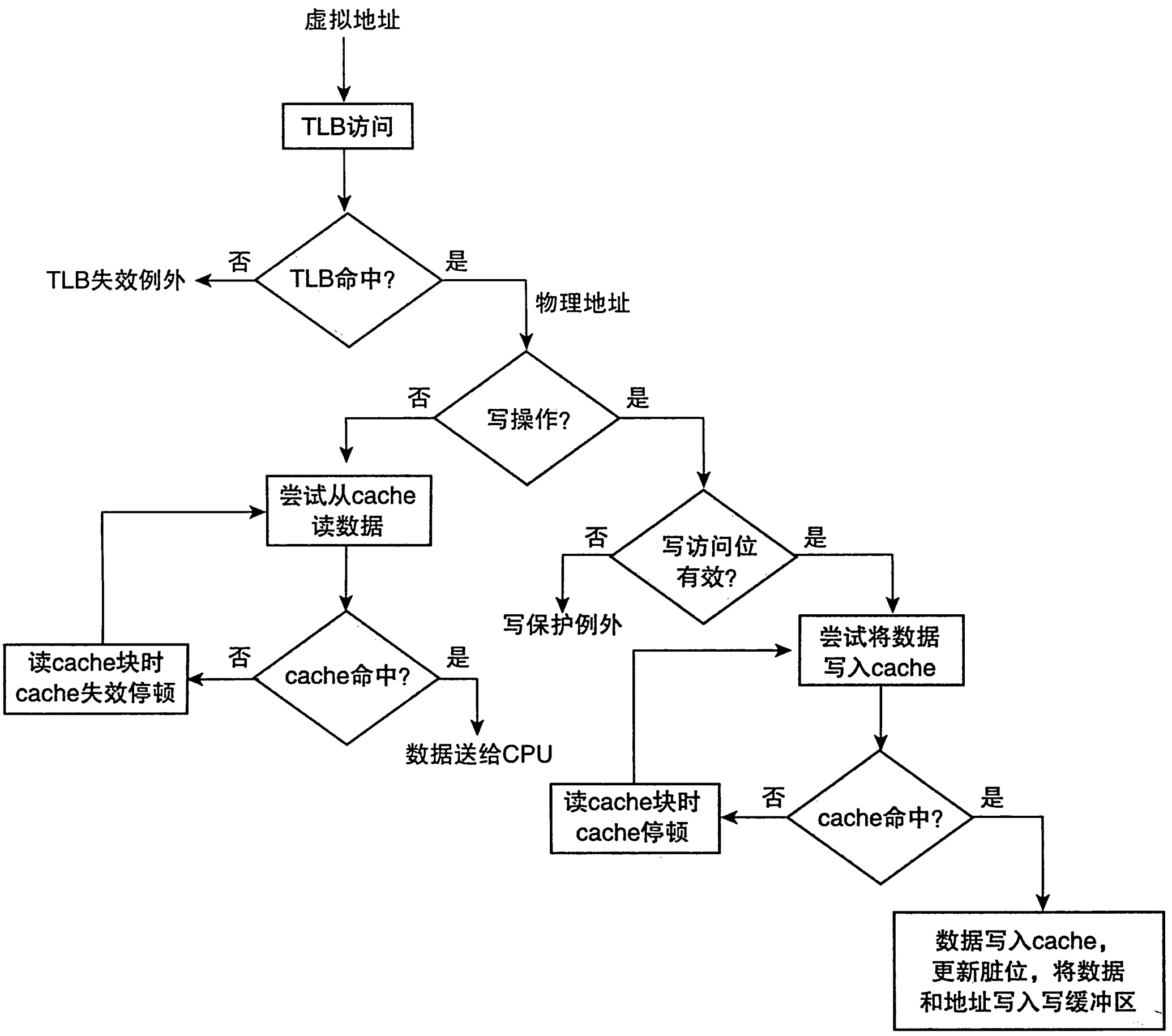

图 5-32 在 Intrinsity FastMATH 的 TLB 和 cache 中处理读或者写穿透操作。如果 TLB 命中，可以用最终的物理地址来访问 cache。对于读操作，cache 发生命中则提供数据，若发生失效，则在从内存中取数据时引起停顿。对于写操作，若命中，cache 中某数据项的一部分内容将被重写，如果采用写穿透策略，还要将数据送到写缓冲区。写失效和读失效类似，只是数据块从内存中读出后会被修改。写回策略需要将 cache 的脏位置位，并且只有当读或写失效时，如果被替换的块是脏块，才将整块写入写缓冲。注意，TLB 命中和 cache 命中是相互独立的事件，但是 cache 命中只能发生在 TLB 命中之后，这就意味着数据必须在内存中。TLB 失效和 cache 失效之间的联系将在接下来的例子和本章最后的习题中进一步研究。请注意，此计算机的地址宽度仅为 32 位

5.7.7 集成虚拟存储、TLB 和 cache

虚拟存储和 cache 系统就像一个层次结构一样共同工作，除非数据在主存中，否则它不可能在 cache 中出现。操作系统帮助管理该层次结构，当它决定将某一页移到磁盘上去时，就在 cache 中将该页的内容刷新。同时，操作系统修改页表和 TLB，而后尝试访问该页上的数据都将发生缺页。

在最好的情况下，虚拟地址由 TLB 进行转换，然后被送到 cache，找到相应的数据，取回并发给处理器。在最坏的情况下，访问在存储层次结构的三个部件中都发生失效：TLB、

页表和 cache。下面的例子将详细介绍这些交互作用。

例题　存储层次结构的全部操作

在像图 5-31 那样的存储层次结构中，包含一个 TLB 和一个按照图示组织的 cache。一个访存请求可能会遇到三种不同类型的失效：TLB 失效、缺页失效和 cache 失效。考虑这三种失效的所有组合：一个或多个事件发生（七种可能性）。对于每种可能性，说明此事件是否会真的发生以及在何种情况下发生。

答案　图 5-33 显示了所有可能发生的组合以及事实上它们是否真的可能发生。

TLB	页表	cache	可能发现吗？如果可能，在什么情况下发生
命中	命中	失效	可能，但是如果TLB命中则不会检查页表
失效	命中	命中	TLB失效，但是在页表中找到了这一项，重试后在cache中找到了数据
失效	命中	失效	TLB失效，但是在页表中找到了这一项，重试后在cache中没有找到数据
失效	失效	失效	TLB失效，接着发生缺页失效，重试后，在cache中没有找到数据
命中	失效	失效	不可能，如果页不在内存中，TLB中没有此转换
命中	失效	命中	不可能，如果页不在内存中，TLB中没有此转换
失效	失效	命中	不可能，如果页不在主存中，数据不允许在cache中存在

图 5-33　TLB、虚拟存储系统和 cache 中事件的可能组合。其中三种组合是不可能的，一种是可能的（TLB 命中，页表命中，cache 失效）但不可能被检测到

