第二章：Coherence 基础

在本章中，我们介绍了众多关于 cache coherence 的内容，以了解 consistency 模型如何与缓存交互。我们从 2.1 节开始，介绍我们在本入门指南中考虑的系统模型。为了简化本章和后续章节的阐述，我们选择了足以说明重要问题的最简单的系统模型；我们把与更复杂的系统模型相关的问题推迟到第 9 章。2.2 节解释了必须解决的 cache coherence 问题以及 incoherence 的可能性是如何产生的。2.3 节精确定义了 cache coherence。

2.1 基准系统模型

在本入门指南中，我们考虑了具有共享内存的多处理器核心的系统。也就是说，所有核心都可以对所有（物理）地址执行 (perform) 加载 (load) 和 存储 (store) 操作。基准系统模型包括单个多核处理器芯片和片外主存，如图 2.1 所示。多核处理器芯片由多个单线程核心组成，每个核心都有自己的私有数据缓存，以及一个由所有核心共享的最后一级缓存 (last-level cache, LLC)。在本入门指南中，当我们使用术语“缓存 (cache)”时，我们指的是核心的私有数据缓存，而不是 LLC。每个核心的数据缓存都使用物理地址进行访问，并且采用写回 (write-back) 方式。核心和 LLC 通过互连网络相互通信。尽管 LLC 位于处理器芯片上，但在逻辑上是“内存侧缓存 (memory-side cache)”，因此不会引入另一个级别的 coherence 问题。LLC 在逻辑上就在内存之前，旨在减少内存访问的平均延迟并增加内存的有效带宽。LLC 还用作芯片上的内存控制器。

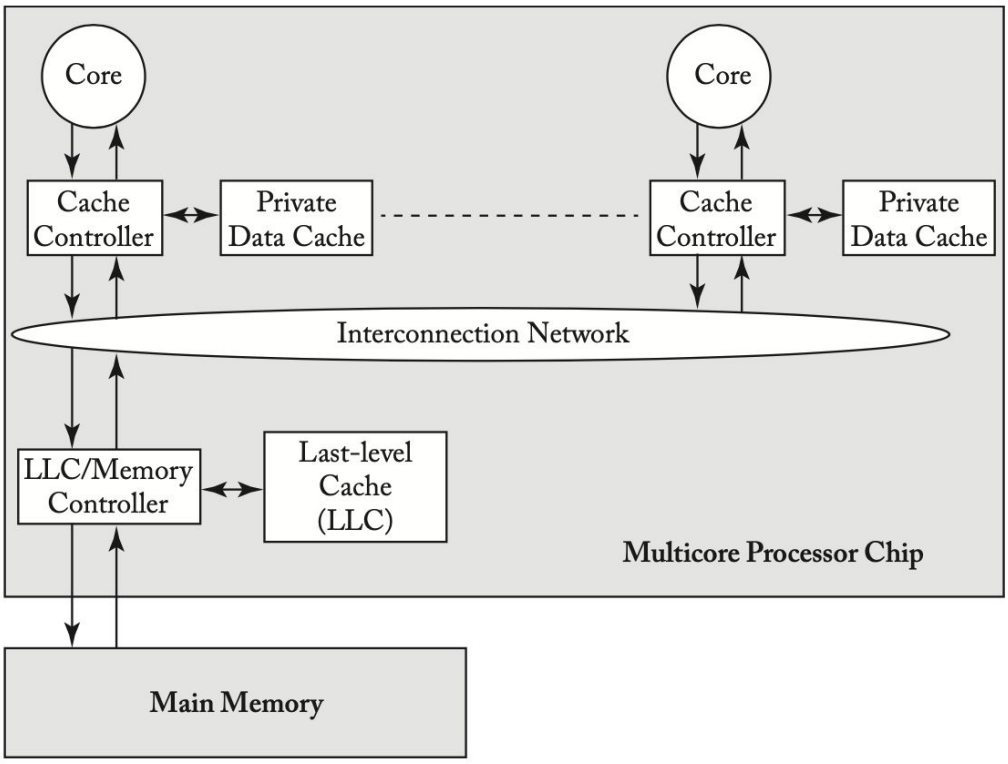

Figure 2.1: Baseline system model used throughout this primer.

