第一章：Consistency 和 Coherence简介

许多现代计算机系统和大多数多核芯片（芯片多处理器）都支持硬件共享内存 (shared memory)。在共享内存系统中，每个处理器核心都可以读取和写入单个共享地址空间。这些设计寻求各种优良特性，例如高性能、低功耗和低成本。当然，在不保证正确性 (correctness) 的前提下提供这些优良特性是没有价值的。正确的共享内存不深入细想的话，似乎很直观，但是，正如本书将展示的那样，即使在定义共享内存系统正确的含义时也存在一些微妙的问题，以及在设计一个正确的共享内存实现时也存在一些微妙的极端情况。此外，必须在错误修复成本高昂的硬件实现中掌握这些微妙之处。即使是学者也应该掌握这些微妙之处，以使他们提出的设计更有可能奏效。

设计和评估正确的共享内存系统需要架构师理解 **memory consistency** 和 **cache coherence**，这正是本书的两个主题。Memory consistency (consistency, memory consistency model, memory model) 是对共享内存正确性的精确的、架构上可见的定义。Consistency 的定义提供了有关 loads 和 stores (or memory reads and writes) 的规则，以及它们如何作用于内存。在理想情况下，consistency 的定义简单易懂。但是，定义共享内存正确运行的含义比定义单线程处理器核心的正确行为更为微妙。单个处理器核心的正确性标准将行为划分为一个正确的结果和许多不正确的选择。这是因为，处理器的架构要求线程的执行将给定的输入状态转换为单个明确定义的输出状态，即使在乱序核心上也是如此。然而，shared memory consistency model 涉及多个线程的 loads 和 stores，通常允许许多正确的执行，而不允许更多不正确的执行。多次正确执行的可能性是由于 ISA 允许多个线程同时执行，通常来自不同线程的指令有许多可能的合法交织。大量正确的执行使以前确定执行是否正确的简单挑战变得复杂。然而，必须掌握 consistency 以实现共享内存，并且在某些情况下，编写使用共享内存的正确程序。

微架构（处理器核心和共享内存系统的硬件设计）必须强制实现所需的 consistency model。为了支持 consistency model，硬件需要提供 cache coherence (or coherence)。在具有缓存的共享内存系统中，当其中一个处理器更新其缓存值时，缓存值可能会过期 (or incoherent)。Coherence 试图使共享内存系统的缓存在功能上与单核系统中的缓存一样不可见。它通过将一个处理器的 write 传播到其他处理器的缓存来实现这个特性。值得强调的是，consistency 定义了共享内存的正确性，需要在架构规范中描述。而 coherence 与 consistency 不同，coherence 是支持 consistency model 的一种手段。

尽管 consistency 是这本入门书的第一个主要主题，但我们会首先在第 2 章简要介绍 coherence，因为 coherence 协议在提供 consistency 方面发挥着重要作用。第 2 章的目标是充分解释 coherence 以了解 consistency model 如何与 coherent cache 交互，而不是探索特定的 coherence 协议或实现。关于 coherence 的协议或实现的相关主题，我们将推迟到第 6-9 章，作为本书的第二部分。

1.1 Consistency (A.K.A., Memory Consistency, Memory Consistency Model, or Memory Model)

Consistency model 根据 load 和 store（内存读取和写入）定义正确的共享内存行为，而不涉及缓存或 coherence。可以从现实世界中直观的感受一下我们为什么需要 consistency model。请考虑一所在线发布其课程安排的大学，假设计算机体系结构课程原定在 152 室。开课前一天，大学注册处决定将课程搬到 252 室。注册处发送电子邮件，要求网站管理员更新在线课程表，几分钟后，注册处会向所有注册的学生发送一条短信，要求学生去检查新的课程表。想象这样一个场景，网站管理员太忙而无法立即发布更新，一个勤奋的学生收到短信后，立即查看在线课程表，会观察到课程位置认为 152 室。尽管在线课程表最终更新为 252 室，并且注册处以正确的顺序执行“写入”，但这位勤奋的学生以不同的顺序观察它们，因此去了错误的房间。Consistency model 定义了这种行为是正确的（因此用户是否必须采取其他行动来实现预期的结果）还是不正确的（在这种情况下，系统必须排除这些重排序）。

尽管这个示例使用了多种媒体，但在具有乱序处理器核心、write buffer、prefetching 和 multiple cache banks 的共享内存硬件中也可能发生类似的行为。因此，我们需要定义共享内存的正确性，即允许哪些共享内存行为，以便程序员知道期望什么，实现者知道他们可以提供的限制。

