

第三章：内存 Consistency 动机和顺序 Consistency

本章深入研究了内存 consistency 模型（又名内存模型），这些模型为程序员和实现者定义了共享内存系统的行为。这些模型定义了正确性，以便程序员知道期望什么，实现者知道提供什么。我们首先激发了定义内存行为的需求（第 3.1 节），说明内存 consistency 模型应该做什么（第 3.2 节），以及比较和对比 consistency 和 coherence（第 3.3 节）。

然后，我们探索（相对）直观的顺序 consistency (sequential consistency, SC) 模型。SC 很重要，因为它是许多程序员对共享内存的期望，并为理解接下来两章中介绍的更宽松（弱, weak）内存 consistency 模型提供了基础。我们首先介绍 SC 的基本思想（第 3.4 节），并介绍它的一种形式，我们也将后续章节（第 3.5 节）中使用它。然后我们讨论 SC 的实现，从用作操作模型的朴素实现开始（第 3.6 节），具有缓存 coherence 的 SC 的基本实现（第 3.7 节），具有缓存 coherence 的更优化的 SC 实现（第 3.8 节），以及原子操作的实现（第 3.9 节）。我们通过提供 MIPS R10000 案例研究（第 3.10 节）并指出一些进一步的阅读材料（第 3.11 节）来结束对 SC 的讨论。

3.1 共享内存行为的问题

要了解为什么必须定义共享内存行为，请考虑表 3.1 中描述的两个核心（注1）的示例执行。（这个例子和本章中的所有例子一样，假设所有变量的初始值都是零。）大多数程序员会期望核心 C2 的寄存器 r2 应该得到值 NEW。然而，在当今的某些计算机系统中，r2 可以为 0。

原书作者注 1：“核心”指的是软件视角下的核心，它可能是一个实际的核心或多线程核心的线程上下文。

Table 3.1: Should r2 always be set to NEW?

Core C1	Core C2	Comments
S1: Store data = NEW; S2: Store flag = SET;	L1: Load r1 = flag; B1: if (r1 ≠ SET) goto L1; L2: Load r2 = data;	/* Initially, data = 0 & flag ≠ SET */ /* L1 & B1 may repeat many times */

硬件可以通过重新排序核心 C1 的存储 S1 和 S2 来使 r2 的值为 0。在本地（即，如果我们只查看 C1 的执行而不考虑与其他线程的交互），这种重新排序似乎是正确的，因为 S1 和 S2 访问不同的地址。（原书）第 18 页的侧边栏描述了硬件可能重新排序内存访问的几种方式，包括这些存储。非硬件专家的读者可能希望相信这种重新排序会发生（例如，使用不是先进先出的写入缓冲区）。

Sidebar: How a Core Might Reorder Memory Access

这个侧边栏描述了现代核心可以重新排序对不同地址的内存访问的几种方式。那些不熟悉这些硬件概念的人可能希望在第一次阅读时跳过这一点。现代核心可能会重新排序许多内存访问，但只需要考虑重新排序的两个内存操作即可。在大多数情况下，我们只需要推理一个核心将两个内存操作重新排序到两个不同的地址，因为顺序执行（即冯诺依曼）模型通常要求对同一地址的操作以原始程序顺序执行。根据重新排序的内存操作是加载还是存储，我们将可能的重新排序分为三种情况。

Store-store reordering. 如果一个核心有一个非 FIFO 写缓冲区，那么两个存储可能会被重新排序，使得存储以不同于它们进入的顺序离开。如果第一个存储在缓存中未命中而第二个命中或第二个存储可以与较早的存储合并（即，在第一个存储之前），则可能会发生这种情况。请注意，即使核心按程序顺序执行所有指令，这些重新排序也是可能的。将存储重新排序到不同的内存地址对单线程执行没有影响。然而，在表 3.1 的多线程示例中，重新排序 Core C1 的存储允许 Core C2 在

看到存储到数据之前将标志视为 SET。请注意，即使写入缓冲区排入完全 coherent 内存层次结构，问题也没有解决。Coherence 将使所有缓存不可见，但存储已重新排序。

Load-load reordering. 现代动态调度的核心可能会不按程序顺序执行指令。在表 3.1 的示例中，Core C2 可以乱序执行加载 L1 和 L2。仅考虑单线程执行，这种重新排序似乎是安全的，因为 L1 和 L2 指向不同的地址。但是，重新排序 Core C2 的加载与重新排序 Core C1 的存储的行为相同；如果内存引用按 L2、S1、S2 和 L1 的顺序执行，则将 r2 分配为 0。如果省略了分支语句 B1，则这种情况更加合理，因此没有控制依赖性将 L1 和 L2 分开。

Load-store 和 store-load reordering. 乱序核心也可以重新排序来自同一线程的加载和存储（到不同的地址）。将较早的加载与较晚的存储重新排序（加载-存储重新排序）可能会导致许多错误行为，例如在释放保护它的锁后加载一个值（如果存储是解锁操作）。表 3.3 中的示例说明了重新排序较早存储与较晚加载（存储加载重新排序）的效果。重新排序 Core C1 的访问 S1 和 L1 以及 Core C2 的访问 S2 和 L2 会导致 r1 和 r2 都为 0 的违反直觉的结果。请注意，由于通常实现的 FIFO 写入缓冲区中的本地旁路，即使使用按程序顺序执行所有指令的核心，存储加载重新排序也可能出现。

读者可能会假设硬件不应该允许部分或全部这些行为，但是如果没有更好地理解允许哪些行为，就很难确定硬件可以做什么和不能做什么。

通过 S1 和 S2 的重新排序，执行顺序可能是 S2、L1、L2、S1，如表3.2所示。

Table 3.2: One possible execution of program in Table 3.1

Cycle	Core C1	Core C2	Coherence State of Data	Coherence State of Flag
1	S2: Store flag = SET		Read-only for C2	Read-write for C1
2		L1: Load r1=flag	Read-only for C2	Read-only for C2
3		L2: Load r2=data	Read-only for C2	Read-only for C2
4	S1: Store data = NEW		Read-write for C1	Read-only for C2

Table 3.3: Can both r1 and r2 be set to 0?

