TSO 和 x86 内存模型

一种广泛实现的内存一致性模型是总存储顺序 (total store order, TSO)。 TSO 最早由 SPARC 引入，更重要的是，它似乎与广泛使用的 x86 架构的内存一致性模型相匹配。 RISC-V 还支持 TSO 扩展 RVTSO，部分是为了帮助移植最初为 x86 或 SPARC 架构编写的代码。 本章使用类似于前一章顺序一致性的模式来介绍这个重要的一致性模型。我们首先通过指出 SC 的局限性部分激励 TSO/x86（第 4.1 节）。 然后，我们在更正式的描述（第 4.3 节）之前以直观的级别介绍 TSO/x86（第 4.2 节），解释系统如何实现 TSO/x86，包括原子指令和用于强制指令之间排序的指令（第 4.4 节）。 最后，我们讨论了其他资源，以了解有关 TSO/x86 的更多信息（第 4.5 节）并比较 TSO/x86 和 SC（第 4.6 节）。

4.1 TSO/X86 的动机

处理器核心长期以来一直使用 write buffer (或 store buffer，写入缓冲区) 来保存已提交（retired）的存储，直到内存系统的其余部分可以处理这些存储。当存储提交时，存储进入写入缓冲区，当要写I'm 入的块处于读写 coherence 状态的高速缓存中时，存储退出写入缓冲区。值得注意的是，一个存储可以在缓存获得写入块的读写 coherence 权限之前进入写入缓冲区；因此，写缓冲区隐藏了为存储未命中提供服务的延迟。因为存储很常见，所以能够避免在大多数存储中停滞不前是一个重要的好处。此外，不停止核心似乎是明智的，因为核心不需要任何东西，因为存储试图更新内存而不是核心状态。

对于单核处理器，可以通过确保对地址 A 的加载将最近存储的值返回给 A，即使对 A 的一个或多个存储在写缓冲区中，也可以使写缓冲区在架构上不可见。这通常通过将最近存储到 A 的值绕过到从 A 加载来完成，其中"最近"由程序顺序确定，或者如果到 A 的存储在写缓冲区中，则停止加载 A 。

在构建多核处理器时，使用多个核心似乎很自然，每个核心都有自己的旁路写入缓冲区（bypass write buffer），并假设写入缓冲区在架构上仍然是不可见的。

Table 4.1: Can both r1 and r2 be set to 0?

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

这个假设是错误的。考虑表 4.1 中的示例代码（与上一章的表 3.3 相同）。假设多核处理器具有顺序核心，其中每个核心都有一个单入口写入缓冲区并按以下顺序执行代码。

1. 核心 C1 执行存储 S1，但在其写缓冲区中缓冲新存储的 NEW 值。
2. 同样，核心 C2 执行存储 S2 并将新存储的 NEW 值保存在其写缓冲区中。
3. 接下来，两个核心执行各自的加载 L1 和 L2，并获得旧值 0。
4. 最后，两个核心的写缓冲区使用新存储的值 NEW 更新内存。

最终结果是 (r1, r2) = (0, 0)。正如我们在上一章中看到的，这是 SC 禁止的执行结果。没有写缓冲区，硬件就是 SC，但有了写缓冲区，它就不是了，这使得写缓冲区在多核处理器中在架构上是可见的。

对可见的写入缓冲区的一种响应是关闭它们，但由于潜在的性能影响，供应商一直不愿这样做。另一种选择是使用激进的、推测性的 SC 实现，使写入缓冲区再次不可见，但这样做会增加复杂性，并且会浪费电力来检测违规和处理错误推测。

SPARC 和后来的 x86 选择的选项是放弃 SC，转而支持内存一致性模型，允许在每个核心上直接使用先进先出 (FIFO) 写入缓冲区。这个被称为 TSO 的新模型允许结果"(r1, r2) = (0, 0)"。这个模型让一些人感到惊讶，但事实证明，对于大多数编程习惯来说，它的行为就像 SC 一样，并且在所有情况下都得到了明确的定义。

4.2 TSO/X86 的基本思想

随着执行的进行，SC 要求每个核心为连续操作的所有四种组合保留其加载和存储的程序顺序：

- Load -> Load
- Load -> Store
- Store -> Store
- Store -> Load /* Included for SC but omitted for TSO */