共享内存的正确性由 memory consistency model 或更简单的内存模型指定。内存模型指定使用共享内存执行的多线程程序的允许行为。对于使用特定输入数据执行的多线程程序，内存模型指定动态加载可能返回的值，以及可选的内存可能的最终状态。与单线程执行不同，多线程通常允许多个正确的行为，这使得理解 memory consistency model 变得微妙。

第 3 章介绍了 memory consistency model 的概念，并介绍了最强、最直观的连贯性模型——sequential consistency, SC（顺序一致性、顺序连贯性）。本章首先解释了为什么需要明确规定共享内存的行为，并精确定义了 memory consistency model 是什么。接下来，本章深入研究了直观的 SC 模型，该模型指出，多线程执行应该看起来像是将每个线程的顺序执行交织在一起，就好像这些线程在单核处理器上进行时分复用一样。简单来说，SC 模型认为多线程执行的结果应该就像是将每个线程的操作顺序交错进行，就好像它们在单核处理器上轮流执行一样。除了这个直观的理解外，本章还对 SC 进行了形式化的阐述，并探讨了如何以简单和激进的方式在具有 coherence 的情况下实现 SC，最后，通过对 MIPS R10000 处理器的案例研究总结了本章内容。

在第 4 章中，我们将超越 SC ，专注于 x86 和历史上的 SPARC 系统所实现的 memory consistency model。这种被称为总存储顺序 (total store order, TSO) 的 memory consistency model，它的动机是为了使用先入先出的 write buffer 来保存已提交的 store 操作的结果，然后再将这些结果写入缓存当中。这种优化违反了 SC，但它承诺带来足够的性能提升，从而激发了在架构设计中定义 TSO，以实现这种优化。在本章中，我们将展示如何从我们的 SC 形式化中定义出 TSO，TSO 如何影响实现，以及 SC 和 TSO 之间的比较。

最后，第 5 章介绍了“宽松 (relaxed)”或“弱 (weak)”的 memory consistency model。它通过展示在强 (strong) 模型中，大多数内存顺序 (memory ordering) 是不必要的来解释这些模型的动机。例如，如果一个线程更新了十个数据项，然后更新了一个同步标志，程序员通常不关心数据项是否按顺序更新，而只关心所有数据项在更新标志之前更新。宽松模型旨在捕捉这种增强的顺序灵活性，以获得更高的性能或更简单的实现。在提供了这个动机之后，本章展开介绍了一个名为 XC 的 relaxed consistency model（宽松连贯性模型）示例，在 XC 模型中，程序员只有在使用 FENCE 指令（例如，在最后一次数据更新之后但在标志写入之前的 FENCE）时才能明确地指定顺序。接着，本章将前两章的形式化方法扩展到 XC 模型上，并讨论了如何实现 XC（包括在核心和 coherence 协议之间进行了大量的重排序）。然后，本章讨论了许多程序员可以避免直接考虑宽松模型的一种方法：如果他们添加足够多的 FENCE 以确保他们的程序是无数据竞争 (data-race free, DRF) 的，那么大多数宽松模型将表现得像 SC 模型。通过使用“SC for DRF”，程序员既可以获得 SC 的（相对）简单的正确性模型和 XC 的（相对）更高的性能。对于那些希望进行更深入地推理的人，本章最后通过区分获取 (acquire)操作和释放 (release)操作，讨论写入原子性 (write atomicity) 和因果关系 (causality)，提供商业示例（包括 IBM Power 的案例研究），并涉及高级语言模型（Java 和 C++），从多个角度对 XC 模型进行了总结。

回到真实世界中班级课程表的 consistency 示例，我们可以观察到电子邮件系统、人工网络管理员和短信系统的组合，代表了一个极其弱的 consistency model。为了防止勤奋的学生走错房间，大学注册处需要在发送电子邮件之后执行 FENCE 操作，以确保在线课程表在发送短信之前已经更新。

1.2 Coherence (A.K.A., CACHE COHERENCE)

如果不小心处理，当多个参与者（例如，多个核心）可以访问某数据项的多个副本（例如，在多个缓存中）并且至少一个访问是写入操作时，就可能会出现 coherence 问题。考虑一个类似于 memory consistency 的示例，一名学生查看在线课程表，发现计算机体系结构课程正在 152 教室上课（读取数据项），并将此信息复制到她手机中的日历应用程序中（缓存数据项）。随后，大学注册处决定将班级搬到 252 教室，更新了在线时间表（写入数据项）并通过短信通知学生。学生手上的数据项副本现在已经过时，这就造成了 incoherent 的情况。如果她去 152 教室上课，她将找不到她的班级。在计算领域中还有其他的 incoherence 示例（不包括计算机体系结构），比如过时的 Web 缓存和程序员使用但未更新代码存储库。

