计算技术研究所，微处理器中心

论文阅读报告　其1
Shared Memory Consistency Models: A Tutorial

谭弘泽
201828013229048

October 17, 2018

Contents

1 前言

讲述了一点发展历史和发展目标等，略。这篇文章主要作为一片研究报告。

2 摘要

并行系统支持共享内存的抽象在许多领域正变得广为接受。为这样的系统写正确而高效的程序需要一个形式化的关于内存语义的标准，称作内存一致性模型。最直观的模型——顺序一致性——极大地限制了许多被单处理器硬件而编译器设计者广泛使用的性能优化的使用，从而降低了使用多处理器的好处。要缓和这个问题，许多现有的多处理器支持了更多的松弛一致性模型。不幸的是，被大量的系统支持的这些模型彼此在一些微妙却又重要的地方不尽相同。此外，精确地定义每个模型的语义经常导致令典型用户和计算机系统构建者难以理解复杂的标准。
这篇辅导性的论文的目的是要通过一种让大多数计算机专业人士可以理解的方式描述关于内存一致性模型的事情。我们专心于基于硬件的共享内存系统所提出的一致性模型。许多这些模型根源被指定了着重于一种它们允许的系统优化。我们保留面向系统的重点，但是使用统一而简单地属于来描述这些不同的模型。我们也简单地讨论一种代替的面向程序员的通过程序行为而不是特定的系统优化的术语来描述这些模型的视角。

