第五章：宽松内存一致性

前两章探讨了内存一致性模型中的连续一致性 (sequential consistency, SC) 和总存储顺序 (total store order, TSO)。这些章节介绍了直观的 SC 和广泛实现的 TSO（例如，在 x86 中）。两种模型有时都被称为强模型，因为每个模型的全局内存顺序 (program order) 通常尊重（保留）每个线程的程序顺序 (program order)。回想一下，对于加载和存储的所有四种组合（Load -> Load、Load -> Store、Store -> Store 和 Store -> Load），SC 保留了来自同一线程的两个内存操作的所有顺序，而 TSO 保留了除 Store -> Load 顺序外的前三个顺序。

译者注：本文将 sequential 翻译为连续的，ordering 翻译为顺序，以作区分。
本章研究了更宽松（弱）的内存一致性模型，这些模型试图只保留程序员"需要"的顺序。这种方法的主要好处是，通过允许更多的硬件和软件（编译器和运行时系统）优化，要求更少的顺序约束可以促进更高的性能。主要缺点是，当"需要"顺序时，宽松模型必须形式化，并为程序员或低层次软件提供机制以将这种顺序传达给实现，并且供应商未能就统一的宽松模型达成一致，从而损害了可移植性。

对宽松一致性模型的全面探索超出了本章的范围。相反，本书是一本入门书，旨在提供基本的直觉并帮助读者了解对这些模型的简单理解的局限性。特别是，我们为宽松模型提供了动机（第 5.1 节），展示并形式化了一个宽松一致性 (relaxed consistency, XC) 模型的示例（第 5.2 节），讨论了 XC 的实现，包括原子指令和用于强制保持顺序的指令（第 5.3 节），介绍了无数据竞争程序的连续一致性（第 5.4 节），介绍了其他宽松模型的概念（第 5.5 节），介绍了 RISC-V 和 IBM Power 内存模型案例研究（第 5.6 节），指向进一步阅读和其他商业模型（第 5.7 节），比较模型（第 5.8 节），并触及高级语言的内存模型（第 5.9 节）。

5.1 动机

正如我们很快将看到的，掌握宽松一致性模型可能比理解 SC 和 TSO 更具挑战性。 这些缺点引出了一个问题：为什么要使用宽松的模型呢？ 在本节中，我们将鼓励宽松模型，首先通过展示程序员不关心指令顺序的一些常见情况（第 5.1.1 节），然后讨论一些可以在不强制保持不需要的顺序时可以利用的优化（第 5.1.2 节）。

5.1.1 重新排序内存操作的机会

考虑表 5.1 中描述的示例。大多数程序员会期望 r2 将始终获得值 NEW，因为 S1 在 S3 之前，而 S3 在加载 SET 值的 L1 的动态实例之前，而 L1 又在 L2 之前。我们可以这样表示：

S1 -> S3 -> L1 loads SET -> L2.
类似地，大多数程序员会期望 r3 总是得到值 NEW，因为：

S2 -> S3 -> L1 loads SET -> L3.
除了上面这两个预期的顺序，SC 和 TSO 还需要顺序 S1 -> S2 和 L2 -> L3。保留这些附加顺序可能会限制实现优化以提高性能，但程序不需要这些附加顺序来进行正确操作。

Table 5.1: What order ensures r2 and r3 always get NEW?

Core C1	Core C2	Comments
S1: data1 = NEW; S2: data2 = NEW; S3: flag = SET;	 L1: r1 = flag B1: if (r1 ≠ SET) goto L1; L2: r2=data1; L3: r3=data2;	/* Initially, data1 and data2 = 0 & flag ≠ SET */ /* spin loop: L1 & B1 may repeat many times */

表 5.2 描述了使用相同锁在两个临界区 (critical section) 之间进行切换的更一般情况。假设硬件支持锁获

取 (acquire)（例如，使用 test-and-set 执行 read-modify-write 并循环直到成功）和锁释放 (release)（例如，存储值 0）。让核心 C1 获取锁，执行临界区 1，任意交织加载 (L1i) 和存储 (S1j)，然后释放锁。类似地，让核心 C2 执行临界区 2，包括加载 (L2i) 和存储 (S2j) 的任意交织。

从临界区 1 到临界区 2 的切换的正确操作取决于这些操作的顺序：

All L1i, All S1j -> R1 -> A2 -> All L2i, All S2j.
其中逗号 (",") 分隔未指定顺序的操作。

正确的操作不依赖于每个临界区中加载和存储的任何顺序——除非操作是针对相同的地址（在这种情况下需要顺序以保持连续的处理器顺序）。即：

所有 L1i 和 S1j 可以以任何顺序相互关联，并且
所有 L2i 和 S2j 可以以任何顺序相互关联。
如果正确的操作不依赖于许多加载和存储之间的顺序，也许可以通过放宽它们之间的顺序来获得更高的性能，因为加载和存储通常比锁获取和释放要频繁得多。这就是宽松或弱模型所做的。

Table 5.2: What order ensures correct handoff from critical section 1 to 2?

Core C1	Core C2	Comments
A1: acquire(lock)		
/* Begin Critical Section 1 */		
Some loads L1i interleaved with some stores S1j		/* Arbitrary interleaving of L1i's & S1j's */
/* End Critical Section 1 */		/* Handoff from critical section 1 */
R1: release(lock)	A2: acquire(lock)	/* To critical section 2 */
	/* Begin Critical Section 2 */	
	Some loads L2i interleaved with some stores S2j	/* Arbitrary interleaving of L2i's & S2j's */
	/* End Critical Section 2 */	
	R2: release(lock)	

5.1.2 利用重新排序的机会

现在假设一个宽松的内存一致性模型，它允许我们重新排序任何内存操作，除非它们之间存在 FENCE。这种宽松的模型迫使程序员推断需要对哪些操作进行保持顺序，这是一个缺点，但它也启用了许多可以提高性能的优化。我们讨论了一些常见且重要的优化，但对该主题的深入处理超出了本入门的范围。

Non-FIFO, Coalescing Write Buffer

回想一下，TSO 启用了 FIFO 写入缓冲区，它通过隐藏提交存储的部分或全部延迟来提高性能。尽管 FIFO 写入缓冲区提高了性能，但更优化的设计将使用允许合并写入的非 FIFO 写入缓冲区（即，在程序顺序上不连续的两个存储可以写入写入缓冲区中的同一个表项）。但是，非 FIFO 合并写入缓冲区通常违反 TSO，因为 TSO 要求存储按程序顺序出现。我们的示例宽松模型允许存储在非 FIFO 写入缓冲区中合并，只要存储没有被 FENCE 分隔。

Simpler Support for Core Speculation

在具有强一致性模型的系统中，核心可能会在准备好提交之前推测性地执行超出程序顺序的加载。回想一下支持 SC 的 MIPS R10000 核心如何使用这种推测来获得比没有推测的朴素实现更好的性能。然而，问题是支持 SC 的推测核心通常必须包含检查推测是否正确的机制，即使错误推测很少见 [15, 21]。R10000 通过将逐出缓存块的地址与核心已推测加载但尚未提交的地址列表（即内核加载队列的内容）进行比较来检查推测。这种机制增加了硬件的成本和复杂性，它消耗了额外的功率，并且它代表了另一种可能限制指令级并行性的有限资源。在具有宽松内存一致性模型的系统中，核心可以不按程序顺序执行

加载，而无需将这些加载的地址与传入的连贯性请求的地址进行比较。这些加载相对于宽松一致性模型不是推测性的（尽管它们可能是推测性的，例如，分支预测或同一线程对同一地址的早期存储）。

Coupling Consistency and Coherence

我们之前提倡将一致性和连贯性解耦以管理劳神的复杂性。或者，通过“打开连贯性魔盒”，宽松模型可以提供比强模型更好的性能。例如，一个实现可能允许一个核心子集从存储中加载新值，即使其余核心仍然可以加载旧值，暂时打破了连贯性的单写多读不变量。例如，当两个线程上下文在逻辑上共享每个核心的写入缓冲区或两个核心共享 L1 数据缓存时，可能会发生这种情况。然而，“打开连贯性魔盒”会带来相当大的劳神工作和验证复杂性，让人想起关于潘多拉魔盒的希腊神话。正如我们将在第 5.6.2 节中讨论的那样，IBM Power 允许进行上述优化。我们还将在第 10 章探讨 GPU 和异构处理器为何以及如何在强制一致性的同时打开连贯性魔盒。但首先，我们探索紧密密封的连贯性魔盒的宽松模型。