Core C1	Core C2	Comments
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0 */

这个执行满足 coherence，因为没有违反 SWMR 属性，所以 incoherence 不是这个看似错误的执行结果的根本原因。

让我们考虑另一个受 Dekker 算法启发的用于确保互斥的重要示例，如表 3.3 所示。执行后，r1 和 r2 中允许哪些值？直觉上，人们可能会认为存在三种可能性：

- (r1, r2) = (0, NEW)，执行顺序是 S1、L1、S2，然后是 L2
- (r1, r2) = (NEW, 0)，执行顺序是 S2、L2、S1 和 L1
- (r1, r2) = (NEW, NEW)，例如，执行顺序可以是 S1、S2、L1 和 L2

令人惊讶的是，大多数实际硬件，例如 Intel 和 AMD 的 x86 系统，也允许 (r1, r2) = (0, 0)，因为它使用先进先出 (FIFO) 写入缓冲区来提高性能。与表 3.1 中的示例一样，所有这些执行都满足缓存 coherence，甚至 (r1, r2) = (0, 0)。

一些读者可能会反对这个例子，因为它是不确定的（允许多个结果）并且可能是一个令人困惑的编程习惯。然而，首先，所有当前的多处理器在默认情况下都是非确定性的。我们所知的所有体系结构都允许并发线程的执行有多种可能的交错。确定性的错觉有时由具有适当同步习惯的软件创建，但并非总是如此。因此，我们在定义共享内存行为时，必须考虑非确定性。

此外，内存行为通常为所有程序的所有执行而定义，即使是那些不正确或故意微妙的（例如，对于非阻塞同步算法）。然而，在第 5 章中，我们将看到一些高级语言模型，它们允许某些执行具有未定义的行为，例如，执行具有数据竞争的程序。

3.2 什么是内存 Consistency 模型？

上一小节中的示例说明共享内存行为是微妙的，有助于精确定义 (a) 程序员可以预期的行为和 (b) 系统实现者可以使用的优化。内存 consistency 模型消除了这些问题。

内存 consistency 模型，或者更简单地说，内存模型，是对使用共享内存执行的多线程程序的允许行为的规范。对于使用特定输入数据执行的多线程程序，它指定动态加载可能返回的值。与单线程执行不同，通常允许多个正确的行为。

通常，内存 consistency 模型 MC 给出了将执行划分为服从 MC（MC 执行）和不服从 MC（非 MC 执行）的规则。这种执行分区反过来又对具体实现进行分区。MC 实现是只允许 MC 执行的系统，而非 MC 实现有时允许非 MC 执行。

最后，我们一直对编程层面含糊其辞。我们首先假设程序是硬件指令集架构中的可执行文件，并且我们假设内存访问是对由物理地址标识的内存位置（即，我们不考虑虚拟内存和地址转换的影响）。在第 5 章中，我们将讨论高级语言（High-level language, HLL）的问题。然后我们将看到，例如，将变量分配给寄存器的编译器会以类似于硬件重新排序内存引用的方式影响 HLL 内存模型。

3.3 Consistency vs Coherence

第 2 章用两个不变量定义了缓存 coherence，我们在这里非正式地再重复一下。SWMR (single-writer-multiple-reader) 不变量确保在任何时候对于具有给定地址的内存位置，(a) 一个核心可以写入（和读取）该地址，或者 (b) 零个或多个核心只能读取它。Data-Value 不变量确保正确传递对内存位置的更新，以便内存位置的缓存副本始终包含最新版本。

似乎缓存 coherence 定义了共享内存行为。它不是。正如我们从图 3.1 中看到的，coherence 协议只是为处理器核心流水线提供了一个内存系统的抽象。它本身不能确定共享内存的行为；流水线也很重要。例如，如果流水线以与程序顺序相反的顺序对 coherence 协议重新排序和呈现内存操作——即使 coherence 协议正确地完成了它的工作——共享内存的正确性也可能不会随之而来。

总之：

- 缓存 coherence 不等于内存 consistency。
- 内存 consistency 实现可以将缓存 coherence 用作有用的“黑盒”。

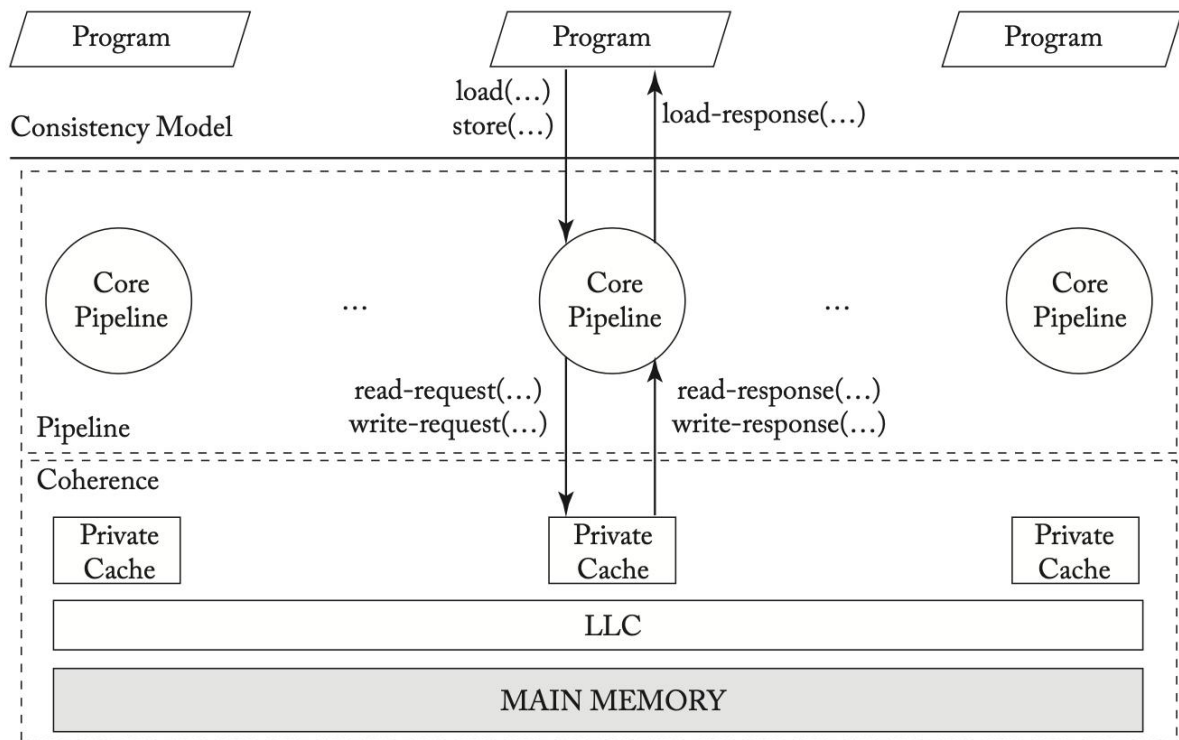


Figure 3.1: A consistency model is enforced by the processor core pipeline combined with the coherence protocol.

3.4 顺序 Consistency (SC) 的基本思想

可以说，最直观的内存 consistency 模型是 SC。它最早由 Lamport [12] 形式化，他将单个处理器（核心）称为顺序执行，如果“执行的结果与按照程序指定的顺序执行操作相同”。然后，他称多处理器为顺序一致（sequentially consistent），前提是“任何执行的结果与所有处理器（核心）的操作都以某种顺序执行的结果相同，并且每个单独的处理器（核心）的操作都以该顺序出现在其程序指定的顺序。”这种操作的总顺序称为内存顺序。在 SC 中，内存顺序遵循每个核心的程序顺序，但其他 consistency 模型可能允许内存顺序并不总是遵循程序顺序。

译者注，第一段原文为：“the result of an execution is the same as if the operations had been executed in the order specified by the program.”