3 简介

像下面这样一段多线程程序，程序员会期望P1在分配任务后，其他处理器能够正确获取到刚刚由P1新填写的任务信息。但是实际上的在许多商业系统中，程序行为会和程序员的预期不一致。

```
   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?

实际的存储系统中的很多变数和复杂性时常带来很多关于松弛一致性的错误观念，文中给出了一个表格如下：

Myth	Reality
A memory consistency model only applies to systems that allow multiple copies of shared data; e.g., through caching.	Figure 5 illustrates several counter-examples.
Most current systems are sequentially consistent.	Figure 9 mentions several commercial systems that are not sequentially consistent.
The memory consistency model only affects the design of the hardware.	The article describes how the memory consistency model affects many aspects of system design, including optimizations allowed in the compiler.
The relationship of cache coherence protocols to memory consistency models: (i) a cache coherence protocol inherently supports sequential consistency, (ii) the memory consistency model depends on whether the system supports an invalidate or update based coherence protocol.	The article discusses how the cache coherence protocol is only a part of the memory consistency model. Other aspects include the order in which a processor issues memory operations to the memory system, and whether a write executes atomically. The article also discusses how a given memory consistency model can allow both an invalidate or an update coherence protocol.
The memory model for a system may be defined solely by specifying the behavior of the processor (or the memory system).	The article describes how the memory consistency model is affected by the behavior of both the processor and the memory system.
Relaxed memory consistency models may not be used to hide read latency.	Many of the models described in this article allow hiding both read and write latencies.
Relaxed consistency models require the use of extra synchronization.	Most of the relaxed models discussed in this article do not require extra synchronization in the program. In particular, the programmer-centric framework only requires that operations be distinguished or labeled correctly. Other models provide safety nets that allow the programmer to enforce the required constraints for achieving correctness.