详细阐述　图 5-33 假设在访问 cache 之前，所有内存地址都转换为物理地址。在这种组织中，cache 是按物理地址索引的并且是按物理地址标记的（cache 索引和标签都是物理地址，而不是虚拟地址）。在这样的系统中，假设 cache 命中，访问内存的时间必须同时包括 TLB 访问时间和 cache 访问时间；当然，这些访问可以流水化。

虚拟寻址 cache：一种使用虚拟地址而不是物理地址访问的 cache。

别名：两个地址访问同一个目标的情况，一般发生在虚拟存储中两个虚拟地址对应同一个物理页时。

或者，处理器可以使用完全或部分虚拟的地址来索引 cache。这被称为虚拟寻址 cache，它使用虚拟地址的标签；因此，这种 cache 是按虚拟地址索引并且是按虚拟地址标记的。在这样的 cache 中，地址转换硬件（TLB）在正常 cache 访问期间未被使用，因为使用尚未转换为物理地址的虚拟地址来访问 cache。这将 TLB 从关键路径上移开，减少了 cache 的延迟。然而，当发生 cache 失效时，处理器需要将地址转换为物理地址，来从主存获取 cache 块。

物理寻址 cache：使用物理地址寻址的 cache。

当使用虚拟地址访问 cache 并且在进程之间共享页（可能使用不同的虚拟地址访问页）时，可能发生别名。当一个对象具有两个名称时就会发生别名——在这种情况下，同一页有两个虚拟地址。这种多义性产生一个问题，由于这种页上的一个字可能在 cache 中的两个位置上，每个位置对应于不同的虚拟地址。这将导致一个程序写入数据，而其他程序没有意识到数据已经改变。完全虚拟寻址的 cache 会导致 cache 和 TLB 的设计受限于减少别名，或导致需要操作系统（可能是用户）采取措施来确保不发生别名。

这两种设计观点常用的折中方法是采用虚拟索引的 cache——有时仅使用地址的页内偏移部分，这实际上是一个物理地址，因为它没有被转换——但使用物理标签。这些设计是按虚拟地址索引但按物理地址标记的，它试图利用虚拟寻址 cache 的性能优势，以及物理寻址

cache 的简单结构。例如，在这种情况下没有别名问题。图 5-31 假设页大小为 4KiB，但实际上是 16KiB，因此 Intrinsity FastMATH 就使用了这种方法。要实现这种方法，必须在最小页大小、cache 大小和相联度之间进行谨慎权衡。RISC-V 要求 cache 的行为就像物理标签和物理索引的一样，但它并不强制要求这样实现。例如，虚拟索引的、物理标签的数据 cache 可以使用额外的逻辑来确保软件无法区分差异。

5.7.8 虚拟存储中的保护

虚拟存储最重要的功能就是允许多个进程共享一个主存，同时为这些进程和操作系统提供内存保护。保护机制必须确保：尽管多个进程共享相同的主存，但无论有意或无意，一个恶意进程不能写另一个用户进程或操作系统的地址空间。TLB 中的写访问位可以防止一个页被写入。如果没有这一级保护，计算机病毒将更加泛滥。

硬件/软件接口 为了使操作系统能够在虚拟存储系统中实现保护，硬件至少要提供以下三个基本能力。请注意，由于前两者都需要虚拟机，因此其需求相同（5.6 节）。

管理模式：也称为内核模式，是一种运行操作系统进程的模式。

1. 支持至少两种模式，指示正在运行的进程是用户进程还是操作系统进程，操作系统进程也可称为管理态进程、内核进程或主管进程。

系统调用：将控制权从用户模式转换到管理模式的特殊指令，触发进程中的一个例外机制。

2. 提供用户进程可读但不能写的一部分处理器状态。这包括用户/管理程序模式位，用来指示处理器处于用户态还是管理态、页表指针和 TLB。为了写这些结构，操作系统使用仅在管理态下可用的特殊指令进行写操作。

3. 提供能让处理器在用户态和管理态之间相互转换的机制。从用户态到管理态的转换通常由系统调用的例外处理完成，它由特殊指令（RISC-V 指令集中的 `ecall`）将控制转移到管理态的代码空间的指定位置。与其他例外处理一样，系统调用处的程序计数器中的值被保存在管理态例外程序计数器（SEPC）中，并且处理器被置于管理态。从例外返回用户模式，使用管理态例外返回 `sret`（supervisor exception return）指令，将重置为用户模式，并跳转到 SEPC 中的地址。

通过使用这些机制并将页表存储在操作系统的地址空间中，操作系统可以更改页表、防止用户进程改写页表、确保用户进程只能访问操作系统提供给它的存储空间。

我们同样要防止一个进程读另一个进程的数据。例如，若教师将成绩存在处理器主存中，我们不希望学生程序读到它们。一旦开始共享主存，就必须赋予进程保护数据的能力，防止被另一个进程读写；否则，共享主存将是喜忧参半！

请记住，每个进程都有自己的虚拟地址空间。因此，如果操作系统将页表组织好，使得独立的虚拟页映射到不相交的物理页，那么一个进程将无法访问另一个进程的数据。当然，这也要求用户进程不能更改页表映射。如果操作系统能阻止用户进程修改自己的页表，那么安全性就有了保证。但是，这样一来，操作系统必须能够修改页表。将页表放在操作系统的保护地址空间就能满足所有要求。

当进程想要以受限的方式共享信息时，操作系统必须协助它们。这是因为访问另一个进程的信息需要更改访问进程的页表。写访问位可用来将共享限制为只读，并且与页表的其余部分一样，该位只能由操作系统更改。为了允许另一个进程（例如 P1）读进程 P2 的一页，P2 就要请求操作系统在 P1 的地址空间中为一个虚拟页创建页表表项，指向 P2 想要共享的

物理页。如果P2要求，操作系统可以使用写保护位来防止P1对数据进行改写。确定页访问权限的任何位都必须包含在页表和TLB中，因为只有在TLB失效时才访问页表。

详细阐述 当操作系统决定从运行进程P1切换为运行进程P2（称为上下文切换或进程切换）时，它必须确保P2不能访问P1的页表，否则不利于数据保护。如果没有TLB，只需将页表寄存器改为指向P2的页表（而不是P1）即可；如果有TLB，必须清除属于P1的TLB表项，目的是保护P1的数据，并迫使TLB载入P2的表项。如果进程切换的频率很高，这一举措的效率就很低。例如，在操作系统切换回P1之前，P2可能只装入了少量的TLB表项。不幸的是，P1随后发现它的所有TLB表项都不见了，因此不得不通过TLB失效来重新加载它们。出现这个问题是因为P1和P2使用相同的虚拟地址时，我们必须清除TLB以防止地址混淆。

