Shared Memory Consistency Models: A Tutorial

摘要

支持共享内存抽象的并行系统正在出现在许多计算领域被广泛接受。写作正确高效这类系统的程序需要正式的内存语义规范，称为内存一致性模型。最直观的模型——顺序模型一致性——通常极大地限制了许多性能优化的使用由单处理器硬件和编译器设计者使用，从而减少使用多处理器的好处。为了缓解这一问题，目前有很多多处理器支持更宽松的一致性模型。不幸的是,由不同系统支持的模型在细微而重要的方面存在差异的方式。此外，经常精确地定义每个模型的语义导致典型用户难以理解的复杂规范计算机系统的建造者。

本教程的目的是描述与内存一致性相关的问题模型的方式对大多数计算机专业人员来说是可以理解的。我们主要研究基于硬件的共享内存的一致性模型系统。这些模型中的许多最初都是强调指定的在系统优化方面，他们允许。我们仍然强调以系统为中心，但是使用统一而简单的术语来描述不同的模型。我们也简要讨论另一个以程序员为中心的视图，该视图描述了其中的模型程序行为的术语而不是特定的系统优化。

1.介绍

共享内存或单个地址空间抽象提供了一些优于消息传递(或私有存储)抽象的优点，通过提供从单处理器到更自然的转换，以及通过简化诸如数据分区和动态负载分配等困难的编程任务。因此，支持共享内存的并行系统在技术和商业计算中得到了广泛的接受。

为了编写正确和高效的共享内存程序，程序员需要精确了解内存在从多个处理器读写操作时的行为。例如，考虑图 1 中的共享内存程序片段，它表示来自 SPLASH 应用程序套件的 LocusRoute 程序片段。图中显示处理器 P1 反复分配任务记录，更新记录中的数据字段，并将记录插入到任务队列中。当不再剩下任务时，处理器 P1 更新一个指针 Head，指向任务队列中的第一个记录。与此同时，其他处理器等待 Head 具有非空值，在一个关键部分中对 head 指针指向的任务进行去队列处理，最后访问去队列记录中的数据字段。程序员希望从内存系统中得到什么来确保这个程序片段的正确执行?一个重要的需求是，从数据字段中

读取的数据值应该与 P1 在该记录中写入的值相同。然而，在许多商业共享内存系统中，处理器可以观察数据字段的旧值(即，在 P1 写字段之前的值)，导致与程序员期望不同的行为。

```
    Initially all pointers = null, all integers = 0.

P1                                  P2, P3, ..., Pn

while (there are more tasks) {      while (MyTask == null) {
  Task = GetFromFreeList();           Begin Critical Section
  Task → Data = ...;                  if (Head != null) {
  insert Task in task queue             MyTask = Head;
}                                       Head = Head → Next;
Head = head of task queue;            }
                                      End Critical Section
                                    }
                                    ...  = MyTask → Data;
```

Figure 1: What value can a read return?

2.内存一致性模型——谁应该关心？

作为程序员和系统之间的接口，内存一致性模型在共享内存系统中影响广泛。模型影响可编程性，因为程序员必须使用它来推断他们程序的正确性。模型影响系统的性能，因为它决定了硬件和系统软件可能利用的优化类型。最后，由于对单个模型缺乏共识，在跨支持不同模型的系统移动软件时，可移植性会受到影响。在程序员和系统之间定义接口的每个级别都需要一个内存一致性模型规范。在 themachine code 接口中，themachine model specification 会影响到 themachine hardware 的设计人员和编写或编写机器代码的程序员。在高级语言接口上，规范影响使用高级语言的程序员和将高级语言代码转换为机器代码的软件和执行此代码的硬件的设计人员。因此，可编程性、性能和可移植性问题可能存在于几个不同的级别。总之，内存模型从程序员的角度影响并行程序的编写，从系统设计人员的角度影响并行系统设计的所有方面(包括处理器、内存系统、互连网络、编译器和编程语言)。

3.单处理器系统中的内存语义

大多数高级单处理器语言都为内存提供了简单的顺序语义。因此程序员就认为所有的内存操作都是按照顺序执行了。但是实际上并不是这样，只不过给程序员这种假象。这种假象的实现主要是通过例如编译器调度，流水线，多发射等等实现的。

4.理解顺序一致性

顺序一致性有两个方面:

