

第七章：监听一致性协议

在本章中，我们介绍了监听一致性协议 (snooping coherence protocol)。监听协议是第一个广泛部署的协议类别，并且它们继续在各种系统中使用。监听协议提供了许多吸引人的特性，包括低延迟一致性事务和比替代目录协议（第 8 章）概念上更简单的设计。

我们首先在高层次上介绍监听协议（第 7.1 节）。然后，我们展示了一个简单的系统，它具有完整但简单的三态 (MSI) 监听协议（第 7.2 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。我们讨论的协议优化包括独占状态（第 7.3 节）和拥有状态（第 7.4 节）的添加，以及更高性能的互连网络（第 7.5 节和 7.6 节）。然后，我们讨论带有监听协议的商用系统（第 7.7 节），然后以讨论监听及其未来（第 7.8 节）结束本章。

鉴于一些读者可能不希望深入研究监听，我们对本章进行了组织，以便读者可以跳过或跳过第 7.3-7.6 节，如果他们愿意的话。

7.1 监听简介

监听协议基于一个想法：所有一致性控制器以相同的顺序观察（监听）一致性请求，并集体“做正确的事”以保持一致性。通过要求对给定块的所有请求按顺序到达，监听系统使分布式一致性控制器能够正确更新共同表示缓存块状态的有限状态机。

传统的监听协议将请求广播到所有一致性控制器，包括发起请求的控制器。一致性请求通常在有序的广播网络上传播，例如总线。有序广播确保每个一致性控制器以相同顺序观察相同系列的一致性请求，即存在一致性请求的总顺序 (total order)。由于总顺序包含所有每个块的顺序，因此该总顺序保证所有一致性控制器都可以正确更新缓存块的状态。

为了说明以相同的每个块顺序 (per-block order) 处理一致性请求的重要性，请考虑表 7.1 和 7.2 中的示例，其中核心 C1 和核心 C2 都希望在状态 M 中获得相同的块 A。在表 7.1 中，所有三个一致性控制器观察一致性请求的相同的每个块顺序，并共同维护单写多读 (SWMR) 不变量。块的所有权从 LLC/内存到核心 C1 再到核心 C2。作为每个观察到的请求的结果，每个一致性控制器独立地得出关于块状态的正确结论。相反，表 7.2 说明了如果核心 C2 观察到与核心 C1 和 LLC/内存不同的每个块的请求顺序，可能会出现不一致性。首先，我们遇到核心 C1 和核心 C2 同时处于状态 M 的情况，这违反了 SWMR 不变量。接下来，我们有一种情况，没有一致性控制器认为它是所有者，因此此时的一致性请求不会收到响应（可能导致死锁）。

Table 7.1: Snooping coherence example. All activity involves block A (denoted “A.”).

Time	Core C1	Core C2	LLC/Memory
0	A:I	A:I	A:I (LLC/memory is owner)
1	A:GetM from Core C1/M	A:GetM from Core C1/I	A:GetM from Core C1/M (LLC/memory is not owner)
2	A:GetM from Core C2/I	A:GetM from Core C2/M	A:GetM from Core C2/M

Table 7.2: Snooping (In)coherence example. All activity involves block A (denoted “A.”).

Time	Core C1	Core C2	LLC/Memory
0	A:I	A:I	A:I (LLC/memory is owner)
1	A:GetM from Core C1/M	A:GetM from Core C2/M	A:GetM from Core C1/M (LLC/memory is not owner)
2	A:GetM from Core C2/I	A:GetM from Core C1/I	A:GetM from Core C2/M

传统的监听协议在所有块中创建了一个总的一致性请求顺序，即使一致性只需要每个块的请求顺序。具有总顺序可以更容易地实现需要内存引用的总顺序的内存连贯性模型 (memory consistency model)，例如 SC 和 TSO。考虑表 7.3 中涉及两个块 A 和 B 的示例；每个块只被请求一次，因此系统很容易观察每个块的请求顺序。然而，由于核心 C1 和 C2 观察到 GetM 和 GetS 请求乱序，因此该执行违反了 SC 和 TSO 内存连贯性模型。

Table 7.3: Per-block order, coherence, and consistency. States and operations that pertain to address A are preceded by the prefix “A:”，and we denote a block A in state X with value V as “A:X[V].” If the value is stale, we omit it (e.g., “A:I”). Note that there are two blocks in this example, A and B, with A initially in state S at Core C2 and B initially in state M at Core C1.