上下文切换：为允许另一个不同的进程使用处理器，改变处理器内部的状态，并保存当前进程返回时需要的状态。

另一种常用的方法是通过添加进程标识符或任务标识符来扩展虚拟地址空间。为此，Intrinsity FastMATH有8位地址空间ID（ASID）字段。这个字段标识当前正在运行的进程；在切换进程时，它保存在由操作系统载入的寄存器中。RISC-V也提供ASID以减少上下文切换时的TLB刷新。进程标识符与TLB的标签部分相连接，因此当页号和进程标识符同时匹配时，TLB才发生命中。除极少数情况外，这种组合消除了清除TLB的需要。

同样的问题可能在cache中发生，因为在进程切换时，cache包含正在运行的进程的数据。对于物理寻址和虚拟寻址的cache，这些问题以不同方式出现，并且通过不同的解决方案（如进程标识符）来确保进程获取自己的数据。

5.7.9　处理TLB失效和缺页失效

当发生TLB命中时，使用TLB将虚拟地址转换为物理地址很简单，但正如我们之前所见，处理TLB失效和缺页失效却很复杂。当TLB中没有表项能与虚拟地址匹配时，将发生TLB失效。回想一下，TLB失效表明两种可能之一：

1. 页在内存中，只需创建缺少的TLB表项。
2. 页不在内存中，需要将控制转移给操作系统来处理缺页失效。

处理TLB失效或缺页失效需要使用例外机制来中断活跃进程，将控制转移到操作系统，然后再恢复执行被中断的进程。缺页将在主存访问时钟周期的某一时刻被发现。为了在缺页失效处理结束后重启指令，必须保存导致缺页失效的指令的程序计数器。管理态例外程序计数器（SEPC）用于保存该值。

此外，TLB失效或缺页失效例外必须在访存发生的同一个时钟周期的末尾被判定，这样下一个时钟周期将开始进行例外处理而不是继续正常的指令执行。如果在此时钟周期内未识别出缺页失效，一条load指令可能会改写寄存器，而当我们尝试重新启动指令时，这可能是灾难性的错误。例如，考虑指令`lb x10,0(x10)`：计算机必须防止写流水线阶段的发生；否则，就无法正确重启指令，因为x10的内容会被破坏。store指令也有类似的复杂情况。当发生缺页失效时，我们必须阻止写内存的操作的完成，这通常是通过令到内存的写控制线为无效来完成的。

硬件/软件接口 从操作系统开始执行例外处理程序，到操作系统保存了进程的所有状态这段时间内，操作系统特别脆弱。例如，如果在操作系统处理第一个例外时发生了另一个例外，控制单元将改写例外链接寄存器，从而无法返回到导致缺页失效的指令！我们可以通

过提供禁止例外和使能例外来避免这种错误的发生。首次发生例外时，处理器会设置一个管理态模式位，来禁止其他例外发生，这可以与处理器设置管理态模式位同时进行。随后操作系统将保存足够多的状态，即记录例外原因的 SEPC 和管理态例外原因（SCAUSE）寄存器，以便在发生另一个例外时允许恢复，这正如我们在第 4 章中看到的那样。RISC-V 中的 SEPC 和 SCAUSE 是两个特殊的控制寄存器，可以帮助处理例外、TLB 失效和缺页失效。然后，操作系统可以重新允许例外发生。这些步骤确保例外不会导致处理器丢失任何状态，因此也不会导致无法重启被中断指令的执行。

使能例外：也称为中断使能，用于控制处理器是否响应例外的信号或动作；在处理器安全地保存重启所需信息之前，必须阻止例外的发生。

一旦操作系统知道引起缺页失效的虚拟地址，它必须完成以下三个步骤：

1. 使用虚拟地址找到对应的页表表项，并在辅助存储中找到引用页的位置。

2. 选择要替换的物理页；如果所选页是脏的，则必须先将其写入辅助内存，然后才能将新的虚拟页写到此物理页中。

3. 启动读操作，将被访问的页从磁盘上取回到所选择的物理页的位置上。

当然，最后一步将花费数百万个处理器时钟周期（如果被替换的页是脏的，那么第二步也是如此）；因此，操作系统通常会选择另一个进程在处理器中执行直到磁盘访问结束。由于操作系统已经保存了当前进程的状态，所以它可以方便地将处理器的控制权交给另一个进程。

当从辅助存储器读页完成后，操作系统可以恢复最初导致缺页失效的进程的状态并执行从例外返回的指令。该指令将处理器从内核态重置为用户态，同时也恢复程序计数器的值。然后，用户进程重新执行引发缺页失效的指令，成功访问所请求的页面，并继续执行。

数据访问引起的缺页失效例外很难处理，是由于以下三个特征：

1. 它们发生在指令的中间，与指令缺页失效不同。

2. 在例外处理结束之前无法完成指令。

3. 例外处理结束后，指令必须重新启动，就像什么都没发生过一样。

使指令**可重新启动**，这样例外被处理之后，指令也能继续执行，这在类似 RISC-V 的体系结构中相对容易实现。因为每条指令只写一个数据项，并且这个写操作发生在指令周期的末尾，所以我们可以简单地阻止指令完成（不执行写操作）并在开始处重新启动指令。

可重启指令：一种在例外被处理之后能从例外中恢复而不会影响指令的执行结果的指令。

详细阐述 对于有着更复杂指令的处理器，其指令可能访问许多存储位置并写许多数据项，这使指令可重启要困难得多。处理一条指令可能在指令中间产生多次缺页失效。例如，x86 处理器有访问数千个数据字的块移动指令。在这样的处理器中，指令通常不能从开始处重新启动，像 RISC-V 那样。相反，指令必须被中断，然后从执行中断处继续执行。在执行的中间恢复指令通常需要保存一些特殊状态、处理例外，然后恢复那些特殊状态。要使处理器正常地执行这项工作，需要在操作系统的例外处理代码与硬件之间进行仔细的协调。

详细阐述 与每次访存都需要一次间接寻址不同，虚拟机监视器（5.6 节）支持影子页表，用于进行用户虚拟地址到物理地址的硬件转换。通过检测对用户页表的所有修改，虚拟机可以确保硬件正在用于转换的影子页表表项与用户操作系统中的页表表项一致，不同的是在用户页表中使用正确的物理地址替代了实地址。因此，虚拟机必须在用户操作系统试图更改页表或访问页表指针时产生自陷。这通常由用户操作系统通过对用户页表进行写保护，以及对页表指针

的任何访问产生自陷来实现。如上所述，如果是特权操作访问页表指针后，会发生后面一种情况。

详细阐述 体系结构中需要虚拟化的最后一部分是I/O。由于计算机中I/O设备数量和类型不断增加，I/O虚拟化是迄今为止系统虚拟化中最困难的部分。另一个困难是在多个虚拟机之间共享真实设备，还有一个困难是要支持大量的设备驱动程序，特别是在一个支持不同用户操作系统的虚拟机上就更加困难。它为每种虚拟机中各种类型的I/O设备提供一个通用的驱动，并且将其留给VMM以管理实际的I/O。

