Parallel Processing

Lecture: **Shared Memory Multiprocessors and Programming**

王展

前期内容回顾

- 并行处理基础：基础历史和基础编程介绍
- 并行程序性能评价方法
- 并行体系结构及编程
  - 数据并行结构及编程：向量部件、GPU（部件、芯片层次—关注计算架构）
  - 共享内存结构及编程：SMP（节点层次—关注访存架构）
  - 消息传递结构及编程：MPP和Cluster（大规模整机层次—关注通信架构）

本次授课提纲

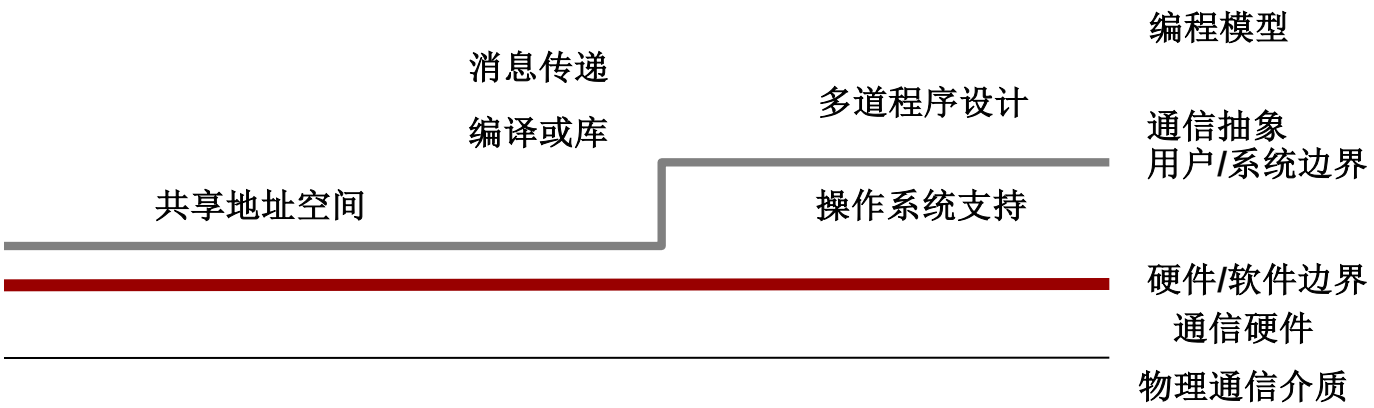
- 共享存储多处理器架构简介
- 存储系统的一致性简介
- 基于侦听的高速缓存一致性
- 共享存储多处理器架构下的OpenMP编程

共享存储多处理器架构简介（从SIMD到MIMD）

共享存储多处理器架构 Shared Memory Multiprocessors

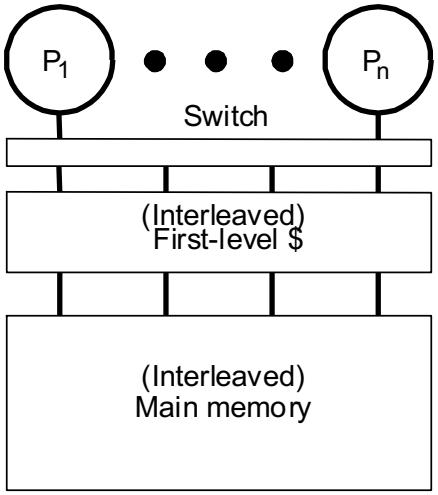
- **对称多处理器 Symmetric Multiprocessors (SMPs)**
  - 提供全局物理地址空间
  - 任一处理器对同一块主存的访问是“对称”的
  - 对称：访存行为一致，访存性能一致
- **对每个处理器来说，都使用和单处理器同样的访存机制**
  - 通过在存储架构上使用相关技术实现多处理器扩展
- **是当前市场上的主流架构**
  - Server、desktop、laptop
- **是一种对并行程序非常有吸引力的架构**
  - 具有很强的吞吐能力（相比单处理器）
  - 细粒度的资源共享（相比MPP或Cluster）
  - 通过普通的loads/stores操作高效的访问共享的数据
  - 能够在不同处理器Cache之间实现自动的数据移动和Cache一致性拷贝

从通信体系结构层次来看SMP

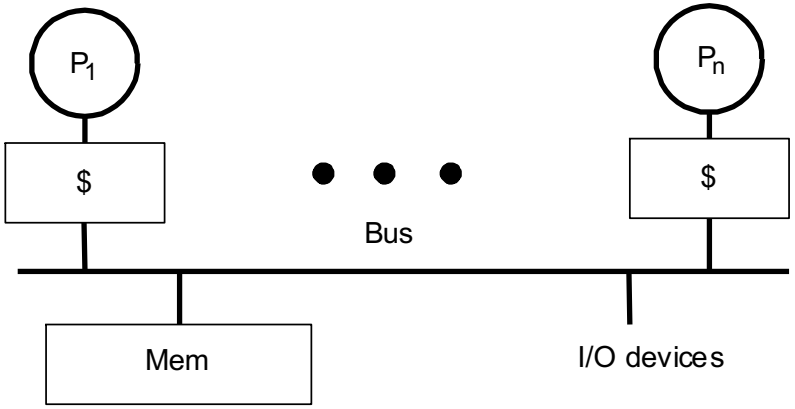


- 共享地址空间的地址翻译和保护直接由硬件支持(hardware SAS)
- 消息传递可以用共享地址空间buffer中的数据拷贝来实现
  - 只要对共享总线和存储的争用不是瓶颈，这样的实现无需涉及操作系统，优于分布式存储消息传递系统，非常高效