TSO 包括前三个约束，但不包括第四个。对于大多数程序来说，这种省略并不重要。表 4.2 重复了上一章表 3.1 的示例程序。在这种情况下，TSO 允许与 SC 相同的执行，因为 TSO 保留了核心 C1 的两次存储和核心 C2 的两次（或更多）加载的顺序。图 4.1（与上一章的图 3.2 相同）说明了该程序的执行。

Table 4.2: 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 */

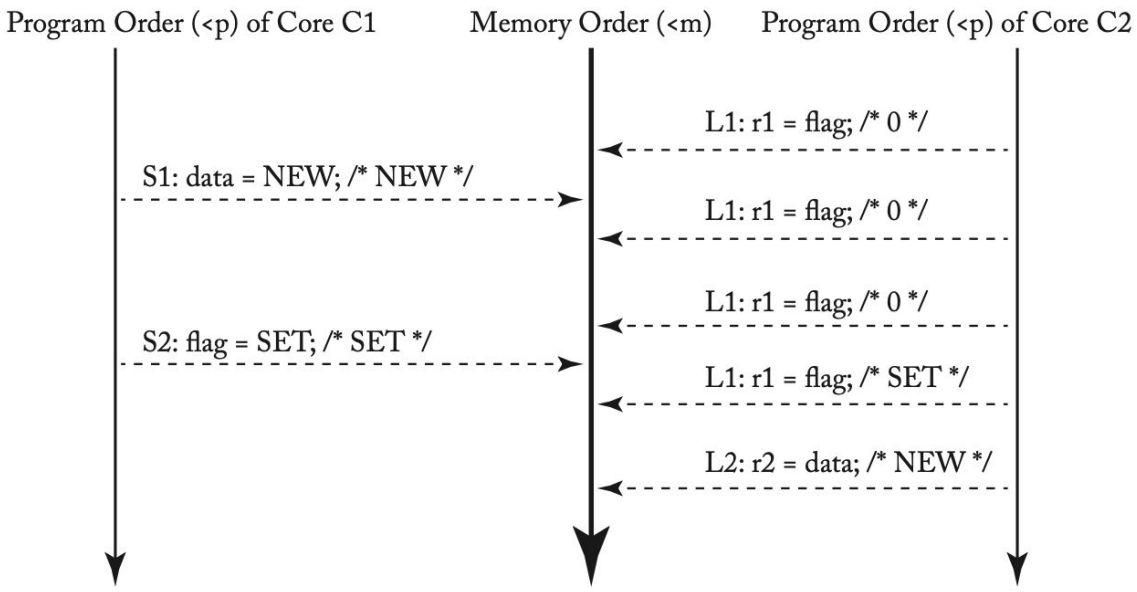


Figure 4.1: A TSO execution of Table 4.2's program.

更一般地说，对于以下常见的编程习惯，TSO 的行为与 SC 相同：

- C1 加载和存储到内存位置 D1, ..., Dn（通常是数据），
- C1 存储到 F（通常是一个同步标志），表示上述工作已经完成，
- C2 从 F 加载以观察上述工作是否完成（有时先自旋并经常使用 RMW 指令），并且
- C2 加载和存储到部分或全部内存位置 D1, ..., Dn。

然而，TSO 允许一些非 SC 执行。在 TSO 下，表 4.1 中的程序（重复上一章的表 3.3）允许图 4.2 中描述的所有四种结果。在 SC 下，只有前三个是合法结果（如上一章图 3.3 所示）。图 4.2d 中的执行说明了一个符合 TSO 但由于不遵守第四个（即，store -> load）约束而违反 SC 的执行。省略第四个约束允许每个核心使用一个写缓冲区。请注意，第三个约束意味着写入缓冲区必须是 FIFO（而不是，例如，合并）以保持 store-store 顺序。