5.2 宽松一致性模型 (XC) 示例

出于教学目的，本节介绍了一个示例宽松一致性模型 (XC)，该模型捕获了宽松内存一致性模型的基本思想和一些实现潜力。XC 假设存在全局内存顺序，对于 SC 和 TSO 的强模型以及 Alpha [33] 和 SPARC 的宽松内存顺序 (relaxed memory order, RMO) [34] 的大部分已失效的宽松模型也是如此。

5.2.1 XC 模型的基本思想

XC 提供了 FENCE 指令，以便程序员可以指示何时需要顺序；否则，默认情况下，加载和存储是无序的。其他的宽松一致性模型将 FENCE 称为屏障、内存屏障、membar 或同步。让核心 Ci 执行一些加载和/或存储 Xi，然后是 FENCE 指令，然后再执行一些加载和/或存储 Yi。FENCE 确保内存顺序将所有 Xi 操作排序在 FENCE 之前，而 FENCE 又在所有 Yi 操作之前。FENCE 指令不指定地址。同一核心的两个 FENCE 也保持有序。但是，FENCE 不会影响其他核心的内存操作顺序（这就是为什么 "fence" 可能是比 "barrier" 更好的名称）。一些架构包括多个具有不同顺序属性的 FENCE 指令；例如，一个体系结构可以包括一个 FENCE 指令，该指令强制执行所有顺序，除了从存储到后续加载。然而，在本章中，我们只考虑对所有类型的操作进行排序的 FENCE。

XC 的内存顺序保证尊重（保留）程序顺序：

- Load -> FENCE
- Store -> FENCE
- FENCE -> FENCE
- FENCE -> Load
- FENCE -> Store

XC 维护 TSO 规则，用于仅对同一地址进行两次访问：

- Load -> Load to the same address
- Load -> Store to the same address
- Store -> Store to the same address
  这些规则强制执行顺序处理器模型（即顺序核心语义）并禁止可能让程序员感到惊讶的行为。 例如，Store -> Store 规则防止执行 "A = 1" 然后 "A = 2" 的临界区在 A 设置为 1 的情况下奇怪地完成。同样，Load -> Load 规则确保如果 B 最初为 0 并且另一个线程执行 "B = 1"，则当前线程无法执行 "r1 = B" 然后 "r2 = B"，其中 r1 得到 1，r2 得到 0，就好像 B 的值从新变旧。

XC 确保加载由于它们自己的存储而立即看到更新（如 TSO 的写缓冲区旁路）。该规则保留了单线程的顺序性，也避免了程序员的惊讶。

5.2.2 在 XC 下使用 Fence 的示例

表 5.3 显示了程序员或低层次软件应如何在表 5.1 的程序中插入 FENCE，以便它在 XC 下正确运行。 这些 FENCE 确保：

S1, S2 -> F1 -> S3 -> L1 loads SET -> F2 -> L2, L3.
使得存储保持顺序的 F1 FENCE 对大多数读者来说很有意义，但有些人对使得加载保持顺序的 F2 FENCE 的需求感到惊讶。 但是，如果允许加载乱序执行，它们可以使看起来像顺序的存储实际上是乱序执行的。 例如，如果执行可以按 L2、S1、S2、S3、L1 和 L3 进行，则 L2 可以得到值 0。这个结果对于不包含 B1 控制依赖的程序尤其可能，因此 L1 和 L2 是对不同地址的连续加载，其中重新排序似乎是合理的，但实际上并非如此。

Table 5.3: Adding FENCEs for XC to Table 5.1's program

Core C1	Core C2	Comments
S1: data1 = NEW;		/* Initially, data1 & data2 = 0 & flag ≠ SET */
S2: data2 = NEW;		
F1: FENCE		
S3: flag = SET;	L1: r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1≠ SET) goto L1;	
	F2: FENCE	
	L2: r2 = data1;	
	L3: r3 = data2;	

表 5.4 显示了程序员或低层次软件如何在表 5.2 的临界区程序中插入 FENCE，以便它在 XC 下正确运行。 这个 FENCE 插入策略，其中 FENCE 围绕每个锁获取和锁释放，出于说明目的是保守的；我们稍后将展示其中一些 FENCE 可以被移除。 特别是，FENCE F13 和 F22 确保了关键部分之间的正确切换，因为：

All L1i, All S1j -> F13 -> R11 -> A21 -> F22 -> All L2i, AllS2j.
接下来，我们将 XC 形式化，然后说明上述两个示例为何有效。

Table 5.4: Adding FENCEs for XC to Table 5.2's critical section program. (Note that not all FENCES are strictly necessary for correctness.)

Core C1	Core C2	Comments
F11: FENCE		
A11: acquire(lock)		
F12: FENCE		
Some loads L1i interleaved		/* Arbitrary interleaving of L1i's & S1j's */
with some stores S1j		
F13: FENCE		
R11: release(lock)	**F21: FENCE**	/* Handoff from critical section 1 */
F14: FENCE	A21: acquire(lock)	/* To critical section 2 */
	F22: FENCE	
	Some loads L2i interleaved	/* Arbitrary interleaving of L2i's & S2j's */
	with some stores S2j	
	F23: FENCE	
	R22: release(lock)	
	F24: FENCE	

5.2.3 形式化 XC

在这里，我们以与前两章的符号和方法一致的方式将 XC 形式化。再一次，让 L(a) 和 S(a) 分别代表加载和存储，以寻址 a。命令 <p 和 <m 分别定义了每个处理器的程序顺序和全局内存顺序。

程序顺序 <p 是每个处理器的总顺序，它捕获每个核心在逻辑上（按顺序）执行内存操作的顺序。全局内存顺序 <m 是所有核心的内存操作的总顺序。

更正式地说，XC 执行需要以下内容。

1. 所有核心都将它们的加载、存储和 FENCE 插入到 <m 的顺序中：

If L(a) <p FENCE => L(a) <m FENCE /* Load -> FENCE /
If S(a) <p FENCE => S(a) <m FENCE / Store -> FENCE /
If FENCE <p FENCE => FENCE <m FENCE / FENCE -> FENCE /
If FENCE <p L(a) => FENCE <m L(a) / FENCE -> Load /
If FENCE <p S(a) => FENCE <m S(a) / FENCE -> Store */

2. 所有核心将它们的加载和存储插入到相同的地址到 <m 的顺序中：

If L(a) <p L'(a) => L(a) <m L' (a) /* Load -> Load to same address /
If L(a) <p S(a) => L(a) <m S(a) / Load -> Store to same address /
If S(a) <p S'(a) => S(a) <m S' (a) / Store -> Store to same address */

3. 每个加载从它之前的最后一个存储中获取它的值到相同的地址：

Value of L(a) = Value of MAX <m {S(a) | S(a) <m L(a) or S(a) <p L(a)} /* Like TSO */
我们在表 5.5 中总结了这些顺序规则。该表与 SC 和 TSO 的类似表有很大不同。从视觉上看，该表显示仅对相同地址的操作或使用 FENCE 的操作强制保持执行顺序。和 TSO 一样，如果操作 1 是“存储 C”，操作 2 是“加载 C”，则存储可以在加载后进入全局顺序，但加载必须已经看到新存储的值。

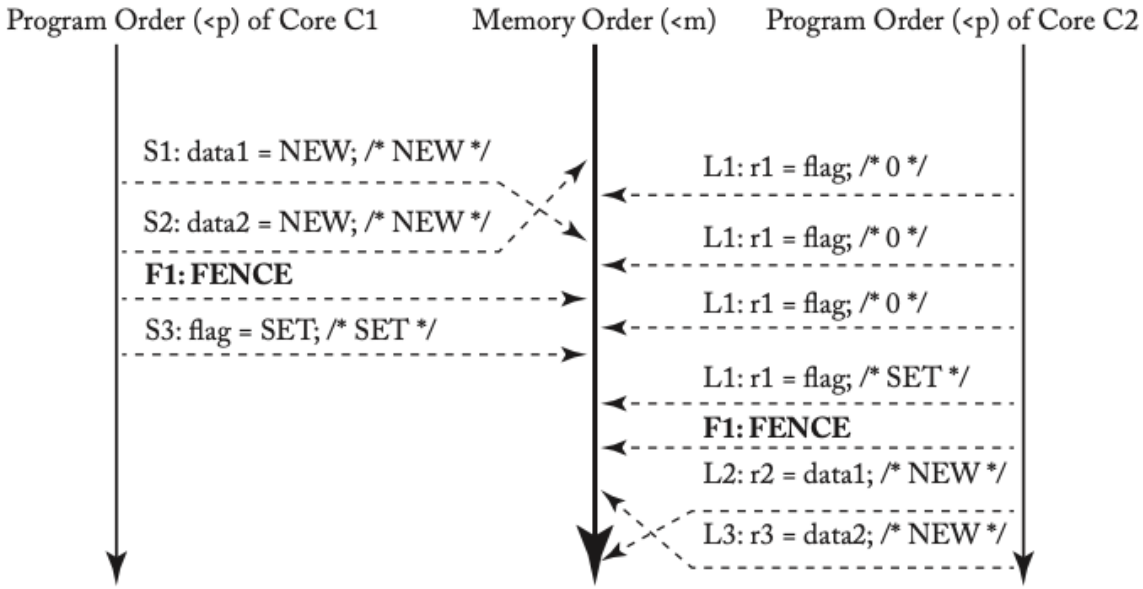
仅允许 XC 执行的实现是 XC 实现。

Table 5.5: XC ordering rules. An "X" denotes an enforced ordering. A "B" denotes that bypassing is required if the operations are to the same address. An "A" denotes an ordering that is enforced only if the operations are to the same address. Entries different from TSO are shaded and shown in bold.