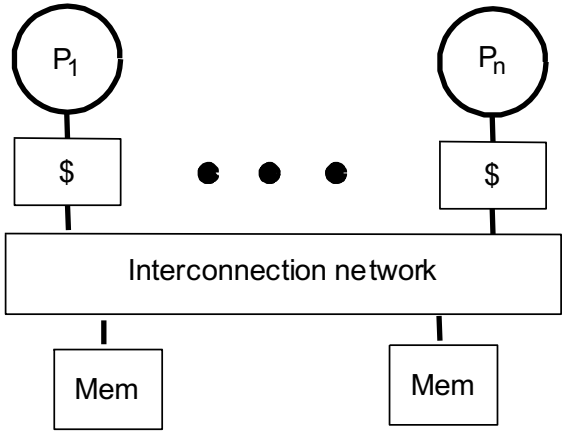
常见的四种共享存储多处理器系统结构



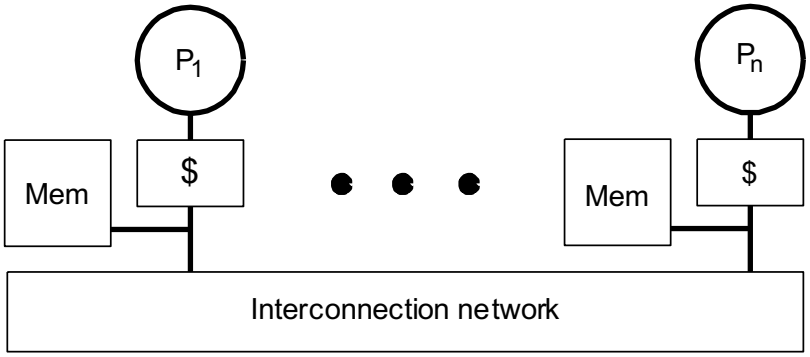
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall

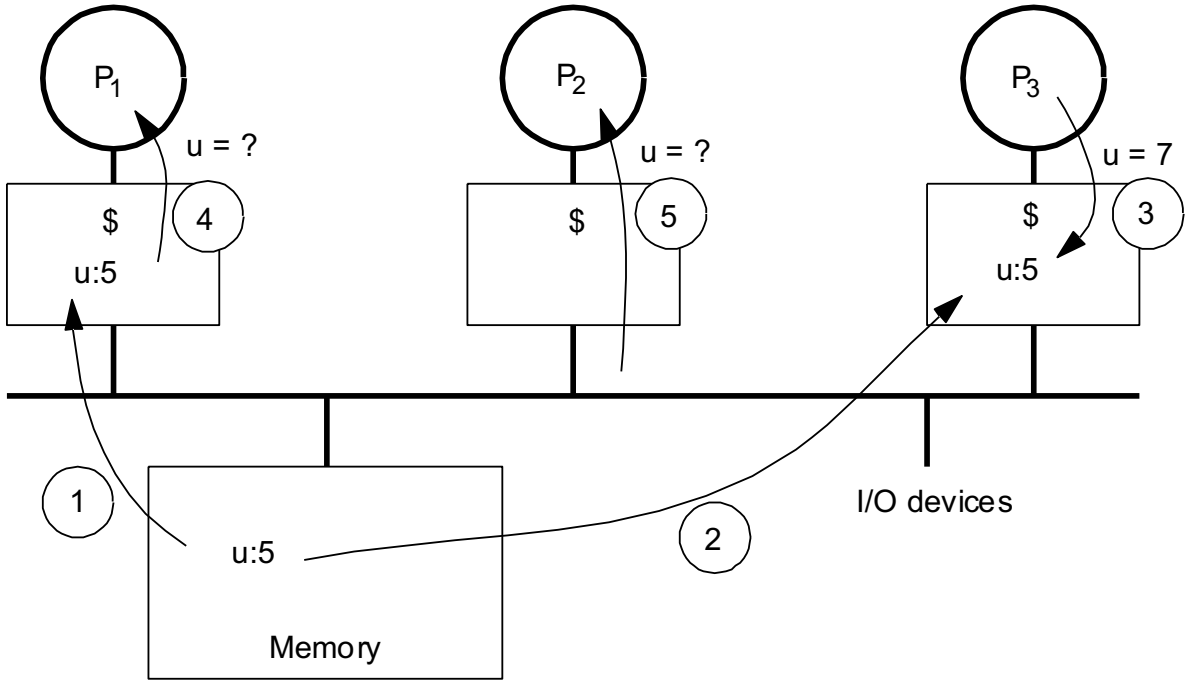


(d) Distributed-memory

Caches在SMP系统中的作用和问题

- **Caches在SMP系统中的关键作用**
  - 减少平均数据读写时间
  - 减少各个处理器或处理器核对共享总线带宽资源的占用
- **但是各个处理器或处理器核独有Cache，带来新的问题**
  - 同一变量的不同拷贝可能出现在多个Cache里
  - 某个处理器对该变量的写可能对其他保有该变量的处理器是不可见的
    - 其他处理器会继续使用该变量的旧值
  - 这就是**Cache一致性问题**（*Cache coherence* problem）