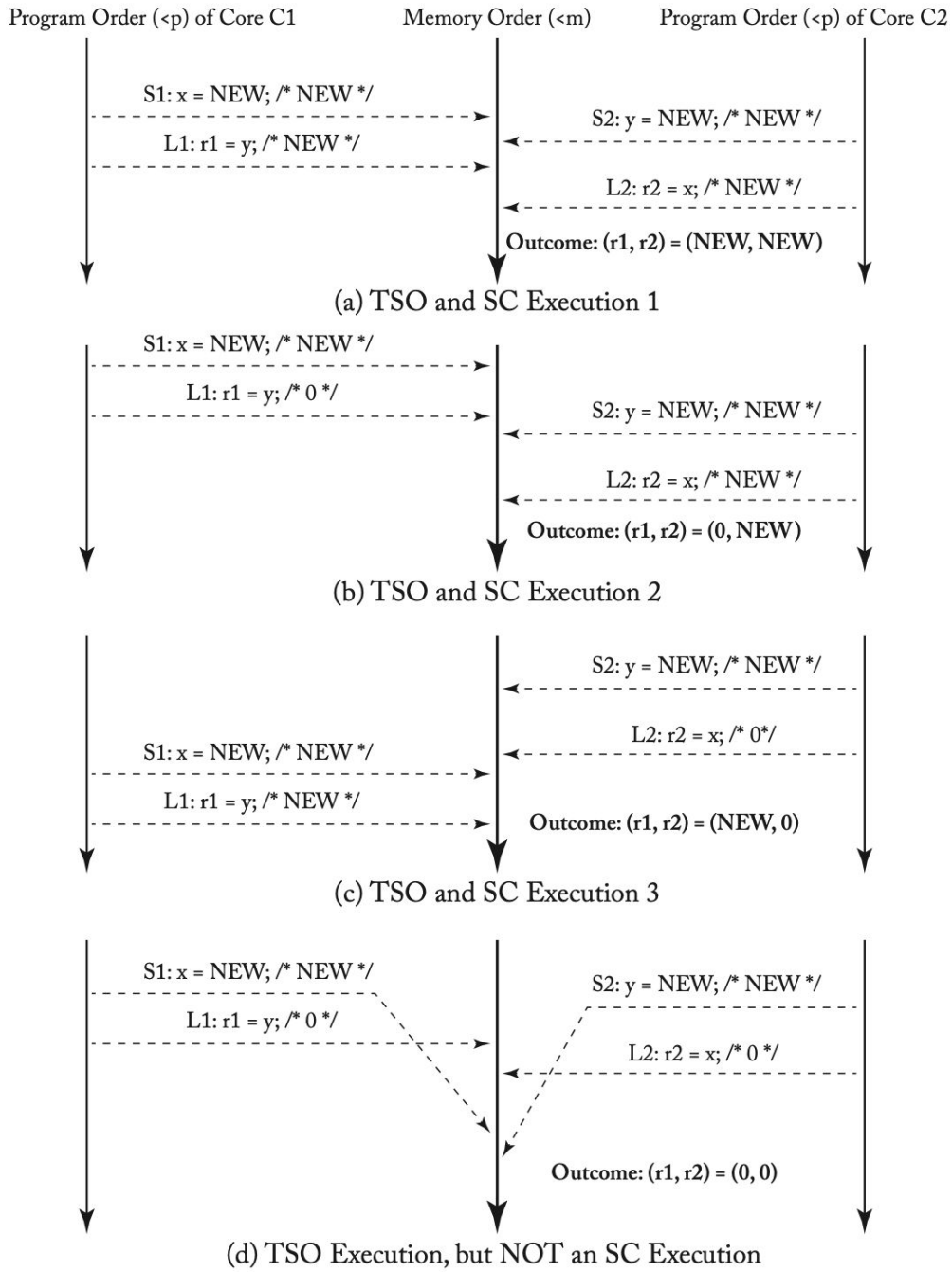


Figure 4.2: Four alternative TSO executions of Table 4.1's program.

程序员（或编译器）可以通过在核心 C1 上的 S1 和 L1 之间以及核心 C2 上的 S2 和 L2 之间插入 FENCE 指令来阻止图 4.2d 中的执行。在核心 Ci 上执行 FENCE 可确保 Ci 上 FENCE 之前的内存操作（按程序顺序）按照内存顺序放置在 Ci 上 FENCE 之后的内存操作之前。使用 TSO 的程序员很少使用 FENCE（又名内存屏障），因为 TSO 对大多数程序"做正确的事"。尽管如此，FENCE 在下一章讨论的宽松模型中发挥着重要作用。

TSO 确实允许一些非直观的执行结果。表 4.3 说明了表 4.1 中程序的修改版本，其中核心 C1 和 C2 分别制作 x 和 y 的本地副本。许多程序员可能假设如果 r2 和 r4 都等于 0，那么 r1 和 r3 也应该为 0，因为存储 S1 和 S2 必须在加载 L2 和 L4 之后插入到内存顺序中。然而，图 4.3 展示了一个执行，显示 r1 和 r3 绕过每个核心写入缓冲区中的值 NEW。事实上，为了保持单线程顺序语义，每个核心都必须按照程序顺序看到自己存储的效果，即使存储还没有被其他核心观察到。因此，在所有 TSO 执行下，本地副本 r1 和 r3 将始终设置为 NEW 值。

> 译者点评：这个例子妙啊！

Table 4.3: Can r1 or r3 be set to 0?