Relaxed memory consistency models do not allow chaotic (or asynchronous) algorithms.	The models discussed in this article allow chaotic (or asynchronous) algorithms. With system-centric models, the programmer can reason about the correctness of such algorithms by considering the optimizations that are enabled by the model. The programmer-centric approach simply requires the programmer to explicitly identify the operations that are involved in a race. For many chaotic algorithms, the former approach may provide higher performance since such algorithms do not depend on sequential consistency for correctness.

Figure 2: Some myths about memory consistency models.

这张表对应于中文如下：

传言	真实
存储一致性模型只应用于允许共享数据有多份拷贝的系统中	有几个反例，如
大多数当前的系统是顺序一致的	介绍几个并非顺序一致的一些商用系统
存储一致性模型只影响硬件设计	文章描述了存储一致性模型如何影响系统设计的诸多方面，包括编译器允许的优化
Cache一致性(Coherence)协议和存储一致性(Consistency)模型的关系： 1. Cache一致性(Coherence)协议内禀的支持顺序一致性 2. 存储一致性(Consistency)模型取决于系统支持的一致性(Coherence)协议是基于无效化的还是更新的	文章讨论Cache一致性(Coherence)协议如何只是存储一致性(Consistency)模型的一部分。其他方面包括处理器向内存系统编排内存操作的顺序，以及写操作是否是被原子地执行的。这篇文章也讨论一个给定的存储一致性(Consistency)模型如何允许无效化的或更新的Cache一致性(Coherence)协议
一个系统的存储模型可以被处理器（或者存储器系统）的行为标准唯一地定义	这篇文章描述了存储一致性模型如何被处理器和存储器系统二者的行为共同影响。
松弛一致性模型不会被用来隐藏读取延迟	这篇文章中描述的许多模型允许隐藏读取和写入延迟。
松弛一致性模型需要使用额外的同步	这篇文章中讨论的大多数松弛模型不需要在程序中进行额外的同步。实践中，面向程序员的框架只需要操作被正确地区分或标记。其他模型提供允许程序员加强需求的约束来达成正确性的安全网络。
松弛存储一致性模型不需要允许混沌(chaotic)（异步(asynchronous)）算法	这篇文章中讨论的模型允许混沌（异步）算法。对于面向系统的模型，考虑被模型允许的优化，程序员能推出这些算法的正确性。面向程序员的版本简单地需要程序员去明确地识别那些存在某些竞争的操作。对于许多混沌算法，一个比较正式的版本会由于这些算法的正确性不依赖于顺序一致性而提供更高的性能。

4 存储一致性模型——谁需要关心?

在共享内存系统中，存储一致性模型的影响普遍存在。在不同层次上，影响着可编程性、性能和可移植性。
总结一下，存储模型从程序员的角度来看影响并行程序的编写，从系统设计者的角度来看事实上影响所有的方面（包括处理器、内存系统、互联网络、编译器、程序语言）。无论是（需要或将来可能需要编写并行程序的）程序员还是各个层次的系统设计者都需要了解存储一致性模型。

5 单处理器系统中的存储语义

大多数高层单处理器语言对内存操作体现一种简单的顺序语义。这种顺序语义允许程序员假设所有内存操作会一次一个地按照程序指定的顺序完成，允许编译器进行各种优化。总体上，这种顺序语义给程序员提供一个简单而直观的模型并允许一大类高效的系统设计。

6 理解顺序一致性

定义：[一个多处理器系统是顺序一致的如果]任何执行的结果是相同的，就好像所有处理器的操作在按照某个序列的顺序执行，并且每个独立的存储器在序列中的出现按照程序指定的顺序。

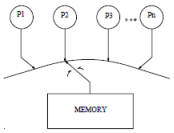


Figure 3: Programmer's view of sequential consistency.

直观理解就是，内存同一时刻只能给一个处理器使用，并且每一个处理器的请求都是依次地一次一个地发送的。
文中给了两个例子，一个是两处理器系统中的锁，一个是三处理器系统中的依赖于顺序一致性的逻辑传递，如图。

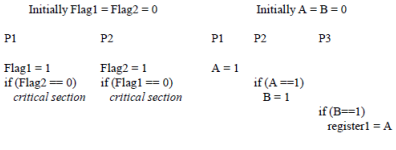