详细阐述 除了虚拟化指令集之外，另一个挑战是虚拟存储的虚拟化，这是因为虚拟机上的每个用户操作系统都管理自己的一组页表。为了实现这一目标，VMM将真实内存和物理内存的概念（通常被同义地处理）分开，并使真实内存成为虚拟内存和物理内存之间的一个独立层次。（有些使用术语虚拟内存、物理内存和机器内存来命名相同的三个层次。）用户操作系统通过其页表将虚拟内存映射到真实内存，VMM页表将用户的真实内存映射到物理内存。虚拟存储体系结构通常由页表实现，如IBM VM / 370、x86和RISC-V。

5.7.10　总结

虚拟存储是在主存和辅助存储之间进行数据缓存管理的一级存储层次。虚拟存储允许单个程序将其地址空间扩展到超出主存的限制。更重要的是，虚拟存储支持以受保护的方式在多个同时活跃的进程之间共享主存。

由于缺页失效的成本很高，管理主存和磁盘之间的存储层次结构很具有挑战性。通常采用下面一些技术来降低缺页失效率：

- 增大页的容量以利用空间局部性，并减少失效率。
- 使用页表实现的虚拟地址和物理地址之间的映射是全相联的，这样虚拟页可以放置在主存中的任何位置。
- 操作系统使用如LRU和引用位等技术来选择要替换的页。

写入辅助存储的成本很高，因此虚拟存储使用写回方案，并且跟踪记录页是否被改过（使用脏位）以避免向磁盘写入干净的页。

虚拟存储机制提供了从程序使用的虚拟地址到用于访存的物理地址之间的转换。该地址转换机制允许对主存进行受保护的共享，并提供了一些额外的好处，例如简化了内存分配。为了保证进程间受到保护，要求只有操作系统才能改变地址转换，这是通过阻止用户程序更改页表来实现的。可以在操作系统的帮助下实现进程之间受控地共享页，并且页表中的访问位指示用户程序是否具有对页的读或写访问权。

如果对于每一次访问，处理器都要访问内存中的页表来进行转换，这样的虚拟存储开销将很大，cache也将失去意义！相反，对于页表，TLB扮演了地址转换cache的角色，利用TLB中的转换将虚拟地址转换为物理地址。

cache、虚拟存储和TLB都建立在一组共同的原理和策略基础上。下一节将讨论这个通用框架。

理解程序性能 尽管虚拟存储是为了使小容量的存储看起来像大容量的存储，但是辅助存储和主存之间的性能差异意味着，如果程序经常访问比它拥有的物理存储更多的虚拟存储，程序运行速度会非常慢。这样的程序将不断地在主存和辅助存储之间交换页面，称为“颠簸”。如果发生颠簸将是一场灾难，但这种情况很少发生。如果你的程序发生颠簸，最简

单的解决方案是在内存更大的计算机上运行它，或为计算机增加内存。更复杂的选择是重新检查算法和数据结构，以查看是否可以更改局部性，从而减少程序同时使用的页数。这一组页被非正式地称为工作集。

一个更常见的性能问题是TLB失效。由于TLB同时只能处理32～64个页表表项，程序很容易有较高的TLB失效率，因为处理器可以直接访问不到64×4KiB = 0.25MiB的空间。例如，对于基数排序，TLB失效通常是一个挑战。为了缓解这个问题，现在大多数计算机体系结构都支持更大的页。例如，除了最小4KiB页外，RISC-V硬件还支持2MiB和1GiB大小的页。因此，如果程序使用大页，就可以直接访问更多内存而不会发生TLB失效。

让操作系统允许程序选择这些更大的页也是一个实际的难题。同样，减少TLB失效的更复杂的解决方案是重新检查算法和数据结构，以减少工作集中的页；另外，考虑内存访问对性能和TLB失效率的重要影响，一些工作集较大的程序已经针对该目标进行了重新设计。

详细阐述 RISC-V通过图5-29的多级页表支持更大的页。除了指向第1级和第2级中的下一级页表之外，它还支持超页转换，即将虚拟地址映射到1GiB物理地址（如果块转换在第1级）或2MiB物理地址（如果块转换在第2级）。

5.8 存储层次结构的一般框架

到目前为止，我们已经看到不同类型的存储层次结构有很多共同点。虽然存储层次结构的很多方面都有量的区别，但是很多决定层次结构功能的策略和特征在本质上是相似的。图5-34显示了存储层次结构的某些定量特征的区别。在本节的剩余部分，我们将讨论关于存储层次结构如何工作的一些选项，以及它们如何决定其行为。我们通过适用于存储层次结构两层之间的四个问题来研究这些策略，为了简单起见，我们将主要使用cache中的术语。

特征	二级cache的典型值	三级cache的典型值	页式存储的典型值	TLB的典型值
块的总大小	250～2000	2500～25 000	16 000～250 000	40～1024
以KiB计量的总容量	16～64	125～2000	1 000 000～1 000 000 000	0.25～16
块的字节数	16～64	64～128	4000～64 000	4～32
失效代价的周期数	10～25	100～1000	10 000 000～100 000 000	10～1000
失效率（二级cache失效被认为是全局失效）	2%～5%	0.1%～2%	0.000 01%～0.0001%	0.01%～2%

图5-34 表征计算机存储层次结构主要组成部分的关键定量设计参数。本图是这些层次截至2012年的典型值。值的范围很宽，一部分原因是很多随时间变化的值是互相关联的；例如，随着cache容量变大以克服更大的失效代价，块大小也随之增长。图中没有显示的是，服务器的微处理器现在还有三级cache，其容量可以是2～8MiB，并且包含比二级cache更多的块。三级cache将二级cache的失效代价降低到30～40个时钟周期

5.8.1 问题一：块可以被放在何处

我们已经看到，在较高存储层次结构中，块的放置可以使用一系列方案，从直接映射到组相联，再到全相联。如上所述，整个方案范围可以被认为是组相联方案的变体，其中组的

数量和每组中块的数量不相同：

机制	组的数	每组中块的数
直接映射	cache中的块数	1
组相联	$\frac{\text{cache中的块数}}{\text{相联度}}$	相联度（一般为2～16）
全相联	1	cache中的块数

增加相联度的好处是通常会降低失效率。失效率的改进来自于减少竞争同一位置而产生的失效。我们稍后将详细讨论。首先，来看能获得多少性能改进。图 5-35 显示了不同 cache 容量时，相联度从直接映射到八路组相联变化时的失效率。最大的改进出现在直接映射变化到两路组相联时，失效率降低了 20% ～ 30%。随着 cache 容量的增加，相联度的提高对性能改进作用很小；这是因为大容量 cache 的总失效率较低，因此改善失效率的机会减少，并且由相联度引起的失效率的绝对改进明显减少。如前所述，相联度增加的潜在缺点是增加了代价和访问时间。