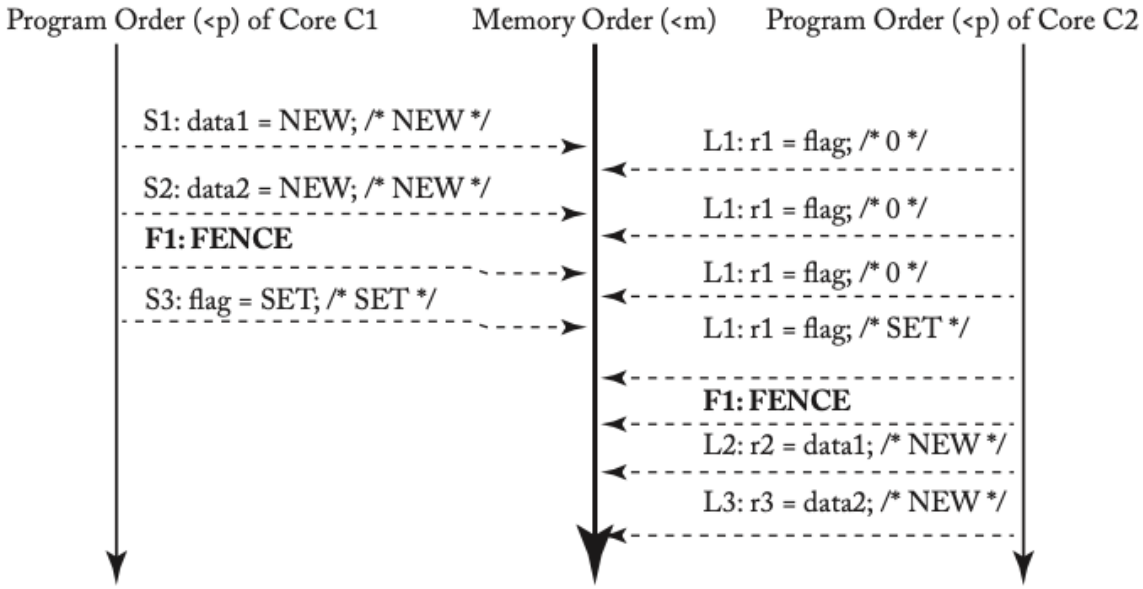
		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	**A**	**A**	**A**	X
	Store	B	**A**	**A**	X
	RMW	**A**	**A**	**A**	X
	FENCE	X	X	X	X

5.2.4 显示 XC 正常操作的示例

借助上一节的形式，我们现在可以揭示为什么第 5.2.2 节的两个示例可以正常工作。图 5.1a 显示了表 5.3 中示例的 XC 执行，其中核心 C1 的存储 S1 和 S2 被重新排序，核心 C2 的加载 L2 和 L3 也是如此。然而，重新排序都不会影响程序的结果。因此，就程序员所知，这种 XC 执行等同于图 5.1b 中描述的 SC 执行，其中两对操作没有重新排序。



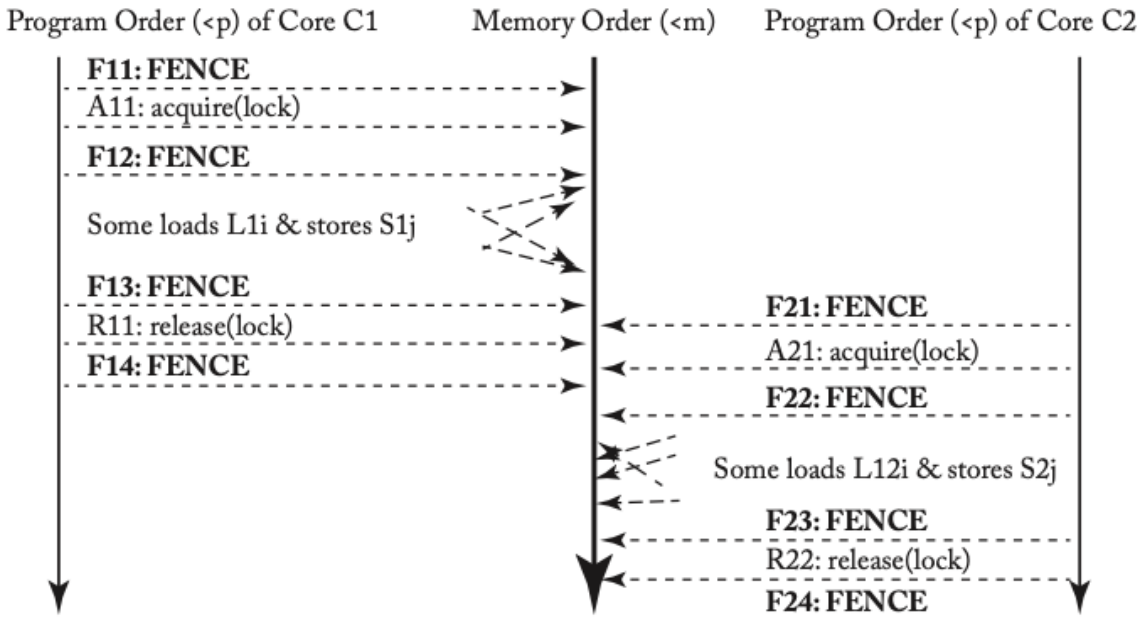
(a) An XC Execution



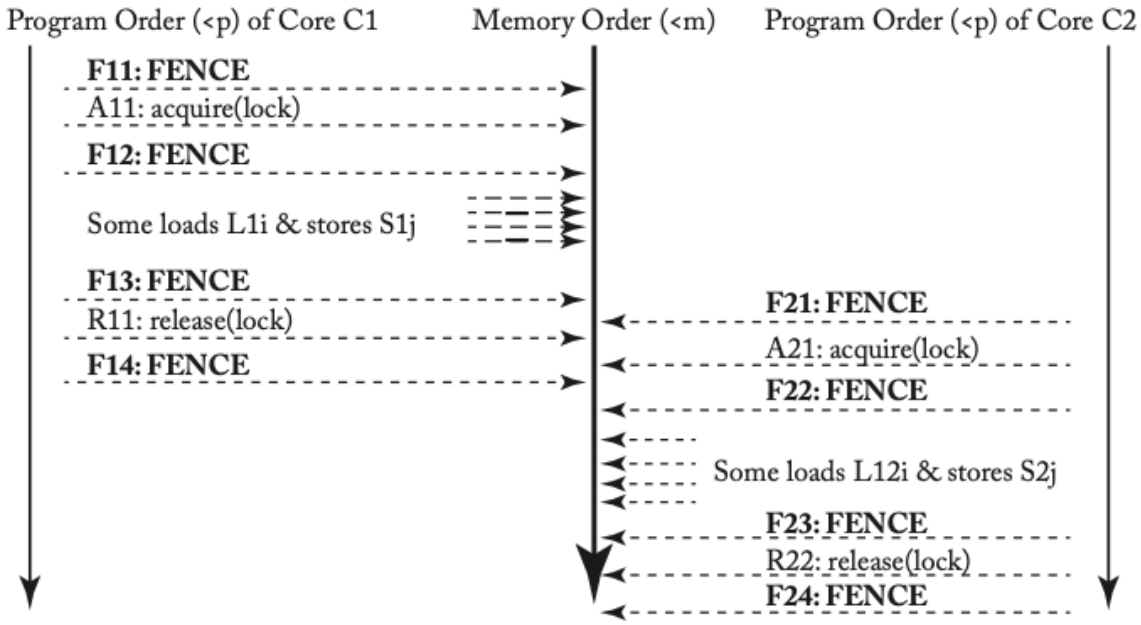
(b) An SC Execution

Figure 5.1: Two equivalent executions of Table 5.3's program.