译者注，第二段原文为：“the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program.”

图 3.2 描述了执行表 3.1 中的示例程序。中间的垂直向下箭头表示内存顺序 ($<_m$)，而每个核心的向下箭头表示其程序顺序 ($<_p$)。我们使用运算符 $<_m$ 表示内存顺序，因此 $op1 <_m op2$ 意味着 $op1$ 在内存顺序上先于 $op2$ 。类似地，我们使用运算符 $<_p$ 来表示给定核心的程序顺序，因此 $op1 <_p op2$ 意味着在该核心的程序顺序中 $op1$ 在 $op2$ 之前。在 SC 下，内存顺序遵循每个核心的程序顺序。“遵循”意味着 $op1 <_p op2$ 意味着 $op1 <_m op2$ 。注释 ($/* \dots */$) 中的值给出了加载或存储的值。此执行以 $r2$ 为 NEW 结束。更一般地，表 3.1 程序的所有执行都以 $r2$ 作为 NEW 终止。唯一的不确定性——L1 在加载值 SET 一次之前加载 flag 为 0 的次数——并不重要。

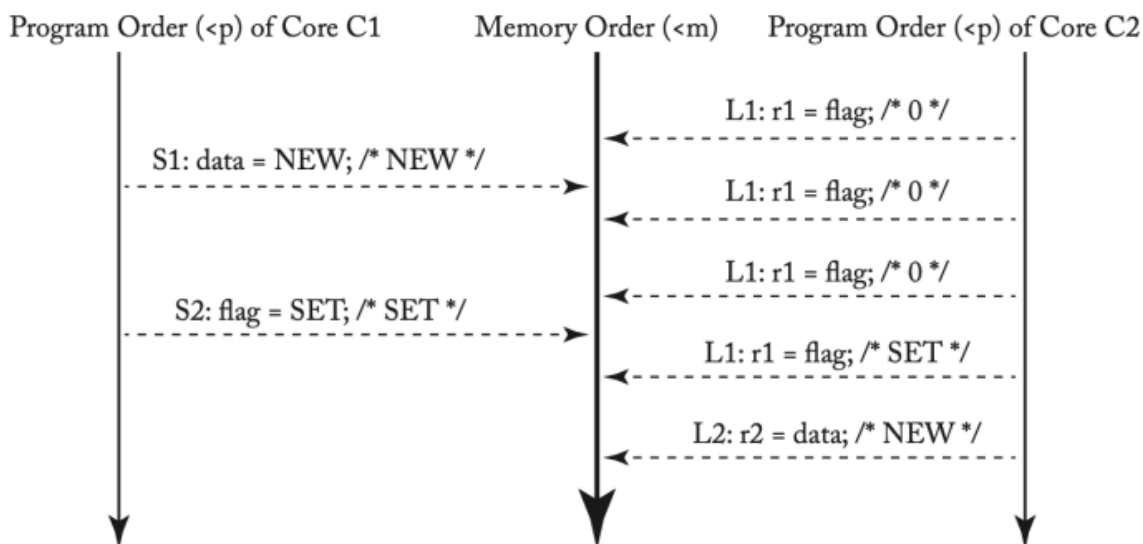


Figure 3.2: A sequentially consistent execution of Table 3.1's program.

这个例子说明了 SC 的价值。在 3.1 节中，如果您预计 r2 一定是 NEW，那么您可能独立发明了 SC，尽管不如 Lamport 精确。

图 3.3 进一步揭示了 SC 的值，该图说明了表 3.3 中程序的四次执行。图 3.3a-c 描述了对应于三个直观输出的 SC 执行：(r1, r2) = (0, NEW)、(NEW, 0) 或 (NEW, NEW)。请注意，图 3.3c 仅描述了导致 (r1, r2) = (NEW, NEW); 的四种可能的 SC 执行中的一种。本次执行为{S1, S2, L1, L2}，其他为{S1, S2, L2, L1}, {S2, S1, L1, L2}, 和{S2, S1, L2, L1}。因此，在图 3.3a-c 中，有六次合法的 SC 执行。