Time	Core C1	Core C2	LLC/Memory
0	A:I B:M[0]	A:S[0] B:I	A:S[0] B:M
1	A:GetM from Core C1/M[0] store A = 1 B:M[0]	A:S[0] B:I	A:S[0] B:M
2	A:M[1] store B = 1 B:M[1]	A:S[0] B:I	A:GetM from Core C1/M B:M
3	A:M[1] B:GetS from Core C2/S[1]	A:S[0] B:I	A:M B:GetS from Core C2/S[1]
4	A:M[1] B:S[1]	A:S[0] B:GetS from Core C2/S[1] r1 = B[1]	A:M B:S[1]
5	A:M[1] B:S[1]	A:S[0] r2 = A[0] B:S[1]	A:M B:S[1]
6	A:M[1] B:S[1]	A:GetM from Core1/I B:S[1]	A:M B:S[1]
r1 = 1, r2 = 0 violates SC and TSO			

我们在侧边栏中讨论了一些关于需要总顺序的微妙问题。

原书侧边栏：监听如何依赖于一致性请求的总顺序

乍一看，读者可能会认为表 7.3 中的问题的出现是因为在周期 1 中块 A 的 SWMR 不变量被违反，因为 C1 有一个 M 副本，而 C2 仍然有一个 S 副本。但是，表 7.4 说明了相同的示例，但强制执行一致性请求的总顺序。此示例在第 4 周期之前是相同的，因此具有相同的明显 SWMR 违规。然而，就像众所周知的“森林中的树”一样，这种违规行为不会引起问题，因为它没有被观察到（即“没有人听到它”）。具体来说，因为核心以相同的顺序看到两个请求，所以 C2 在看到块 B 的新值之前使块 A 无效。因此，当 C2 读取块 A 时，它必须获取新值，因此产生正确的 SC 和 TSO 执行。

传统的监听协议使用一致性请求的总顺序来确定何时在基于监听顺序的逻辑时间内观察到特定请求。在表 7.4 的例子中，由于总顺序，核心 C1 可以推断 C2 将在 B 的 GetS 之前看到 A 的 GetM，因此 C2 在收到一致性消息时不需要发送特定的确认消息。这种对请求接收的隐式确认 (implicit acknowledgment) 将监听协议与我们在下一章研究的目录协议区分开来。

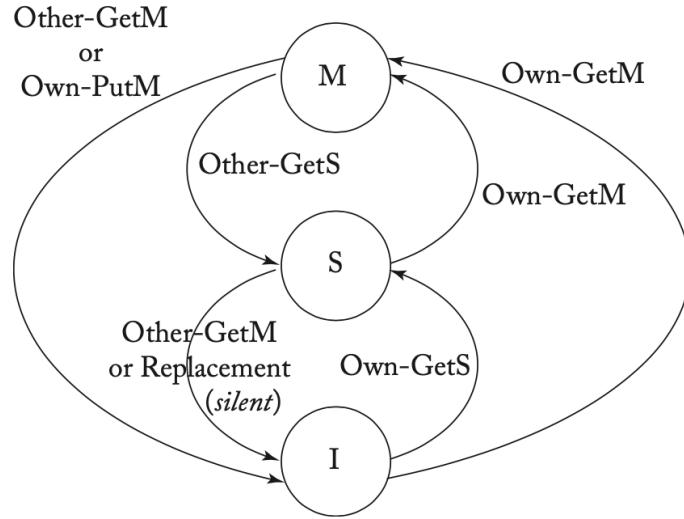


Figure 7.1: MSI: Transitions between stable states at cache controller.

要求以总顺序观察广播一致性请求对于用于实现传统监听协议的互连网络具有重要意义。由于许多一致性控制器可能同时尝试发出一致性请求，互连网络必须将这些请求序列化为某种总顺序。然而，网络决定了这个顺序，这个机制被称为协议的序列化顺序点 (serialization ordering point)。在一般情况下，一致性控制器发出一致性请求，网络在序列化点对该请求进行排序并将其广播给所有控制器，发射控制器 (issuing controller) 通过监听从控制器接收到的请求流来了解其请求的排序位置。作为一个具体而简单的例子，考虑一个使用总线来广播一致性请求的系统。一致性控制器必须使用仲裁逻辑来确保一次在总线上只发出一个请求。该仲裁逻辑充当序列化点，因为它有效地确定了请求在总线上出现的顺序。一个微妙但重要的一点是，一致性请求在仲裁逻辑序列化它的瞬间就被排序，但控制器可能只能通过监听总线以观察在它自己的请求之前和之后出现哪些其他请求来确定这个顺序。因此，一致性控制器可以在序列化点确定之后的几个周期内观察总请求顺序。

到目前为止，我们只讨论了一致性请求，而不是对这些请求的响应。这种看似疏忽的原因是监听协议的关键方面围绕着请求。响应消息几乎没有限制。他们可以在不需要支持广播也不需要任何顺序要求的单独互连网络上 travel。由于响应消息携带数据，因此比请求长得多，因此能够在更简单、成本更低的网络上发送它们有很大的好处。值得注意的是，响应消息不会影响一致性事务的序列化。从逻辑上讲，无论响应何时到达请求者，当请求被排序时，都会发生一个由广播请求和单播响应组成的一致性事务。请求出现在总线上和响应到达请求者之间的时间间隔确实会影响协议的实现（例如，在这个间隙期间，是否允许其他控制器请求此块？如果是，请求者如何响应？），但不影响事务的序列化。（注1）