Cache一致性问题



- 如果使用的是write-back cache，缓存在cache中的数据在刷新相关cache line时才写回存储
- 在事件3发生后，处理器之间看到的变量u的值就有可能不同，有些处理器拿到了变量u的旧值

存储系统的一致性简介

直觉上的存储行为模型

- 提供一组保存数据的单元，当处理器读取某个单元时，返回写入该单元的最新值
- 在单处理器/单核处理器上很容易实现
  - 但也有例外: I/O设备与处理器之间的存储一致性（eg. 富岳）
  - 上述问题的一些软件解决方法
    - 使用不可缓存的存储（ uncacheable ），不可缓存的操作，强制刷Cache，强制数据读写绕开Cache
- 我们希望多进程运行多处理器上时，获得的结果与单处理器上一致
- 但事与愿违：**Next↘**

问题的出现关乎“序（Order）”

- 直观概念中：每个读操作必须返回“最后（Last）”写入此位置的值
- 如何理解“最后（last）”的含义?
- 在串行程序下，“last”也不是通过时间（Time）来定义，而是通过程序操作的次序（Order）来定义，
- 在并行情况下，定义好了进程内的自己独立操作顺序，还需要定义清楚进程之间的操作顺序

如何定义序，先回顾一些单处理器下

在单处理器环境下：

- **存储（访存）操作（Memory Operation）**
  - a single read (load), write (store) or read-modify-write access to a memory location
  - 假设这些操作相互之间都是原子操作
- **发射（Issue）**
  - 存储操作离开处理器内部环境并呈现给存储系统（cache，buffer…）
- **执行（Perform）：**
  - 一个存储操作相对于某处理器执行了，只要它能感受到所发出的存储操作的效果
  - 例如：写操作执行：该处理器后续的读操作能够返回此写操作写入的值
  - 例如：读操作执行：该处理器后续的写操作不会影响该读操作的返回值

- **在串行情况下，“后续”一词的含义是明确的，因为读和写操作的次序是由程序中语句的次序决定的**
- **多处理器环境下：replace “the” by “a” processor**
  - 完成（*complete*）：所有处理器都执行了相关操作
  - 需要理解不同进程操作的顺序

迁移到多处理器并行程序上

为了简化讨论，假设系统只有单一共享存储，没有缓存

- **存储可以在所有处理器对一个单元发出的读写操作上强加一个序**
  - 来自某个处理器的读写操作遵从自己的程序操作序
  - 存储单元成为硬件上的自然参照点，可以确定跨处理器访问此单元的操作顺序

- **这个序的一些性质**
  - 不同处理器之间对存储单元的任何交替访问都是合理的
  - 只要此访问序列能够体现每个处理器的程序操作序即可

- **“最后的（Last）”是指在保持这些性质下的一个假定串行序中最近的元素，“后续的”是指在保持这些性质下的一个假定串行序中两个操作的后一个操作**
  - 对所有处理器来说，这个序都是一致的
  - 所有处理器看到对某个单元的写操作应该是相同的

