第八章：目录一致性协议

在本章中，我们介绍目录一致性协议 (directory coherence protocol)。 最初开发目录协议是为了解决监听协议缺乏可扩展性的问题。 传统的监听系统在一个完全有序的互连网络上广播所有请求，并且所有请求都被所有一致性控制器监听。 相比之下，目录协议使用一定程度的间接性来避免有序广播网络和让每个缓存控制器处理每个请求。

我们首先在高层次介绍目录协议（第 8.1 节）。 然后，我们展示了一个具有完整但简单的三态 (MSI) 目录协议的系统（第 8.2 节）。 该系统和协议作为我们稍后添加系统功能和协议优化的基准。 然后我们解释如何将独占状态（第 8.3 节）和拥有状态（第 8.4 节）添加到基准 MSI 协议。 接下来我们讨论如何表示目录状态（第 8.5 节）以及如何设计和实现目录本身（第 8.6 节）。 然后，我们描述了提高性能和降低实现成本的技术（第 8.7 节）。 然后，在讨论目录协议及其未来（第 8.9 节）结束本章之前，我们将讨论具有目录协议的商用系统（第 8.8 节）。

那些满足于学习目录一致性协议基础知识的读者可以略读或跳过第 8.3 节到第 8.7 节，尽管这些部分中的一些材料将帮助读者更好地理解第 8.8 节中的案例研究。

8.1 目录协议简介

目录协议的关键创新是建立一个目录，维护每个块的一致性状态的全局视图。 目录跟踪哪些缓存保存每个块以及处于什么状态。 想要发出一致性请求（例如，GetS）的缓存控制器将其直接发送到目录（即，单播消息），并且目录查找块的状态以确定接下来要采取的操作。 例如，目录状态可能表明请求的块由核心 C2 的缓存拥有，因此应将请求转发 (forward) 到 C2（例如，使用新的 Fwd-GetS 请求）以获取块的副本。 当 C2 的缓存控制器接收到这个转发的请求时，它会向请求的缓存控制器单播响应。

比较目录协议和监听协议的基本操作是有启发性的。在目录协议中，目录维护每个块的状态，缓存控制器将所有请求发送到目录。目录要么响应请求，要么将请求转发给一个或多个其他一致性控制器然后响应。一致性事务通常涉及两个步骤（单播请求，随后是单播响应）或三个步骤（单播请求，K >= 1 个转发请求和 K 个响应，其中 K 是共享者的数量）。一些协议甚至还有第四步，因为响应是通过目录间接进行的，或者因为请求者在事务完成时通知目录。相比之下，监听协议将块的状态分布在可能的所有一致性控制器上。因为没有对这种分布式状态的中央总结，所以必须将一致性请求广播到所有一致性控制器。因此，监听一致性事务总是涉及两个步骤（广播请求，随后是单播响应）。

与监听协议一样，目录协议需要定义一致性事务何时以及如何相对于其他事务进行排序。在大多数目录协议中，一致性事务是在目录中排序的。多个一致性控制器可以同时向目录发送一致性请求，事务顺序由请求在目录中的序列化顺序决定。如果两个请求竞争 (race) 到目录，互连网络有效地选择目录将首先处理哪个请求。第二个到达的请求的命运取决于目录协议和竞争的请求类型。第二个请求可能会（a）在第一个请求之后立即处理，（b）在等待第一个请求完成时保留在目录中，或者（c）否定确认 (negatively acknowledged, NACKed)。在后一种情况下，目录向请求者发送否定确认消息 (NACK)，请求者必须重新发出其请求。在本章中，我们不考虑使用 NACK 的协议，但我们会在第 9.3.2 节讨论 NACK 的可能使用以及它们如何导致活锁 (livelock) 问题。

使用目录作为排序点代表了目录协议和监听协议之间的另一个关键区别。 传统的监听协议通过序列化有序广播网络上的所有事物来创建总顺序。 监听的总顺序不仅可以确保每个块的请求按块顺序处理，而且还有助于实现内存连贯性模型 (memory consistency model)。 回想一下，传统的监听协议使用完全有序的广播来序列化所有请求； 因此，当请求者观察到其自己的一致性请求时，这将作为其一致性时期可能开始的通知。 特别是，当一个监听控制器看到自己的 GetM 请求时，它可以推断出其他缓存将使其 S 块无效。 我们在表 7.4 中证明了这个序列化通知足以支持强 SC 和 TSO 内存连贯性模型。

相反，目录协议对目录中的事务进行排序，以确保所有节点按块顺序处理冲突请求。 然而，缺少总顺序意味着目录协议中的请求者需要另一种策略来确定其请求何时被序列化，从而确定其一致性时期何时可以安全开始。 因为（大多数）目录协议不使用完全有序的广播，所以没有序列化的全局概念。 相反，必须针对（可能）具有块副本的所有缓存单独序列化请求。 需要显式消息来通知请求者其请求已被每个相关缓存序列化。 特别是，在 GetM 请求中，每个具有共享 (S) 副本的缓存控制器必须在序列化失效消息后发送显式确认 (Ack) 消息。