Core C1	Core C2	Comments
S1: x = NEW; L1: r1 = x; L2: r2 = y;	S2: y = NEW; L3: r3 = y; L4: r4 = x;	/* Initially, x = 0 & y = 0 */ /* Assume r2 = 0 & r4 =0 */

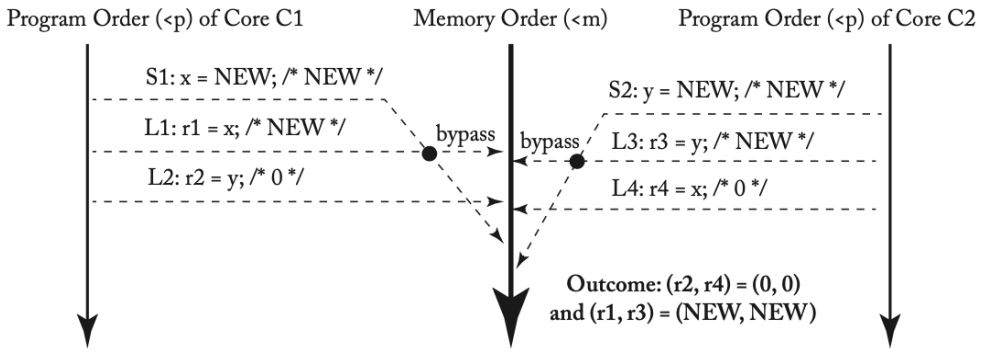


Figure 4.3: A TSO execution of Table 4.3's program (with "bypassing").

4.3 一点 TSO/X86 正式化方法

在本节中，我们更精确地定义了 TSO，其定义仅对第 3.5 节的 SC 定义进行了三处更改。

TSO execution 需要以下内容。

> 译者注：请与3.5节比较，体会两者的差异。

1. 所有核心都将它们的加载和存储插入到内存顺序 `<m` 中，考虑到它们的程序顺序，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 */
- 删除：If `S(a) <p L(b)` => `S(a) <m L(b)` /* Store -> Load */ => 修改1：Enable FIFO Write Buffer

2. 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：

- 删除：Value of `L(a)` = Value of `MAX <m {S(a) | S(a) <m L(a)}` => 修改2：Need Bypassing
- Value of `L(a)` = Value of `MAX <m {S(a) | S(a) <m L(a) or S(a) <p L(a)}`

最后一个令人费解的等式表明，加载的值是最后存储到同一地址的值，该地址要么是（a）按内存顺序在它之前，要么（b）按程序顺序在它之前（但可能在它之后）内存顺序，选项（b）优先（即，写缓冲区绕过覆盖内存系统的其余部分）。

3. 第 (1) 部分必须扩充以定义 FENCE：/* 修改3: FENCEs Order Everything */

- 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 */

因为 TSO 已经需要除 Store -> Load 之外的所有顺序，也可以将 TSO FENCE 定义为仅排序：

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

我们选择让 TSO FENCE 对所有内容进行冗余排序，因为这样做不会造成伤害并使它们像我们在下一章中为更轻松的模型定义的 FENCE 一样。

我们在表 4.4 中总结了 TSO 的排序规则。该表与 SC 的类似表（表 3.4）有两个重要区别。首先，如果操作 #1 是存储，操作 #2 是加载，则该交叉点处的条目是 “B” 而不是 “X” ；如果这些操作指向相同的地址，则即使这些操作进入内存顺序超出程序顺序，加载也必须获取刚刚存储的值。其次，该表包括在 SC 中不需要的 FENCE ； SC 系统的行为就像在每次操作之前和之后已经存在 FENCE 一样。

> 译者注：请与3.5节比较，体会两者的差异。

Table 4.4: TSO ordering rules. An “X” denotes an enforced ordering. A “B” denotes that bypassing is required if the operations are to the same address. Entries that are different from the SC ordering rules are shaded and shown in bold.