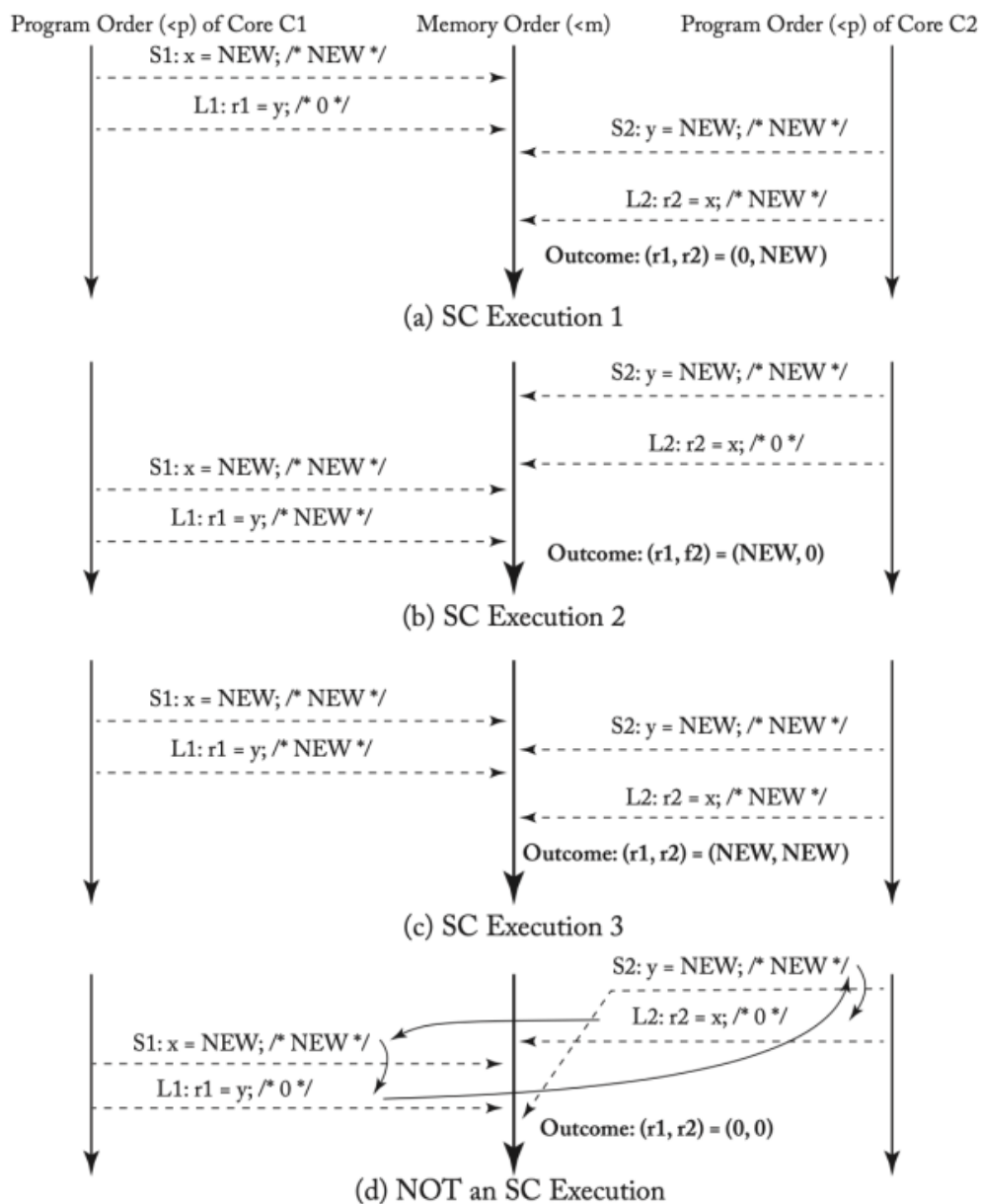


Figure 3.3: Four alternative executions of Table 3.3's program.

图 3.3d 显示了对应于输出 $(r1, r2) = (0, 0)$ 的非 SC 执行。对于此输出，无法创建尊重程序顺序的内存顺序。程序顺序规定：

- $S1 <_p L1$
- $S2 <_p L2$

但内存顺序规定：

- $L1 <_m S2$ (so $r1$ is 0)
- $L2 <_m S1$ (so $r2$ is 0)

遵守所有这些约束会导致循环，这与总顺序不一致。图 3.3d 中的额外弧线说明了循环。

我们刚刚看到了六次 SC 执行和一次非 SC 执行。这可以帮助我们理解 SC 实现：一个 SC 实现必须允许前六次执行中的一个或多个，但不能允许第七次执行。

3.5 一点 SC 形式化方法

在本节中，我们更精确地定义了 SC，特别是为了让我们能够在接下来的两章中将 SC 与较弱的 consistency 模型进行比较。我们采用 Weaver 和 Germond [20] 的形式化方法——一种指定 consistency 的公理方法，我们将在第 11 章中详细讨论——使用以下符号：L(a) 和 S(a) 分别表示加载和存储，以地址 a。命令 <p 和 <m 分别定义程序和全局内存顺序。程序顺序 <p 是每个核心的总顺序，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。

SC 执行 (SC execution) 需要以下内容。

(1) 所有核心都按照程序顺序将它们的加载和存储插入到 <m 的顺序中，无论它们是相同还是不同的地址（即 a=b 或 a != b）。有四种情况：

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

(2) 每个加载都从它之前的最后一个存储（按全局内存顺序）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)}, where MAX <m denotes "latest in memory order"

原子 read-modify-write (RMW) 指令，我们将在 3.9 节中更深入地讨论，进一步限制了允许的执行。例如，test-and-set 指令的每次执行都要求 test 的加载和 set 的存储在内存顺序中逻辑上连续出现（即，相同或不同地址的其他内存操作不会插入它们之间）。

我们在表 3.4 中总结了 SC 的 ordering 要求。该表指定了 consistency 模型强制执行的程序顺序。例如，如果给定线程在程序顺序中在存储之前有一个加载（即，加载是“操作 1”，存储是表中的“操作 2”），那么这个交点处的表条目是“X”这表示这些操作必须按程序顺序执行。对于 SC，所有内存操作都必须按照程序顺序执行；在我们在接下来的两章中研究的其他 consistency 模型下，其中一些排序约束是放松的（即，它们的排序表中的一些条目不包含“X”）。

Table 3.4: SC ordering rules. An “X” denotes an enforced ordering.

		Operation 2		
Operation 1		Load	Store	RMW
	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

SC 实现 (SC implementation) 只允许 SC 执行。严格来说，这是 SC 实现的 safety 属性（无害）。SC 实现也应该有一些 liveness 属性（做一些好事）。具体来说，存储必须最终对反复尝试加载该位置的加载可见。这种被称为最终写入传播 (eventual write-propagation) 的属性通常由 consistency 协议确保。更一般地说，避免饥饿和一些公平也很有价值，但这些问题超出了本次讨论的范围。

3.6 朴素的 SC 实现

SC 允许两个简单的实现，这使得更容易理解 SC 允许哪些执行。

多任务单处理器 (The Multitasking Uniprocessor)

首先，可以通过在单个顺序核心（单处理器）上执行所有线程来为多线程用户级软件实现 SC。线程 T1 的指令在核心 C1 上执行，直到上下文切换到线程 T2 等。在上下文切换时，必须在切换到新线程之前完成任何挂起的内存操作。因为每个线程在其量子中的指令作为一个原子块执行（并且因为单处理器正确地尊重内存依赖性），所以所有的 SC 规则都被强制执行。

交换机 (The Switch)

其次，可以使用一组核心 C、单个交换机和内存来实现 SC，如图 3.4 所示。假设每个核心按其程序顺序一次向交换机呈现内存操作。每个核心都可以使用任何不影响它向交换机呈现内存操作的顺序的优化。例如，可以使用带有分支预测的简单五级顺序流水线。