此基准系统模型省略了许多常见的特性，但这些特性不是本入门指南的大部分内容所必需。这些特性包括指令缓存、多级缓存、多核之间共享的缓存、虚拟寻址缓存、TLB 和 coherent DMA。基准系统模型还忽略了多个多核芯片的可能性。我们稍后会讨论所有这些特性，但现在，它们会增加不必要的复杂性。

2.2 问题：Incoherence 是如何发生的

出现 incoherence 的可能性仅源于一个基本问题：存在多个可以访问缓存和内存的参与者。在现代系统中，这些参与者是处理器核心、DMA 引擎和可以读取和/或写入缓存和内存的外部设备。在本入门指南的其余部分中，我们通常关注作为核心的参与者，但值得注意的是，其他参与者也可能存在。

表 2.1 说明了一个 incoherence 的简单示例。最初，内存位置 A 在内存以及两个核心的本地缓存中的值为 42。在时间 1，Core 1 将其缓存中内存位置 A 的值从 42 更改为 43，并在其缓存中更新，从而使 Core 2 缓存中的 A 值变为陈旧。Core 2 执行一个 while 循环，从其局部缓存中反复加载 A 的（陈旧）值。显然，这是一个 incoherence 的例子，因为来自 Core 1 的 store 对 Core 2 不可见，因此 Core 2 卡在 while 循环中。

Table 2.1: Example of incoherence. Assume the value of memory at memory location A is initially 42 and cached in the local caches of both cores.

Time	Core C1	Core C2
1	S1: A = 43;	L1: while (A == 42);
2		L2: while (A == 42);
3		L3: while (A == 42);
4		…
n		Ln: while (A == 42);

为了防止 incoherence，系统必须实现一个 cache coherence 协议，使 Core 1 的 store 操作对 Core 2 可见。这些 cache coherence 协议的设计和实现是第 6-9 章的主要主题。

2.3 Cache Coherence 接口

在非正式情况下，coherence 协议必须确保 writes 对所有处理器都可见。在本节中，我们将通过它们展示的抽象接口来更正式地理解 coherence 协议。

处理器核心通过 coherence 接口（图 2.2）与 coherence 协议交互，该接口提供两种方法：(1) *read-request* 方法，接受内存位置作为参数并返回一个值；(2) *write-request* 方法，它接受内存位置和要写入的值作为参数，并返回一个确认 (acknowledgment)。