原书作者注1：这种一致性事务的逻辑序列化类似于处理器核心中指令执行的逻辑序列化。即使核心执行乱序执行，它仍然按程序顺序提交（序列化）指令。

7.2 基准监听协议

在本节中，我们提出了一个简单的、未经优化的监听协议，并描述了它在两种不同系统模型上的实现。第一个简单的系统模型说明了实现监听一致性协议的基本方法。第二个稍微复杂的基准系统模型说明了即使是相对简单的性能改进也可能影响一致性协议的复杂性。这些示例提供了对监听协议的关键特性的深入了解，同时揭示了促使本章后续部分介绍的特性和优化的低效率。第 7.5 节和第 7.6 节讨论了如何使这个基准协议适应更高级的系统模型。

7.2.1 高层次协议规范

基准协议只有三个稳定状态：M、S 和 I。这样的协议通常称为 MSI 协议。与第 6.3 节中的协议一样，该协议假定一个回写缓存。一个块由 LLC/内存拥有，除非该块在状态 M 的缓存中。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 7.1 和 7.2 中，我们分别展示了缓存和内存控制器的稳定状态之间的转换。

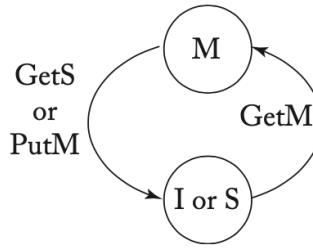


Figure 7.2: MSI: Transitions between stable states at memory controller.

Table 7.4: Total order, coherence, and consistency. States and operations that pertain to address A are preceded by the prefix “A:”, and we denote a block A in state X with value V as “A:X[V].” If the value is stale, we omit it (e.g., “A:I”).

Time	Core C1	Core C2	LLC/Memory
0	A:I B:M[0]	A:S[0] B:I	A:S[0] B:M
1	A:GetM from Core C1/M[0] store A = 1 B:M[0]	A:S[0] B:I	A:S[0] B:M
2	A:M[1] store B = 1 B:M[1]	A:S[0] B:I	A:GetM from Core C1/M B:M
3	A:M[1] B:GetS from Core C2/S[1]	A:S[0] B:I	A:M B:GetS from Core C2/S[1]
4	A:M[1] B:S[1]	A:GetM from Core1/I B:I	A:M B:S[1]
5	A:M[1] B:S[1]	A:I B:GetS from Core C2/S[1] r1 = B[1]	A:M B:S[1]
6	A:GetS from Core C2/S[1] B:S[1]	A:GetS from Core C2/S[1] r2 = A[1] B:S[1]	A:GetS from Core C2/S[1] B:S[1]
r1 = 1, r2 = 1 satisfies SC and TSO			

需要注意三个符号问题。首先，在图 7.1 中，弧被标记为在总线上观察到的一致性请求。我们有意省略了其他事件，包括加载、存储和一致性响应。其次，缓存控制器上的一致性事件被标记为“Own”或“Other”，以表示观察请求的缓存控制器是否是请求者。第三，在图 7.2 中，我们使用以缓存为中心的符号来指定内存中的块的状态（例如，内存状态 M 表示存在状态为 M 的块的缓存）。

7.2.2 简单的监听系统模型：原子请求，原子事务

图 7.3 说明了简单的系统模型，它几乎与图 2.1 中介绍的基准系统模型相同。唯一的区别是图 2.1 中的通用互连网络被指定为总线。每个核心都可以向其缓存控制器发出加载和存储请求；当缓存控制器需要为另一个块腾出空间时，它会选择一个块来驱逐。总线促进了被所有一致性控制器监听的一致性请求的总顺序。与前一章中的示例一样，该系统模型具有简化一致性协议的原子性属性。具体来说，该系统实现了两个原子性属性，我们将其定义为原子请求 (Atomic Request) 和原子事务 (Atomic Transaction)。Atomic Request 属性声明一致性请求在其发出的当个周期中排序。此属性消除了在发射请求 (issue request) 和排序请求 (order request) 之间（由于另一个核心的一致性请求）而导致块状态发生变化的可能性。Atomic Transaction 属性指出一致性事务是原子的，因为对同一块的后续请求可能不会出现在总线上，直到第一个事务完成之后（即，直到响应出现在总线上之后）。由于一致性涉及对单个块的操作，因此系统是否允许对不同块的后续请求不会影响协议。尽管比大多数当前系统更简单，但该系统模型类似于 1980 年代成功的机器 SGI Challenge [5]。