目录和监听协议之间的这种比较突出了它们之间的基本权衡。 目录协议以间接级别（即，对于某些事务具有三个步骤，而不是两个步骤）为代价实现了更大的可扩展性（即，因为它需要更少的带宽）。 这种额外的间接级别增加了一些一致性事务的延迟。

8.2 基准目录系统

在本节中，我们将介绍一个带有简单、适度优化的目录协议的基准系统。 该系统提供了对目录协议关键特性的深入了解，同时揭示了其低效率以促使本章的后续部分介绍特性和优化。

8.2.1 目录系统模型

我们在图 8.1 中说明了我们的目录系统模型。 与监听协议不同的是，互连网络的拓扑是有意模糊的。 它可以是 mesh、torus 或架构师希望使用的任何其他拓扑。 我们在本章中假设的互连网络的一个限制是它强制执行点对点排序。 也就是说，如果控制器 A 向控制器 B 发送两条消息，则这些消息以与发送它们相同的顺序到达控制器 B。（注1） 点对点排序降低了协议的复杂性，我们将没有排序的网络讨论推迟到第 8.7.3 节。

原书作者注1：严格来说，我们只需要某些类型的消息的点对点顺序，但这是我们推迟到第 8.7.3 节的细节。

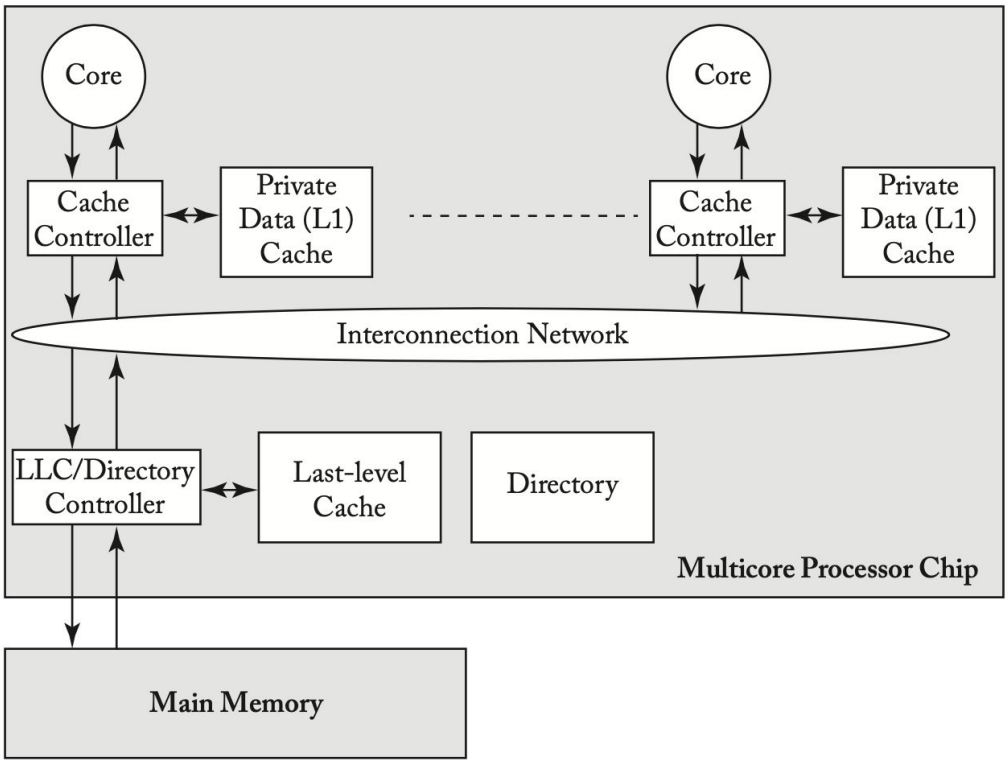

Figure 8.1: Directory system model.

此目录系统模型与图 2.1 中的基准系统模型之间的唯一区别是我们添加了一个目录并将内存控制器重命名为目录控制器 (directory controller)。 有很多方法来调整和组织目录的大小，现在我们假设最简单的模型：对于内存中的每个块，都有一个对应的目录条目。 在 8.6 节中，我们检查和比较了更实用的目录组织选项。 我们还假设一个具有单个目录控制器的单片 LLC；在第 8.7.1 节中，我们解释了如何在 LLC 的多个 bank 和多个目录控制器之间分配此功能。

8.2.2 高层次协议规范

基准目录协议只有三个稳定状态：MSI。 除非块处于状态 M 的缓存中，否则块由目录控制器拥有。每个块的目录状态包括稳定的一致性状态、所有者的身份（如果块处于状态 M）和 共享者编码为 one-hot 位向量（如果块处于状态 S）。 我们在图 8.2 中说明了一个目录条目。 在 8.5 节中，我们将讨论目录条目的其他编码。

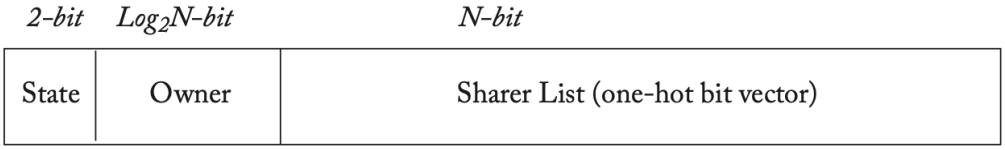

Figure 8.2: Directory entry for a block in a system with N nodes.