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

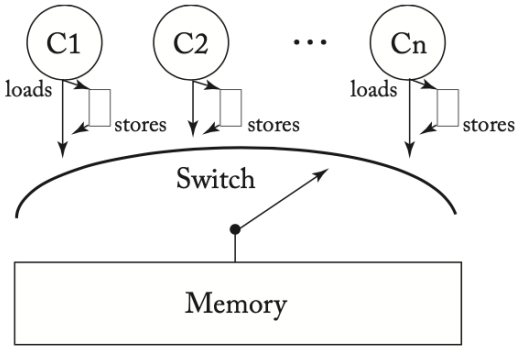
人们普遍认为 x86 内存模型等效于 TSO（用于普通可缓存内存和普通指令），但据我们所知，AMD 和 Intel 都没有对此做出保证，也没有发布正式的 x86 内存模型规范。AMD 和 Intel 通过示例和说明公开定义 x86 内存模型，这一过程在 Sewell 等人的第 2 节中得到了很好的总结 [15]。所有的示例都符合 TSO，所有的说明也都与 TSO 一致。只有当 x86 内存模型的公开、正式的描述可用时，才能证明这种等价性。如果有反例显示出 x86 执行不符合 TSO 模型，或者 TSO 执行不符合 x86 模型，或者两者皆然，这种等价关系就可能被否定。

Sewell 等人支持 x86 等效于 TSO [15]，在 CACM 中进行了总结，并在其他地方提供了更多详细信息 [9, 14]。特别是，作者提出了 x86-TSO 模型。该模型有两种形式，作者证明是等效的。第一种形式提供了一个类似于下一节的图 4.4a 的抽象机器，并添加了一个用于建模 x86 LOCK'd 指令的全局锁。第二种形式是有标签的过渡系统。第一种形式使从业者可以使用模型，而后者简化了形式证明。一方面，x86-TSO 似乎与 x86 规范中的非正式规则和试金石一致。另一方面，在几个 AMD 和 Intel 平台上的实证测试没有发现任何违反 x86-TSO 模型的行为（但这并不能证明它们不能）。总之，像 Sewell 等人一样，我们敦促 x86 硬件和软件的创建者采用明确且可访问的 x86-TSO 模型。

4.4 实现 TSO/X86

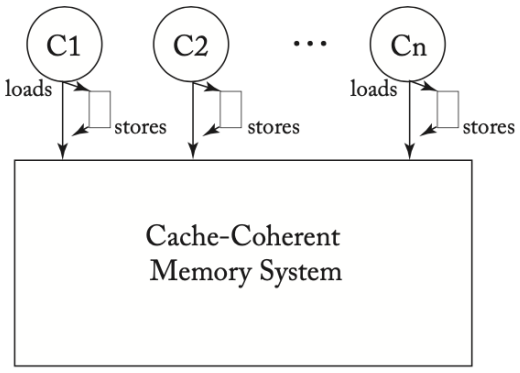
TSO/x86 的实现问题与 SC 类似，只是增加了每个核心中的 FIFO writer buffer。图 4.4a 更新了图 3.4 的 Switch 以适应 TSO 并按如下方式操作。

- Load 和 store 将每个核心留在该核心的程序顺序 `<p` 中。
- Load 要么 bypass write buffer 中的值，要么像以前一样等待切换。
- 如果 buffer 未满，则 store 进入 FIFO write buffer 的尾部；如果 buffer 已满，则 store stall 核心。
- 当 switch 选择核心 Ci 时，它要么执行下一次 load，要么执行 write buffer 头部的 store。



This implementation is the same as for Figure 3.4, except that each core Ci has a FIFO write buffer that buffers stores until they go to memory.

(a) A TSO implementation using a switch



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

(b) A TSO implementation using cache coherence

Figure 4.4: Two TSO implementations.