类似地，图 5.2a 描述了表 5.4 中临界区示例的执行，其中核心 C1 的加载 L1i 和存储 S1j 相互重新排序，核心 C2 的加载 L2i 和存储 S2j 也是如此。再一次，这些重新排序不会影响程序的结果。因此，就程序员所知，这种 XC 执行等同于图 5.2b 中描述的 SC 执行，其中没有重新排序加载或存储。



(a) An XC Execution



(b) An SC Execution

Figure 5.2: Two equivalent executions of Table 5.4's critical section program.

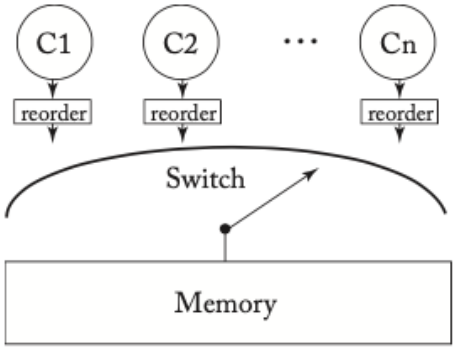
这些例子表明，如果有足够的 FENCE，像 XC 这样的宽松模型可以在程序员看来就像 SC。5.4 节讨论了从这些示例中进行泛化，但首先让我们实现 XC。

5.3 实现 XC

本节讨论实现 XC。 我们采用类似于前两章实现 SC 和 TSO 的方法，其中我们将核心操作的重新排序与缓存连贯性分开。回想一下，TSO 系统中的每个核心都通过 FIFO 写缓冲区与共享内存分开。对于 XC，每个核心将通过一个更通用的重新排序单元与内存分开，该重新排序单元可以重新排序加载和存储。

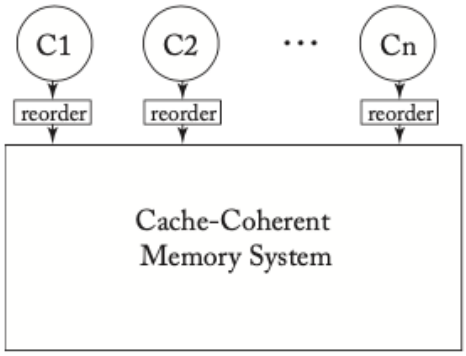
如图 5.3a 所示，XC 操作如下。

加载、存储和 FENCE 将每个核心 Ci 留在 Ci 的程序顺序 <p 中，并进入 Ci 的重新排序单元的尾部 (tail)。Ci 的重新排序单元对操作进行排队，并将它们从尾部传递到头部 (head)，按照程序顺序或按照下面指定的规则重新排序。当 FENCE 到达重新排序单元的头部时，它会被丢弃。
当交换机选择核心 Ci 时，它在 Ci 的重新排序单元的头部执行加载或存储。



This XC implementation is modeled after the SC and TSO switch implementations of the previous chapter, except that a more-general reorder unit separates cores and the memory switch.

(a) An XC implementation using a switch



This XC implementation replaces the switch above with a cache-coherent memory system in a manner analogous to what was done for SC and TSO in the previous chapters.

(b) An XC implementation using cache coherence

Figure 5.3: Two XC implementations.

重新排序单元要求这些操作遵循以下规则：(1) FENCE，(2) 对同一地址的操作，以及 (3) 旁路。

1. FENCE 可以通过几种不同的方式实现（参见第 5.3.2 节），但它们必须强制执行。具体来说，无论地址如何，重新排序单元都可能不会重新排序：

Load -> FENCE
Store -> FENCE
FENCE -> FENCE
FENCE -> Load
FENCE -> Store

2. 同一地址，重新排序单元都可能不会重新排序：

Load -> Load
Load -> Store
Store -> Store (to the same address)

3. 重新排序单元必须确保加载由于它们自己的存储而立即看到更新。

毫不奇怪，所有这些规则都反映了第 5.2.3 节的规则。

在前两章中，我们认为 SC 和 TSO 实现中的交换机和内存可以被缓存连贯的内存系统所取代。XC 也有同样的论点，如图 5.3b 所示。因此，对于 SC 和 TSO，XC 实现可以将核心（重新）排序规则与缓存连贯性的实现分开。和以前一样，缓存连贯性实现了全局内存顺序。新的是，由于重新排序单元重新排序，内存顺序可能更频繁地不尊重程序顺序。

那么从 TSO 迁移到像 XC 这样的宽松模型能提供多少性能呢？不幸的是，正确答案取决于第 5.1.2 节中讨论的因素，例如 FIFO 与合并写入缓冲区和推测支持。

在 1990 年代后期，我们中的一个人认为推测核心的趋势正在减少宽松模型（更好的性能）的存在理由，并主张回归 SC 或 TSO [22] 的更简单接口。尽管我们仍然相信简单的界面是好的，但这一举动并没有发生。原因之一是企业势头。另一个原因是，由于嵌入式芯片和/或具有许多（非对称）核心的芯片中的功率受限部署，并非所有未来的核心都具有高度推测性。

5.3.1 XC 的原子指令

在支持 XC 的系统中，有几种可行的方法来实现原子 RMW 指令。RMW 的实现还取决于系统如何实现 XC；在本节中，我们假设 XC 系统由动态调度的核心组成，每个核心都通过非 FIFO 合并写入缓冲区连接到内存系统。

在这个 XC 系统模型中，一个简单可行的解决方案是借用我们用于 TSO 的实现。在执行原子指令之前，核心会排空写缓冲区，获得具有读写连贯性权限的块，然后执行加载部分和存储部分。因为块处于读写状态，所以存储部分直接执行到缓存，绕过 (bypassing) 写缓冲区。在加载部分执行和存储部分执行之间，如果存在这样的窗口的话，则缓存控制器不得驱逐该块；如果传入的连贯性请求到达，则必须推迟到 RMW 的存储部分执行。

借用 TSO 解决方案来实现 RMW 很简单，但它过于保守并且牺牲了一些性能。值得注意的是，不需要排空写缓冲区，因为 XC 允许 RMW 的加载部分和存储部分都穿过 (pass) 较早的存储。因此，只需简单地获得对块的读写连贯性权限，然后执行加载部分和存储部分，而不放弃这两个操作之间的块就足够了。

译者注：原子 RWM 的实现。
原子 RMW 的其他实现是可能的，但它们超出了本入门书的范围。XC 和 TSO 之间的一个重要区别是如何使用原子 RMW 来实现同步。在表 5.6 中，我们描述了 TSO 和 XC 的典型临界区，包括锁获取和锁释放。使用 TSO，原子 RMW 用于尝试获取锁，并使用存储来释放锁。有了 XC，情况就更复杂了。对于获取，默认情况下，XC 不会限制 RMW 相对于临界区中的操作重新排序。为了避免这种情况，锁获取后必须跟一个 FENCE。类似地，默认情况下，锁释放不受限制于针对临界区中它之前的操作不进行重新排序。为了避免这种情况，锁释放之前必须有一个 FENCE。

Table 5.6: Synchronization in TSO vs. synchronization in XC

Code	TSO	XC
Acquire lock	RMW: test-and-set L / * read L, write L=1 */ if L==1, goto RMW /* if lock held, try again */	RMW: test-and-set L / * read L, write L=1 */ if L==1, goto RMW /* if lock held, try again */ **FENCE**
Critical Section	Loads and stores	Loads and stores
Release lock	Store L=0	**FENCE** Store L=0

5.3.2 XC FENCE

如果核心 C1 执行一些内存操作 Xi，然后是 FENCE，然后是内存操作 Yi，则 XC 实现必须保持顺序。具体来说，XC 实现必须满足顺序 Xi <m FENCE <m Yi。我们看到三种基本方法：

一个实现可以实现 SC 并将所有 FENCE 视为无操作 (nop)。这还没有在商业产品中完成，但是已经有学术建议，例如，通过隐式事务记忆 [16]。
一个实现可以等待所有内存操作 Xi 执行，认为 FENCE 完成，然后开始内存操作 Yi。这种 "FENCE as drain" 的方法很常见，但它使 FENCE 成本高昂。
实现可以积极推动必要的事情，强制执行 Xi <m FENCE <m Yi，而不会耗尽 (draining)。究竟如何做到这一点超出了本入门书的范围。虽然这种方法的设计和验证可能更复杂，但它可以带来比耗尽更好的性能。
在所有情况下，FENCE 实现必须知道每个操作 Xi 何时完成（或至少是有序的）。对于绕过 (bypass) 通