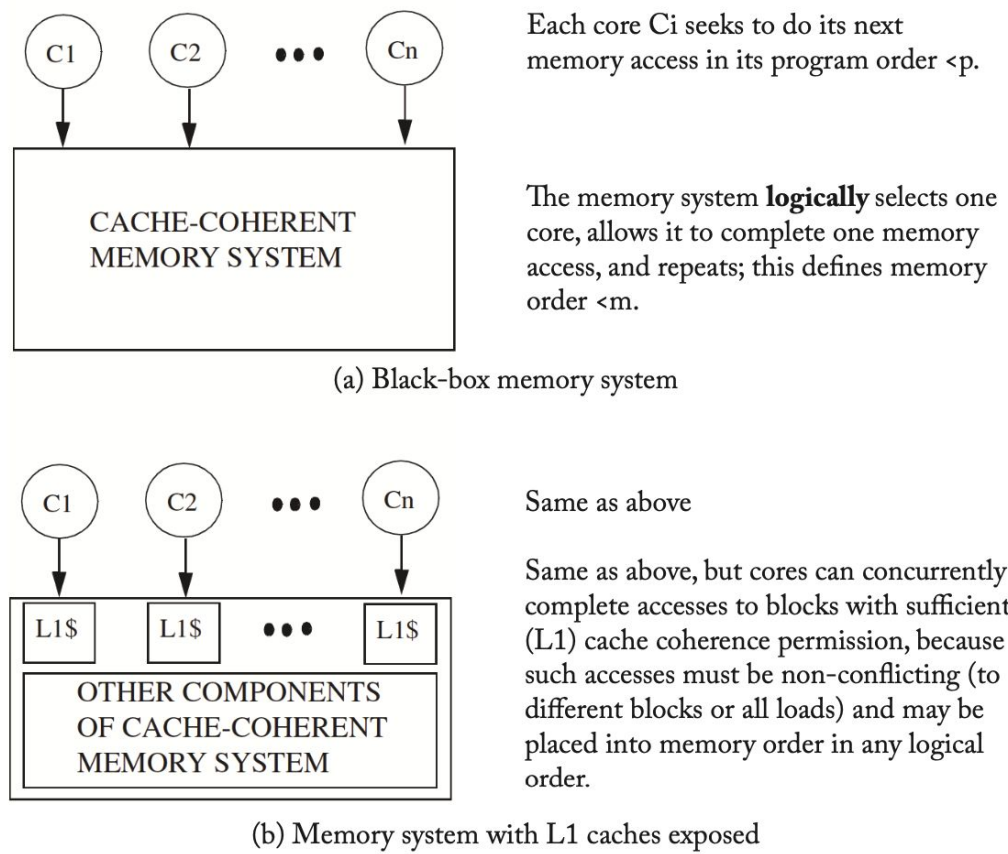


Figure 3.5: Implementing SC with cache coherence.

接下来假设交换机选择一个核心，让内存完全满足加载或存储，只要请求存在，就重复这个过程。交换机可以通过任何方法（例如，随机）选择核心，该方法不会因就绪请求而使核心饿死。此实现通过构造在操作上实现 SC。

评估 (Assessment)

这些实现的好消息是它们提供了操作模型，定义了 (1) 允许的 SC 执行和 (2) SC 实施“黄金标准”。（在第 11 章中，我们将看到这样的操作模型可以用来正式指定 consistency 模型。）交换机实现还作为存在证明，SC 可以在没有缓存或 coherence 的情况下实现。

当然，坏消息是，由于在第一种情况下使用单个核心和在第二种情况下使用单个交换机/内存的顺序瓶颈，这些实现的性能不会随着核心数量的增加而扩展。这些瓶颈导致一些人错误地认为 SC 排除了真正的并行执行。它没有，我们将在下面看到。

3.7 具有缓存 Coherence 的基本 SC 实现

缓存 coherence 促进了可以执行非冲突加载和存储的 SC 实现——两个操作冲突，如果它们指向相同的地址并且其中至少一个是存储——完全并行。此外，创建这样一个系统在概念上很简单。

在这里，我们将 coherence 主要视为实现第 2 章的 SWMR 不变量的黑盒。我们通过稍微打开 coherence 块盒以显示简单的一级 (L1) 缓存来提供一些实现直觉：

- 使用状态修改 (M) 表示一个核心可以读写的 L1 块，
- 使用状态共享 (S) 表示一个或多个核心只能读取的 L1 块，以及
- GetM 和 GetS 分别表示在 M 和 S 中获得块的 coherence 请求。

正如第 6 章及以后讨论的那样，我们不需要深入了解如何实现 coherence。

图 3.5a 描绘了图 3.4 的模型，其中交换机和内存被一个缓存 coherent 内存系统取代，表示为一个黑盒子。每个核心按其程序顺序一次向缓存 coherent 内存系统提供内存操作。在开始对同一核心的下一个请求之前，内存系统会完全满足每个请求。

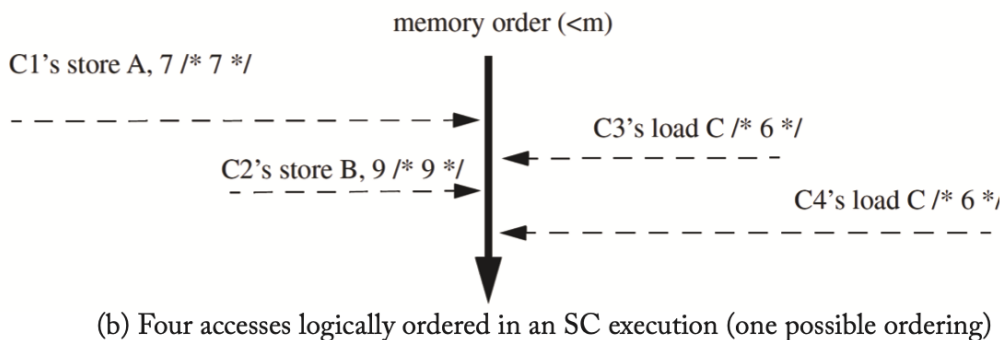
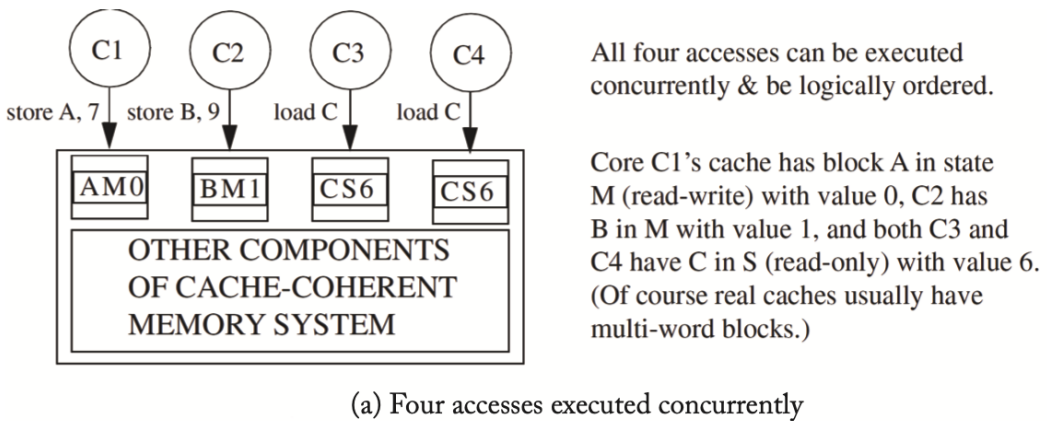


Figure 3.6: A concurrent SC execution with cache coherence.

图 3.5b 稍微“打开”了内存系统黑匣子，显示每个核心都连接到自己的 L1 缓存（稍后我们将讨论多线程）。如果内存系统具有 B 适当的 coherence 权限（加载或存储状态 M 或 S，存储状态 M），则存储系统可以响应对 B 的加载或存储。此外，内存系统可以并行响应来自不同核心的请求，前提是相应的 L1 缓存具有适当的权限。例如，图 3.6a 描绘了四个核心各自寻求执行内存操作之前的缓存状态。这四个操作不冲突，可以由各自的 L1 缓存来满足，因此可以同时进行。如图 3.6b 所示，我们可以任意排序这些操作以获得合法的 SC 执行模型。更一般地说，L1 缓存可以满足的操作总是可以同时完成，因为 coherence 的 SWMR 不变量确保它们是不冲突的。

译注：可以回顾 2.4 小节，串在一起理解。M 状态即为单核 read-write 时的状态；S 状态即为多核 read-only 时的状态。

Flashback to Quiz Question 1: In a system that maintains sequential consistency, a core must issue coherence requests in program order. *True or false?*
Answer: *False!* A core may issue coherence requests in any order.