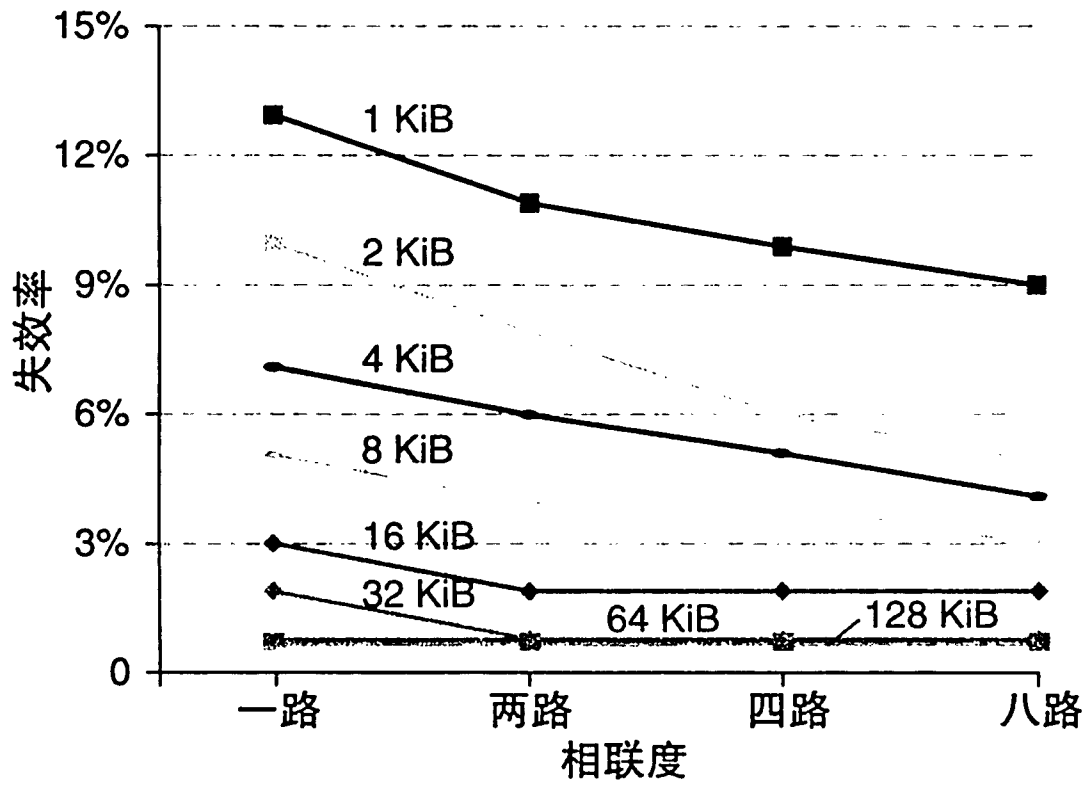

图 5-35　随着相联度的增加，8 种容量的数据 cache 的失效率都会改进。从一路（直接映射）到两路组相联变化时获益明显，进一步增加相联度所获得的好处就小一些了（例如，从两路到四路提高了 1% ～ 10%，而从一路到两路提高了 20% ～ 30%）。从四路到八路组相联的改进甚至更少，它们反而接近全相联 cache 的失效率。小容量的 cache 由于其本身失效率较高，因此从相联度所获得的益处就很明显。图 5-16 解释了这些数据是如何收集的

5.8.2　问题二：如何找到块

我们如何找到某个块取决于块的放置方案，因为这决定了可能存放位置的数量。我们可以把这些方案总结如下：

相联度	定位方法	需要比较的次数
直接映射	索引	1
组相联	索引组，查找组中元素	相联度
全相联	查找所有cache表项	cache容量
	独立的查找表	0

在存储层次结构中，直接映射、组相联或全相联映射的选择取决于失效代价与相联度实现代价之间的权衡，包括时间和额外硬件开销。在片上包含二级 cache 允许实现更高的相联

度，这是因为命中时间不再关键，设计者也不必依靠标准 SRAM 芯片来构建模块。除非容量很小，否则 cache 不使用全相联映射方式，其中比较器的成本并不是压倒性的，而绝对失效率的改进才是最明显的。

在虚拟存储系统中，页表是一张独立的映射表，它用来索引内存。除了表本身需要的存储空间外，使用索引表还会引起额外的存储访问。使用全相联映射和额外的页表有以下几个原因：

- 全相联有其优越性，因为失效代价非常高。
- 全相联允许软件使用复杂的替换策略以降低失效率。
- 全相联很容易索引，不需要额外的硬件，也不需要进行查找。

因此，虚拟存储系统通常使用全相联。

组相联映射通常用于 cache 和 TLB，访问时包括索引和组内查找。一些系统使用直接映射 cache，这是因为访问时间短并且实现简单。访问时间短是因为查找时不需要进行比较。这样的设计选择取决于很多实现细节，例如，cache 是否集成在片上、实现 cache 的技术以及 cache 的访问时间对处理器周期时间的重要性。

5.8.3 问题三：当 cache 发生失效时替换哪一块

当相联的 cache 发生失效时，我们必须决定要替换哪个块。在全相联的 cache 中，所有的块都是替换的候选者。如果 cache 是组相联的，则必须在一组的块中进行选择。当然，直接映射 cache 中的替换很容易，因为只有一个候选者。

在组相联或全相联的 cache 中有两种主要的替换策略：

- 随机：随机选择候选块，可能使用一些硬件辅助实现。
- 最近最少使用（LRU）：被替换的块是最久没有被使用过的块。

实际上，在相联度不低（典型的是两路到四路）的层次结构中实现 LRU 的代价太高，这是因为追踪记录使用信息的代价很高。即使对于四路组相联，LRU 通常也是近似实现的，例如，追踪记录哪一对块是最近最少使用的（需要使用 1 位），然后追踪记录每对块中哪一块是最近最少使用的（每对需要使用 1 位）。

对于较大的相联度，LRU 是近似的或使用随机替换策略。在 cache 中，替换算法由硬件实现，这意味着方案应该易于实现。随机替换算法用硬件很容易实现，对于两路组相联 cache，随机替换的失效率比 LRU 替换策略的失效率高约 1.1 倍。随着 cache 容量变大，两种替换策略的失效率下降，并且绝对差异也变小。实际上，随机替换算法的性能有时可能比用硬件简单实现的近似 LRU 更好。

在虚拟存储中，LRU 的一些形式都是近似的，因为当失效代价很大时，失效率的微小降低都很重要。通常提供参考位或其他等价的功能使操作系统更容易追踪记录一组最近使用较少的页。由于失效代价很高且相对不频繁发生，主要由软件来近似这项信息的做法是可行的。