常缓存连贯性的存储（例如，存储到 I/O 设备或使用一些花哨的写更新优化的存储），知道何时完成操作可能特别棘手。

5.3.3 警告

最后，XC 实现者可能会说，“我正在实现一个宽松的模型，所以一切都会发生。” 这不是真的。必须遵守许多 XC 规则，例如 Load -> Load order to the same address（这个特定的顺序实际上对于在乱序核心中强制执行是非常重要的）。此外，所有 XC 实现必须足够强大，以便为在每对指令之间具有 FENCE 的程序提供 SC，因为这些 FENCE 需要内存顺序以尊重所有程序顺序。

5.4 无数据竞争程序的 Sequential Consistency

孩子们和计算机架构师都希望“拥有他们的蛋糕并吃掉它”。对于 memory consistency models，这可能意味着程序员能够使用（相对）直观的 SC 模型进行推理，同时仍能获得在 XC 等宽松模型上执行的性能优势。

幸运的是，对于重要的*无数据竞争 (data-race-free, DRF)* 程序 [3]，可以同时实现这两个目标。通俗地说，当两个线程访问同一个内存位置、其中至少一个访问是写入、并且没有干预的同步操作时，便会发生数据竞争。数据竞争通常（但不总是）是编程错误的结果，许多程序员试图编写 DRF 程序。*用于 DRF 编程的 SC (SC for DRF programming)* 要求程序员通过编写正确同步了的程序、并标记同步操作、来确保程序是 SC 下的 DRF (DRF under SC)；然后它要求实现者通过将标记的同步操作映射到宽松内存模型支持的 FENCE 和 RMW 上、来确保宽松模型上 DRF 程序的所有执行也是 SC 执行。XC 和大多数商用的宽松内存模型都有必要的 FENCE 指令和 RMW 来恢复 SC。此外，这种方法还可以作为 Java 和 C++ 等高级语言 (HLL) 内存模型的基础（第 5.9 节）。

让我们用两个例子来深入理解 "SC for DRF"。表 5.7 和表 5.8 都描述了 Core C1 stores 两个位置（S1 和 S2）而 Core C2 以相反顺序 loads 这两个位置（L1 和 L2）的示例。这两个示例不同，因为 Core C2 在表 5.7 中没有同步，但在表 5.8 中获取了与 Core C1 相同的锁。

Table 5.7: Example with four outcomes for XC with a data race

Core C1	Core C2	Comments
F11: FENCE		/* Initially, data1 & data2 = 0 */
A11: acquire(lock)		
F12: FENCE		
S1: data1 = NEW;	L1: r2 = data2;	
S2: data2 = NEW;	L2: r1 = data1;	/* Four Possible Outcomes under XC:
F13: FENCE		(r1, r2) =
R11: release(lock)		(0, 0), (0, NEW), (NEW, 0), or (NEW, NEW)
F14: FENCE		But has a Data Race */

Table 5.8: Example with two outcomes for XC without a data race, just like SC

Core C1	Core C2	Comments
F11: FENCE		/* Initially, data1 & data2 = 0 */
A11: acquire(lock)		
F12: FENCE		
S1: data1 = NEW;		
S2: data2 = NEW;		
F13: FENCE		
R11: release(lock)	**F21: FENCE**	
F14: FENCE	A21: acquire(lock)	
	F22: FENCE	
	L1: r2 = data2;	
	L2: r1 = data1;	/* Two Possible Outcomes under XC:
	F23: FENCE	(r1, r2) =
	R22: release(lock)	(0, 0) or (NEW, NEW)
	F24: FENCE	Same as with SC */

由于 Core C2 在表 5.7 中没有同步，它的 loads 可以与 Core C1 的 stores 同时执行。由于 XC 允许 Core C1 重排序 stores S1 和 S2（或不），并且 Core C2 重排序 loads L1 和 L2（或不），因此可能有四种结果，其中 (r1, r2) = (0, 0)、(0, NEW)、(NEW, 0) 或 (NEW, NEW)。例如，如果 loads 和 stores 按顺序 S2、L1、L2 再 S1 或顺序 L2、S1、S2 再 L1 执行，则会出现输出 (0, NEW)。但是，此示例包含两个数据争用（S1 与 L2 和 S2 与 L1），因为 Core C2 未获取 Core C1 使用的锁。

表 5.8 描述了 Core C2 获取的锁与 Core C1 获取的锁相同的情况。在这种情况下，Core C1 的临界区将在 Core C2 之前完全执行，反之亦然。这允许两种结果：(r1, r2) = (0, 0) 或 (NEW, NEW)。重要的是，这些结果不受在它们在各自的临界区内重排序的影响，即 Core C1 重排序 stores S1 和 S2 和/或 Core C2 重排序 loads L1 和 L2。“一棵树倒在树林里（重排序的 stores），但没有人听到它（没有并发的 loads）。”此外，XC 允许的结果与 SC 允许的结果相同。"SC for DRF" 从这两个例子中概括为：

- 要么一个执行具有数据竞争、暴露了 XC 的 loads 或 stores 重排序的，要么
- 这个 XC 执行是无数据竞争的，并且与 SC 执行没有区别。

对 "SC for DRF" 的更具体理解需要一些定义。

- 一些内存操作被标记为*同步操作 ("synchronization operations")*，而其余的则默认标记为*数据操作 ("data operations")*。同步操作包括锁的获取和释放。
- 如果两个数据操作 Di 和 Dj 来自不同的核心（线程）（即不按程序顺序排序），访问相同的内存位置，并且至少有一个是 store，则它们会发生*冲突 (conflict)*。
- 如果两个同步操作 Si 和 Sj 来自不同的核心（线程），访问相同的内存位置（例如，相同的锁），并且至少有一个同步操作是 write，则它们会发生*冲突 (conflict)*（例如，获取和释放一个自旋锁会冲突，而两个 read_locks 对于 reader-writer 锁则不会冲突）。
- 如果 Si 和 Sj 冲突，或者如果 Si 与某些同步操作 Sk 冲突、Sk <p Sk'（即，在核心 K 的程序顺序中，Sk 比 Sk' 早）、并且 Sk' 与 Sj 传递冲突，则两个同步操作 Si 和 Sj *传递冲突 (transitively conflict)*。
- 如果两个数据操作 Di 和 Dj 发生冲突，并且它们出现在全局内存顺序中、但却没有一对由相同核心（线程）i 和 j 介入的可传递冲突的同步操作，则两个数据操作会产生*竞争 (race)*。换句话说，当且仅当存在一对传递冲突的同步操作 Si 和 Sj 使得 Di <m Si <m Sj <m Dj 时，一对冲突的数据操作 Di <m Dj 不是数据竞争。
- 如果没有数据操作竞争，则一个 SC 执行是无数据竞争 (DRF) 的。

- 如果一个程序的所有 SC 执行都是 DRF，那么它就是 DRF。
- 如果所有 DRF 程序的所有执行都是 SC 执行，则一个 memory consistency model 支持 "SC for DRF programs"。这种支持通常需要一些特殊动作来进行同步操作。

考虑内存模型 XC。要求程序员或底层软件确保在所有同步操作之前和之后都加上 FENCEs，如表 5.8 中所示。

通过围绕同步操作的 FENCEs，XC 支持 "SC for DRF programs"。虽然证明超出了这项工作的范围，但该结果背后的直觉来自上面讨论的表 5.7 和表 5.8 中的示例。

支持 SC for DRF programs 允许许多程序员使用 SC 而不是更复杂的 XC 规则来推理他们的程序，同时受益于 XC 相对于 SC 启用的任何硬件性能改进或简化。问题是（不是总是有问题吗？）保证 DRF 的高性能（即，不将太多操作标记为同步）可能具有挑战性。

- *准确 (exactly)* 确定哪些内存操作存在竞争是不可判定 (undecidable) 的，因此必须标记为同步。图 5.4 描述了一个执行，其中，只有当我们可以确定 C1 的初始代码块是否没有停机 (halt) 时，Core C2 的 store 才应该被标记为同步（这决定了 FENCEs 是否实际上是必要的），这当然是不可判定的。当不确定是否需要 FENCE 时，可以通过添加 FENCEs 来避免不确定性。这总是正确的，但可能会损害性能。在极限情况下，可以用 FENCEs 包围所有内存操作，以确保任何程序的 SC 行为。
- 最后，程序可能存在由于 bugs 而违反 DRF 的数据竞争。坏消息是，在数据竞争之后，执行可能不再服从 SC，迫使程序员去推理底层的宽松内存模型（例如 XC）。好消息是，至少在第一次数据竞争之前，所有执行都将服从 SC，允许仅使用 SC 推理一些调试 [5]。