多处理器存储系统一致性的形式化定义

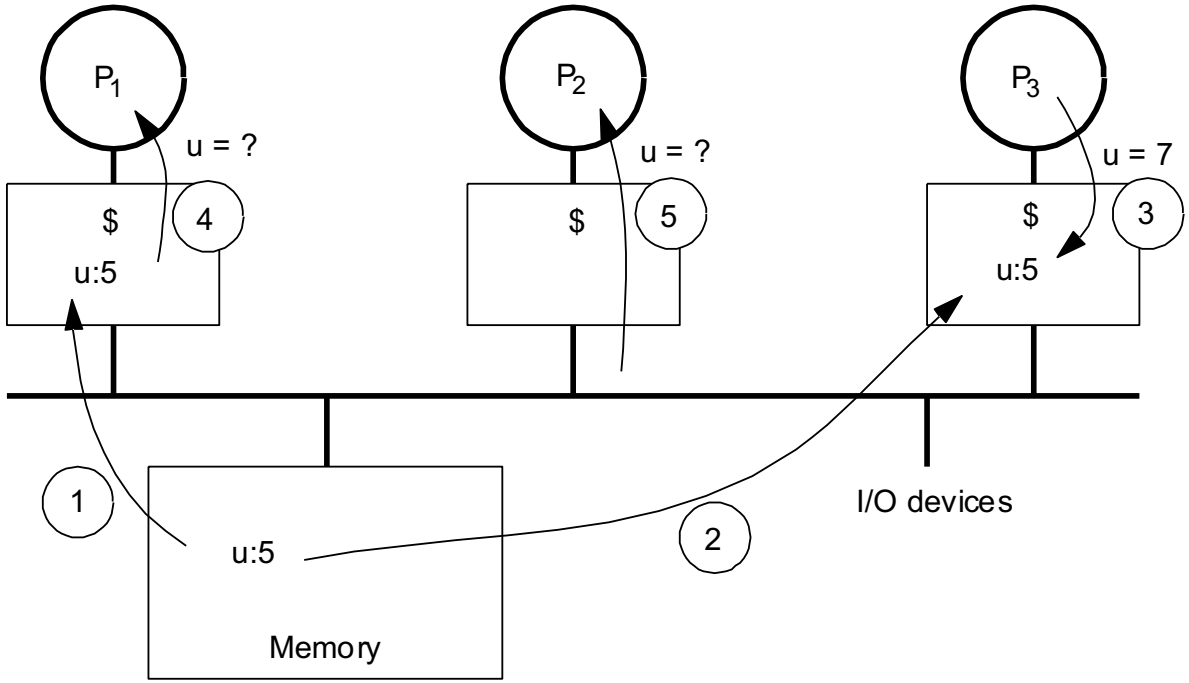
- 我们说多处理器存储系统是一致的，如果某个程序的任何执行结果都满足下列条件：对于任何单元，有可能建立一个假想的操作序列（也就是说，将所有进程发出的读写操作排成一个全序），此序列与执行结果一致，并且在此序列中：

1）任何特定进程发出的操作所表现出来的序和该进程向存储系统发出他们的序相同

2）每个读操作返回的值是对应单元按串行顺序写入的最后一个值

- 上述定义中隐藏的两个性质：
  - 写传播（*Write propagation*）：写操作的效果对其他进程可见
  - 写串行化（*Write serialization*）：对于某个单元的所有写操作（来自相同或者不同的进程）而言，所有进程都是以相同顺序看到这些操作
    - 如果P1看到w1在w2之后，P2就不应该看到w1在w2之前
    - 没有必要把写串行化的概念推广到读操作上，因为读操作的结果只有发出此操作的进程可见

再来回顾Cache一致性问题



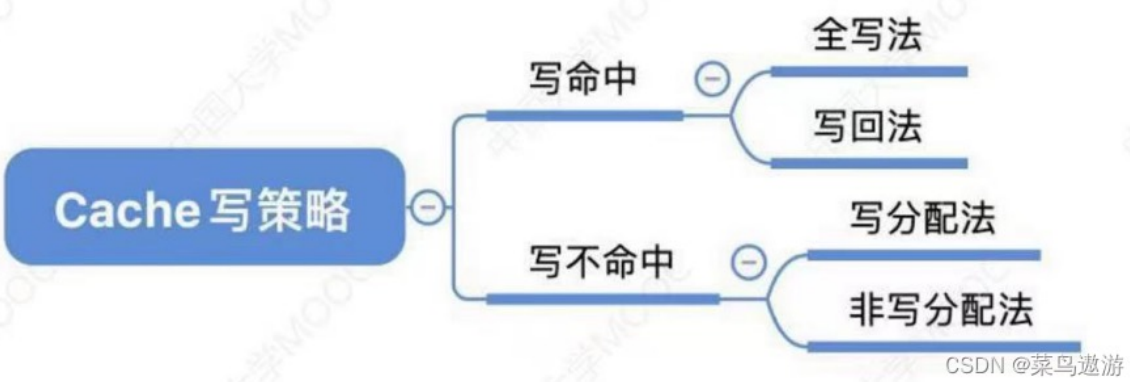
- 如果使用的是write-back cache，缓存在cache中的数据在刷新相关cache line时才写回存储
- 在事件3发生后，处理器之间看到的变量u的值就有可能不同，有些处理器拿到了变量u的旧值