在介绍详细规范之前，我们首先说明协议的更高层次的抽象，以了解其基本行为。 在图 8.3 中，我们展示了缓存控制器发出一致性请求以将权限从 I 更改为 S、I 或 S 更改为 M、M 更改为 I 以及 S 更改为 I 的事务。与上一章中的监听协议一样，我们使用以缓存为中心的符号指定块的目录状态（例如，目录状态 M 表示存在一个缓存，该块处于状态 M）。 请注意，缓存控制器可能不会静默地驱逐共享块；也就是说，有一个明确的 PutS 请求。 我们将讨论具有静默驱逐共享块的协议，以及静默与显式 PutS 请求的比较，直到第 8.7.4 节。

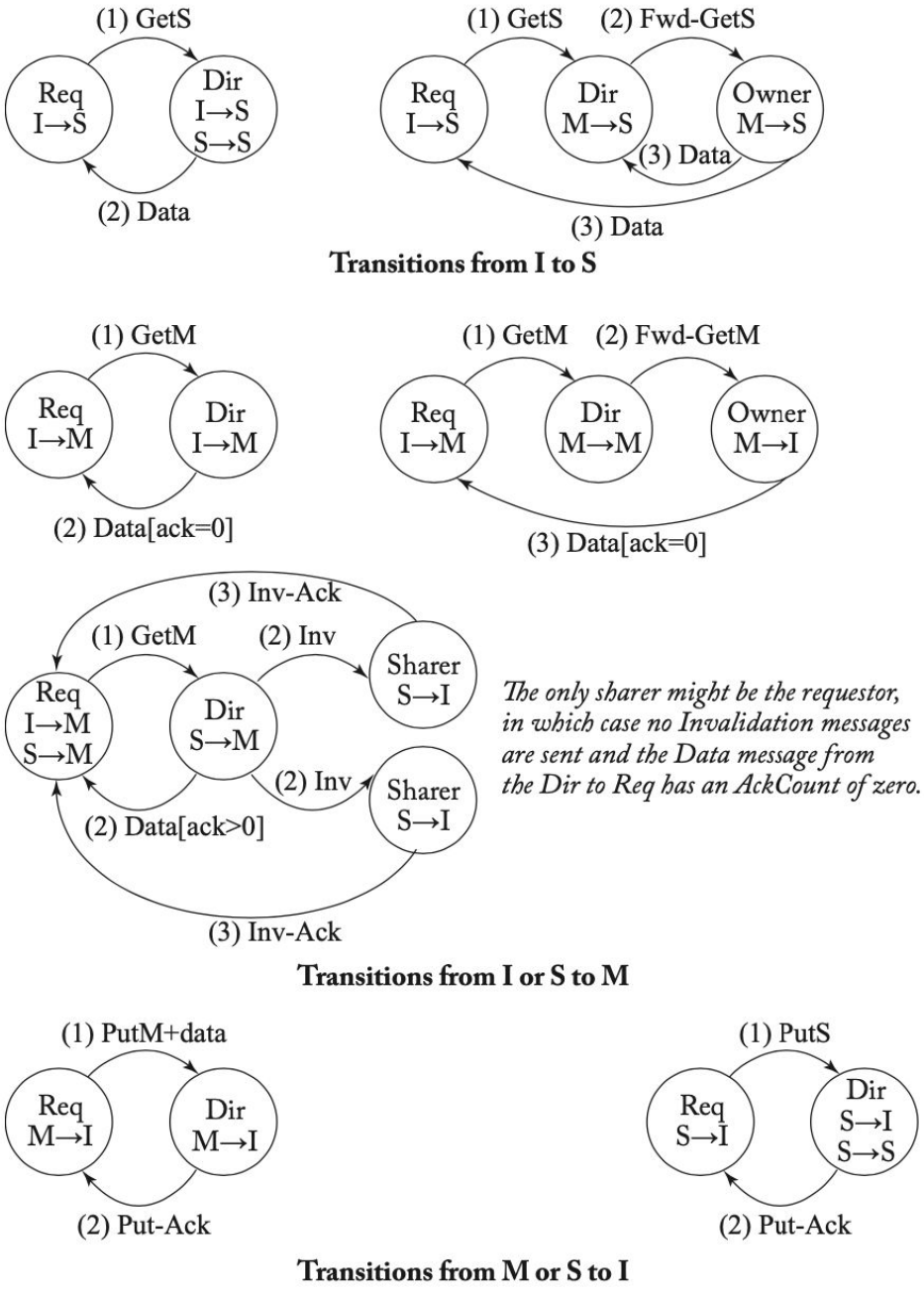

Figure 8.3: High-level description of MSI directory protocol. In each transition, the cache controller that requests the transaction is denoted "Req."