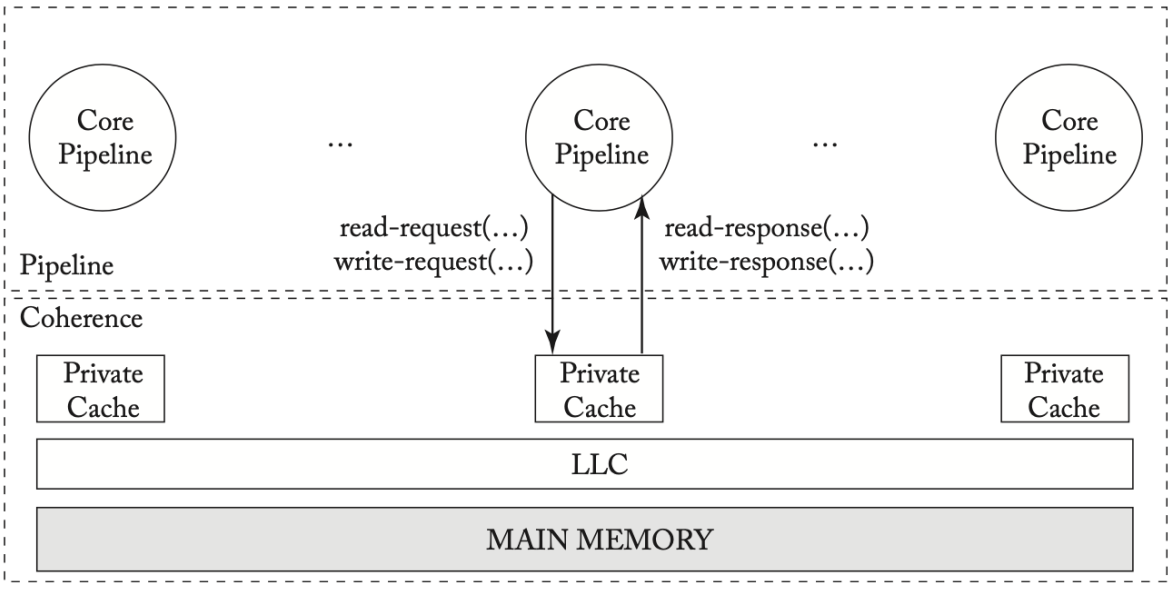

Figure 2.2: The pipeline-coherence interface.

文献中出现了许多 coherence 协议，并已在实际处理器中使用。我们根据它们的 coherence 接口的性质将这些协议分为两类。具体来说，是基于 consistency model 和 coherence 是否可以清晰地分离，或者它们是否不可分割。

Consistency-agnostic coherence. 在第一类中，一个 write 在返回之前对所有其他核心可见。因为 writes 是同步传播的，所以第一类提供了一个与原子内存系统（没有缓存）相同的接口。因此，任何与 coherence 协议交互的子系统，例如处理器核心流水线，都可以假设它正在与不存在缓存的原子内存系统交互。从 consistency 强制实施的角度来看，这种 coherence 接口可以很好地分离关注点。Cache coherence 协议将缓存完全抽象出来并呈现出原子内存的错觉，就好像缓存被移除一样，只有内存包含在 coherence 框内（图 2.2），而处理器核心流水线强制实施 consistency model 规范中的排序。

Consistency-directed coherence. 在第二类这种较新的类别中，writes 是异步传播的。因此，一个 write 可以在所有处理器可见之前返回，从而允许（实时）观察过时的值。但是，为了正确强制实施 consistency，此类中的 coherence 协议必须确保最终使 writes 可见的顺序符合 consistency model 规定的排序规则。回到图 2.2，流水线和 coherence 协议一起强制实施 consistency model 要求的排序。第二类的出现是为了支持基于吞吐量的通用图形处理单元 (GP-GPUs)，并在本入门指南第一版出版后受到重视。（注1）

> 原书作者注 1：对于那些关心 consistency 影响的人，请注意，使用这种方法可以强制实施各种 consistency model，包括 SC 和 TSO 等强模型。

本入门书（以及本章的其余部分）侧重于第一类 coherence 协议。我们在异构 coherence 的背景下讨论第二类 coherence 协议（第 10 章）。

2.4 Consistency-agnostic coherence 不变量

Coherence 协议必须满足哪些不变量 (invariants) 才能使缓存不可见并呈现原子内存系统的抽象？

在教科书和已发表的论文中出现了几种 coherence 的定义，我们不想一一介绍。相反，我们提出了我们更喜欢的定义，因为它洞察了 coherence 协议的设计。在侧边栏中，我们讨论了可替代的定义以及它们与我们首选定义的关系。