5.8.4 问题四：写操作如何处理

任何存储层次结构的一个关键特性是如何处理写操作。我们已经看到两个基本选项：

- 写穿透：信息将写入 cache 中的块和存储层次结构中较低层的块（对 cache 而言是主存）。5.3 节中的 cache 使用这种方案。

- 写返回：信息仅写入 cache 中的块。修改后的块只有在它被替换时才会写入层次结构中的较低层。由于 5.7 节中讨论的原因，虚拟存储系统总是采用写返回策略。

写返回和写穿透都有其各自的优点。写返回的主要优点如下：

- 处理器可以按 cache 而不是内存能接收的速率写单个的字。
- 块内的多次写操作只需对存储层次结构中的较低层进行一次写操作。
- 当写回块时，由于写一整个块，系统可以有效地利用高带宽传输。

写穿透具有以下优点：

- 失效比较简单，代价也比较小，这是因为不需要将块写回到存储层次结构中的较低层。
- 写穿透比写返回更容易实现，尽管实际上写穿透 cache 仍然需要写缓冲区。

在虚拟存储系统中，只有写返回策略才是实用的，这是因为写到存储层次结构较低层的延迟很大。尽管允许存储器的物理和逻辑宽度更宽，并对 DRAM 采用突发模式，处理器产生写操作的速率通常还是超过存储系统可以处理它们的速率。因此，现在最低一级的 cache 通常采用写回策略。

重点 cache、TLB 和虚拟存储最初可能看起来非常不同，但它们都基于相同的两个局部性原理，并且可以通过它们对四个问题的各自回答来理解：

问题 1：块可以被放在哪里？

答案：一个位置（直接映射）、一些位置（组相联）或任何位置（全相联）。

问题 2：如何找到块？

答案：有四种方法：索引（在直接映射的 cache 中），有限的检索（在组相联 cache 中），全部检索（在全相联的 cache 中），单独的查找表（在页表中）。

问题 3：失效时替换哪一块？

答案：通常是最近最少使用的块，或随机选取的一块。

问题 4：如何处理写操作？

答案：层次结构中的每一层都可以使用写穿透策略或写返回策略。

5.8.5　3C：一种理解存储层次结构的直观模型

在本节中，我们将介绍一种模型，该模型可以很好地洞察存储层次结构中引起失效的原因，以及存储层次结构中的变化对失效的影响。我们将用 cache 来解释这些想法，尽管这些想法对其他层次也都直接适用。在此模型中，所有失效都被分为以下三类（**3C 模型**）：

- **强制失效**：对没有在 cache 中出现过的块进行第一次访问时产生的失效，也称为冷启动失效。
- **容量失效**：cache 无法包含程序执行期间所需的所有块而引起的失效。当某些块被替换出去，随后再被调入时，将发生容量失效。
- **冲突失效**：在组相联或者直接映射 cache 中，很多块为了竞争同一个组导致的失效。冲突失效是直接映射或组相联 cache 中的失效，而在相同大小的全相联 cache 中不存在。这种 cache 失效也称为**碰撞失效**（collision miss）。

3C 模型：将所有的 cache 失效都归为三种类型的 cache 模型，三类分别为强制失效、容量失效和冲突失效。因其三类名称的英文单词首字母均为 C 而得名。

强制失效：也称为冷启动失效。对没有在 cache 中出现过的块进行第一次访问时产生的失效。

容量失效：由于 cache 在全相联时都不可能容纳所有请求的块而导致的失效。

图 5-36 显示了失效率如何按照引起的原因被分为三种。改变 cache 设计中的某一方面可以直接影响这些失效的原因。由于冲突失效来自对同一 cache 块的争用，因此提高相联度可减少冲突失效。但是，提高相联度可能会延长访问时间，从而降低整体性能。