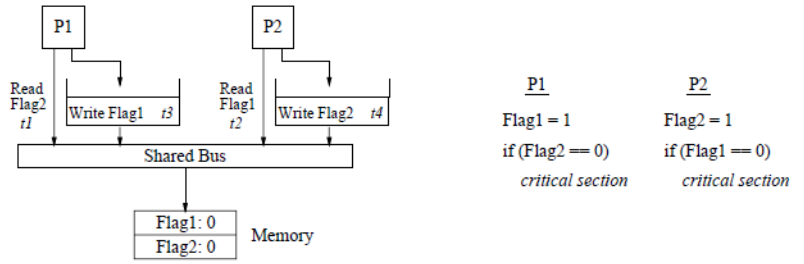


Figure 4: Examples for sequential consistency.

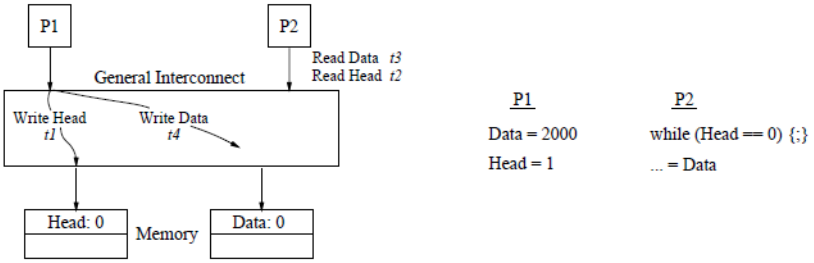
两个处理器分别有锁Flag1和Flag2，为进入临界区，两者将各自的锁置位，并检测对方是否加锁，如果是独占则成功进入缓冲区。保证这样的临界区确实是临界区的就是顺序一致性。或者三个处理器，第一个写A,第二个如果看到A被写为1则将B写为1，第三个如果看到B被写为1，读出A，这个A是1，是被顺序一致性保护的。如果没有了顺序一致性，以上程序可能出现不同的结果，这可能会对程序员带来极大的负担。

7 实现顺序一致性

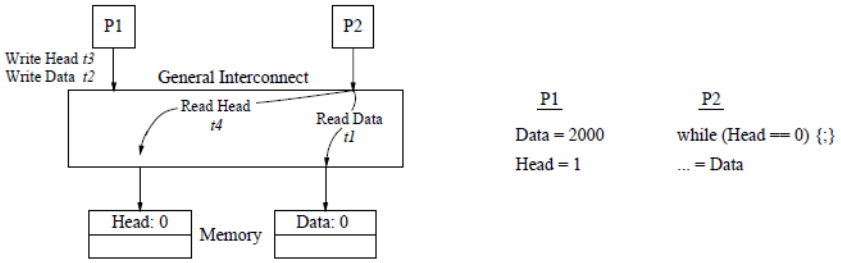
本章描述顺序一致性的直观抽象如何在实际系统中被实现。论文围绕下图三个例子进行讨论。



(a) write buffer



(b) overlapped writes



(c) non−blocking reads

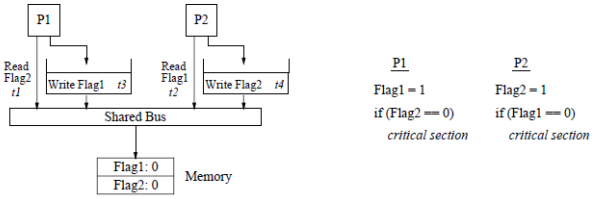
Figure 5: Canonical optimizations that may violate sequential consistency.

7.1 无高速缓存架构

一类典型系统中即使没有cache也可能破坏顺序一致性。

7.1.1 可穿透写缓冲

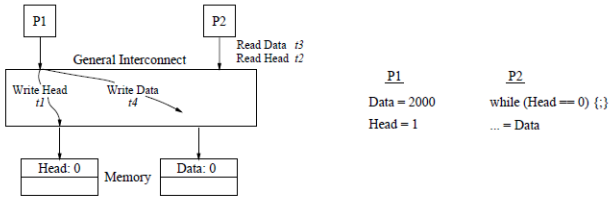
如果结构中每个处理器有独自的写缓冲，而读可以越过写缓冲执行，这个读又不会检测是否和其他处理器的写缓冲发生竞争现象，可能就会破坏顺序一致性，出现每个处理器的读都越过了对方的写的现象、。



(a) write buffer

7.1.2 交叠写操作

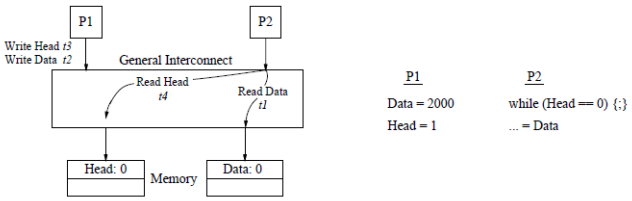
如果允许片上网络中后发生的写早于同一处理器先发生的写在RAM中生效，可能出现违反顺序一致性的情况。如Head和Data的例子，先准备好数据Data再把Head指向Data，但是另一处理器可能同时看到新的Head和旧的Data。
如果将时间上不连续的两个写操作并进同一Cache行，也有同样的问题。
要解决这个问题，一个方法是，一直等到接受信号（acknowledge）返回再发送下一个写。



(b) overlapped writes

7.1.3 非阻塞读操作

受到了网络中延迟的影响后，两个读操作实际发生的时间交换，也可能导致违反顺序一致性。



(c) non−blocking reads

7.2 有高速缓存架构

一旦有了高速缓存，又会带来很多可能违反顺序一致性的问题。其一，多核需要一种机制来传播所有的写操作，其二，检测写操作被完成的时机需要更多的通信。其三，将写操作传播给多个副本内禀的不是原子性的，如果将这些写保持一个原子性的幻像就会有更大的挑战性。

7.2.1 高速缓存一致性和顺序一致性

一个比较强的高速缓存一致性的定义是把它作为顺序一致性的同义词。其他定义会极大地放宽要求。
特别地，有几个条件会跟高速缓存一致性协议联系起来。

1. 写最终对所有处理器可见
2. 写同一地址在所有处理器看起来是同一顺序

显然，满足这些条件也完全有可能违反顺序一致性。
文章只把高速缓存一致性协议看作达成一致性模型的机制，而不把它作为任何一致性模型的定义。

7.2.2 检测写操作的完成

如果要避免顺序一致性被破坏，每个Cache的写操作需要生成给其他所有Cache的消息，并且需要收集齐所有的acknowledge信号才能继续下一个写。

7.2.3 保持写操作的原子性幻象

有多个可能需要更新的副本内禀地没有原子性。
即使每一个处理器都让写操作一次一个地完成，如果两个处理器写同一个地址，可能导致由于高速缓存更新信号的到达顺序不同，而使不同处理器得到不同的结果。如P1和P2同时写A，P1、P2消息分别距离P3、P4较近，从而P3、P4分别读出了P2、P1写的数据。

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.

7.3 编译器

很多地方其实不允许编译器进行如寄存器分配、消除公共子表达式等编译器优化。例如，如果给某个变量赋值并没有被写回内存，或者读了一次以后，一直用寄存器里的值，可能导致死循环。（如果在C语言中，记得声明成volatile）

7.4 顺序一致性总结

顺序一致性有若干个需要注意的要点：

1. 每个处理器需要确保上一个内存操作被完成后再完成下一个内存操作。

   附加地，在有高速缓存系统中，每个高速缓存都需要生成给其他所有高速缓存的无效化或更新信息，直到完成才能进行下一个内存操作。

2. 在有高速缓存系统中我们需要关心操作的原子性，写同一个地址的操作需要被序列化，必须有一个确切的顺序，而让所有的处理器看起来结果一致。

3. 对于编译器，消除内存操作可能违反顺序一致性。