在第 3.7 节中，我们展示了对于 SC，交换机可以被缓存 coherent 内存系统替换，然后认为核心可以是推测性的和/或多线程的，并且非绑定预取可以由核心、缓存或软件启动。

如图 4.4b 所示，相同的论点适用于 TSO，在每个核心和缓存 coherent 内存系统之间插入一个 FIFO 写入缓冲区。因此，除了写缓冲区之外，所有前面的 SC 实现讨论都适用于 TSO，并提供了一种构建 TSO 实现的方法。此外，大多数当前的 TSO 实现似乎只使用上述方法：采用 SC 实现并插入写缓冲区。

关于写缓冲区，关于推测核心如何实现它们的文献和产品空间超出了本章的范围。例如，微架构可以物理地结合存储队列（未提交的存储）和写入缓冲区（已提交的存储），和/或物理分离的加载和存储队列。

最后，多线程为 TSO 引入了一个微妙的写入缓冲区问题。 TSO 写缓冲区在逻辑上对每个线程上下文（虚拟内核）都是私有的。因此，在多线程内核上，一个线程上下文不应该绕过另一个线程上下文的写缓冲区。这种逻辑分离可以通过每个线程上下文的写入缓冲区来实现，或者更常见的是，通过使用共享的写入缓冲区来实现，该缓冲区的条目由线程上下文标识符标记，只有在标记匹配时才允许绕过。

Flashback to Quiz Question 4: In a TSO system with multithreaded cores, threads may bypass values out of the write buffer, regardless of which thread wrote the value. *True or false?* **Answer:** *False!* A thread may bypass values that it has written, but other threads may not see the value until the store is inserted into the memory order.

4.4.1 实现原子指令

TSO 中原子 RMW 指令的实现问题类似于 SC 中原子 RMW 指令的实现问题。关键区别在于 TSO 允许 load pass（即，在之前排序）已写入 write buffer 的较早的 store。对 RMW 的影响是“写入”（即存储）可能会被写入 write buffer。

为了理解 TSO 中原子 RMW 的实现，我们将 RMW 视为一个 load 后面紧跟着一个 store。

由于 TSO 的排序规则，RMW 的加载部分不能传递早期的加载。乍一看，RMW 的加载部分可能会传递写入缓冲区中较早的存储，但这是不合法的。如果 RMW 的加载部分通过较早的存储，则 RMW 的存储部分也必须通过较早的存储，因为 RMW 是原子对。但是因为在 TSO 中 stores 是不允许互相传递的，所以 RMW 的 load 部分也不能传递一个更早的 store。

这些对 RMW 的排序限制会影响实施。因为 RMW 的加载部分在更早的存储被排序（即退出写缓冲区）之前无法执行，原子 RMW 在它可以执行 RMW 的加载部分之前有效地耗尽了写缓冲区。此外，为了确保存储部分可以在加载部分之后立即 ordered，加载部分需要读写 coherence 权限，而不仅仅是足以满足正常加载的读取权限。最后，为了保证 RMW 的原子性，缓存控制器可能不会放弃对加载和存储之间块的 coherence 权限。

更优化的 RMW 实现是可能的。例如，写缓冲区不需要排空，只要 (a) 已经在写缓冲区中的每个条目在缓存中都具有读写权限，并在 RMW 提交之前保持缓存中的读写权限，并且 (b) 核心执行 MIPS R10000 式的加载推测检查（第 3.8 节）。从逻辑上讲，所有早期的存储和加载都将作为一个单元（有时称为“块”）在 RMW 之前立即提交。

4.4.2 实现 Fence

支持 TSO 的系统不提供存储和后续（按程序顺序）加载之间的排序，尽管它们确实需要加载来获取较早存储的值。在程序员希望这些指令被排序的情况下，程序员必须通过在存储和后续加载之间放置一条 FENCE 指令来明确指定该排序。 FENCE 的语义规定，在程序顺序中 FENCE 之前的所有指令必须在程序顺序中 FENCE 之后的任何指令之前排序。对于支持 TSO 的系统，FENCE 因此禁止负载绕过较早的存储。在表 4.5 中，我们重新审视了表 4.1 中的示例，但我们添加了两个之前没有出现的 FENCE 指令。如果没有这些 FENCE，两个负载（L1 和 L2）可以绕过两个存储（S1 和 S2），从而导致 r1 和 r2 都设置为零的执行。添加的 FENCE 禁止重新排序，因此禁止执行。

 Uploading image.png...