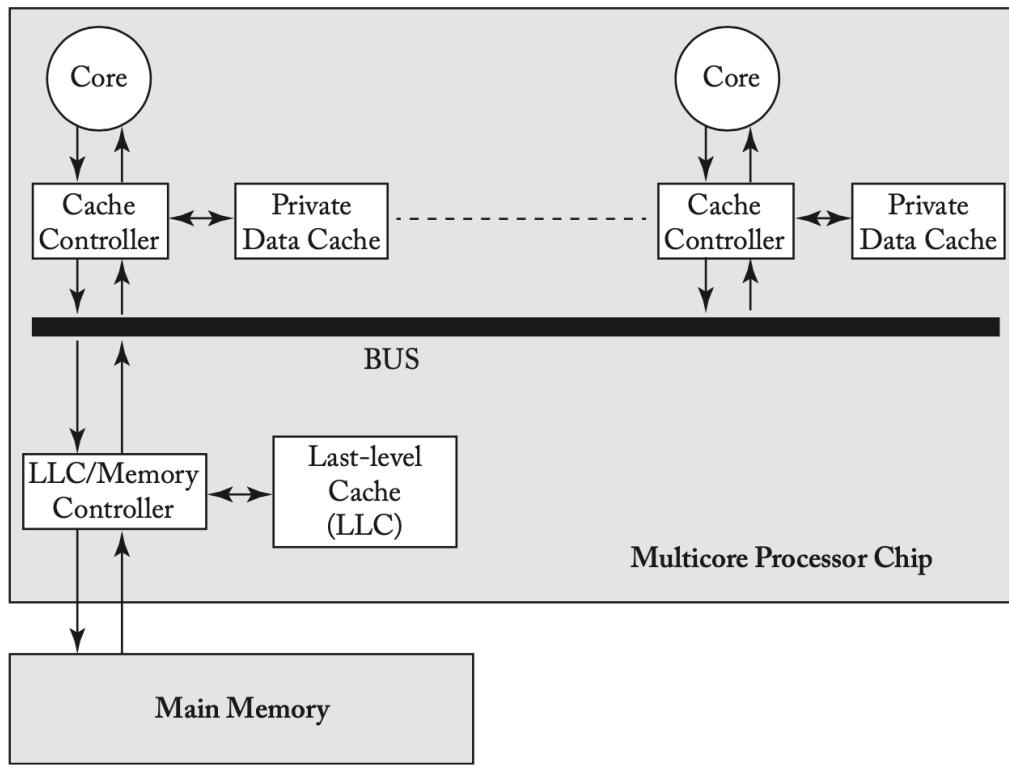


Figure 7.3: Simple snooping system mode.

详细的协议规范

表 7.5 和 7.6 给出了简单系统模型的详细一致性协议。与第 7.2.1 节中的高层次描述相比，最显着的区别是在缓存控制器中添加了两个瞬间状态 (transient state)，在内存控制器中添加了一个瞬间状态。该协议的瞬间状态很少，因为简单系统模型的原子性约束极大地限制了可能的消息交织的数量。

Table 7.5: Simple snooping (atomic requests, atomic transactions): Cache controller

States	Processor Core Events			Bus Events							
				Own Transaction				Transactions for Other Cores			
	Load	Store	Replacement	Own-GetS	Own-GetM	Own-PutM	Data	Other-GetS	Other-GetM	Other-PutM	
I	Issue GetS /IS ^D	Issue GetM /IM ^D									
IS ^D	Stall Load	Stall Store	Stall Evict				Copy data into cache, load hit /S	(A)	(A)	(A)	
IM ^D	Stall Load	Stall Store	Stall Evict				Copy data into cache, store hit /M	(A)	(A)	(A)	
S	Load hit	Issue GetM /SM ^D	-/I						-/I		
SM ^D	Load hit	Stall Store	Stall Evict				Copy data into cache, store hit /M	(A)	(A)	(A)	
M	Load hit	Store Hit	Issue PutM, send Data to memory /I					Send Data to req and memory /S	Send Data to req /I		

Table 7.6: Simple snooping (atomic requests, atomic transactions): Memory controller

State	Bus Events			
	GetS	GetM	PutM	Data from Owner
IoS	Send data block in Data message to requestor/IoS	Send data block in Data message to requestor/M		
IoS ^D	(A)	(A)		Update data block in memory/IoS
M	-/IoS ^D		-/IoS ^D	

表中的阴影条目表示不可能的（或至少是错误的）转换。例如，缓存控制器永远不应该收到它没有请求的块（即，在其缓存中处于状态 I 的块）的 Data 消息。类似地，Atomic Transaction 约束阻止另一个核心在当前事务完成之前发出后续请求；由于此约束，无法出现标记为“(A)”的表条目。空白条目表示不需要任何操作的合法转换。这些表省略了许多理解协议所需要的实现细节。此外，在本协议和本章的其余协议中，我们省略了另一个核心事务的 Data 对应的事件；一个核心从不采取任何行动来响应在总线上观察另一个核心事务的 Data。