评估 (Assessment)

我们创建了一个 SC 实现：

- 充分利用缓存的延迟和带宽优势，
- 与它使用的缓存 coherence 协议一样具有可扩展性，并且
- 将实现核心的复杂性与实现 coherence 分离开来。

3.8 使用缓存 Coherence 优化 SC 实现

大多数真正的核心实现比我们具有缓存 coherence 的基本 SC 实现更复杂。核心采用预取、推测执行和多线程等功能来提高性能并容忍内存访问延迟。这些特性与内存接口交互，我们现在讨论这些特性如何影响 SC 的实现。值得记住的是，任何功能或优化都是合法的，只要它不会产生违反 SC 的最终结果（加载返回的值）。

非绑定预取 (Non-Binding Prefetching)

块 B 的非绑定预取是对 coherent 内存系统的请求，以更改 B 在一个或多个缓存中的 coherence 状态。最常见的是，软件、核心硬件或缓存硬件请求预取以更改 B 在一级缓存中的状态，以允许通过以下方式加载（例如，B 的状态是 M 或 S）或加载和存储（B 的状态是 M）发出 coherence 请求，例如 GetS 和 GetM。重要的是，在任何情况下，非绑定预取都不会更改块 B 中的寄存器或数据的状态。非绑定预取的影响仅限于图 3.5a 的“缓存 coherence 内存系统”块内，使得非绑定预取对内存 consistency 模型的影响相当于 no-op 的功能。只要按程序顺序执行加载和存储，以什么顺序获得 coherence 权限都没有关系。

实现可以在不影响内存 consistency 模型的情况下进行非绑定预取。这对于内部缓存预取（例如，流缓冲区和更激进的核心都很有用。

译者注：非绑定预取的概念参考以下文献 1.2.3 小节：

最简单，也可能是最古老的软件策略就是直接将预取值加载到处理器的寄存器中，就如其它显式加载操作一样。许多架构，尤其是现代的乱序架构，在 LOAD 指令发射后并不会暂停执行，而是在不得需要用到 LOAD 所加载的数据时，才暂停相关执行。

这种预取策略通常被称为 绑定预取（binding prefetch），因为在发出预取时，数据的后续使用价值将会被绑定。这种方法存在很多缺点：它会消耗宝贵的处理器寄存器空间；它会强制硬件进行预取，即便内存系统的负载很重；在预取地址错误时会导致语义上的困难（例如，预取的无效地址是否会导致内存保护故障（fault）？）；

不清楚如何将这种策略用于指令。

与此相反，大多预取技术都会将预取值直接放入 cache，或用于增强 cache 层次的补充缓冲区（supplemental buffer）中，并同时访问。在多核和多处理器系统中，这些 cache 和 buffer 还将参与 cache 一致性协议，因此预取值的存储位置可能在随后访问的期间发生改变；硬件需要保证被访问的值是最新的。这种预取策略被称作非绑定策略（nonbinding）。对于这些方案，预取将只是一种纯粹的性能方案，不影响程序语义。

再次感谢 @VN Vortex 的分享。

推测核心 (Speculative Cores)

考虑一个按程序顺序执行指令的核心，但也执行分支预测，其中后续指令（包括加载和存储）开始执行，但可能会因分支错误预测而被丢弃（即，使其效果无效）。这些丢弃的加载和存储可以看起来像非绑定预取，使这种推测是正确的，因为它对 SC 没有影响。分支预测之后的加载可以呈现给 L1 缓存，其中它要么未命中（导致非绑定 GetS 预取）要么命中，然后将值返回到寄存器。如果加载被丢弃，核心会丢弃寄存器更新，从而消除加载的任何功能影响——就好像它从未发生过一样。缓存不会撤消非绑定预取，因为这样做不是必需的，并且如果重新执行加载，预取块可以提高性能。对于存储，核心可能会提前发出非绑定 GetM 预取，但它不会将存储呈现给缓存，直到存储被保证提交。

Flashback to Quiz Question 2: The memory consistency model specifies the legal orderings of coherence transactions. *True or false?*

Answer: *False!*

动态调度核心 (Dynamically Scheduled Cores)

为了实现比静态调度的核心更高的性能，许多现代核心会动态地按照程序顺序以外的顺序调度指令执行。使用动态或乱序（程序）调度的单核处理器只需在程序内强制执行真实的数据依赖关系。然而，在多核处理器的上下文中，动态调度引入了一个新的问题：内存 consistency 的推测。考虑一个希望动态重新排序两个加载指令 L1 和 L2 的核心（例如，因为在计算 L1 地址之前计算 L2 地址）。许多核心将会在 L1 之前推测性地执行 L2，它们预测这种重新排序对其他核心不可见，这将违反 SC。

对 SC 的推测需要核心验证预测是否正确。Gharachorloo 等人 [8] 提出了两种执行此检查的技术。首先，在核心推测性地执行 L2 之后，但在它提交 L2 之前，核心可以检查推测性访问的块是否没有离开缓存。只要块保留在缓存中，它的值就不会在加载执行和提交之间发生变化。为了执行此检查，核心跟踪 L2 加载的地址，并将其与被驱逐的块和传入的 coherence 请求进行比较。传入的 GetM 表明另一个核心可以观察 L2 乱序，并且此 GetM 将暗示错误推测并丢弃推测执行。

第二种检查技术是在核心准备好提交加载时重放每个推测性加载（注2）[2, 17]。如果提交时加载的值不等于先前推测加载的值，则预测不正确。在示例中，如果重放的 L2 加载值与 L2 的原始加载值不同，则加载-加载重新排序导致执行明显不同，推测执行必须被丢弃。

原书作者注2：Roth [17] 展示了一种通过确定何时不需要重放来避免许多加载重放的方案。

动态调度核心中的非绑定预取 (Non-Binding Prefetching in Dynamically Scheduled Cores)

动态调度的核心可能会遇到程序顺序不正确的加载和存储缺失。例如，假设程序顺序是加载 A，存储 B，然后存储 C。核心可能会“乱序”启动非绑定预取，例如，首先是 GetM C，然后是并行的 GetS A 和 GetM B。SC 不受非绑定预取顺序的影响。SC 只要求核心的加载和存储（看起来）按程序顺序访问其一级缓存。Coherence 要求一级缓存块处于适当的状态以接收加载和存储。