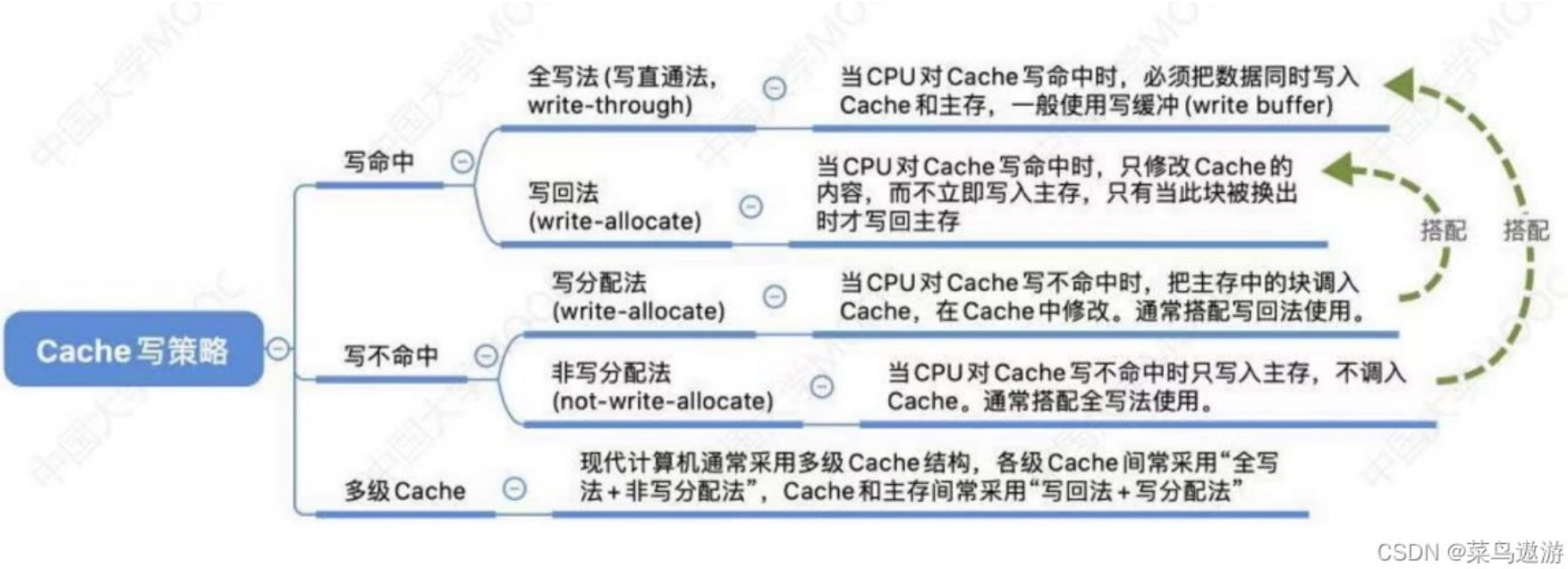
基于侦听的高速缓存一致性

首先回顾一下单处理器高速缓存的基本知识

- **单处理器高速缓存的两个基本方面**
  - 总线事务
  - 高速缓存块相关的状态转换图
- **总线事务:**
  - 由三个阶段组成: 仲裁（arbitration）, 发出命令/地址（command/address）, 传输数据（data transfer）
  - 所有设备都会看到相关地址，但只有一个会响应
- **高速缓存块的状态转换图:**
  - 是指示一个数据块的特征（例如：无效、有效、脏）如何改变的有限状态自动机
  - 写穿透（Write-through）, 写不分配（write no-allocate）的高速缓存只需要两个状态: valid, invalid
  - 写返回（Writeback） caches 会有一个额外的状态: modified (“dirty”)
- **多处理器系统对上述两个方面进行扩展实现Cache一致性**

Cache写策略回顾


Cache写策略
写命中
全写法
写回法
写不命中
写分配法
非写分配法
CSDN @菜鸟遨游


Cache写策略
写命中
全写法(写直通法，write-through)
当CPU对Cache写命中时，必须把数据同时写入Cache和主存，一般使用写缓冲(write buffer)
写回法(write-allocate)
当CPU对Cache写命中时，只修改Cache的内容，而不立即写入主存，只有当此块被换出时才写回主存
写不命中
写分配法(write-allocate)
当CPU对Cache写不命中时，把主存中的块调入Cache，在Cache中修改。通常搭配写回法使用。
非写分配法(not-write-allocate)
当CPU对Cache写不命中时只写入主存，不调入Cache。通常搭配全写法使用。
多级Cache
现代计算机通常采用多级Cache结构，各级Cache间常采用“全写法+非写分配法”，Cache和主存间常采用“写回法+写分配法”
搭配
搭配
CSDN @菜鸟遨游