两个硬件优化：

1. 在进行写操作前进行预取独占请求。

   这一条只会应用在基于无效化的高速缓存系统中。

2. 顺序一致性可以被回滚和重新安排读操作来保证，就像一次分支预测错误。

其他的隐藏延迟的策略还有，通过软件进行非绑定预取,和硬件支持多任务。这些策略对松弛一致性系统也起作用。
有一些能够让编译器检测可能破坏顺序一致性的地方的算法，但是这些算法有指数复杂度的，由多项式复杂度的，但是至少都需要像重影分析一样进行全局依赖关系分析，效率都不太高，实用性并不强。

8 松弛一致性模型

有很多种松弛一致性模型，就好比对的总是差不多的，而错的各有各的错, 虽然不符合顺序一致性不一定算是错误。放宽条件大体上分为以下5类，如下图。

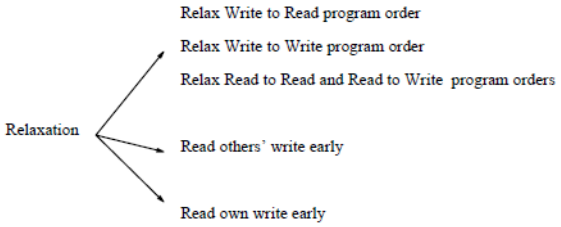


Figure 7: Relaxations allowed by memory models. The first three (program order) relaxations apply only to operation pairs accessing different locations.

实际的商业系统个对一致性的要求不尽相同，放宽的条件可能是以上5个可能放宽约束中的一个或多个。

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.

Relaxation	Example Commercial Systems Providing the Relaxation
W → R Order	AlphaServer 8200/8400, Cray T3D, Sequent Balance, SparcCenter1000/2000
W → W Order	AlphaServer 8200/8400, Cray T3D
R → RW Order	AlphaServer 8200/8400, Cray T3D
Read Others' Write Early	Cray T3D
Read Own Write Early	AlphaServer 8200/8400, Cray T3D, SparcCenter1000/2000

Figure 9: Some commercial systems that relax sequential consistency.

作者尽可能以比较统一的术语来描述各种松弛一致模型。

8.1 刻画不同的内存一致性模型

在考虑松弛一致模型的时候，有放松程序顺序和放松写的原子性两种选择。最终可能两

种都放松。
松弛一致模型的系统中一般都会提供一些能够重载掉这些放松的机制，比如引入屏障指令等。

8.2 放松写到读的程序顺序

第一类要讨论的模型放松了写和之后不同位置的读的程序顺序约束。这些模型包括IBM 370模型和SPARC V8全体存储顺序模型（TSO），以及处理器一致性模型（PC）（这个定义和Goodman定义的处理器一致性模型不一样）
IBM370最严，直到前面所有的写都对所有处理器可见都会禁止读返回结果。TSO模型部分放松了这些要求，允许一个读返回同一处理器的写结果，甚至在写操作跟其他处理器在同一位置的写序列化之前。PC模型同时放松了这两个约束，一个读可以返回任何写的值而不用等到序列化或者对其他处理器可见。
IBM370模型提供了序列化指令，可以放在两个操作之间来保证程序顺序的约束。
不同于IBM370，TSO和PC模型没有提供明确的安全网络。不过，程序员可以用读改写(read-modify-write)操作来提供一个写和接着的读的程序顺序被保持的幻象。其中，读改写(read-modify-write)操作是一种原子操作，包含读取修改和写回三部分，就好像MIPS指令集中的LL/SC指令块一样。对于TSO，程序顺序看起来会是被保持的，如果读或者写之一要么是读改写操作的一部分，要么被替换乘一个读改写操作。要去把读替换成读改写，读改写里面的写必须是写回读出的内容的“虚设”的写。相似的，要去把写替换成读改写，读改写里面的写必须是写入期望的内容而不去管读入的值。
对于PC，程序顺序看起来会是被保持的，如果读要么是读改写操作的一部分，要么被替换乘一个读改写操作。不同于TSO，把写替换成读改写不足以保证PC的程序顺序。这个差异的出现是因为TSO的读改写有更加严格的约束。
IBM370不破坏写的原子性，TSO只有在同一个处理器先写再读回来的时候需要注意原子性，PC需要借助读改写操作。
讨论读改写操作保持程序顺序的原因超过了论文范文。总得来说依赖于读改写操作来保持程序顺序有不少缺点。

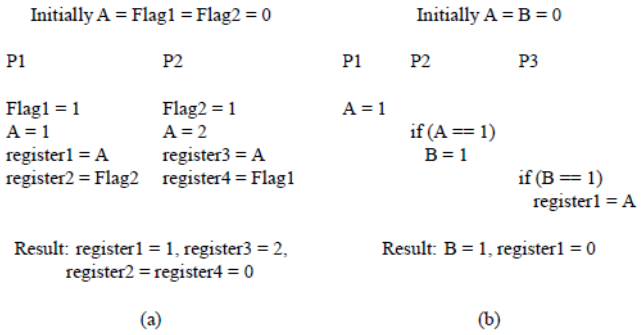