大多数事务都相当简单，但有两个事务值得在这里进一步讨论。第一个是当缓存试图将权限从 I 或 S 升级到 M 并且目录状态为 S 时发生的事务。缓存控制器向目录发送 GetM，目录执行两个操作。首先，它用包含数据和"AckCount"的消息响应请求者；AckCount 是块当前共享者的数量。目录将 AckCount 发送给请求者，以通知请求者有多少共享者必须确认已使他们的块无效以响应 GetM。其次，目录向所有当前共享者发送无效 (Inv) 消息。每个共享者在收到 Invalidation 后，会向请求者发送 Invalidation-Ack (Inv-Ack)。一旦请求者收到来自目录的消息和所有 Inv-Ack 消息，它就完成了事务。收到所有 Inv-Ack 消息的请求者知道该块不再有任何读者，因此它可以在不违反一致性的情况下写入该块。

值得进一步讨论的第二个事务发生在缓存试图驱逐处于状态 M 的块时。在此协议中，我们让缓存控制器向目录发送包含数据的 PutM 消息。 该目录以 Put-Ack 响应。 如果 PutM 没有携带数据，那么协议将需要在 PutM 事务中发送第三条消息——从缓存控制器到目录的数据消息，被逐出的块已处于状态 M。 此目录协议中的 PutM 事务与监听协议中发生的不同，其中 PutM 不携带数据。

8.2.3 避免死锁

在这个协议中，消息的接收会导致一致性控制器发送另一个消息。一般来说，如果事件 A（例如，消息接收）可以导致事件 B（例如，消息发送）并且这两个事件都需要资源分配（例如，网络链接和缓冲区），那么我们必须小心避免可能发生的死锁，如果出现循环资源依赖。例如，GetS 请求会导致目录控制器发出 Fwd-GetS 消息；如果这些消息使用相同的资源（例如，网络链接和缓冲区），那么系统可能会死锁。在图 8.4 中，我们说明了一个死锁，其中两个一致性控制器 C1 和 C2 正在响应彼此的请求，但传入队列已经充满了其他一致性请求。如果队列是 FIFO，则响应无法通过请求。由于队列已满，每个控制器都会停止尝试发送响应。因为队列是先进先出的，控制器无法切换到处理后续请求（或获取响应）。因此，系统死锁。

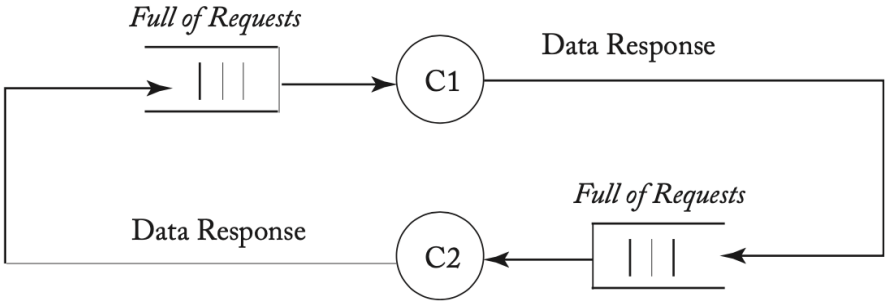

Figure 8.4: Deadlock example.

避免一致性协议中的死锁的一个众所周知的解决方案是为每类消息使用单独的网络。 网络可以在物理上分离或在逻辑上分离（称为虚拟网络），但关键是避免消息类别之间的依赖关系。 图 8.5 说明了一个系统，其中请求和响应消息在不同的物理网络上传播。 因为一个响应不能被另一个请求阻塞，它最终会被它的目的节点消费，打破了循环依赖。

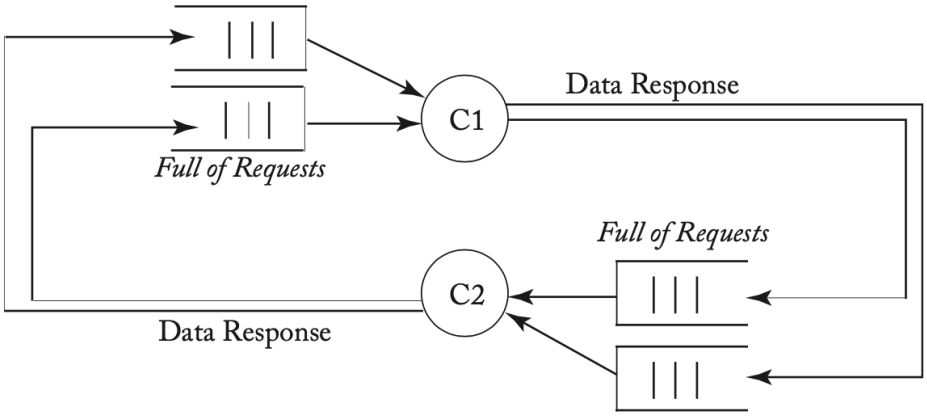

Figure 8.5: Avoiding deadlock with separate networks.