基于侦听的高速缓存一致性

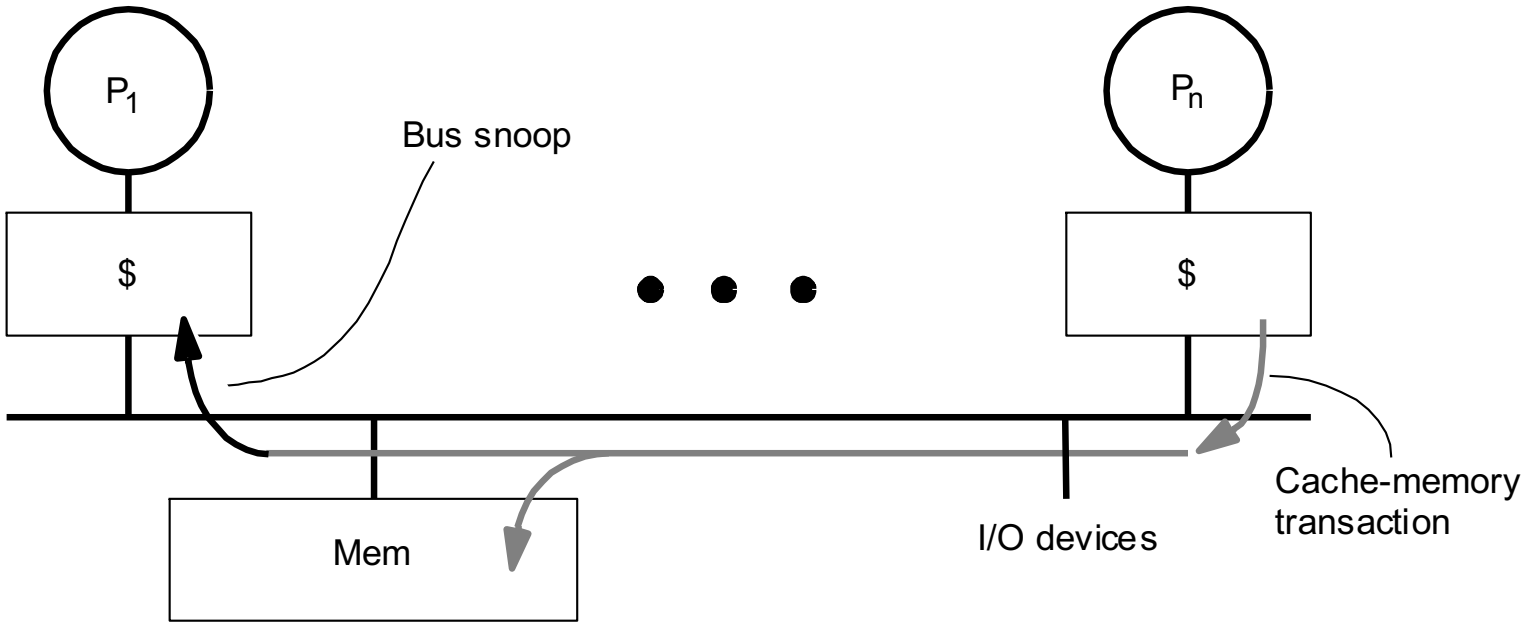
基本思路

- 所有处理器都可以观察到总线上的所有事务
- 处理器及其高速缓存控制器侦听（或监控）总线上发生的事务，并针对与其相关事务采取策略 **(e.g. change state)**
- 满足写传播和写串行化，通过策略实现结果一致

实现方式（协议）

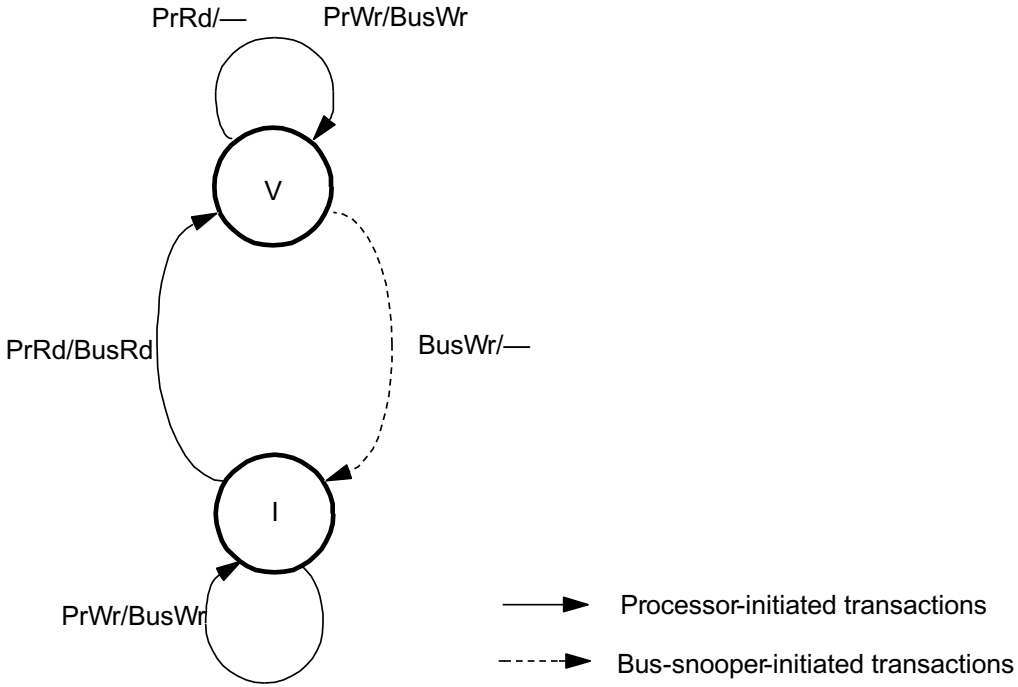
- 高速缓存控制器同时接收两个输入（原来是一个）：
  - 由处理器发出的存储请求
  - 由总线侦听器通知关于从其他高速缓存发起的总线事务
  - 根据当前的状态和状态转换图采取行动
    - Updates state, responds with data, generates new bus transactions
- 该协议是一种分布式算法: 由一组相互作用的有限状态自动机来表示
  - 和存在于Cache的存储块相联系的一组状态、状态转换图、和状态转关相关的动作
- 维护一致性的粒度是cache块

例子：单级写穿透Cache一致性系统



- 相比单处理器缓存系统: 增加了总线侦听, 作废/更新 caches
  - 不需要增加新的总线事务
  - 由两种实现方式：基于作废的协议和基于更新的协议
  - 写作废通过一次本地处理器的缓存扑空，从主存获得更新后的数据