Figure 10: Differences between 370, TSO, and PC. The result for the program in part (a) is possible with TSO and PC because both models allow the reads of the flags to occur before the writes of the flags on each processor. The result is not possible with IBM 370 because the read of A on each processor is not issued until the write of A on that processor is done. Consequently, the read of the flag on each processor is not issued until the write of the flag on that processor is done. The program in part (b) is the same as in Figure 4(b). The result shown is possible with PC because it allows P2 to return the value of P1's write before the write is visible to P3. The result is not possible with IBM 370 or TSO.

上图中左边(a)这种情况会在TSO和PC模型中发生。右边(b)这种情况只会在PC模型中发生。

8.3 放松写到读和写到写的程序顺序

第二类模型通过消除不同位置的写的顺序约束进一步放松了程序顺序要求。SPARC V8部分存储顺序模型（PSO）是这里讨论的唯一的例子。
PSO允许了像之前图5(a)和图5(b)那样的非顺序一致的结果。
PSO提供了一个明确的STBAR指令来保持两个写的程序顺序，在这条指令之前的写操作必须全部落实才能继续进行后续的写操作。
和前一个模型一样，PSO允许的优化对于编译器不足以灵活到有用的地步。

8.4 放松全部程序顺序

我们考虑的最后一类模型放松不同位置的所有操作的程序顺序。
文章讨论弱顺序（WO，Weak Ordering）模型，两种释放一致性模型（RCsc/RCpc），和三种由商业架构提出的模型：Digital Alpha，SPARC V9松弛内存模型（RMO），和IBM PowerPC模型。除了Alpha都允许同一位置的两个读交换顺序。往回看图5，所有的模型都会在所有的示例代码中违反顺序一致性。

8.4.1 弱顺序(WO,Weak Ordering)

弱顺序模型把内存操作分为两个范畴：数据操作和同步操作。要加强两个操作间的程序顺序，程序员需要让至少一个操作等于同步操作。这个模型基于重排同步操作之间的数据区域内存操作一般不影响程序正确性的直观。
被划分为同步的操作有效地提供一个加强程序顺序的安全网络。这里简要地描述了通过硬件支持适当的功能的简单方式。每个处理器提供一个保持跟踪未完成的操作的计数器。这个计数器当处理器发出一个操作的时候自增，当先前的操作完成时自减。每个处理器要发出同步操作，必须保证直到先前所有的操作都被完成，也就是以计数器的零值作为信号。进一步，每个处理器要发出任何操作必须保证先前的同步操作完成。注意两个同步操作之间的内存操作仍然可能被重排或者交叠。
弱一致模型保证写操作让程序员看起来总是原子的，从而不需要用安全网络来保证写原子性。
MIPS指令集会提供SYNC或者SYNCI之类的指令，如果在一个弱顺序模型中实现，应该就可以用类似于这样的方式来完成。

8.4.2 释放一致性(RCsc/RCpc)

释放一致性下有更多的操作类别，如下图。

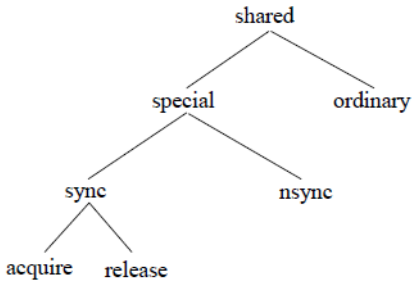


Figure 11: Distinguishing operations for release consistency.

最终同步（sync）分为了获取(acquire)和释放(release)两类。
对于RCsc，顺序如下：
获取操作→所有操作，所有操作→释放操作，特殊操作→特殊操作。
对于RCpc，顺序如下：
获取操作→所有操作，所有操作→释放操作，特殊操作→特殊操作，除非特殊写后面跟着特殊读。
其中，对于两个操作A、B，$A \rightarrow B$表示如果按照程序顺序，操作类型A在操作类型B之前发生，那么这两个操作的程序顺序必须被保持。
acquire可以一定程度上超前，release也可以被常规操作超前。

8.4.3 Alpha, RMO, 和PowerPC

Alpha, RMO, 和PowerPC提供了明确的栅栏指令。
Alpha有内存屏障（MB，memory barrier）和写内存屏障（WMB，write memory barrier）两种。
SPARC V9 RMO模型提供了更多种的栅栏指令，比如MEMBAR。
PowerPC提供了一条单独的栅栏指令，SYNC。

8.4.4 编译器优化

这些模型可以允许编译器做一些诸如存取顺序重排的优化。

9 松弛一致模型的一个代替的抽象

9.1 一个面向程序员的框架的例子

如下图这个例子中，Head依赖于另外一个线程，算是同步信号，而Data的值算是数据操作。