> **Sidebar: Consistency-Like Definitions of Coherence**
>
> 我们对 coherence 的首选定义是从实现的角度来定义的。它指定硬件强制的不变量，关于不同核心对内存位置的访问权限以及核心之间传递的数据值。
>
> 存在另一类从程序员的角度定义 coherence 的定义，类似于 memory consistency model 如何指定架构上可见的 loads 和 stores 的顺序。
>
> 指定 coherence 的一种 consistency-like 的方法与 sequential consistency 的定义有关。Sequential consistency (SC) 是我们将在第 3 章中深入讨论的 memory consistency model，它指定系统必须以尊重每个线程的程序顺序的总序 (total order) 执行所有线程的到所有内存位置的 loads 和 stores。每个 load 都会获取该总序中最近 store 的值。与 SC 的定义类似的 coherence 的定义是，一个 coherent 的系统必须看起来以尊重每个线程的程序顺序的总序执行所有线程的到单个内存位置的 loads 和 stores。这个定义强调了文献中 coherence 和 consistency 之间的一个重要区别：coherence 是在每个内存位置的基础上指定的，而 consistency 是针对所有内存位置指定的。值得注意的是，任何满足 SWMR 和数据值不变量 (data-value invariants)（并与不对任何特定位置的访问重排序的流水线相结合）的 coherence 协议也可以保证满足这种 consistency-like 的 coherence 的定义。（然而，反过来不一定是真的。）
>
> Coherence 的另一个定义 [1, 2] 用两个不变量定义了 coherence：(1) 每个 stores 最终都对所有核心可见，(2) 对相同内存位置的 writes 被*序列化 (serialized)*（即，所有核心以相同的顺序观察）。IBM 在 Power 架构 [4] 中采取了类似的观点，部分原因是为了便于实现，其中一个核心的一系列 stores 可能已经到达某些核心（它们的值对这些核心的 loads 可见）但没有到达其他核心。不变量

> 2 等价于我们之前描述的 consistency-like 的定义。不变量 2 是一个*安全性 (safety)* 不变量（坏事不得发生），与之相比，不变量 1 是一个*活跃性 (liveness)* 不变量（好事最终必须发生）。
>
> Hennessy 和 Patterson [3] 规定的另一个 coherence 的定义由三个不变量组成：(1) 一个核心到内存位置 A 的一个 load 会获得该核心先前 store 到 A 的值，除非另一个核心中间已 store 到 A；(2) 如果 store S 和 load "在时间上充分分离"并且如果 S 和 load 之间没有发生其他 store，则对 A 的 load 获得另一个核心对 A 的 store S 的值；(3) 到相同的内存位置的 stores 被序列化（与前面定义中的不变量 2 相同）。与前面的定义一样，这组不变量同时捕获了安全性和活跃性。

我们通过*单写多读 (single-writer-multiple-reader, SWMR)* 不变量来定义 coherence。对于任何给定的内存位置，在任何给定的时刻，要么有一个核心可以写入它（也可以读取它），要么有一些核心可以读取它。因此，永远不会有一个给定的内存位置可以由一个核心写入并同时由任何其他核心读取或写入的时间。查看此定义的另一种方法是考虑，对于每个内存位置，该内存位置的生命周期被划分为多个时期 (epochs)。在每个时期中，要么单个核心具有读写访问权限，要么一些核心（可能为零）具有只读访问权限。图 2.3 说明了一个示例内存位置的生命周期，分为四个维持 SWMR 不变性的时期。

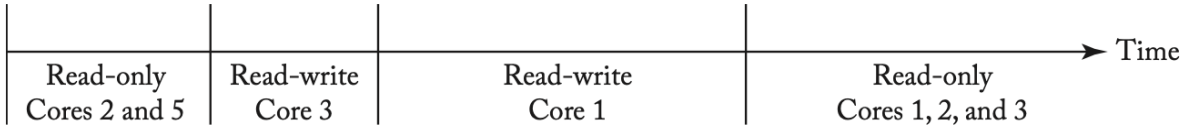

Figure 2.3: Dividing a given memory location's lifetime into epochs.

除了 SWMR 不变量之外，coherence 还要求给定内存位置的值被正确传播。为了解释为什么值很重要，让我们重新考虑图 2.3 中的例子。即使 SWMR 不变量成立，如果在第一个只读时期期间核心 2 和 5 可以读取不同的值，那么系统就不是 coherent 的。类似地，如果核心 1 在其读写时期未能读取核心 3 写入的最后一个值，或者核心 1、2 或 3 中的任何一个未能读取核心 1 在其读写时期期间执行的最后一次写入，则系统是 incoherent 的。

因此，coherence 的定义必须用与值如何从一个时期传播到下一个时期有关的数据值不变量 (data value invariant) 来增强 SWMR 不变量。该不变量表明，一个时期开始时的内存位置值与其最后一个读写时期结束时的内存位置值相同。

Coherence invariants

1. **Single-Writer, Multiple-Read (SWMR) Invariant.** For any memory location A, at any given time, there exists only a single core that may write to A (and can also read it) or some number of cores that may only read A.
2. **Data-Value Invariant.** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of the its last read-write epoch.