使用 coherence 协议可以防止对过时数据的访问 (incoherence)，而 coherence 协议则是由系统内分布的参与者所实现的一组规则。尽管 Coherence 协议存在许多变种，但它们都遵循一些主要原则，正如第 6-9 章所探讨的。从本质上讲，所有这些变种都通过将一个处理器的写入操作传播到所有缓存中，来使得其他处理器也能够看见该写操作，以此来保持诸如本地日历与在线时间表之类的数据同步，从而防止不一致性问题的出现。然而，不同的协议在同步发生的时间 (when) 和方式 (how) 上有所差异。总体上，Coherence 协议分为两大类别。在第一类中，cache coherence 确保写入操作同步地传播到缓存中。当在线时间表更新时，cache coherence 确保学生的日历也会得到更新。在第二类中，cache coherence 将写入操作异步地传播到缓存，同时仍然遵循 consistency model。尽管这种情况下 Coherence 协议不能保证在线课程表更新后，新值也会立即传播到学生的日历；但是，协议确保在短信到达她的手机之前传播新值。本入门书侧重于第一类 coherence 协议（第 6-9 章），而在第 10 章讨论新兴的第二类。

第 6 章介绍了 cache coherence 协议的整体架构，并为后续章节中有关特定 coherence 协议的内容奠定了基础。本章涵盖了大多数 coherence 协议共有的问题，包括缓存控制器和内存控制器的分布式操作，以及常见的 MOESI coherence 状态：修改 (modified, M)、拥有 (owned, O)、独占 (exclusive, E)、共享 (shared, S) 和无效 (invalid, I)。重要的是，本章还介绍了我们的表驱动方法，用于呈现具有稳定 (stable)（例如 MOESI）和瞬态 (transient) coherence 状态的协议。在实际实现中需要瞬态状态，因为现代系统很少允许从一个稳定状态到另一个稳定状态的原子转换（例如，状态 Invalid 中的读取未命中将花费一些时间等待数据响应，然后才能进入状态 Shared）。在Coherence 协议中，许多真实的复杂性隐藏在这些瞬态状态中，类似于处理器核心的复杂性隐藏在微架构状态中的程度。

第 7 章介绍了最初主导商业市场的 snooping cache coherence（监听式缓存一致性/嗅探式缓存一致性）协议。不深入细想的话，监听协议很简单。当缓存未命中发生时，一个核心的缓存控制器会仲裁共享总线并广播其请求。共享总线确保所有控制器以相同的顺序观察所有请求，因此所有控制器可以协调它们各自的分布式动作，以确保它们保持全局 consistent 状态。然而，监听逐步变得复杂，因为系统可能使用多条总线，并且现代总线并不能原子地处理请求。现代总线具有用于仲裁的队列，可以发送单播响应，这些响应可能因为流水线延迟或无序处理 (out-of-order) 而有所延迟。所有这些特性都会导致更多的瞬态 coherence 状态。第 7 章以 Sun UltraEnterprise E10000 和 IBM Power5 的案例研究结束。

第 8 章深入探讨了 directory cache coherence（目录缓存一致性）协议，这些协议承诺，相较于依赖广播的监听协议，能够扩展到更多处理器核心和其他参与者。有一个笑话说，计算机科学中的所有问题都可以通过引入一层间接引用来解决。目录协议支持这个笑话：当缓存未命中时，会从下一级缓存（或内存）控制器请求内存位置，该控制器维护一个目录，以跟踪哪些缓存保存哪些位置。根据所请求的内存位置的目录表项，控制器向请求者发送响应消息或将请求消息转发给当前缓存该内存位置的一个或多个参与者。每条消息通常都只有一个目的地（即没有广播或多播），但是瞬态 coherence 状态比比皆是，因为从一种稳定的 coherence 状态到另一种稳定的 coherence 状态的转换，可以生成与系统中参与者数量成比例的多条消息。本章从一个基本的 MSI 目录协议开始，然后对其进行细化以处理 MOESI 的状态 E 和 O、分布式目录、更少的请求停滞、近似目录表项表示等内容。此外，本章还探讨了目录本身的设计，包括目录缓存技术。本章以旧 SGI Origin 2000 和较新的 AMD HyperTransport、HyperTransport Assist 和 Intel QuickPath Interconnect (QPI) 的案例研究结束。

第 9 章涉及了一些 coherence 高级主题，但并未涵盖全部内容。为了便于解释，前面关于 coherence 的章节故意限制在最简单的系统模型上，以解释基本问题。第 9 章深入研究了更复杂的系统模型和优化，重点关注监听和目录协议共同面临的问题。初始的主题包括处理指令缓存、多级缓存、直写缓存、TLB、coherent DMA、虚拟缓存和 hierarchical coherence 协议。最后，本章深入探讨了性能优化（例如，针对 migratory sharing 和 false sharing）和一个名为 Token Coherence 的新协议族，它综合了目录协议和监听协议的 coherence 特性。