本节中的目录协议使用三个网络来避免死锁。 因为请求可以导致转发 (forward) 请求，而转发请求可以导致响应，所以存在三个消息类别，每个类别都需要自己的网络。 请求消息是 GetS、GetM 和 PutM。转发的请求消息是 Fwd-GetS、Fwd-GetM、Inv (Invalidation) 和 Put-Ack。 响应消息是 Data 和 Inv-Ack。 本章中的协议要求转发请求网络提供点对点排序； 其他网络没有排序约束，在不同网络上传输的消息之间也没有任何排序约束。

我们推迟到第 9.3 节对避免死锁进行更彻底的讨论，包括对虚拟网络的更多解释和避免死锁的确切要求。

8.2.4 详细的协议规范

我们在表 8.1 和 8.2 中提供了详细的协议规范，包括所有瞬间状态。与第 8.2.2 节中的高层次描述相比，最显着的区别是瞬间状态。一致性控制器必须管理处于一致性事务中的块的状态，包括缓存控制器在将其一致性请求发送到目录和接收其所有必要的响应消息之间接收来自另一个控制器的转发请求的情况，包括数据和可能的 Inv-Ack。缓存控制器可以在未命中状态处理寄存器 (miss status handling register, MSHR) 中维护此状态，核心使用该寄存器跟踪未完成的一致性请求。在符号上，我们以 XY^AD 的形式

表示这些瞬间状态，其中上标 A 表示等待确认，上标 D 表示等待数据。（这种表示法不同于监听协议，其中上标 A 表示等待请求出现在总线上。）

Table 8.1: MSI directory protocol—cache controller

	Load	Store	Replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	Send GetS to Dir/IS^D	Send GetM to Dir/IM^{AD}										
IS^D	Stall	Stall	Stall			Stall		-/S		-/S		
IM^{AD}	Stall	Stall	Stall	Stall	Stall			-/M	-/IM^A	-/M	ack--	
IM^A	Stall	Stall	Stall	Stall	Stall						ack--	-/M
S	Hit	Send GetM to Dir/SM^{AD}	Send PutS to Dir/SI^A			Send Inv-Ack to Req/I						
SM^{AD}	Hit	Stall	Stall	Stall	Stall	Send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A		ack--	
SM^A	Hit	Stall	Stall	Stall	Stall						ack--	-/M
M	Hit	Hit	Send PutM +data to Dir/MI^A	Send data to Req and Dir/S	Send data to Req/I							
MI^A	Stall	Stall	Stall	Send data to Req and Dir/SI^A	Send data to Req/II^A		-/I					
SI^A	Stall	Stall	Stall			Send Inv-Ack to Req/II^A	-/I					
II^A	Stall	Stall	Stall				-/I					

Table 8.2: MSI directory protocol—directory controller

	GetS	GetM	PutS-NotLast	PutS-Last	Put M+ data from Owner	PutM+data from Non-Owner	Data
I	Send data to Req, add Req to Sharers/S	Send data to Req, set Owner to Req/M	Send Put-Ack to Req	Send Put-Ack to Req		Send Put-Ack to Req	
S	Send data to Req, add Req to Sharers	Send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	Remove Req from Sharers, sent Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req/I		Remove Req from sharers, send Put-Ack to Req	
M	Send Fwd-GetS to Owner, add Req and Owner to Sharers, clear Owner/S^D	Send Fwd-GetM to Owner, set Owner to Req	Send Put-Ack to Req	Send Put-Ack to Req	Copy data to memory, clear Owner, send Put-Ack to Req/I	Send Put-Ack to Req	
S^D	Stall	Stall	Remove Req from Sharers, send Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req		Remove Req from Sharers, send Put-Ack to Req	Copy data to memory/S

因为这些表格乍一看可能有些令人生畏，所以下一节将介绍一些示例场景。

8.2.5 协议操作

该协议使缓存能够获取处于状态 S 和 M 的块，并将块替换到处于这两种状态中的目录中。

I to S (common case #1)

缓存控制器向目录发送 GetS 请求并将块状态从 I 更改为 IS^D。 目录接收到这个请求，如果该目录是所有者（即当前没有缓存在 M 中拥有该块），则该目录以 Data 消息响应，将块的状态更改为 S（如果它已经不是 S），并且 将请求者添加到共享者列表 (sharer list)。 当数据到达请求者时，缓存控制器将块的状态更改为 S，完成事务。

I to S (common case #2)