重要的是，SC（或任何其他内存 consistency 模型）：

- 规定加载和存储（出现）应用于 coherent 内存的顺序，但
- 不规定 coherence 活动的顺序。

Flashback to Quiz Question 3: To perform an atomic read-modify-write instruction (e.g., test-and-set), a core must always communicate with the other cores. *True or false?*

Answer: *False!*

多线程 (Multithreading)

SC 实现可以容纳多线程——粗粒度、细粒度或同时的。每个多线程核心应该在逻辑上等同于多个（虚拟）核心，它们通过一个交换机共享每个一级缓存，其中缓存选择接下来要服务的虚拟核心。此外，每个缓存实际上可以同时服务多个非冲突请求，因为它可以假装它们是按某种顺序服务的。一个挑战是确保线程 T1 在存储对其他核心上的线程“可见”之前无法读取同一核心上另一个线程 T2 写入的值。因此，虽然线程 T1 可以在线程 T2 以内存顺序插入存储后立即读取该值（例如，通过将其写入状态 M 的缓存块），但它无法从处理器核心中的共享加载存储队列中读取该值。

Sidebar: Advanced SC Optimizations

此侧边栏描述了一些高级 SC 优化。

Post-retirement speculation. 单核处理器通常采用一种称为写入（存储）缓冲区的结构来隐藏存储未命中的延迟。存储从处理器流水线退出到写入缓冲区，从那里它从关键路径流入高速缓存/内存系统。这在单核上是安全的，只要加载检查写入缓冲区以查找相同地址的未完成存储。然而，在多核上，SC 排序规则排除了对写入缓冲区的幼稚使用。动态调度的核心可以隐藏一些但不是全

部的存储未命中延迟。为了隐藏更多的存储未命中延迟，已经有许多建议积极实施 SC，利用指令窗口之外的推测。关键思想是推测性地退休加载和存储过去的未决存储未命中，同时以细粒度 [9, 16] 或粗粒度块 [1, 3, 11, 19] 分别维护推测性退出指令的状态。

Non-speculative reordering. 甚至可以在执行 SC 时非推测地乱序地执行内存操作，只要重新排序对其他核心不可见 [7, 18]。在没有回滚恢复的情况下，如何确保重新排序对其他核心不可见？

一种方法（称为 coherence 延迟）涉及延迟 coherence 请求：具体而言，当较年轻的内存操作在未决的较旧的操作之后退出时，对较年轻的位置的 coherence 请求会延迟，直到较旧的内存操作退出。Coherence 延迟存在固有的死锁风险，因此需要谨慎的死锁避免机制。在表 3.3 所示的示例中，如果加载 L1 和 L2 都退出存储，并且对各自位置的 coherence 请求被延迟，这可能会阻止任何一个存储完成，从而导致死锁。

另一种方法（称为前驱序列化）要求旧的内存操作做得足够——通常在一个中心点进行序列化——以确保年轻的操作可以安全地完成它。冲突排序 [6] 允许加载和存储在挂起存储未命中后退出，只要挂起存储在目录中序列化并确定挂起存储的全局列表；只要年轻的内存操作不与这个列表冲突，就可以安全地退休。Gope 和 Lipasti [4] 提出了一种为顺序处理器量身定制的方法，其中每个加载或存储都按程序顺序从目录中获取一个互斥体，但可以乱序退休。


最后，可以利用编译器或内存管理单元的帮助来确定可以安全地重新排序的访问 [5]。例如，对线程私有或只读变量的两次访问可以安全地重新排序。

3.9 SC 的原子操作

要编写多线程代码，程序员需要能够同步线程，而这种同步通常涉及原子地执行成对的操作。此功能由原子执行 "read-modify-write" (RMW) 的指令提供，例如众所周知的 "test-and-set"、"fetch-and-increment" 和 "compare-and-swap"。这些原子指令对于正确同步至关重要，用于实现自旋锁和其他同步原语。对于自旋锁，程序员可以使用 RMW 原子地读取锁的值是否已解锁（例如，等于 0）并写入锁定的值（例如，等于 1）。为了使 RMW 是原子的，RMW 的读（加载）和写（存储）操作必须按照 SC 要求的操作的总顺序连续出现。

在微架构中实现原子指令在概念上很简单，但幼稚的设计会导致原子指令的性能不佳。实现原子指令的正确但简单的方法是让核心有效地锁定内存系统（即阻止其他核心发出内存访问）并执行其对内存的读取、修改和写入操作。这种实现虽然正确且直观，但牺牲了性能。

更积极的 RMW 实现利用了 SC 只需要出现所有请求的总顺序的洞察。因此，原子 RMW 可以通过首先让核心在其缓存中获取处于状态 M 的块来实现，如果该块尚未处于该状态。然后，核心只需要在其缓存中加载和存储该块 - 没有任何 coherence 消息或总线锁定 - 只要它等待为缓存块的任何到来的 coherence 请求提供服务，直到存储完成。这种等待不会有死锁的风险，因为存储是保证完成的。

 Uploading image.png...

更加优化的 RMW 实现可以在加载部分和存储部分执行之间留出更多时间，而不会违反原子性。考虑块在缓存中处于只读状态的情况。RMW 的加载部分可以推测性地立即执行，而缓存控制器发出 coherence 请求，将块的状态升级为读写。当块以读写状态获取后，RMW 的写入部分执行。只要核心能够保持原子性的假象，这种实现就是正确的。为了检查原子性的假象是否保持，核心必须检查加载的块是否从加载部分和存储部分之间的缓存中被逐出；这种推测支持与在 SC 中检测错误推测所需的支持相同（第 3.8 节）。

3.10 将它们放在一起：MIPS R10000

MIPS R10000 [21] 是一个具有崇高历史地位但清晰的商业示例，它是一款实现了 SC 的推测性微处理器，与具有 cache-coherent 的内存层次结构相协作。在这里，我们关注于与实现内存一致性相关的 MIPS R10000 的方面。

R10000 是一个四路超标量 RISC 处理器核心，具有分支预测和乱序执行功能。该芯片支持 L1 指令和 L1 数据的写回缓存，以及与（片外, off-chip）统一的 L2 缓存之间的私有接口。

芯片的主系统接口总线支持多达四个处理器的缓存一致性，如图 3.7 所示（改编自 Yeager [21] 的图 1）。要构建一个基于 R10000 的系统，包含更多处理器，比如 SGI Origin 2000（在第 8.8.1 节中详细讨论），架构师们实现了一个目录一致性协议，通过系统接口总线和专用的 Hub 芯片将 R10000 处理器连接起来。在这两种情况下，R10000 处理器核心看到的是一个 coherent memory 系统，部分位于芯片上，部分位于芯片外。

在执行过程中，R10000 核心按照程序顺序将（推测性的）加载和存储指令排入地址队列。加载指令从之前相同地址的最近一次存储指令中获得（推测性的）值，如果没有则从数据缓存中获取。加载和存储指令按程序顺序提交，然后删除它们在地地址队列中的条目。要提交存储指令，L1 缓存必须将块保持在 M 状态，并且存储指令的值必须与提交同时进行，以保证原子性。

重要的是，由于 coherence invalidation 或为了为另一个块腾出空间，缓存块的驱逐会导致地址队列中包含一个加载指令的地址被取消，并且后续的指令也会被取消，然后重新执行。因此，当一个加载指令最终提交时，加载的块在执行和提交之间一直保持在缓存中，因此它必须获得与在提交时执行相同的值。由于存储指令实际上在提交时写入缓存，R10000 在逻辑上将加载和存储指令按程序顺序呈现给 coherent memory 系统，从而实现了前面讨论过的 SC 内存模型。

 Uploading image.png...