1.3 异构系统的 Consistency 和 Coherence

现代计算机系统主要是异构的。今天的手机处理器不仅包含多核 CPU，还包含 GPU 和其他加速器（例如神经网络硬件）。在寻求可编程性的过程中，这种异构系统开始支持共享内存。第 10 章讨论这种异构处理器的 consistency 和 coherence。

本章首先关注 GPU，可以说是当今最流行的加速器。本章观察到 GPU 最初选择不支持硬件 cache coherence，因为 GPU 是为尴尬的并行图形工作负载而设计的，这些工作负载并不需要同步或共享数据。然而，当 GPU 用于具有细粒度同步和数据共享的通用工作负载时，缺乏硬件 cache coherence 会导致可编程性和/或性能挑战。本章详细讨论了一些有希望的 coherence 替代方案，这些方案克服了这些限制——特别是解释了为什么候选协议直接执行 consistency model ，而不是以 consistency-agnostic (不考虑一致性)的方式实现 coherence。本章最后简要讨论了 CPU 和加速器之间的 consistency 和 coherence。

1.4 指定和验证 Memory Consistency Model 和 Cache Coherence

Consistency model 和 coherence 协议是复杂而微妙的。 然而，必须管理这种复杂性，以确保多核是可编程的，并且它们的设计可以得到验证。为了实现这些目标，形式化地指定 consistency model 至关重要。形式化的规范将使程序员能够清楚而详尽地（通过工具支持）了解内存模型允许哪些行为，不允许哪些行为。其次，一个精确的形式化规范对于验证实现是强制性的。

第 11 章首先讨论了指定系统的两种方法——公理 (axiomatic) 和操作 (operational)——重点关注如何将这些方法应用于 consistency model 和 coherence 协议。 然后本章讨论了验证实现的技术——包括处理器流水线和 coherence 协议实现——是否符合它们的规范。 本章讨论了形式化方法和非形式化测试。

1.5 Consistency 和 Coherence 小测验

问题 1：在维护 sequential consistency (SC) 的系统中，核心必须按程序顺序 (program order) 发出 coherence 请求。对或错？（答案在第 3.8 节）

问题 2：Memory consistency model 指定了 coherence transaction 的合法顺序。对或错？（第 3.8 节）

问题 3：要执行原子 read-modify-write 指令（例如 test-and-set），核心必须始终与其他核心通信。对或错？（第 3.9 节）

问题 4：在具有多线程核心的 TSO 系统中，线程可能会 bypass（旁路）write buffer 中的值，而不管哪个线程写入了该值。对或错？（第 4.4 节）

问题 5：编写与高级语言的 consistency model（例如 Java）相关的正确同步代码的程序员不需要考虑架构的 memory consistency model。对或错？（第 5.9 节）

问题6：在 MSI 监听协议中，cache block 可能仅处于三种 coherence 状态之一。对或错？（第 7.2 节）

问题 7：Snooping cache coherence 协议要求核心在总线上进行通信。对或错？（第 7.6 节）

问题 8：GPU 不支持硬件 cache coherence。因此，他们无法强制执行 memory consistency model。对或错？（第 10.1 节）。

尽管后续的内容会提供答案，但是我们鼓励读者在查看答案之前尝试回答这些问题。

1.6 本书不能做什么

本书不包括以下内容。

- 同步。Coherence使缓存不可见。Consistency 可以使共享内存看起来像一个单独的内存模块。然而，程序员可能需要锁、屏障和其他同步技术来使他们的程序有用。读者可以参考关于共享内存同步的综合讲座 [2]。
- 商用的 Relaxed Consistency Model。本入门书不涉及 ARM、PowerPC 和 RISC-V 内存模型的微妙之处，但确实描述了它们提供哪些机制来强制执行顺序。

- 并行编程。本书不讨论并行编程模型、方法或工具。
- 分布式系统的 Consistency。本入门书仅限于共享内存多核内的 consistency，不包括在一般分布式系统中的 consistency 模型及其实现。读者可参考数据库复制综合讲座 [1] 和仲裁系统 [3]。

1.7 参考文献

[1] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Database Replication. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010. DOI: 10.1007/978-1-4614-8265-9_110. 8

[2] M. L. Scott. Shared-Memory Synchronization. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013. DOI: 10.2200/s00499ed1v01y201304cac023. 7

[3] M. Vukolic. Quorum Systems: With Applications to Storage and Consensus. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. DOI: 10.2200/s00402ed1v01y201202dct009. 8