与所有 MSI 协议一样，可以在状态 S 和 M 中执行加载（即命中），而存储仅在状态 M 中命中。在加载和存储未命中时，缓存控制器分别通过发送 GetS 和 GetM 请求来启动一致性事务。瞬间状态 IS^D、IM^D 和 SM^D 表示请求消息已发送，但尚未收到数据响应（Data）。在这些瞬间状态下，因为请求已经被排序，所以事务已经被排序并且块在逻辑上分别处于状态 S、M 或 M。（注2）但是，加载或存储必须等待数据到达。（注3）一旦数据响应出现在总线上，缓存控制器就可以将数据块复制到缓存中，根据需要转换到稳定状态 S 或 M，并执行挂起的加载或存储。

原书作者注2：我们在此协议中不包含升级事务 (Upgrade transaction)，该事务将通过不向请求者发送不必要的数据来优化 S 到 M 的转换。对于具有原子请求的此系统模型，添加升级将相当简单，但如果没有任何原子请求，则要复杂得多。当我们提出一个没有原子请求的协议时，我们会讨论这个问题。

原书作者注3：从技术上讲，只要对请求进行排序，只要新存储的值在数据到达时没有被覆盖，就可以执行存储。类似地，允许对新写入的值进行后续加载。

系统模型的原子性属性以两种方式简化了缓存未命中处理。首先，Atomic Request 属性确保当缓存控制器试图升级块的权限（从 I 到 S、I 到 M 或 S 到 M）时它可以发出请求，而不必担心可能会排序另一个核心的请求领先于自己。因此，缓存控制器可以根据需要立即转换到状态 IS^D、IM^D 或 SM^D，以等待数据响应。类似地，Atomic Transaction 属性确保在当前事务完成之前不会发生对块的后续请求，从而无需在处于这些瞬间状态之一时处理来自其他核心的请求。

数据响应可能来自内存控制器或具有处于状态 M 的块的另一个缓存。具有处于状态 S 的块的缓存可以忽略 GetS 请求，因为需要内存控制器响应，但必须使 GetM 请求上的块无效，以强制执行一致性不变量。具有处于状态 M 的块的缓存必须响应 GetS 和 GetM 请求，发送数据响应并分别转换到状态 S 或状态 I。

LLC/内存有两种稳定状态，M 和 IoS，以及一种瞬间状态 IoS^D。在状态 IoS 中，内存控制器是所有者并响应 GetS 和 GetM 请求，因为此状态表明没有缓存具有状态 M 的块。在状态 M 中，内存控制器不响应数据，因为缓存处于状态 M 是所有者并且拥有数据的最新副本。但是，状态 M 中的 GetS 意味着缓存控制器将转换到状态 S，因此内存控制器还必须获取数据、更新内存并开始响应所有未来的请求。它通过立即转换到瞬间状态 IoS^D 并等待直到它从拥有它的缓存中接收到数据来做到这一点。

当缓存控制器由于替换决定而驱逐一个块时，这会导致协议的两种可能的一致性降级：从 S 到 I 和从 M 到 I。在这个协议中，S-to-I 降级在该块被从缓存中逐出，而不与其他一致性控制器进行任何通信。通常，只有在所有其他一致性控制器的行为保持不变时，才有可能进行静默状态转换；例如，不允许无声地驱逐拥有的区块。M-to-I 降级需要通信，因为块的 M 副本是系统中唯一有效的副本，不能简单地丢弃。因此，另一个一致性控制器（即内存控制器）必须改变其状态。为了替换处于状态 M 的块，缓存控制器在总线上发出 PutM 请求，然后将数据发送回内存控制器。在 LLC，当 PutM 请求到达时，块进入状态 IoS^D，然后在 Data 消息到达时转换到状态 IoS。（注4）Atomic Request 属性简化了缓存控制

器，通过在 PutM 在总线上排序之前阻止可能降级状态的干预请求（例如，另一个核心的 GetM 请求）。类似地，Atomic Transaction 属性通过阻止对块的其他请求，直到 PutM 事务完成并且内存控制器准备好响应它们来简化内存控制器。

原书作者注4：我们做了一个简化的假设，即这些消息不能乱序到达内存控制器。

运行示例

在本节中，我们将展示一个系统执行示例，以展示一致性协议在常见场景中的行为方式。我们将在后续部分中使用此示例来理解协议并突出它们之间的差异。该示例仅包括一个块的活动，并且最初，该块在所有缓存中处于状态 I，在 LLC/内存处处于状态 IorS。