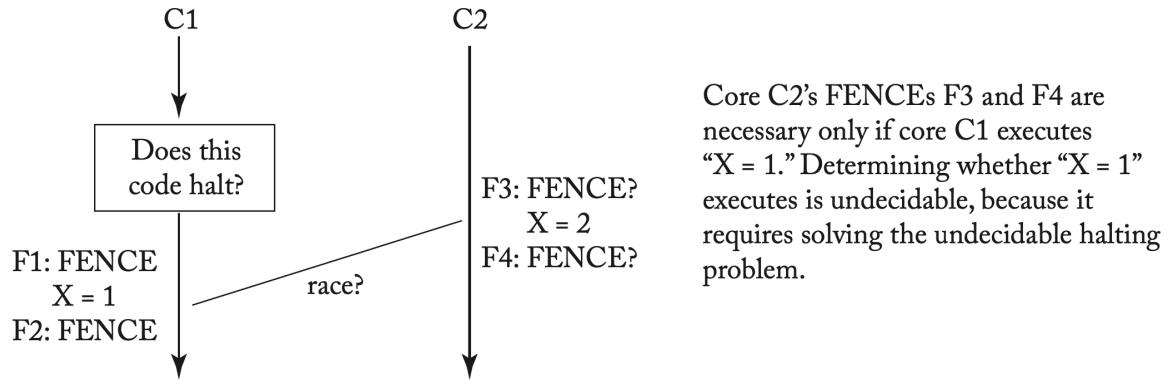


Figure 5.4: Optimal placement of FENCEs is undecidable.

总而言之，使用宽松内存系统的程序员可以通过两种选择来推理他们的程序：

- 他们可以直接使用规则来推理模型允许做什么排序和不允许做什么排序（例如，表 5.5 等），或
- 他们可以插入足够的同步以确保没有数据竞争（仍然允许同步竞争）并使用相对简单的 sequential consistency 模型来推理他们的程序，这种模型似乎永远不会出现程序顺序之外的内存操作。

我们几乎总是推荐后一种 "sequential consistency for data-race-free" 方法，将前一种方法留给编写同步库或设备驱动程序等代码的专家。

5.5 一些宽松模型的概念

学术文献提供了许多可供选择的宽松内存模型和概念。在这里，我们从大量文献中回顾一些宽松内存的概念，以提供基本的理解，但全面而正式的探索超出了本入门书的范围。幸运的是，SC for DRF 的用户，可能是大多数程序员，不必掌握这个困难部分中的概念。在第一次通过时，读者可能希望略过或跳过本节。

5.5.1 释放一致性

在 Adve 和 Hill 提出 "SC for DRF" 的同一个 ISCA 1990 会议中，Gharachorloo 等人 [20] 提出了释放一致性 (release consistency, RC)。使用本章的术语，RC 的主要观察是使用 FENCE 围绕所有同步操作是多余的。随着对同步的深入了解，同步获取 (acquire) 只需要一个后续的 FENCE，而同步释放 (release) 只需要一个前面的 FENCE。

对于表 5.4 的临界区示例，可以省略 FENCE F11、F14、F21 和 F24。让我们关注 "R11: release(lock)"。FENCE F13 很重要，因为它在锁释放之前对临界区的加载 (L1i) 和存储 (S1j) 进行排序。FENCE F14 可以省略，因为如果核心 C1 的后续内存操作（表中未显示）在释放 R11 之前执行，则没有问题。

RC 实际上允许这些后续操作早在临界区开始时就执行，而 XC 的 FENCE 不允许这样的排序。RC 提供了类似于 FENCE 的 ACQUIRE 和 RELEASE 操作，但仅在一个方向而不是像 FENCE 那样在两个方向上对内存访问进行排序。更一般地说，RC 只需要：

ACQUIRE -> Load, Store (but not Load, Store -> ACQUIRE)
Load, Store -> RELEASE (but not RELEASE -> Load, Store) and
SC ordering of ACQUIREs and RELEASEs:
ACQUIRE -> ACQUIRE
ACQUIRE -> RELEASE
RELEASE -> ACQUIRE, and
RELEASE -> RELEASE

5.5.2 因果关系和写原子性

在这里，我们说明了宽松模型的两个微妙属性。第一个属性，因果关系 (causality)，要求“如果我看到它并把它告诉你，那么你也会看到它。”例如，考虑表 5.9，其中核心 C1 执行存储 S1 以更新 data1。让核心 C2 自旋，直到看到 S1 的结果 (r1NEW)，执行 FENCE F1，然后执行 S2 更新 data2。类似地，核心 C3 在加载 L2 上自旋，直到它看到 S2 的结果 (r2NEW)，执行 FENCE F2，然后执行 L3 以观察存储 S1。如果核心 C3 保证观察到 S1 完成 (r3==NEW)，那么因果关系成立。另一方面，如果 r3 为 0，则违反了因果关系。

Table 5.9: Causality: If I see a store and tell you about it, must you see it too?

Core C1	Core C2	Core C3
S1: data1 = NEW;	L1: r1 = data1; B1: if (r ≠ NEW) goto L1; **F1: FENCE** S2: data2 = NEW;	/* Initially, data1 & data2 = 0 */ L2: r2 = data2; B2: if (r2 ≠ NEW) goto L2; **F2: FENCE** L3: r3 = data1; /* r3==NEW? */

第二个属性，写原子性 (write atomicity, also called store atomicity or multi-copy atomicity)，要求一个核心的存储在逻辑上同时被所有其他核心看到。XC 根据定义是写入原子的，因为它的内存顺序 (<m) 指定了存储在内存中生效的逻辑原子点。在此之前，没有其他核心可以看到新存储的值。在此之后，所有其他核心必须看到新值或来自以后存储的值，但不能看到被存储破坏的值。按照 XC 的要求，写入原子性允许一个核心在其他核心看到它之前看到它自己存储的值，这导致一些人认为“写入原子性”是一个糟糕的名字。

写原子性的必要但非充分条件是正确处理独立读写 (Independent Read Independent Write, IRIW) 示例。IRIW 在表 5.10 中进行了描述，其中核心 C1 和 C2 分别存储 S1 和 S2。假设核心 C3 的加载 L1 观察到 S1 (r1NEW)，核心 C4 的 L3 观察到 S2 (r3NEW)。如果 C3 的 L2 加载 0 (r20) 而 C4 的 L4 加载 0 (r40) 怎么办？前者意味着核心 C3 在看到存储 S2 之前看到存储 S1，而后者意味着核心 C4 在看到存储 S1 之前看到存储 S2。在这种情况下，存储 S1 和 S2 不仅被“重新排序”，甚至不存在存储顺序，并且违反了写入原子性。反过来不一定正确：正确处理 IRIW 并不自动意味着存储原子性。更多事实（可能会让您头疼并渴望 SC、TSO 或 SC for DRF）：

写原子性意味着因果关系。例如，在表 5.9 中，核心 C2 观察存储 S1，执行 FENCE，然后存储 S2。通过写原子性，这确保了 C3 将存储 S1 视为已完成。
因果关系并不意味着写原子性。对于表 5.10，假设核心 C1 和 C3 是共享写入缓冲区的多线程核心的两个线程上下文。假设核心 C2 和 C4 相同。让 C1 将 S1 放在 C1-C3 写缓冲区中，所以它只被 C3 的 L1 观察到。同样，C2 将 S2 放入 C2-C4 写缓冲区，因此 S2 仅由 C4 的 L3 观察。在任一存储离开写缓冲区之前，让 C3 执行 L2 和 C4 执行 L4。这种执行违反了写原子性。然而，使用表 5.9 中的示例，可以看出这种设计提供了因果关系。

Table 5.10: IRIW example: Must stores be in some order?