(1)维护单个处理器操作之间的程序顺序;

(2)维护所有处理器操作之间的单个顺序顺序。后一个方面使它看起来像一个内存操作相对于其他内存操作是原子性的或即时执行的。

图 3 展示了序列一致性为程序员提供了一个简单的系统视图。

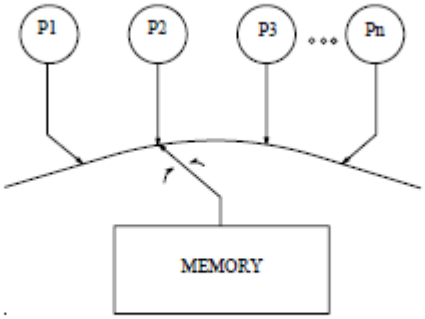


Figure 3: Programmer's view of sequential consistency.

图 4 提供了两个示例来说明顺序一致性的语义。图 4(a)说明了程序顺序在单个处理器操作之间的重要性。图 4(b)说明了内存操作的原子执行的重要性。

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(a)

Initially A = B = 0

P1	P2	P3
A = 1		
	if (A ==1)	
	B = 1	
		if (B==1)
		register1 = A

(b)

Figure 4: Examples for sequential consistency.

5.顺序一致性的实现

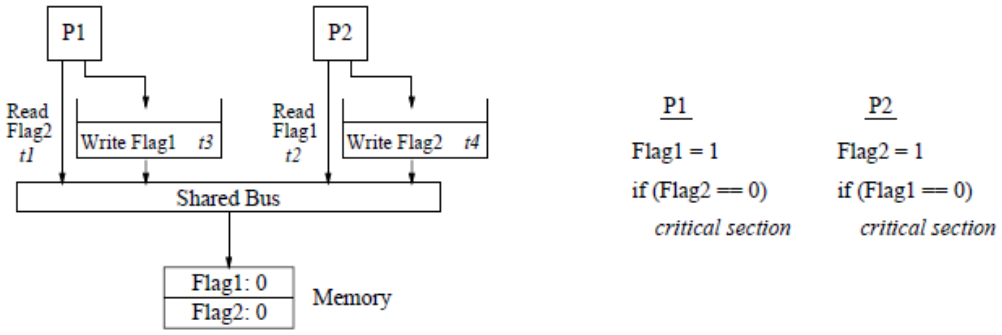
本节描述如何在实际系统中实现图 3 所示的顺序一致性的直观抽象。

5.1 没有缓存的体系架构

选择了三种典型的硬件优化作为例子来说明在没有数据缓存的情况下实现顺序一致性时出现的典型交互。

5.1.1 具有旁路功能的写缓冲区

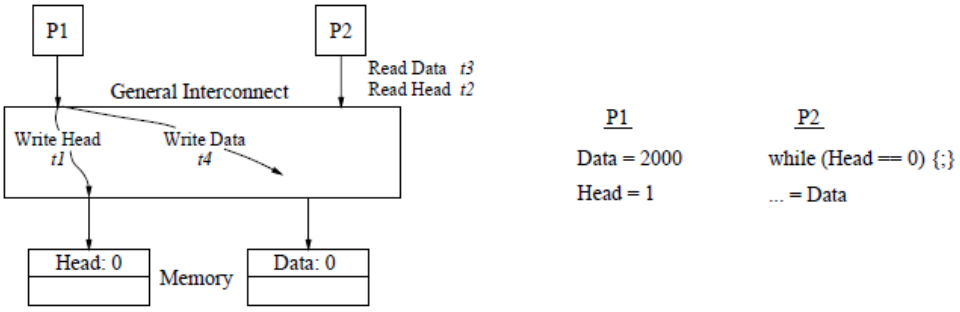
第一个优化说明了在写操作和后续读操作之间保持程序顺序的重要性。



(a) write buffer

5.1.2 重叠写操作

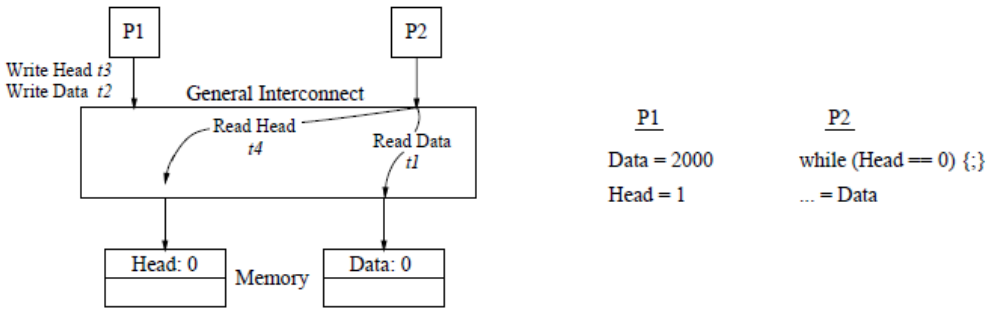
第二个优化说明了维护两个写操作之间的程序顺序的重要性。



(b) overlapped writes

5.1.3 非阻塞读取操作

第三个优化说明了保持读操作和后续读或写操作之间的程序顺序的重要性。



(c) non−blocking reads

Figure 5: Canonical optimizations that may violate sequential consistency.

5.2 带有缓存的体系架构

上一节描述了在没有缓存的情况下，在实现顺序一致性模型时，由于内存操作重新排序而产生的并发症。共享数据的缓存(或复制)可能会出现类似的重新排序行为，这会违反顺序一致性。

5.2.1 缓存一致性和顺序一致性

顺序一致性：:(1)写对于所有处理器都是可见的；(2)对相同的位置进行写，对于不同的处理器看到的顺序是相同的。

顺序一致性要求所有处理器以相同的顺序对所有位置(而不仅仅是相同的位置)进行写入，而且明确要求单个处理器的操作似乎按照程序顺序执行。