在此示例中，如表 7.7 所示，核心 C1 和 C2 分别发出加载和存储指令，这些指令在同一块上未命中。核心 C1 尝试发出 GetS，核心 C2 尝试发出 GetM。我们假设核心 C1 的请求恰好首先被序列化，并且 Atomic Transaction 属性阻止核心 C2 的请求到达总线，直到 C1 的请求完成。内存控制器在周期 3 响应 C1 完成事务。然后，核心 C2 的 GetM 在总线上被序列化；C1 使其副本无效，并且内存控制器响应 C2 以完成该事务。最后，C1 发出另一个 GetS。所有者 C2 以数据响应并将其状态更改为 S。C2 还将数据的副本发送到内存控制器，因为 LLC/内存现在是所有者并且需要块的最新副本。在此执行结束时，C1 和 C2 处于状态 S，LLC/内存处于状态 IorS。

Table 7.7: Simple snooping: Example execution. All activity is for one block.

Cycle	Core C1	Core C2	LLC/memory	Request on Bus	Data on Bus
Initial	I	I	IorS		
1	Load miss; issue GetS/IS ^D				
2				GetS (C1)	
3		Store miss; stall due to <i>Atomic Transactions</i>	Send response to C1		
4					Data from LLC/ mem
5	Copy data to cache; perform load/S	Issue GetM/IM ^D			
6				GetM (C2)	
7	-/I		Send response to C2/M		
8					Data from LLC/ mem
9		Copy data to cache; perform store/M			
10	Load miss; issue GetS/IS ^D				
11				GetS (C1)	
12		Send data to C1 and to LLC/mem/S	-/IorS ^D		
13					Data from C2
14	Copy data from C2; perform load/S		Copy data from C2/IorS		

7.2.3 基准监听系统模型：非原子请求、原子事务

我们在本章其余部分中使用的基准监听系统模型与简单的监听系统模型不同，它允许非原子请求。非原子请求来自许多实现优化，但最常见的原因是在缓存控制器和总线之间插入消息队列（甚至单个缓冲区）。通过将发出请求的时间与发出请求的时间分开，协议必须解决简单监听系统中不存在的漏洞窗口（window of vulnerability）。基准监听系统模型保留了原子事务属性，直到第 7.5 节我们才放松。

我们在表 7.8 和 7.9 中介绍了详细的协议规范，包括所有瞬间状态。与第 7.2.2 节中的简单监听系统协议相比，最显着的区别是瞬间状态的数量要多得多。放宽 Atomic Request 属性引入了许多情况，其中缓存控制器在发射其一致性请求和在总线上观察其自己的一致性请求之间观察来自总线上另一个控制器的请求。

Table 7.8: MSI snooping protocol with atomic transactions-cache controller. A shaded entry labeled “(A)” denotes that this transition is impossible because transactions are atomic on bus.

	Load	Store	Replacement	Own-GetS	Own-GetM	Own-PutM	Other-GetS	Other-GetM	Other-PutM	Own Data Response
I	Issue GetS/IS ^{AD}	Issue GetM/ IM ^{AD}					-	-	-	
IS ^{AD}	Stall	Stall	Stall	-/IS ^D			-	-	-	
IS ^D	Stall	Stall	Stall			(A)	(A)	(A)	-/S	
IM ^{AD}	Stall	Stall	Stall		-/IM ^D		-	-	-	
IM ^D	Stall	Stall	Stall			(A)	(A)	(A)	-/M	
S	Hit	Issue GetM/ SM ^{AD}	-/I				-	-/I	-	
SM ^{AD}	Hit	Stall	Stall		-/SM ^D		-	-/IM ^{AD}	-	
SM ^D	Hit	Stall	Stall			(A)	(A)	(A)	-/M	
M	Hit	Hit	Issue PutM/ MI ^A				Send data to requestor and to memory/S	Send data to requestor/I	-	
MI ^A	Hit	Hit	Stall			Send data to memory/I	Send data to requestor and to memory/II ^A	Send data to requestor/II ^A		
II ^A	Stall	Stall	Stall			Send NoData to memory/I	-	-		

Table 7.9: MSI snooping protocol with atomic transactions-memory controller. A shaded entry labeled “(A)” denotes that this transition is impossible because transactions are atomic on bus.