Core C1	Core C2	Core C3	Core C4
S1: data1 = NEW;	S2: data2 = NEW;		/* Initially, data1=data2=0 */
		L1: r1 = data1; /* NEW */	
		F1: FENCE	L3: r3 = data2; /* NEW */
		L2: r2 = data2; /* NEW? */	**F2: FENCE**
			L4: r4 = data1; /* NEW? */

最后，XC 内存模型既是存储原子的，又是保持因果关系的。我们之前认为 XC 是存储原子的。 XC 保持因果关系，因为存储原子性意味着因果关系。

5.6 宽松内存模型案例研究

在本节中，我们将介绍两个内存模型案例研究：RISC-V 的 RVWMO (RISC-V Weak Memory Order) 和 IBM 的 Power Memory 模型。

译者注：终于讲到 RISC-V 的内存模型了！

5.6.1 RISC-V 弱内存顺序 (RVWMO)

RISC-V 实现了一个内存模型 RVWMO（注1），它可以理解为释放一致性 (RC) 和 XC 的混合。与 XC 类似，RVWMO 是根据全局内存顺序（所有内存操作的总顺序）定义的，并且具有 FENCE 指令的几种变体。与 RC 类似，加载和存储可以携带注释 (annotation)：加载指令可以携带 ACQUIRE 注释，存储指令可以携带 RELEASE 注释，RMW 指令可以使用单独的 RELEASE、单独的ACQUIRE 或 RELEASE 和 ACQUIRE 两者一起进行注释。下面，我们将总结 RVWMO 内存模型，解释 RVWMO 如何结合 XC 和 RC 的各个方面。我们还将讨论 RVWMO 如何在某些方面略强于 XC，而在其他方面则较弱。

原书作者注1：RISC-V 还指定了一种称为 zTSO 的 TSO 变体，本节不讨论。
释放/获取顺序。有两种类型的 ACQUIRE 注释：ACQUIRE-RCpc 和 ACQUIRE-RCsc（注2）。同样，有两种类型的 RELEASE 注释：RELEASE-RCpc 和 RELEASE-RCsc。而加载（存储）可以携带任何一种 ACQUIRE(RELEASE) 注释，而 RMW 只能携带 RCsc注释。注释保留以下顺序：

ACQUIRE -> Load,Store (ACQUIRE refers to both ACQUIRE-RCsc and ACQUIRE-RCpc)
Load,Store -> RELEASE (RELEASE refers to both RELEASE-RCsc and RELEASE-RCpc)
RELEASE-RCsc -> ACQUIRE-RCsc
原书作者注2：PC 是指处理器一致性 (Processor Consistency)，SC 是指连续一致性 (Sequential

Consistency)。PC（第 4.5 节）是 TSO 的不太正式的前身，没有 TSO 的写入原子性属性。
FENCE 顺序。RVWMO 有多种 FENCE 指令变体。有一个强大的 FENCE 指令，

FENCE RW, RW（注3）；
它与 XC 中的指令一样，强制 Load,Store -> Load,Store 顺序。此外，还有其他五种非平凡的组合：

FENCE RW, W;
FENCE R, RW;
FENCE R, R;
FENCE W, W; and
FENCE.TSO.
例如，FENCE R, R 仅强制 Load -> Load 顺序。FENCE.TSO 强制 Load -> Load，Store -> Store 和 Load -> Store 顺序，但不强制 Store -> Load 顺序。

原书作者注3：R 表示读取（加载），W 表示写入（存储）。
一个例子。表 5.11 显示了一个包含 RELEASE-ACQUIRE 和 FENCE 顺序的示例。由于前者，S1 -> L1 在核心 C1 中强制执行；由于后者，S2 -> L2 在核心 C2 中强制执行。因此，这种组合确保 r1 和 r2 不能都读取 0。

Table 5.11: RVWMO: FENCE and RELEASE-ACQUIRE orderings ensure that both r1 and r2 cannot read 0

Core C1	Core C2	Comments
S1: RELEASE-RC$_{SC}$ x = NEW; L1: ACQUIRE-RC$_{SC}$ r1 = y;	S2: y = NEW; FENCE RW, RW; L2: r2=x;	Initially x=y=0

依赖引起的顺序。RVWMO 在某些方面略强于 XC。地址、数据和控制相关性可以在 RVWMO 中引起内存顺序，但在 XC 中则不然。考虑表 5.12 中所示的示例。这里，核心 C1 向 data2 写入 NEW，然后设置指针指向 data2 的位置。（通过 FENCE W, W 指令对两个存储 S1 和 S2 进行排序）。在核心 C2 中，L1 将指针的值加载到 r1 中，然后加载 L2 解引用 (dereference) r1。尽管两个加载 L1 和 L2 没有明确排序，但 RVWMO 隐式强制执行 L1 -> L2，因为 L1 和 L2 之间存在地址依赖 (address dependency)：L1 产生的值被 L2 解引用。

Table 5.12: Address dependency induced ordering. Can r1=&data2 and r2=0?

Core C1	Core C2	Comments
S1: data2 = NEW; FENCE W,W S2: pointer = &data2;	L1: r1 = pointer; L2: r2=*r1;	Initially pointer = & data1, data1 = 0

考虑表 5.13 中显示的示例，也称为加载缓冲 (load buffering)。让我们假设 x 和 y 最初都是 0。可以允许 r1 和 r2 凭空读取任意值（比如 42）吗？有点令人惊讶的是，XC 并不禁止这种行为。因为在 L1 和 S1 以及 L2 和 S2 之间都没有 FENCE，所以 XC 不会强制任何一个 Load -> Store 顺序。这可能会导致执行以下操作：

S1 预测 L1 将读取 42，然后推测性地将 42 写入 y，
L2 从 y 读取 42 到 r2，
S2 将 42 写入 x，并且
L1 将 42 从 x 读入 r1，从而使初始预测“正确”。

Table 5.13: Data dependency induced ordering. Can r1 and r2 read 42?

Core C1	Core C2	Comments
L1: r1 = x; S1: y = r1;	L2: r2 = y; S2: x = r2;	Initially x=y=0

译者注：注意区别，这个例子中是寄存器编号相同、而不是内存地址相同，所以不违反 XC 的规则。
译者注：顺便提一句，这边作者玩了一个 "42" 的梗，哈哈哈。
然而，RVWMO 通过隐式强制 Load -> Store 顺序 (L1 -> S1 和 L2 -> S2) 来禁止这种行为，因为每个加载和存储之间存在数据依赖 (data dependency) 关系：每个加载读取的值由接下来的存储写入。

与此类似，RVWMO 还隐式地强制在加载和后续存储之间进行排序，该存储控制依赖 (control dependent) 于加载。这是为了防止因果循环，例如表 5.14 中所示，其中存储能够影响由先前加载读取的值，该值决定存储是否必须执行。

Table 5.14: Control dependency induced ordering. Can r1 read 42, causing S1 (indirectly) to affect its own execution?

Core C1	Core C2	Comments
L1: r1=x; B1: if r1 ≠ 42 return; S1: y=42;	L2: r2 = y; S2: x= r2;	Initially x = y = 0

值得注意的是，上述所有依赖都是指句法 (syntactic) 依赖而不是语义 (semantic) 依赖，即是否存在寄存器编号依赖的函数，而不是实际值。除了地址、数据和控制之外，RVWMO 还强制执行“流水线依赖 (pipeline dependency)”以反映大多数处理器流水线实现的行为；对此的讨论超出了本入门的范围，读者可以参考 RISC-V 规范 [31]。

相同地址的顺序。回想一下，XC 为访问同一地址维护了 TSO 顺序规则。与 XC 一样，RVWMO 也强制执行

Load -> Store ordering to the same address,
Store -> Store ordering to the same address, and does not enforce
Store -> Load ordering to the same address.
与 XC 不同，RVWMO 不强制

Load -> Load ordering to the same address in all situations
仅在以下情况下才会强制执行：(a) 两次加载之间没有存储到同一地址，以及 (b) 两次加载返回由不同存储写入的值。对于这种微妙的基本原理的详细讨论，读者可以参考 RISC-V 规范 [31]。

RMW。RISC-V 支持两种类型的 RMW：原子内存操作 (AMO) 和保留加载/条件存储 (LdR/StC)。虽然 AMO 来自单个指令（例如，fetch-and-increment），但 LdR/StC 实际上由两个单独的指令组成：LdR 带来一个值并为核心做一个保留记号，而 StC 只有在保留记号仍然存在时才能成功。这两种 RMW 的原子性语义略有不同。与 XC RMW 类似，如果加载和存储在全局内存顺序中连续 (consecutively) 出现，则称 AMO 是原子的。LdR/StC 较弱。 假设 LdR 读取了 store s 产生的值；只要在全局内存顺序中 s 和 StC 之间没有相同地址的存储，就可以说 LdR/StC 是原子的。