缓存控制器向目录发送 GetS 请求并将块状态从 I 更改为 IS^D。 如果目录不是所有者（即存在当前在 M 中具有块的缓存），则目录将请求转发给所有者并将块的状态更改为瞬态 S^D。 所有者通过向请求者发送 Data 并将块的状态更改为 S 来响应此 Fwd-GetS 消息。现在的先前所有者还必须将 Data 发送到目录，因为它正在放弃对目录的所有权，该目录必须具有块的最新副本。 当数据到达请求者时，缓存控制器将块状态更改为 S 并认为事务完成。 当 Data 到达目录时，目录将其复制到内存中，将块状态更改为 S，并认为事务完成。

I to S (race cases)

上述两种 I-to-S 场景代表了常见的情况，其中正在进行的块只有一个事务。 该协议的大部分复杂性源于必须处理一个块的多个正在进行的事务的不太常见的情况。 例如，读者可能会惊讶于缓存控制器可以接收到状态 IS^D 的块的无效 (Invalidation)。 考虑发出 GetS 并转到 IS^D 的核心 C1 和另一个核心 C2，它为在 C1 的 GetS 之后到达目录的同一块发出 GetM。 该目录首先发送 C1 Data 以响应其 GetS，然后发送 Invalidation 以响应 C2 的 GetM。 由于 Data 和 Invalidation 在不同的网络上传输，它们可能会乱序到达，因此 C1 可以在数据之前接收失效。

I or S to M

缓存控制器向目录发送 GetM 请求并将块的状态从 I 更改为 IM^AD。在这种状态下，缓存等待 Data 和（可能的）Inv-Ack，表明其他缓存已经使它们在状态 S 中的块副本无效。缓存控制器知道需要多少 Inv-Ack，因为 Data 消息包含 AckCount，可能为零。图 8.3 说明了目录响应 GetM 请求的三种常见情况。如果目录处于状态 I，它只需发送 AckCount 为零的数据并进入状态 M。如果处于状态 M，目录控制器将请求转发给所有者并更新块的所有者；现在以前的所有者通过发送 AckCount 为零的 Data 来响应 Fwd-GetM 请求。最后一种常见情况发生在目录处于状态 S 时。目录以 Data 和等于共享者数量的 AckCount 进行响应，另外它向共享者列表中的每个核心发送 Invalidation。接收 Invalidation 消息的缓存控制器使其共享副本无效并将 Inv-Ack 发送给请求者。当请求者收到最后一个 Inv-Ack 时，它转换到状态 M。注意表 8.1 中的特殊 Last-Inv-Ack 事件，它简化了协议规范。

这些常见情况忽略了一些突出目录协议并发性的可能竞争。 例如，核心 C1 的缓存块处于 IM^A 状态，并从 C2 的缓存控制器接收 Fwd-GetS。 这种情况是可能的，因为目录已经向 C1 发送了 Data，向共享者发送了 Invalidation 消息，并将其状态更改为 M。当 C2 的 GetS 到达目录时，目录将其简单地转发给所有者 C1。 这个 Fwd-GetS 可能在所有 Inv-Ack 到达 C1 之前到达 C1。 在这种情况下，我们的协议只是停止并且缓存控制器等待 Inv-Ack。 因为 Inv-Ack 在单独的网络上传输，所以保证它们不会阻塞在未处理的 Fwd-GetS 后面。

M to I

为了驱逐处于状态 M 的块，缓存控制器发送一个包含数据的 PutM 请求并将块状态更改为 MI^A。 当目录接收到这个 PutM 时，它会更新 LLC/内存，以 Put-Ack 响应，并转换到状态 I。在请求者收到 Put-Ack 之前，块的状态保持有效 M 并且缓存控制器必须响应转发块的一致性请求。 如果缓存控制器在发送 PutM 和接收 Put-Ack 之间收到转发的一致性请求（Fwd-GetS 或 Fwd-GetM），缓存控制器响应 Fwd-

GetS 或 Fwd-GetM 并更改其块状态分别到 SI^A 或 II^A。 这些瞬间状态实际上分别是 S 和 I，但表示缓存控制器必须等待 Put-Ack 完成到 I 的转换。

S to I

与前一章中的监听协议不同，我们的目录协议不会静默地驱逐状态 S 中的块。相反，为了替换状态 S 中的块，缓存控制器发送 PutS 请求并将块状态更改为 SI^A。 目录接收此 PutS 并以 Put-Ack 响应。 在请求者收到 Put-Ack 之前，block 的状态实际上是 S。如果缓存控制器在发送 PutS 之后并且在收到 Put-Ack 之前收到 Invalidation 请求，它会将 block 的状态更改为 II^A。 这种瞬间状态实际上是 I，但它表示缓存控制器必须等待 Put-Ack 完成从 S 到 I 的事务。