5.2.2 检测写操作的完成情况

对其他处理器缓存中复制的行进行写操作时，系统通常需要一种机制来确认目标缓存接收到无效消息或更新消息。此外，需要收集确认消息(无论是在内存中还是在发出写操作的处理器中)，并且必须通知发出写操作的处理器它们的完成。只有在上述通知之后，处理器才能认为写操作是完整的。一种常见的优化方法是，在处理节点接收到无效消息或更新消息时，在实际缓存副本受到影响之前确认该消息;只要在处理传入消息到缓存[6]时观察到某些排序

约束，这种设计仍然能够满足顺序一致性。

5.2.3 保持原子性书写的假象

虽然顺序一致性要求内存操作出现原子操作或瞬时操作，但将更改传播到多个缓存副本本质上是非原子操作。我们激励并描述了两个条件，这两个条件可以一起确保在存在数据复制的情况下出现原子性。非原子性的问题更容易用基于更新的协议进行说明;因此，下面的示例假设这样一个协议。

Initially A = B = C = 0

P1	P2	P3	P4
A = 1	A = 2	while (B != 1) {;}	while (B != 1) {;}
B = 1	C = 1	while (C != 1) {;}	while (C != 1) {;}
		register1 = A	register2 = A

Figure 6: Example for serialization of writes.

5.3 编译器

程序顺序方面的连续一致性与编译器的交互类似于硬件的交互。具体来说，对于到目前为止讨论的所有程序片段，编译器生成的共享内存操作的重新排序将导致违反顺序一致性，类似于硬件生成的重新排序。因此，在缺乏更复杂的分析的情况下，编译器的一个关键需求是在共享内存操作之间保持程序顺序。这个需求直接限制了任何可能导致内存操作重新排序的单处理器编译器优化。这包括简单的优化，如代码移动、寄存器分配和通用子表达式消除，以及更复杂的优化，如循环阻塞或软件流水线。

5.4 顺序一致性总结

从上面的讨论中可以清楚地看到，顺序一致性限制了许多常见的硬件和编译器优化。顺序一致性的简单硬件实现通常需要满足以下两个需求。首先，在按照程序顺序进行下一个内存操作之前，处理器必须确保其上一个内存操作已经完成。我们称这个需求为程序订单需求。确定写操作的完成通常需要从内存中发出显式的确认消息。此外，在基于缓存的系统中，写操

作必须为所有缓存副本生成失效或更新消息，并且只有在生成的失效和更新得到目标缓存的承认时，写操作才能被认为是完整的。第二个需求只涉及基于缓存的系统和写原子性。它要求对相同位置的写入被序列化(即。，将写入到相同位置的数据以相同的顺序显示给所有处理器)，直到写入生成的所有失效或更新得到确认(即，直到写入对所有处理器可见为止)。我们称之为写原子性要求。对于编译器，程序顺序需求的模拟应用于直接实现。此外，通过优化(如寄存器分配)来消除内存操作也可能违反顺序一致性。