3.11 关于 SC 的进一步阅读

以下我们列举了围绕 SC 的广泛文献中的一些论文。

Lamport [12] 定义了 SC。据我们所知，Meixner 和 Sorin [15] 是第一批证明了，在一个系统中，核心按程序顺序呈现加载和存储指令给一个具有缓存一致性的内存系统，就足以实现 SC 的人。尽管这个结果在某种程度上被人们直观地认为是正确的，但这是第一次得到明确的证明。

SC 可以与数据库串行化 [10] 进行比较。这两个概念在要求所有实体的操作似乎以序列顺序影响共享状态方面是相似的。这些概念之间的区别在于操作和共享状态的性质和预期。在 SC 中，每个操作都是对易失性状态（内存）的单个内存访问，假定不会失败。而在串行化中，每个操作是对数据库上的事务，可以读取和写入多个数据库实体，并且预期遵守 ACID 特性：Atomic - 全部或无操作，即使发生故障；Consistent - 使数据库保持一致；Isolated - 并发事务之间没有影响；Durable - 效果在崩溃和电力中断后仍然存在。

我们遵循了 Lamport 和 SPARC 的方法来定义所有内存访问的总顺序。虽然这可以在某些情况下增加直观性，但并非必要。请记住，如果两个访问来自不同的线程，访问相同的位置，并且至少一个是存储（或 RMW），则它们之间存在冲突。与总顺序不同，可以仅定义冲突访问的约束条件，而将非冲突访问保留为无序状态，这是由 Shasha 和 Snir [18] 提出的先驱方法。对于第 5 章中的放松模型，这种观点尤其有价值。

最后，谨慎的故事。我们之前在第 3.7 节中提到，检查是否可以观察到乱序执行的加载的一种方法是记住加载的临时读取的值 A，并在提交时，如果内存位置的值是相同的值 A，则提交加载。然而，Martin 等人 [14] 指出，这对于执行值预测的核心来说并不成立 [13]。在值预测中，当一个加载执行时，核心可以对其值进行预测。考虑一个核心，假设加载块 X 将产生值 A 的预测值，尽管实际值是 B。在核心对加载 X 进行预测并在提交时重新执行加载之间，另一个核心将块 X 的值更改为 A。然后核心在提交时重新执行加载，比较这两个值，发现它们相等，错误地确定了预测是正确的。如果系统以这种方式进行预测，可能会违反 SC。这种情况类似于所谓的 [ABA 问题](#)，Martin 等人证明在存在值预测的情况下，可以通过检查预测来避免一致性违规的可能性（例如，通过重新执行所有依赖于最初预测加载的加载）。这次讨论的重点不是深入讨论这个特定的边界情况或其解决方案的细节，而是说服您证明您的实现是正确的，而不是依赖于直觉。

3.12 参考文献

- [1] C. Blundell, M. M. K. Martin, and T. F. Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In Proc. of the 36th Annual International Symposium on Computer Architecture, June 2009. DOI: 10.1145/1555754.1555785. 32
- [2] 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. 31
- [3] 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. 32
- [4] D. Gope and M. H. Lipasti. Atomic SC for simple in-order processors. In 20th IEEE International Symposium on High Performance Computer Architecture, 2014. DOI: 10.1109/hpca.2014.6835950. 32
- [5] A. Singh, S. Narayanasamy, D. Marino, T.D Millstein, and M. Musuvathi. End-to-end sequential consistency. In 39th International Symposium on Computer Architecture, 2012. DOI: 10.1109/isca.2012.6237045. 32
- [6] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient sequential consistency via conflict ordering. In Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS, 2012. DOI: 10.1145/2150976.2151006. 32
- [7] K. Gharachorloo, S. V. Adve, A. Gupta, J. Hennessy, and M. D. Hill. Specifying system requirements for memory consistency models. Technical Report CSL-TR93-594, Stanford University, December 1993. 32
- [8] K.Gharachorloo,A.Gupta,andJ.Hennessy.Twotechniquestoenhancetheperformance of memory consistency models. In Proc. of the International Conference on Parallel Processing, vol. I, pp. 355–64, August 1991. 30
- [9] C. Guiady, 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. 32
- [10] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, 1993. 35
- [11] L. Hammond et al. Transactional memory coherence and consistency. In Proc. of the 31st Annual International Symposium on Computer Architecture, June 2004. DOI: 10.1109/isca.2004.1310767. 32
- [12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, C-28(9):690–91, September 1979. DOI: 10.1109/tc.1979.1675439. 22, 35
- [13] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 226–37, December 1996. DOI: 10.1109/micro.1996.566464. 35

- [14] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multi- processing. In Proc. of the 34th Annual IEEE/ACM International Symposium on Microarchi- tecture, pp. 328–37, December 2001. DOI: 10.1109/micro.2001.991130. 35
- [15] 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. 35
- [16] P. Ranganathan, V. S. Pai, and S. V. Adve. Using speculative retirement and larger in- struction windows to narrow the performance gap between memory consistency models. In Proc. of the 9th ACM Symposium on Parallel Algorithms and Architectures, pp. 199–210, June 1997. DOI: 10.1145/258492.258512. 32
- [17] A. Roth. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In Proc. of the 32nd Annual International Symposium on Computer Architecture, June 2005. DOI: 10.1109/isca.2005.48. 31
- [18] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. ACM Transactions on Programming Languages and Systems, 10(2):282–312, April 1988. DOI: 10.1145/42190.42277. 32, 35
- [19] T. F. Wenisch, A. Ailamaki, A. Moshovos, and B. Falsafi. Mechanisms for store-wait-free multiprocessors. In Proc. of the 34th Annual International Symposium on Computer Architec- ture, June 2007. DOI: 10.1145/1250662.1250696. 32
- [20] D.L.WeaverandT.Germond,Eds.SPARCArchitectureManual(Version9).PTRPrentice Hall, 1994. 23
- [21] K. C. Yeager. The MIPS R10000 superscalar microprocessor. IEEE Micro, 16(2):28–40, April 1996. DOI: 10.1109/40.491460. 34