因为 TSO 只允许一种类型的重新排序，所以 FENCE 相当少见，而且 FENCE 指令的实现也不是很关键。一个简单的实现（例如在执行 FENCE 时排空写入缓冲区，并且在较早的 FENCE 提交之前不允许执行后续加载）可能会提供可接受的性能。

然而，对于允许更多重新排序的一致性模型（在下一章中讨论），FENCE 指令更频繁，它们的实现会对性能产生重大影响。

> **Sidebar: Non-speculative TSO Optimizations**
>
> 此侧边栏描述了一些高级的非推测性 TSO 优化。
>
> **Non-speculative TSO reordering**。有论文表明，加载 [11, 12] 和存储 [13] 都可以非推测性地重新排序，同时仍然使用 coherence 延迟来强制执行 TSO。如前所述，关键挑战是确保延迟不会导致导致死锁的循环依赖，上述所有论文都解决了这个问题。
>
> **RMW without write buffer drain**。在 4.4.1 节中，我们展示了如何使用推测将写缓冲区从关键路径中移出。拉贾拉姆等人 [10] 表明，如果重新定义 RMW 的原子性语义，则可以非推测地实现相同的效果。回想一下，为了 RMW 是原子的，我们之前要求 RMW 的读取和写入操作必须在 TSO 全局内存顺序中连续出现。考虑以下松弛，其中只要写入与 RMW 的地址相同的地址不会出现在全局内存顺序中的读取和写入之间，RMW 就被认为是原子的。请注意，这种宽松的定义与 RMW 的直观定义相匹配，足以让它们在同步情况下使用。同时，它允许 RMW 实现，其中 RMW 的加载部分可以绕过写入缓冲区中的早期存储，而无需 MIPS R10000 样式的加载推测检查。但是，确保 RMW 原子性需要 coherence 延迟，直到写入缓冲区耗尽，这会带来死锁风险。拉贾拉姆等人 [10] 展示了如何避免死锁。

> **Reordering past a FENCE**。有几篇论文提出了优化的 FENCE 实现 [3, 4, 6, 8]，使 FENCE 之后的内存操作能够在 FENCE 之前的内存操作之前非推测性地退出。这些技术要么使用 coherence 延迟，要么使用前置序列化，或者两者的组合。

4.5 关于 TSO 的进一步阅读

Collier [2] 通过一个模型描述了替代的内存一致性模型，包括 IBM System/370 的模型，其中每个核心都有一个完整的内存副本，它的加载从本地副本读取，它根据一些定义模型的限制写入更新所有副本。如果使用此模型定义 TSO，则每个存储将立即写入其自己的核心的内存副本，然后可能稍后将所有其他内存一起更新。

Goodman [7] 公开讨论了处理器一致性 (processor consistency, PC) 的概念，其中一个核心的存储按顺序到达其他核心，但不一定在同一"时间"到达其他核心。Gharachorloo 等人 [5] 更精确地定义了 PC。TSO 和 x86-TSO 是 PC 的特殊情况，其中每个核心都会立即看到自己的存储区，而当任何其他核心看到一个存储区时，所有其他核心都会看到它。这个属性在下一章（5.5 节）中称为写原子性 (write atomicity)。

据我们所知，TSO 最初是由 Sindhu 等人 [16] 正式定义的。正如所讨论的，在第 4.3 节中，Sewell 等人 [9, 14, 15] 提出并形式化了 x86-TSO 模型，该模型看起来与 AMD 和 Intel x86 文档和当前实现一致。

4.6 比较 SC 和 TSO

现在我们已经看到了两个内存一致性模型，我们可以进行比较。在 SC、TSO 等之间有什么关系呢？

- 执行：SC 的执行是 TSO 执行的一个真子集；所有的 SC 执行都是 TSO 执行，但有些 TSO 执行是 SC 执行，有些则不是。请参见图4.5a中的 Venn 图。
- 实现：实现遵循相同的规则：SC 实现是 TSO 实现的一个真子集。请参见图4.5b，它与图4.5a相同。

更一般地说，如果内存一致性模型 Y 比模型 X 更宽松（弱），那么所有的 X 执行也都是 Y 执行，但反之不成立。如果 Y 比 X 更宽松，那么所有的 X 实现也都是 Y 实现。还有可能两个内存一致性模型是不可比较的，因为它们都允许对方所不允许的执行。