6.松弛的内存模型

作为顺序一致性的一种替代方法，在学术和商业环境中都提出了几种松弛的内存一致性模型。本节首先描述用于描述各种模型的简单方法，然后使用这种方法描述每个模型。

6.1 描述不同的内存一致性模型

我们根据两个关键特性对放松的内存一致性模型进行分类:(1)它们如何放松程序顺序需求;(2)它们如何放松写原子性需求。关于程序顺序松弛，我们根据模型是否将顺序从写操作放松到后续读操作、两个写操作之间，以及最后从读操作放松到后续读或写操作来区分模型。在所有情况下，松弛只适用于具有不同地址的操作对。

图 7 展示了有哪些松弛操作。

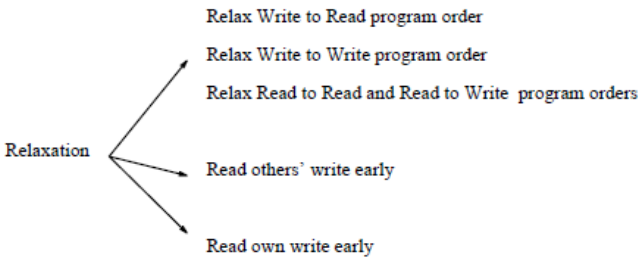


Figure 7: Relaxations allowed by memory models. The first three (program order) relaxations apply only to operation pairs accessing different locations.

图 8 对松弛模型进行了简单的分类。

Relaxation	W — R Order	W — W Order	R — RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Figure 8: Simple categorization of relaxed models. A √ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The "Read Own Write Early" relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The "Read Others' Write Early" relaxation is possible and detectable with complex implementations of RCsc.

6.2 宽松的所有程序顺序

6.2.1 弱序（WO）

弱排序模型将内存操作分为两类:数据操作和同步操作。要在两个操作之间强制执行程序顺序，需要程序员识别至少一个操作作为同步操作。这个模型基于这样一种直觉，即在同步操作之间将内存操作重新排序到数据区域通常不会影响程序的正确性。

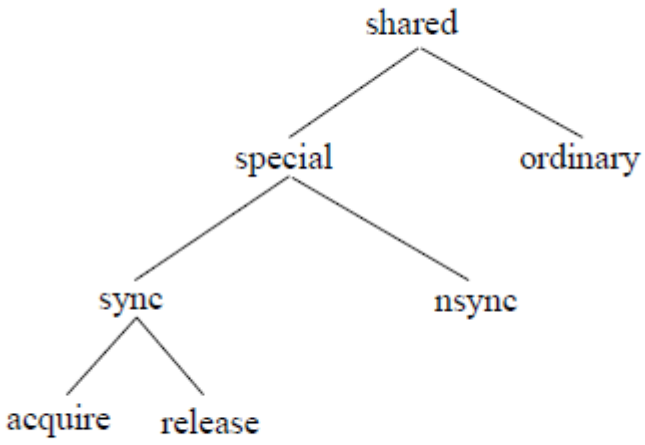


Figure 11: Distinguishing operations for release consistency.

6.2.2 版本一致性（RCsc/RCpc）

在一对操作之间强制执行程序顺序可以通过根据上述信息区分或标记适当的操作来实现。对于 RCpc，将程序顺序从写操作强制到读操作需要使用类似于 PC 模型的读-修改-写操作。此外，如果命令的写是普通的，那么读-修改-写中的写需要是一个释放;否则，在 readmodify-write 中的写操作可以是任何特殊的写操作。类似地，为了使写操作在 RCpc 中呈现原子性，可以使用读-修改-写操作替换类似于 PC 模型的适当操作。正如前面提到的，在 RCsc 的更复杂的实现中，write 也可能是非原子的。通过将充分的操作标记为特殊操作，可以实现保持写入原子性的目的;但是，在本文中介绍的简单框架中，很难精确地解释如何实现这一点。我们应该注意到，RCsc 模型也有 16 个伴随而来的更高级别的抽象，这就减少了程序员直接使用较低级别的规范进行推理的需要。

For RCsc, the constraints are as follows:

- acquire → all, all → release, and special → special.