	GetS	GetM	PutM	Data from Owner	NoData
IorS	Send data to requestor	Send data to requestor/M	-/IorS ^D		
IorS ^D	(A)	(A)		Write data to LLC/memory/IorS	-/IorS
M	-/IorS ^D	-	-/M ^D		
M ^D	(A)	(A)		Write data to LLC/IorS	-/M

以 I-to-S 转换为例，缓存控制器发出 GetS 请求并将块的状态从 I 更改为 IS^{AD}。直到在总线上观察到请求缓存控制器自己的 GetS (Own-GetS) 并序列化，块的状态实际上是 I。也就是说，请求者的块被视为在 I 中；无法执行加载和存储，并且必须忽略来自其他节点的一致性请求。一旦请求者观察到自己的 GetS，请求是有序的，块在逻辑上是 S，但是由于数据还没有到达，所以无法进行加载。缓存控制器将块的状态更改为 IS^D 并等待前一个所有者的数据响应。由于 Atomic Transaction 属性，数据消息是下一个一致性消息（到同一个块）。一旦数据响应到达，事务就完成了，请求者将块的状态更改为稳定的 S 状态并执行加载。I-to-M 转换与 I-to-S 转换类似。

从 S 到 M 的转变说明了在漏洞窗口期间发生状态变化的可能性。如果一个核心试图在状态 S 中存储到一个块，缓存控制器发出一个 GetM 请求并转换到状态 SM^AD。该块有效地保持在状态 S，因此加载可能会继续命中，并且控制器会忽略来自其他核心的 GetS 请求。但是，如果另一个核心的 GetM 请求首先被排序，则缓存控制器必须将状态转换为 IM^AD 以防止进一步的加载命中。正如我们在侧边栏中所讨论的，S 到 M 转换期间的漏洞窗口使添加升级事务变得复杂。

原书侧边栏：没有原子请求的系统中的升级事务

对于具有原子请求 (Atomic Request) 的协议，升级事务 (Upgrade transaction) 是缓存从 Shared 转换为 Modified 的有效方式。升级请求使所有共享副本失效，并且比发出 GetM 快得多，因为请求者只需要等待升级被序列化（即总线仲裁延迟），而不是等待来自 LLC/内存的数据到达。

但是，如果没有原子请求，添加升级事务变得更加困难，因为在发出请求和请求被序列化之间存在漏洞窗口。由于在此漏洞窗口期间序列化的 Other-GetM 或 Other-Upgrade，请求者可能会丢失其共享副本。解决这个问题的最简单的方法是将块的状态更改为一个新状态，在该状态下它等待自己的升级被序列化。当其 Upgrade 被序列化时，这将使其他 S 副本（如果有）无效但不会返回数据，然后核心必须发出后续 GetM 请求以转换到 M。

更有效地处理升级是困难的，因为 LLC/内存需要知道何时发送数据。考虑核心 C0 和 C2 共享一个块 A 并寻求升级它，同时核心 C1 寻求读取它的情况。C0 和 C2 发出升级请求，C1 发出 GetS 请求。假设它们在总线上序列化为 C0、C1 和 C2。C0 的升级成功，因此 LLC/内存（处于 IorS 状态）应将其状态更改为 M 但不发送任何数据，C2 应使其 S 副本无效。C1 的 GetS 在 C0 处找到处于状态 M 的块，它以新的数据值响应并将 LLC/内存更新回状态 IorS。C2 的 Upgrade 终于出现了，但是因为它丢失了共享副本，需要 LLC/内存来响应。不幸的是，LLC/内存处于 IorS 状态，无法判断此升级需要数据。存在解决此问题的替代方案，但不在本入门的范围内。

漏洞窗口也以更显着的方式影响 M-to-I 一致性降级。为替换状态为 M 的块，缓存控制器发出 PutM 请求并将块状态更改为 MI^AA；与第 7.2.2 节中的协议不同，它不会立即将数据发送到内存控制器。在总线上观察到 PutM 之前，块的状态实际上是 M 并且缓存控制器必须响应其他核心对该块的一致性请求。在没有干预一致性请求到达的情况下，缓存控制器通过将数据发送到内存控制器并将块状态更改为状态 I 来响应观察自己的 PutM。如果干预的 GetS 或 GetM 请求在 PutM 排序之前到达，缓存控制器必须像处于状态 M 一样做出响应，然后转换到状态 II^AA 以等待其 PutM 出现在总线上。一旦它看到它的 PutM，直观地，缓存控制器应该简单地转换到状态 I，因为它已经放弃了块的所有权。不幸的是，这样做会使内存控制器卡在瞬间状态，因为它也接收到 PutM 请求。缓存控制器也不能简单地发送数据，因为这样做可能会覆盖有效数据。（注5）解决方案是，当缓存控制器在状态 II^AA 中看到其 PutM 时，它会向内存控制器发送一条特殊的 NoData 消息。NoData 消息向内存控制器表明它来自非所有者并让内存控制器退出其瞬间状态。内存控制器变得更加复杂，因为它需要知道如果它收到 NoData 消息应该返回到哪个稳定状态。我们通过添加第二个瞬态内存状态 M^AD 来解决这个问题。请注意，这些瞬间状态代表了我们通常的瞬间状态命名约定的一个例外。在这种情况下，状态 X^AD 表示内存控制器在收到 NoData 消息时应恢复到状态 X（如果收到数据消息则移动到状态 IorS）。