8.2.6 协议简化

该协议相对简单，并牺牲了一些性能来实现这种简单性。 我们现在讨论两个简化：

除了只有三个稳定状态之外，最重要的简化是协议在某些情况下停止。 例如，缓存控制器在处于瞬间状态时收到转发的请求时会停止。 在第 8.7.2 节中讨论的更高性能选项是处理消息并添加更多瞬间状态。第二个简化是目录发送 Data（和 AckCount）以响应将块状态从 S 更改为 M 的缓存。缓存已经有有效数据，因此目录只需发送 data-less AckCount 就足够了。 我们推迟添加这种新类型的消息，直到我们在第 8.4 节中介绍 MOSI 协议。

8.3 添加独占状态

8.4 添加拥有状态

8.5 代表目录状态

8.6 目录组织

从逻辑上讲，该目录包含每个内存块的单个条目。 许多传统的基于目录的系统（其中目录控制器与内存控制器集成）通过增加内存来保存目录来直接实现这种逻辑抽象。 例如，SGI Origin 添加了额外的 DRAM 芯片来存储每个内存块的完整目录状态 [10]。

然而，对于当今的多核处理器和大型 LLC，传统的目录设计已经没有什么意义了。 首先，架构师不希望目录访问片外存储器时产生延迟和功耗，特别是对于缓存在芯片上的数据。 其次，当几乎所有内存块在任何给定时间都没有被缓存时，系统设计人员会对大型片外目录状态犹豫不决。 这些缺点促使架构师通过在芯片上仅缓存目录条目的子集来优化常见情况。 在本节的其余部分中，我们将讨论目录缓存设计，其中一些设计先前已由 Marty 和 Hill [13] 分类。

与传统的指令和数据缓存一样，目录缓存 [7] 提供对完整目录状态的子集的更快访问。 因为目录总结了一致性缓存的状态，所以它们表现出类似于指令和数据访问的局部性，但只需要存储每个块的一致性状态而不是其数据。 因此，相对较小的目录高速缓存可实现较高的命中率。 目录缓存对一致性协议的功能没有影响；它只是减少了平均目录访问延迟。 在多核处理器时代，目录缓存变得更加重要。 在内核驻留在单独的芯片和/或板上的旧系统中，消息延迟足够长，以至于它们倾向于分摊目录访问延迟。 在多核处理器中，消息可以在几个周期内从一个内核传输到另一个内核，并且片外目录访问的延迟往往会使通信延迟相形见绌并成为瓶颈。 因此，对于多核处理器，有强烈的动机来实现片上目录缓存以避免昂贵的片外访问。

片上目录高速缓存包含完整目录条目集的子集。 因此，关键的设计问题是处理目录高速缓存未命中，即当针对其目录条目不在目录高速缓存中的块的一致性请求到达时。

我们总结了表 8.7 中的设计选项，并在接下来进行描述。

Table 8.7: Comparing directory cache designs

		Inclusive Directory Caches (Section 8.6.2)				
	Directory Cache Backed by DRAM Directory (Section 8.6.1)	**Inclusive Directory Cache Embedded in Inclusive LLC (Section 8.6.2)**		**Standalone Inclusive Directory Cache (Section 8.6.2)**		**Null Directory Cache (Section 8.6.3)**
		No Recalls	**With Recalls**	**No Recalls**	**With Recalls**	
Directory location	DRAM	LLC		LLC		None
Uses DRAM	Yes	No		No		No
Miss at directory implies	Must access DRAM	Block must be I		Block must be I		Block could be in any state→ must broadcast
Inclusion requirements	None	LLC includes L1s		Directory cache includes L1s		None
Implementation costs	DRAM plus separate on-chip cache	Larger LLC blocks; highly associative LLC	Larger LLC blocks	Highly associative storage for redundant tags	Storage for redundant tags	None
Replacement notification	None	None	Desirable	Required	Desirable	None

8.6.1 以 DRAM 为后备的目录缓存

8.6.2 Inclusive 目录缓存

8.6.3 空目录缓存（没有后备存储）

8.6.4 对于 State S 块的静默式 vs. 非静默式驱逐

8.7 性能和可扩展性优化

在本节中，我们将讨论提高目录协议性能和可扩展性的几种优化。

8.7.1 分布式目录

到目前为止，我们假设有一个目录附加到单个整体 LLC。 这种设计显然有可能在这个共享的中央资源上造成性能瓶颈。 集中式瓶颈问题的典型、通用解决方案是分配资源。 给定块的目录仍然固定在一处，但不同的块可以有不同的目录。

在较旧的具有 N 个节点的多芯片多处理器中（每个节点由多个芯片组成，包括处理器核心和内存），每个节点通常具有与其关联的内存的 1/N 以及相应的目录状态的 1/N 。

我们在图 8.11 中说明了这样一个系统模型。 节点的内存地址分配通常是静态的，并且通常可以使用简单的算术轻松计算。 例如，在具有 N 个目录的系统中，块 B 的目录项可能位于目录 B 模 N 处。每个块都有一个主目录，即保存其内存和目录状态的目录。 因此，我们最终得到一个系统，其中有多个独立的目录管理不同块集的一致性。 与要求所有一致性流量通过单个中央资源相比，拥有多个目录可以提供更大的一致性事务带宽。 重要的是，分发目录对一致性协议没有影响。