For RCpc, the write to read program order among special operations is eliminated:

- acquire → all, all → release, and special → special except for a special write followed by a special read.

6.2.3 Alpha, RMO 和 PowerPC

Alpha、RMO 和 PowerPC 模型都提供了明确的围栏指令作为它们的安全网。Alpha 模型提供了两种不同的围栏指令，内存屏障(MB)和写内存屏障(WMB)。MB 指令可用于维护从 MB 前的任何内存操作到 MB 后的任何内存操作的程序顺序。Alpha 模型不需要写原子性的安全网。SPARC V9 RMO 模型提供了更多的围栏指令。有效地，可以定制一个 MEMBAR 指令，以便将以前的读写操作与将来的读写操作结合起来;四比特编码用于指定读到读、读到写、写到读和写到写顺序的任何组合。MEMBAR 可以用于对后续读命令进行写操作，这一事实减轻了使用读-修改-写来实现此顺序的需要，这在 SPARC V8 TSO 或 PSO 模型中是必需的。与 TSO 和 PSO 类似，RMO 模型不需要写原子性的安全网。PowerPC 模型提供了一个单独的 fence 指令，称为同步指令。为了强制执行程序顺序，同步指令的行为与 Alpha 模型的 MB 指令相似，只有一个例外。例外情况是，即使将同步放在两个读到相同位置的读之间，第二次读也可能返回比第一次读更早的写的值;即。，读取似乎发生在程序顺序之外。这可能会在程序中产生微妙的正确性问题，并且可能需要使用读-修改-写操作(类似于 PC 和 RCpc

的使用)来强制两个读到相同位置的程序顺序。PowerPC 在原子性方面也不同于 Alpha 和 RMO，因为它允许另一个处理器提前读取写操作;因此，与 PC 和 RCpc 类似，可能需要使用读-修改-写操作来使写看起来是原子的。

6.2.4 编译优化

与前几节中的模型不同，放松所有程序顺序的模型提供了足够的灵活性，允许对共享内存操作进行通用编译器优化。在 WO、RCsc 和 RCpc 等模型中，编译器具有在两个连续同步或特殊操作之间重新排序内存操作的灵活性。类似地，在 Alpha、RMO 和 PowerPC 模型中，编译器具有完全的灵活性，可以在连续的 fence 指令之间重新排序操作。由于大多数程序很少使用这些操作或指令，所以编译器会得到大量的代码，几乎所有用于单处理器程序的优化都可以安全地应用。

7.总结

有充分的证据表明，放松的内存一致性模型通过启用许多硬件优化，提供了比顺序一致性更好的性能。处理器速度相对于内存和通信速度的提高只会增加这些模型的潜在好处。在除了在硬件级别提供性能提升之外，宽松的内存一致性模型还在启用重要的编译器优化方面发挥了关键作用。以上原因导致许多商业架构，如 Digital Alpha、Sun SPARC 和 IBM PowerPC，支持轻松的内存模型。此外，几乎所有其他体系结构都支持某种形式的明确的围栏指令，这表示将来支持放松的内存模型。遗憾的是，现有的关于内存一致性模型的文献数量庞大且复杂，其中大多数是针对这一领域的研究人员，而不是典型的用户或计算机系统构建人员。本文使用统一的、直观的术语来讨论与内存一致性模型相关的几个问题，这些模型代表了当今工业中使用的模型，目的是为了达到更广泛的计算机专业人员社区。放松内存一致性模型的一个缺点是增加了编程复杂性。这种复杂性的产生主要是由于文献中提出的许多规范将程序员暴露在模型所支持的低级性能优化中。我们以前的工作通过使用更高级别的抽象定义模型来解决这个问题;只要程序员提供了关于内存操作的正确的程序级信息，这种抽象就提供了顺序一致性的假象。与此同时，高性能 Fortran 等语言标准化工作导致了与顺序一致性不同的高级内存模型。例如，高性能 Fortran 的 forall 语句指定了一组数组索引的计算，它具有复制/复制语义，其中一个索引的计算不受其他索引计算产生的值的影响。总的来说，最佳内存一致性模型的选择还远未解决，它将从语言和硬件设计人员之间更积极的协作中获益。