总结。总而言之，RVWMO 是一种结合了 XC 和 RC 方面的最新宽松内存模型。对于详细的提议和正式规范，读者可以参考 RISC-V 规范 [31]。

5.6.2 IBM POWER

5.6.2 IBM POWER

IBM Power 实现了 Power 内存模型 [23]（特别参见 Book II 的第 1 章、第 4.4 节，以及附录 B）。我们试图在这里给出 Power 内存模型的要点，但我们建议读者参阅 Power 手册以获得明确的介绍，尤其是对于 Power 编程。我们不为 SC 提供像表 5.5 这样的排序表，因为我们不确定是否可以正确指定所有条目。我们只讨论正常的可缓存内存（启用“Memory Coherence”、禁用“Write Through Required”，以及禁用“Caching Inhibited”），而不是 I/O 空间等。PowerPC [24] 代表当前 Power 模型的早期版本。在阅读本入门书的第一遍时，读者可能希望浏览或跳过本节；该内存模型比本入门书中迄今为止介绍的模型要复杂得多。

Power 提供了一个宽松的模型，表面上与 XC 类似，但有以下重要区别。

First, stores in Power are performed *with respect to (w.r.t.) other cores*, not w.r.t. memory. A store by core C1 is “performed w.r.t.” core C2 when any loads by core C2 to the same address will see the newly stored value or a value from a later store, but not the previous value that was clobbered by the store. Power ensures that if core C1 uses FENCEs to order store S1 before S2 and before S3, then the three stores will be performed w.r.t. every other core Ci in the same order. In the absence of FENCEs, however, core C1’s store S1 may be performed w.r.t. core C2 but not yet performed w.r.t. to C3. Thus, Power is not guaranteed to create a total memory order (<m) as did XC.

Second, some FENCEs in Power are defined to be *cumulative*. Let a core C2 execute some memory accesses X1, X2, ..., a FENCE, and then some memory accesses Y1, Y2,.... Let set X = {Xi} and set Y = {Yi}. (The Power manual calls these sets A and B, respectively.) Define cumulative to mean three things: (a) add to set X the memory accesses by *other* cores that are ordered before the FENCE (e.g., add core C1’s store S1 to X if S1 is performed w.r.t. core C2 before C2’s FENCE); (b) add to set Y the memory accesses by *other* cores that are ordered after the FENCE by data dependence, control dependence, or another FENCE; and (c) apply (a) recursively backward (e.g., for cores that have accesses previously ordered with core C1) and apply (b) recursively forward. (FENCEs in XC are also cumulative, but their cumulative behavior is automatically provided by XC’s total memory order, not by the FENCEs specifically.)

Third, Power has three kinds of FENCEs (and more for I/O memory), whereas XC has only one FENCE.

- SYNC or HWSYNC (“HW” means “heavy weight” and “SYNC” stands for “synchronization”) orders all accesses X before all accesses Y and is cumulative.
- LWSYNC (“LW” means “light weight”) orders loads in X before loads in Y, orders loads in X before stores in Y, and orders stores in X before stores in Y. LWSYNC is cumulative. Note that LWSYNC does not order stores in X before loads in Y.
- ISYNC (“I” means “instruction”) is sometimes used to order two loads from the same core, but it is not cumulative and, despite its name, it is not a FENCE like HWSYNC and LWSYNC, because it orders instructions and not memory accesses. For these reasons, we do not use ISYNC in our examples.

Fourth, Power orders accesses in some cases even without FENCEs. For example, if load L1 obtains a value used to calculate an effective address of a subsequent load L2, then Power orders load L1 before load L2. Also, if load L1 obtains a value used to calculate an effective address or data value of a subsequent store S2, then Power orders load L1 before store S2.

TODO

5.7 进一步阅读和商用宽松内存模型

TODO

5.8 比较内存模型

TODO

5.9 高层次语言模型

TODO

5.10 参考文献

[1] (Don't) Do It Yourself: Weak Memory Models. diy release seven

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. IEEE

Computer, 29(12):66–76, December 1996. DOI: 10.1109/2.546611. 81

[3] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In Proc. of the 17th Annual International Symposium on Computer Architecture, pp. 2–14, May 1990. DOI: 10.1109/isca.1990.134502. 68, 81, 83

[4] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. IEEE Transactions on Parallel and Distributed Systems, June 1993. DOI: 10.1109/71.242161. 81

[5] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In Proc. of the 18th Annual International Symposium on Computer Archi- tecture, pp. 234–43, May 1991. DOI: 10.1145/115952.115976. 71

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In Proc. of the International Conference on Computer Aided Verification, July 2010. DOI: 10.1007/978-3-642-14295-6_25.

[7] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hard- ware. In Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, March 2011. DOI: 10.1007/978-3-642-19835-9_5.

[8] J. Alglave, L. Maranget, P.E. McKenney, A. Parri, and A. Stern. Frightening small chil- dren and disconcerting grown-ups: Concurrency in the Linux kernel. In ASPLOS, 2018. DOI: 10.1145/3296957.3177156. 82

[9] ARM. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition Errata Markup. Downloaded January 13, 2011. 81, 82

[10] ARM. ARM v7A+R Architectural Reference Manual. Available from ARM Ltd. 81, 82

[11] ARM. ARM v8 Architectural Reference Manual. Available from ARM Ltd. 82

[12] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In ESOP, 2015. DOI: 10.1007/978-3- 662-46669-8_12. 86

[13] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In POPL, 2011. DOI: 10.1145/1926385.1926394. 86

[14] H.-J. Boehm and S. V. Adve. Foundations of the CCC concurrency memory model. In Proc. of the Conference on Programming Language Design and Implementation, June 2008. DOI: 10.1145/1375581.1375591. 83

[15] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In Proc. of the 31st Annual International Symposium on Computer Architecture, June 2004. DOI: 10.1109/isca.2004.1310766. 58

[16] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In Proc. of the 34th Annual International Symposium on Computer Architecture, June 2007. DOI: 10.1145/1250662.1250697. 68

[17] S.ChakrabortyandV.Vafeiadis.Groundingthin-airreadswitheventstructures.InPOPL, 2019. DOI: 10.1145/3290383. 86

[18] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In Proc. of the 13th Annual International Symposium on Computer Architecture, pp. 434–42, June 1986. DOI: 10.1145/285930.285991. 81

[19] S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, and P. Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 429–442, January 2017. DOI: 10.1145/3009837.3009839. 81

[20] 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. 72, 81

[21] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC C ILP D RC? In Proc. of the 26th Annual International Symposium on Computer Architecture, pp. 162–71, May 1999. DOI: 10.1109/isca.1999.765948. 58

[22] M. D. Hill. Multiprocessors should support simple memory consistency models. IEEE Computer, 31(8):28–34, August 1998. DOI: 10.1109/2.707614. 66

[23] IBM. Power ISA Version 2.06 Revision B. http://www.power.org/resources/ downloads/ PowerISA_V2.06B_V2_PUBLIC.pdf, July 2010. 78, 81

[24] IBM Corporation. Book E: Enhanced PowerPC Architecture, Version 0.91, July 21, 2001. 78

[25] B.W.KernighanandD.M.Ritchie.TheCProgrammingLanguage,2nded.,PrenticeHall,

1988. 83

[26] S. Mador-Haim, R. Alur, and M. M. K. Martin. Generating litmus tests for contrasting memory consistency models. In Proc. of the 22nd International Conference on Computer Aided Verification, July 2010. DOI: 10.1007/978-3-642-14295-6_26. 82

[27] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In Proc. of the 32nd Sym- posium on Principles of Programming Languages, January 2005. http://dx.doi.org/10. 1145/1040305.1040336.1040336 DOI: 10.1145/1040305.1040336. 83

[28] P. E. McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It? http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook. 2011.01.%05a.pd, 2011. 81

[29] A. Meixner and D. J. Sorin. Dynamic verification of memory consistency in cache- coherent multithreaded computer architectures. In Proc. of the International Conference on Dependable Systems and Networks, pp. 73–82, June 2006. DOI: 10.1109/dsn.2006.29. 81

[30] W. Pugh. The Java memory model is fatally flawed. Concurrency: Practice and Experience, 12(1):1–11, 2000. DOI: 10.1002/1096-9128(200005)12:6%3C445::aid- cpe484%3E3.0.co;2-a. 83

[31] The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA https: //github.com/riscv/riscv-isa-manual/releases/download/draft-20190521- 21c6a14/riscv-spec.pdf DOI: 10.21236/ada605735. 77, 78

[32] J. Ševčík, and D. Aspinall. On validity of program transformations in the Java memory model. In ECOOP, 2008. DOI: 10.1007/978-3-540-70592-5_3. 85

[33] R. L. Sites, Ed. Alpha Architecture Reference Manual. Digital Press, 1992. 58, 81

[34] D.L.WeaverandT.Germond,Eds.SPARCArchitectureManual(Version9).PTRPrentice Hall, 1994. 58, 81, 82