冲突失效：也称为碰撞失效。在组相联或者直接映射 cache 中，很多块为了竞争同一个组导致的失效。这种失效在使用相同大小的全相联 cache 中是不存在的。

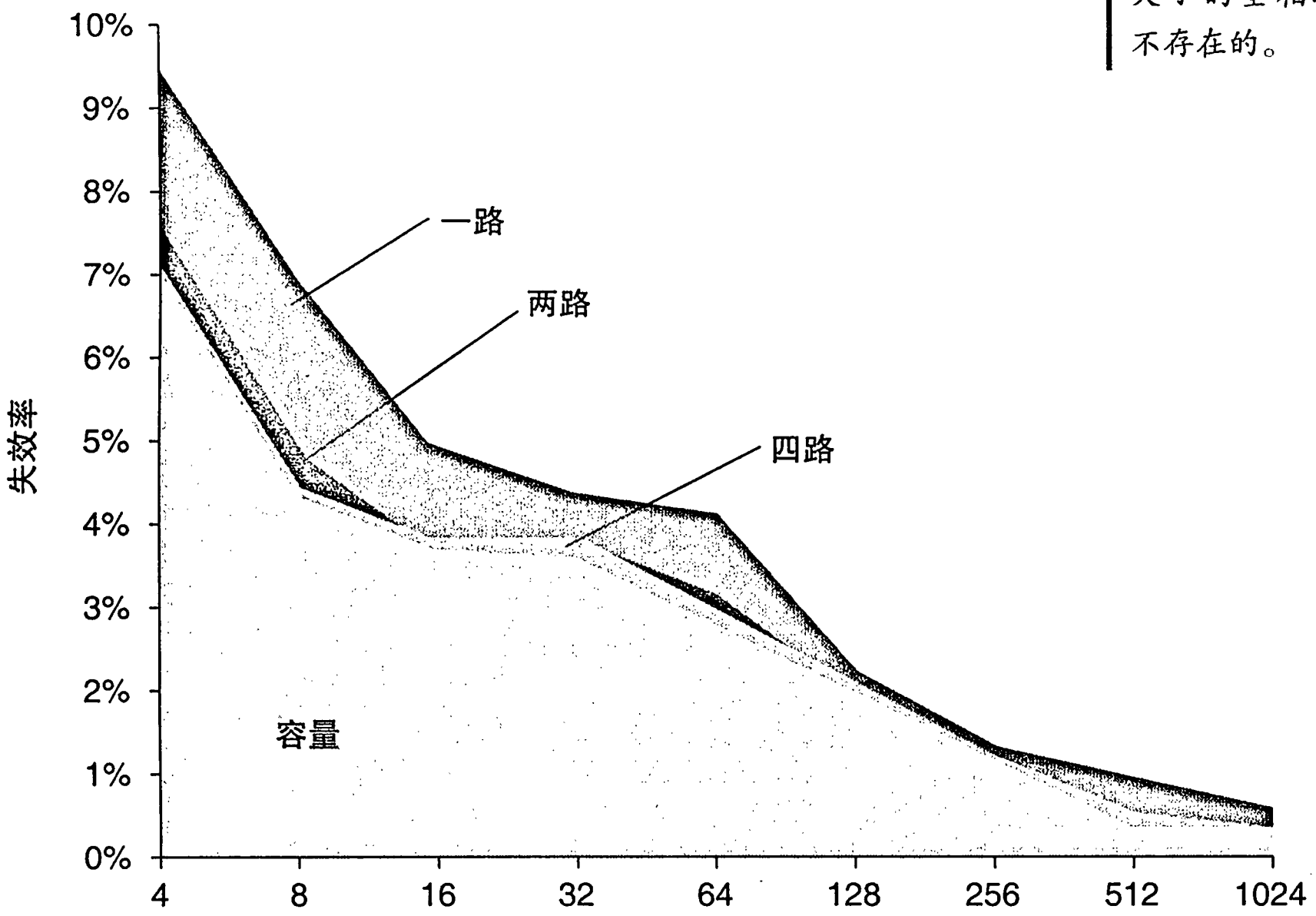

图 5-36 根据失效原因，失效率可被分为三种。此图显示不同容量 cache 的总失效率及其组成部分。数据由 SPEC CPU2000 整数和浮点基准测试得到，与图 5-35 中的数据源相同。强制失效部分只占 0.006%，在图中看不出来。下一部分是容量失效，它取决于 cache 容量。冲突部分取决于相联度和 cache 容量，图中显示了相联度从一路到八路的冲突失效率。在每种情况下，当相联度从下一个较高度变为标记的相联度时，标记部分对应于失效率的增加。例如，标记为两路的部分表示当 cache 从四路变为两路时失效的增加。因此，相同大小的直接映射 cache 与全相联 cache 所引起的失效率的差异由标记为四路、两路和一路的部分的总和给出。八路和四路之间的差异很小，很难在图上看到

简单地增大 cache 容量可以减少容量失效，实际上，多年来二级 cache 容量一直在稳步增长。当然，在增大 cache 的同时，我们也必须注意访问时间的增长，这可能导致整体性能降低。因此，尽管一级 cache 也在增大，但是非常缓慢。

由于强制失效是对块的第一次访问时产生的，因此，对 cache 系统来说，减少强制失效次数的主要方法是增加块大小。由于程序将由较少的 cache 块组成，因此这将减少对程序每一块都要访问一次时的总访问次数。如上所述，块容量增加太多可能对性能产生负面影响，因为失效代价会增加。

重点 设计存储层次结构的挑战在于，任何一个改进失效率的设计可能同时对整体性能产生负面影响，如图 5-37 所示。这种正面和负面效果的结合使得存储层次结构的设计变得有趣。

设计变化	对失效率的影响	可能对性能产生的负面影响
增加cache容量	降低失效率	可能延长访问时间
增加相联度	由于减少了冲突失效，降低了失效率	可能延长访问时间
增加块容量	由于空间局部性，对很宽范围内变化的块大小，降低了失效率	增加失效损失，块太大还会增大失效率

图 5-37　存储层次结构设计的挑战

将失效分为 3C 是个有用的定性模型。在实际 cache 设计中，很多设计选择相互影响，改变一个 cache 特性通常会影响另一些失效率的组成部分。尽管存在这些缺点，但该模型仍是深入了解 cache 设计性能的有效方法。

自我检测　以下哪项表述（如果有的话）是正确的？

1. 没有办法减少强制失效。
2. 全相联 cache 没有冲突失效。
3. 在减少失效方面，相联度比容量更重要。

5.9　使用有限状态自动机控制简单的 cache

现在可以为 cache 建立控制，正如我们在第 4 章中为单时钟周期和流水线的数据通路添加控制那样。本节先对简单 cache 进行定义，之后描述有限状态自动机，并以使用有限状态自动机控制这个简单 cache 作为结束。5.12 节是对本节的深度扩展，使用一种新的硬件描述语言展示 cache 和控制器。

5.9.1　一个简单的 cache

现在我们将要为一个简单的 cache 设计控制器。下面是这个 cache 的主要特征：

- 直接映射 cache
- 使用写分配写回
- 块大小为四字（16 字节或 128 位）
- cache 大小为 16KiB，因此该缓存内包含 1024 个块
- 32 位地址
- 该 cache 的每个块内都包含有效位和脏位

根据 5.3 节中的内容，我们现在可以计算该 cache 地址的字段：

- cache 块索引为 10 位
- 块内偏移为 4 位
- 标签大小为 32−（10 + 4），也就是 18 位

处理器与 cache 之间的控制信号为：

- 1 位读或写信号
- 1 位有效信号，表示该操作是否为 cache 操作
- 32 位地址
- 32 位数据，从处理器传输至 cache
- 32 位数据，从 cache 传输至处理器

- 1 位就绪信号，表示 cache 操作已经完成

存储器与 cache 之间的接口与处理器和 cache 之间的字段基本一致，只不过数据位现在换成了 128 位宽。额外的存储器宽度在当今的微处理器中很常见，它处理 32 位或 64 位字的处理器，而 DRAM 控制器通常为 128 位。将 cache 块与 DRAM 的宽度相匹配可以简化设计。因此将信号设计如下：

- 1 位读或写信号
- 1 位有效信号，表示该操作是否为存储器操作
- 32 位地址
- 128 位数据，从 cache 传输至存储器
- 128 位数据，从存储器传输至 cache
- 1 位就绪信号，表示存储器操作已经完成

需要注意的是，存储器接口所需的时钟周期数不是固定的。我们假设当存储器读写操作完成时，存储器控制器可以通过就绪信号将该事件通知给 cache。

在描述 cache 控制器之前，我们需要回顾一下有限状态自动机的相关知识，有限状态自动机可以使得控制一个需要多时钟周期的操作成为可能。

5.9.2 有限状态自动机

为了设计单时钟周期数据通路的控制单元，我们使用真值表来根据指令类别设置控制信号。cache 的控制将会更为复杂，因为对 cache 的操作可能包含了一系列步骤。cache 控制必须指定每个步骤中需要设置的信号以及下一个将要执行的步骤。

最常见的多步骤控制技术基于**有限状态自动机**，它通常以图的形式表示。有限状态自动机由一系列状态和状态之间改变的方向组成。该方向由**状态转换函数**定义，它是当前状态和指向新状态的输入之间的映射。当使用有限状态自动机进行控制时，该自动机的状态也指定了一系列输出，该输出是当机器处于该状态下的断言。有限状态自动机实现时通常假定所有未明确断言过的输出都是无效的。类似地，数据通路的正确操作也取决于这样一个事实：没有明确断言过的信号都是无效的，而不是针对该信号做无意义的操作。