```
Initially all locations = 0

P1              P2

Data = 2000     while (Head == 0) {;}
Head = 1        ...  = Data
```

Figure 12: Providing information about memory operations.

如下图，对于一个内存操作，如果绝对不会发生竞争现象，那么这个操作就是数据操作。如果可能发生竞争现象，或者不清楚会不会发生竞争现象，那么保险起见都算是同步信号。

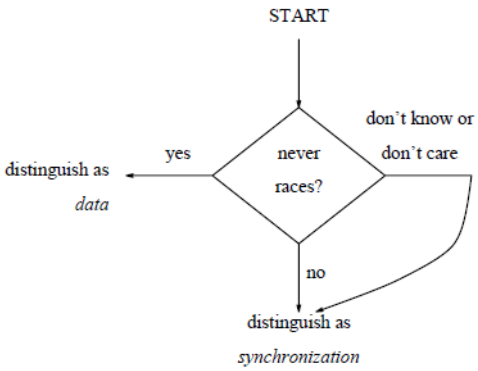


Figure 13: Deciding how to distinguish a memory operation.

9.2 区分内存操作的机制

这节描述一些可能的几个前面章节提到的传递面向程序员的框架所需要的信息的机制。

9.2.1 在编程语言级别传达信息

许多语言规定了一些范式，并且要求程序员去使用这些范式。
低层一些，编程语言可能提供一些公共的同步例程，要求程序员去调用这些例程来实现同步。
程序员也可能被允许直接用一些内存操作来达到同步目的。这种情况下，程序员必须显式地传达这些操作类型的信息。
编程语言也可以默认进行同步，这样程序更简单能够减少错误，但是要使用更紧凑的数据操作就需要显式声明了。

9.2.2 向硬件传达信息

这些信息可能会和特定的地址范围或者内存指令关联起来。

10 讨论

有很强的证据表明松弛内存一致性模型能够通过允许一系列的硬件优化，提供比顺序一致性模型提供更好的性能。处理器速度相对于内存和通讯速度的提升只会进一步提高这些模型潜在的好处。而且不但能够是硬件性能提升，松弛内存一致性模型还能允许一些重要的编译器优化。
不过松弛内存一致性模型的一个缺点是会增加编程复杂度。总之，选出最好的内存一致性模型的问题离解决还有很远，这需要编程语言和硬件设计者的积极合作。

11 附录A-单词翻译对照表

英文	中文
thereby	由此，从而
futhermore	此外，再者，与此同时
myth	传言
chaotic	混沌的
pervasive	普遍的
relaxed	松散的
consistency	一致性、连贯性
coherence	一致性、连贯性
sequential	顺序的
sequential consistency	顺序一致性
relaxed memory consistency	松弛存储一致性
alleviate	缓和
coalesce	联合
stall	拖延
synonym	同义词
impose	强加
convey	表达，传递
with respect to	关于，涉及
explicit	明确的，清楚的
dummy	仿制品，傀儡；虚设的，挂名的
stringent	严格的，迫切的
outstanding	杰出的，显著的，未完成的
depict	描绘
precede	vt.在...之前发生
paradigm	范例，样式

12 附录B-咬文嚼字

12.1 混沌算法(chaotic algorithm)

部分迭代算法不总需要让系统处处同步，例如较大的神经网络集群可能会在梯度下降（反向传播）的时候每个机器的神经网络参数并不完全同步，从而降低网络延迟对系统的影响（引入误差，但是在系统趋近收敛的时候不同步也会越来越少）。

12.2 一致性(Consistency)与一致性(Coherence)

Coherent主要是不同副本之间要保持一致，不能相互矛盾。
Consistent主要是作为整个系统，不能自相矛盾。