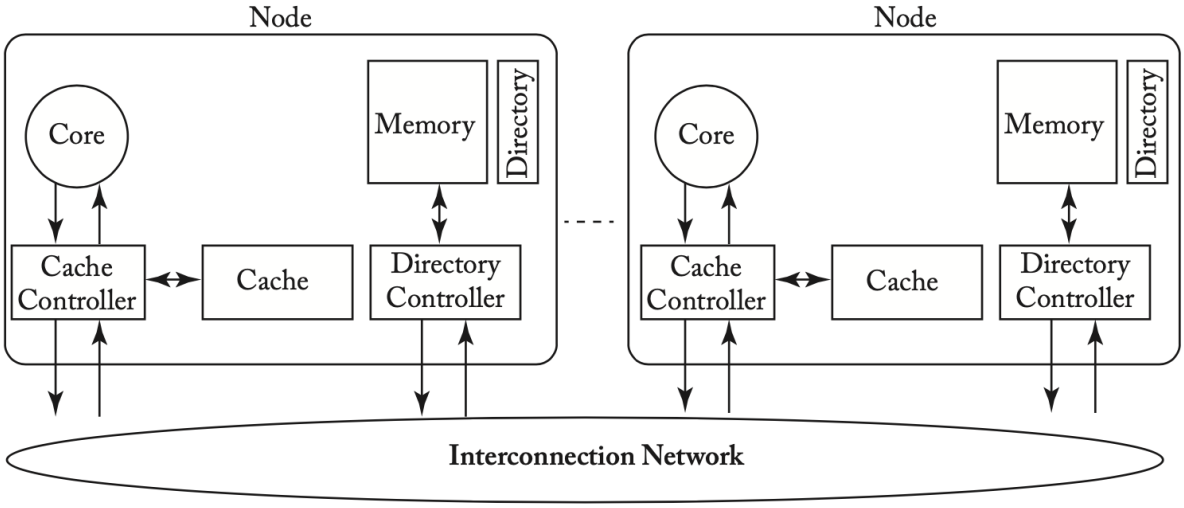

Figure 8.11: Multiprocessor system model with distributed directory.

在当今具有大型 LLC 和目录缓存的多核处理器世界中，分发目录的方法在逻辑上与传统多芯片多处理器中的方法相同。 我们可以分发（存储）LLC 和目录缓存。 每个块都有一个 LLC 主库及其关联的目录缓存库。

8.7.2 Non-Stalling 目录协议

8.7.3 无需点对点 ordering 的互连网络

8.8 案例研究

8.9 目录协议的讨论和未来

8.10 参考文献

[1] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In Proc. of the International Parallel and Distributed Processing Symposium, 2003. DOI: 10.1109/ipdps.2003.1213087. 189

[2] A.Agarwal,R.Simoni,M.Horowitz,andJ.Hennessy.Anevaluationofdirectoryschemes for cache coherence. In Proc. of the 15th Annual International Symposium on Computer Architecture, pp. 280–89, May 1988. DOI: 10.1109/isca.1988.5238. 171

[3] J. K. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In Proc. of the 11th Annual International Symposium on Computer Architecture, pp. 355–62, June 1984. DOI: 10.1145/800015.808205. 171

[4] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In Proc. of the 15th Annual International Symposium on Computer Architecture, pp. 73–80, May 1988. DOI: 10.1109/isca.1988.5212. 174

[5] P. Conway and B. Hughes. The AMD Opteron northbridge architecture. IEEE Micro, 27(2):10–21, March/April 2007. DOI: 10.1109/mm.2007.43. 185, 186

[6] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. IEEE Micro, 30(2):16–29, March/April 2010. DOI: 10.1109/mm.2010.31. 172, 176, 186, 187

[7] A. Gupta, W.-D. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In Proc. of the International Conference on Parallel Processing, 1990. DOI: 10.1007/978-1-4615-3604-8_9. 172

[8] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. ACM Transactions on Computer Systems, 11(4):300–18, November 1993. DOI: 10.1145/161541.161544. 171

[9] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. Document Number 320412–001US, January 2009. 187

[10] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In Proc. of the 24th Annual International Symposium on Computer Architecture, pp. 241–51, June 1997. DOI: 10.1145/264107.264206. 162, 172, 183

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. IEEE Computer, 25(3):63– 79, March 1992. DOI: 10.1109/2.121510. 184

[12] R. A. Maddox, G. Singh, and R. J. Safranek. Weaving High Performance Multiprocessor Fabric: Architecture Insights into the Intel QuickPath Interconnect. Intel Press, 2009. 187

[13] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In Proc. of the 34th Annual International Symposium on Computer Architecture, June 2007. DOI: 10.1145/1250662.1250670. 172

[14] S. L. Scott. Synchronization and communication in the Cray T3E multiprocessor. In Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 26–36, October 1996. 189