有限状态自动机：一个包含了一组输入/输出、状态转换函数（将当前状态和输入映射为新状态）和一个输出函数（将当前状态和输入映射为断言输出）的顺序逻辑函数。

下一状态函数：一个组合函数，给定一个输入和当前状态，可以得出有限状态自动机的下一状态。

多选器的控制与上述行为略有不同，因为它只选择众多输入中的一个，而不考虑该输入是 0 还是 1。因此，在有限状态自动机中，我们总是指定所有需要关注的多选器控制的设置。当使用逻辑器件实现有限状态自动机时，可以默认将控制设置为 0，也因此不需要任何逻辑门。在附录 A 中提供了一个简单的有限状态自动机示例，如果你不熟悉有限状态自动机的概念，那么在继续后面的学习之前，建议你先查看附录 A。

有限状态自动机可以用保存当前状态的临时寄存器和一组组合逻辑实现，该组合逻辑决定被断言的数据通路信号以及下一状态。图 5-38 是该实现的图示。附录 C 详细描述了有限状态自动机是如何使用该结构实现的。在附录 A 的 A.3 节中，有限状态自动机的组合控制逻辑既可以用只读存储器（ROM）也可以用可编程逻辑阵列（PLA）来实现（在附录 A 中同样有关于这些逻辑元件的介绍）。

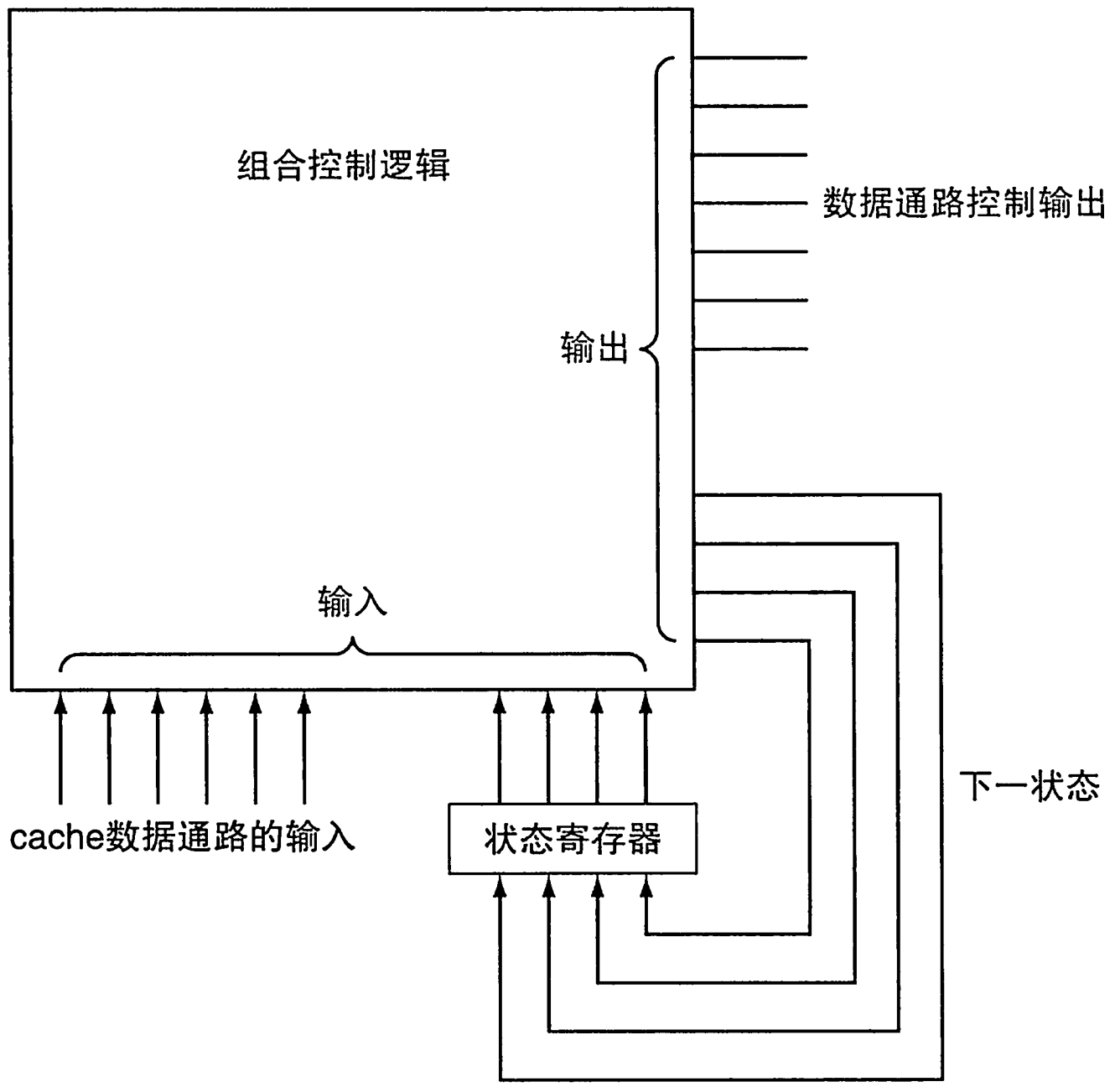

图 5-38　有限状态自动机控制器的典型实现，使用一组组合逻辑和一个保存当前状态的寄存器实现。组合逻辑的输出是当前状态的下一状态编号和被断言的控制信号。组合逻辑的输入是当前状态和任一可以决定下一状态的输入。需要注意的是，在本章的有限状态自动机中，输出只取决于当前状态，而与输入无关。我们用灰色线来表示这些控制线和逻辑以及数据线和逻辑。随后的详细阐述对此有更详细的解释

详细阐述　请注意，这个简单的设计被称为阻塞 cache，在该设计中处理器必须一直等待直到 cache 完成请求。5.12 节描述了一种被称为非阻塞 cache 的替代方案。

详细阐述　本书中描述的这种有限状态自动机被称为 Moore 自动机，以 Edward Moore 的名字命名。这种自动机的标识特征是它的输出只取决于当前状态。对 Moore 自动机而言，被标记为组合控制逻辑的盒子可以被分为两个部分，其中一部分包含控制输出并且仅有状态作为输入，另一部分只包含下一状态输出。

另一种自动机是 Mealy 自动机，以 George Mealy 的名字命名。Mealy 自动机同时使用输入和当前状态来决定输出。Moore 自动机在速度和控制单元规模上具有潜在实现优势：Moore 自动机的速度优势在于控制输出部分，该部分在时钟周期开始时就需要用到，而该部分只取决于当前状态而与输入无关，因此具有速度优势。在附录 A 中，使用逻辑门就可以实现这种有限状态自动机，因此可以很明显地看出它的规模优势。Moore 自动机的潜在劣势是需要额外的状态。例如，在两个状态序列中仅有一个状态不同的情况下，Mealy 自动机会使用输出依赖输入的方式将状态统一。

5.9.3　使用有限状态自动机作为简单的 cache 控制器

图 5-39 描述了简单 cache 控制器的四个状态。

- 空闲：该状态等待处理器发出的有效读或写信号，之后有限状态自动机跳转到标签比较状态。