写穿透一致性协议状态转换图



- 和在单处理中一样，每个缓存块有两个状态：无效I和有效V
- 存在Cache中的数据块都有一个状态位，不在Cache中的使用I状态
- A/B表示如果A被观察到，那么生成B事务，如处理器产生写事务，就生成总线上的写事务
- 在上述转换图中，写操作会无效掉所有所有相关Cache块（写作废）

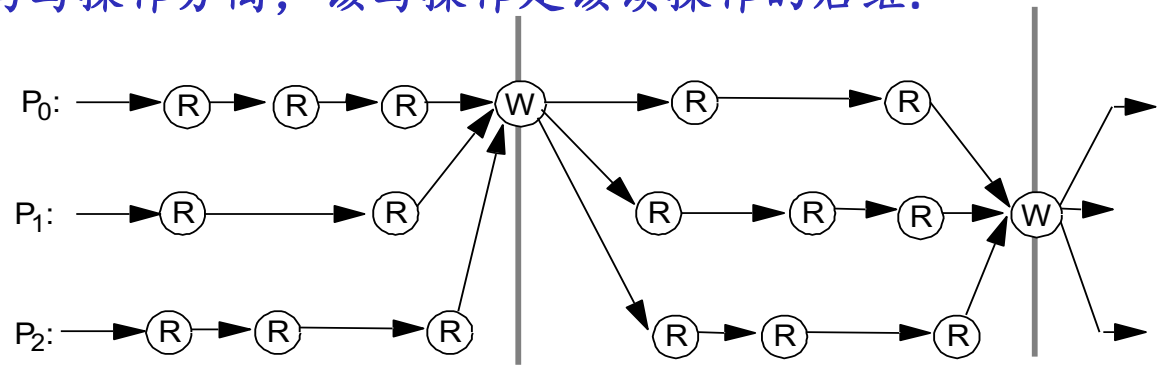
上述协议如何提供一致性

- **目标：验证上述协议能否实现，满足程序语句序和写串行化条件的，多处理器下的程序完全序列**
- **假设总线事务和存储器操作都是原子的**
  - 一旦一个请求被放到总线上，他的所有阶段完成之前，不允许其他请求放到总线上
  - 处理器要等待前面一个存储请求完成再发出下一个
  - 对单级Cache来说，作废应用于Cache，写操作在总线事务期间完成
- **在上述协议中，所有写操作都在总线上出现，并原子化执行**
  - 不管在哪次执行中，所有对某单元的写操作都按它们出现在总线上的顺序串行化（称为总线顺序）
  - 因为侦听控制器在写操作总线事务器间执行作废，所以也按照总线顺序串行化
- **读操作如何插入写操作的串行序列?**
  - 由于读可以在Cache上执行，对一个单元的读并不是完全串行化的
  - 考虑两种情况，出现在总线上的读操作（读扑空）和不出现在总线上的读操作（读命中）
  - 两种操作的结果来源都是总线顺序最近写入该单元的值

写穿透一致性协议下的全序构造

任何保持下述偏序的串行顺序都是一致的

- 如果操作是由同一个处理器发出的，并且在程序顺序中，存储器操作M2在存储器操作M1之后，则M2是M1的后继.
- 如果读操作生成一个过程在写操作W生成的事务之后，则该读操作是W的后继.
- 如果一个读或写操作M生成一个总线事务并且一个写操作的总线事务在M的总线事务之后，则该写操作是M的后继.
- 如果一个读操作并不生成总线事务（是一次命中），且并没有被另一个总线事务将其与写操作分离，该写操作是该读操作的后继.



·写总线事务定义了一个全局事件序列

·其间各个处理器按照程序操作序执行读操作，在保证各自序的前提下，处理器之间的交织得到的任何全序和该执行结果是一致的

写穿透的问题

- **较高的内存带宽要求**
  - 所有处理器的写操作都会放到总线和内存上
  - 考虑 200MHz, 1CPI processor, and 15% instrs. are 8-byte stores
  - 每个处理器每秒产生 30M stores or 240MB 数据
  - 1GB/s 总线仅能支持 4 个左右处理器
  - 所以一般SMP不会采用该协议
- **写返回通过Cache命中吸收了部分对总线和内存的访问**
  - 写命中操作不会放到总线上
  - 如何设计协议满足一致性的写传播和写串行化?
- **在讨论上述问题之前，我们来看下存储同一性**

存储同一性（Memory Consistency）问题

- 写在一个单元的数据将对所有的读取者都会是可见的，但是何时成为可见的，看下面的例子：

P_1	P_2
/*Assume initial value of A and ag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- 从程序员的直觉来看，输出的A的值必须是1，根据程序语句序
- 但一致性并不能保证，一致性并没有隐含P1和P2对不同地址单元的访问顺序，一致性仅仅要求A的当前值最终对P2是可见的，而不一定是在flag的新值被观察到之前

再来看一个例子

P_1	P_2
/*Assume initial values of A and B are 0 */	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