这些不变量还有其他等效的解释。一个值得注意的例子 [5] 用令牌 (tokens) 来解释 SMWR 不变量。不变量如下。对于每个内存位置，存在固定数量的令牌，该令牌至少与核心数量一样大。如果一个核心拥有所有令牌，它可能会写入内存位置。如果一个核心有一个或多个令牌，它可能会读取内存位置。因此，在任何给定时间，一个核心都不可能在任何其他核心读取或写入内存位置的同时写入这个内存位置。

2.4.1 维护 Coherence 不变量

上一节中介绍的 coherence 不变量提供了一些关于 coherence 协议如何工作的直观理解。绝大多数 coherence 协议，称为"无效化协议 (invalidate protocols)"，都是为了维护这些不变量而明确设计的。如果一个核心想要读取一个内存位置，它会向其他核心发送消息以获取该内存位置的当前值，并确保其他核心没有将该内存位置的缓存副本保持在 read-write 状态。这些消息结束了任何活跃的 read-write 时期，并开始一个 read-only 时期。如果一个核心想要写入一个内存位置，它会向其他核心发送消息以获取内存位置的当前值，如果它还没有有效的 read-only 缓存副本，并确保没有其他核心有以 read-only

或 read-write 状态缓存该内存位置的副本。这些消息结束任何活跃的 read-write 或 read-only 时期，并开始一个新的 read-write 时期。这本入门指南关于 cache coherence 的章节（第 6-9 章）极大地扩展了对无效化协议的抽象描述，但基本的直观理解保持不变。

2.4.2 Coherence 的粒度

核心可以以各种粒度呈现 loads 和 stores，通常范围为 1-64 字节。理论上，可以以最精细的 load/store 粒度执行 coherence。然而，在实践中，coherence 通常保持在缓存块的粒度上。也就是说，硬件在逐个缓存块的基础上强制实施 coherence。在实践中，SWMR 不变量很可能是，对于任何内存*块 (block)*，要么有一个 writer，要么有一些 readers。在典型系统中，一个核心不可能写入块的第一个字节，而另一个核心正在写入该块中的另一个字节。尽管缓存块粒度很常见，而且我们在本入门书的其余部分都假设这一点，但应该知道有些协议在更细和更粗的粒度上保持 coherence。

2.4.3 Coherence 何时相关？

Coherence 的定义，无论我们选择哪种定义，仅在某些情况下相关，架构师必须意识到它何时适用，何时不适用。我们现在讨论两个重要问题。

- Coherence 适用于持有共享地址空间中的块的*所有存*储结构。这些结构包括 L1 数据缓存、L2 缓存、共享的最后一级缓存 (LLC) 和主存。这些结构还包括 L1 指令缓存和转换后备缓冲区 (TLB)。（注2）
- 程序员不能直接看到 coherence。相反，处理器流水线和 coherence 协议共同强制实施 consistency model，并且只有 consistency model 对程序员可见。

> 原书作者注 2：在某些架构中，TLB 可以保存不是共享内存中块的严格副本的映射。

2.5 参考文献

[1] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. Ph.D. thesis, Computer System Laboratory, Stanford University, December 1995.

[2] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory. In Proc. of the 17th Annual International Symposium on Computer Architecture, pp. 15–26, May 1990. DOI: 10.1109/isca.1990.134503.

[3] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach, 4th ed. Morgan Kaufmann, 2007.

[4] IBM. Power ISA Version 2.06 Revision B. http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf, July 2010.

[5] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In Proc. of the 30th Annual International Symposium on Computer Architecture, June 2003. DOI: 10.1109/isca.2003.1206999.