原书作者注5：考虑这样一种情况，核心 C1 在 M 中有一个块并发出一个 PutM，但核心 C2 执行 GetM，核心 C3 执行 GetS，两者都在 C1 的 PutM 之前排序。C2 获取 M 中的块，修改块，然后响应 C3 的 GetS，用更新的块更新 LLC/内存。当 C1 的 PutM 最终被排序时，写回数据会覆盖 C2 的更新。

7.2.4 运行示例

回到表 7.10 所示的运行示例，核心 C1 发出 GetS，核心 C2 发出 GetM。与前面的示例（在表 7.7 中）不同，消除 Atomic Request 属性意味着两个核心都会发出它们的请求并更改它们的状态。我们假设核心 C1 的请求恰好首先被序列化，并且 Atomic Transaction 属性确保 C2 的请求在 C1 的事务完成之前不会出现在总线上。在 LLC/内存响应完成 C1 的事务后，核心 C2 的 GetM 在总线上被序列化。C1 使其副本无效，LLC/内存响应 C2 以完成该事务。最后，C1 发出另一个 GetS。当这个 GetS 到达总线时，所有者 C2 以数据响应并将其状态更改为 S。C2 还将数据的副本发送到内存控制器，因为 LLC/内存现在是所有者并且需要一个 up-to-date 的块的副本。在此执行结束时，C1 和 C2 处于状态 S，LLC/内存处于状态 IorS。

Table 7.10: Baseline snooping: Example execution

Cycle	Core C1	Core C2	LLC/memory	Request on Bus	Data on Bus
1	Issue GetS/IS ^{AD}				
2		Issue GetM/IM ^{AD}			
3				GetS (C1)	
4	-/IS ^D		Send data to C1/IorS		
5					Data from LLC/mem
6	Copy data from LLC/mem/S			GetM (C2)	
7	-/I	-/IM ^D	Send data to C2/M		
8					Data from LLC/mem
9		Copy data from LLC/mem/M			
10	Issue GetS/IS ^{AD}				
11				GetS (C1)	
12	-/IS ^D	Send data to C1 and to LLC/mem/S	-/IorS ^D		
13					Data from C2
14	Copy data from C2/S		Copy data from C2/IorS		

7.2.5 协议简化

该协议相对简单，并牺牲了性能来实现这种简单性。最重要的简化是在总线上使用原子事务。拥有原子事务消除了许多可能的转换，在表中用“(A)”表示。例如，当一个核心有一个处于 IM^D 状态的缓存块时，该核心不可能观察到另一个核心对该块的一致性请求。如果事务不是原子的，则可能会发生此类事件，并会迫使我们重新设计协议来处理它们，如第 7.5 节所示。

另一个牺牲性能的显着简化涉及对状态 S 的缓存块的存储请求事件。在此协议中，缓存控制器发出 GetM 并将块状态更改为 SM^{AD}。如前面的侧边栏所述，更高性能但更复杂的解决方案将使用升级事务。

7.3 添加独占状态

7.4 添加拥有状态

7.5 非原子总线

7.6 总线互连网络的优化

7.7 案例研究

7.8 讨论和监听的未来

7.9 参考文献

[1] N. Agarwal, L.-S. Peh, and N. K. Jha. In-network snoop ordering (INSO): Snoopy coherence on unordered interconnects. In Proc. of the 14th International Symposium on High-Performance Computer Architecture, pp. 67–78, February 2009. DOI: 10.1109/hpca.2009.4798238. 143

[2] L. A. Barroso and M. Dubois. Cache coherence on a slotted ring. In Proc. of the 20th International Conference on Parallel Processing, August 1991. 146

[3] A. Charlesworth. Starfire: Extending the SMP envelope. IEEE Micro, 18(1):39–49, January/February 1998. DOI: 10.1109/40.653032. 143, 144

[4] S. Frank, H. Burkhardt, III, and J. Rothnie. The KSR1: Bridging the gap between shared memory and MPPs. In Proc. of the 38th Annual IEEE Computer Society Computer Conference (COMPON), pp. 285–95, February 1993. DOI: 10.1109/cmpcon.1993.289682. 146

[5] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In Proc. of the Hawaii International Conference on System Sciences, 1994. DOI: 10.1109/hicss.1994.323177. 114

[6] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp snooping: An approach for extending SMPs. In Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 25–36, November 2000. DOI: 10.1145/378993.378998. 143

[7] M. R. Marty and M. D. Hill. Coherence ordering for ring-based chip multiprocessors. In Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, December 2006. DOI: 10.1109/micro.2006.14. 146

[8] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. IBM Journal of Research and Development, 49(4/5), July/September 2005. DOI: 10.1147/rd.494.0505. 145