如图4.5所示，TSO 比 SC 更宽松，但比不可比较的模型 MC1 和 MC2 更严格。在下一章中，我们将看到 MC1 和 MC2 的候选模型，包括 IBM Power memory consistency model 的案例研究。

何为良好的内存一致性模型？
一个良好的内存一致性模型应当具备 Sarita Adve 提出的三个P，以及我们的第四个P：

- *可编程性*：一个良好的模型应该使编写多线程程序变得（相对）容易。这个模型应该对大多数用户来说是直观的，即使是那些没有阅读详细细节的用户也是如此。它应该是精确的，以便专家可以拓展所允许的范围。
- *性能*：一个良好的模型应该在合理的功耗、成本等条件下促进高性能的实现。它应该给予实现者广泛的选项余地。
- *可移植性*：一个良好的模型应该被广泛采用，或者至少提供向后兼容性或在不同模型之间进行转换的能力。
- *精确性*：一个良好的模型应该被精确定义，通常通过数学方式。自然语言过于模糊不清，不能让专家拓展所允许的范围。

SC和TSO模型的优劣如何？
运用这四个P来分析：

- *可编程性*：SC 模型是最直观的。TSO 模型也接近，因为对于常见的编程惯例，它的行为类似于 SC 模型。然而，隐含的非 SC 执行可能会对程序员和工具作者产生影响。
- *性能*：对于简单的核心，TSO 模型可以比 SC 模型提供更好的性能，但通过推测可以减小二者之间的差距。
- *可移植性*：SC 模型被广泛理解，而 TSO 模型得到了广泛采用。
- *精确性*：SC 模型和 TSO 模型都有明确的形式定义。

总的来说，SC 模型和 TSO 模型非常接近，尤其是与下一章讨论的更复杂和更宽松的 memory consistency 模型相比较。

4.7 参考文献

[1] S. V. Adve. Designing memory consistency models for shared-memory multiproces- sors. Ph.D. thesis, Computer Sciences Department, University of Wisconsin–Madison, November 1993. 51

[2] W. W. Collier. Reasoning About Parallel Architectures. Prentice-Hall, Inc., 1990. 50

[3] Y. Duan, A. Muzahid, and J. Torrellas. WeeFence: Toward making fences free in TSO. In The 40th Annual International Symposium on Computer Architecture, 2013. DOI: 10.1145/2485922.2485941. 49

[4] Y. Duan, N. Honarmand, and J. Torrellas. Asymmetric memory fences: Optimizing both performance and implementability. In Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems, 2015. DOI: 10.1145/2694344.2694388. 49

[5] 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. 50

[6] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and design of AlphaServer GS320. In Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000. DOI: 10.1145/378993.378997. 49

[7] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Computer Sciences Department, University of Wisconsin–Madison, February 1991. 50

[8] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. In 19th International Conference on Parallel Architectures and Compilation Techniques, 2010. DOI: 10.1145/1854273.1854312. 49

[9] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In Proc. of the Conference on Theorem Proving in Higher Order Logics, 2009. DOI: 10.1007/978-3-642- 03359-9_27. 46, 50

[10] B. Rajaram, V. Nagarajan, S. Sarkar, and M. Elver. Fast RMWs for TSO: Semantics and implementation. In Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013. DOI: 10.1145/2491956.2462196. 49

[11] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras. Non-speculative load-load reordering in TSO. In Proc. of the 44th Annual International Symposium on Computer Architecture, 2017. DOI: 10.1145/3079856.3080220. 49

[12] A. Ros and S. Kaxiras. The superfluous load queue. In 51st Annual IEEE/ACM Interna- tional Symposium on Microarchitecture, 2018. DOI: 10.1109/micro.2018.00017. 49

[13] A. Ros and S. Kaxiras. Non-speculative store coalescing in total store order. In 45th ACM/IEEE Annual International Symposium on Computer Architecture, 2018. DOI: 10.1109/isca.2018.00028. 49

[14] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 379– 391, 2009. DOI: 10.1145/1480881.1480929. 46, 50

[15] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Communications of the ACM, July 2010. DOI: 10.1145/1785414.1785443. 45, 46, 50

[16] P. Sindhu, J.-M. Frailong, and M. Ceklov. Formal specification of memory models. Technical Report CSL-91–11, Xerox Palo Alto Research Center, December 1991. DOI: 10.1007/978-1-4615-3604-8_2. 50