- 这里没有用标记，也没与用显式同步事件，程序员的意图就不太清楚了
- 因此，我们需要某些规则
  - 定义一个序模型，每个进程看到的对存储器并发读写的序
  - 程序员能推断他们程序执行的结果及其正确性
- 这就是存储同一性模型（memory consistency model）

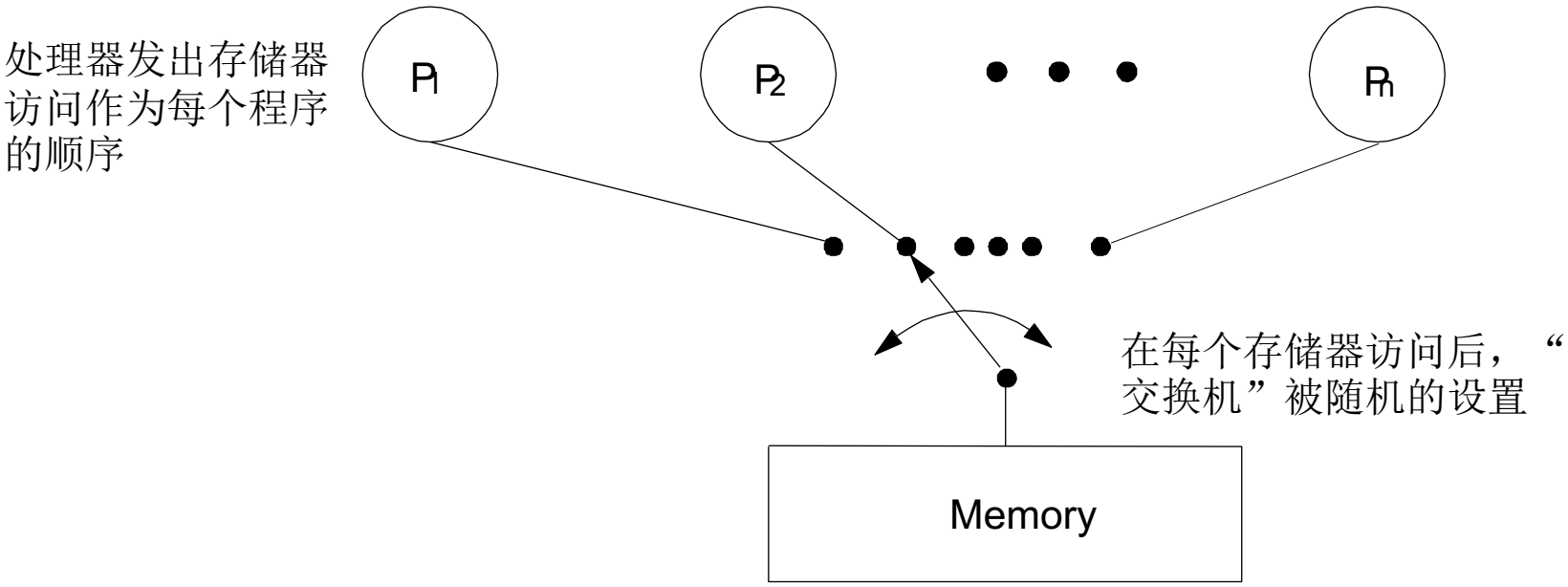
存储同一性模型（Memory Consistency Model）

- 在共享存储空间上，多个进程对存储器的不同单元做并发的读写操作，每个进程都会看到一个这些操作被完成的序，一个存储同一性模型规定了对这种序的若干约束条件
- 对程序员和系统设计人员都有影响
  - 程序员用它来预测程序运行的结果
  - 系统设计人员用它来约束编译器或硬件产生合理的访存编排
  - 是系统和程序员之间的桥梁

顺序同一性（Sequential Consistency）

称一个多处理器是顺序同一的，如果程序在它上面的任何一次执行的结果都和其中所有操作按某种顺序执行的结果一致，并且在这种顺序执行中每个处理器所完成的操作的顺序和它的程序执行序（也叫程序原序）一致。 [Lamport, 1979]

顺序同一性模型下程序员看到的存储子系统抽象



顺序同一性中的“程序执行序”如何理解?

- 从直观上看，某个进程的程序执行序仅仅是源程序中语句执行的的顺序
  - 编译器按照直截了当的方式产生汇编代码
  - 每条指令最多只能执行一次内存操作
- 但这不一定是优化编译器产生的硬件存储操作顺序
- 那么究竟什么才是程序的执行序？
- 事实上，在并行计算机体系结构的每一个接口上都有一个“程序执行序”，这些序模型是可以分别定义的
- 因为程序员总以源程序推断程序执行行为，所以，我们关心的同一性模型使用源程序的程序执行序

顺序同一性的例子

P_1	P_2
/*Assume initial values of A and B are 0*/	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

- (A,B)的可能结果包括: (0,0), (1,0), (1,2); 在顺序同一性下不可能的结果是: (0,2)
  - 因为按照不同处理器上的程序执行序有： 1a->1b and 2a->2b
- 但是，如果实际执行的顺序是1b->1a->2b->2a，尽管和我们看到的程序执行序不同，但结果是一致的，因此也是顺序同一的

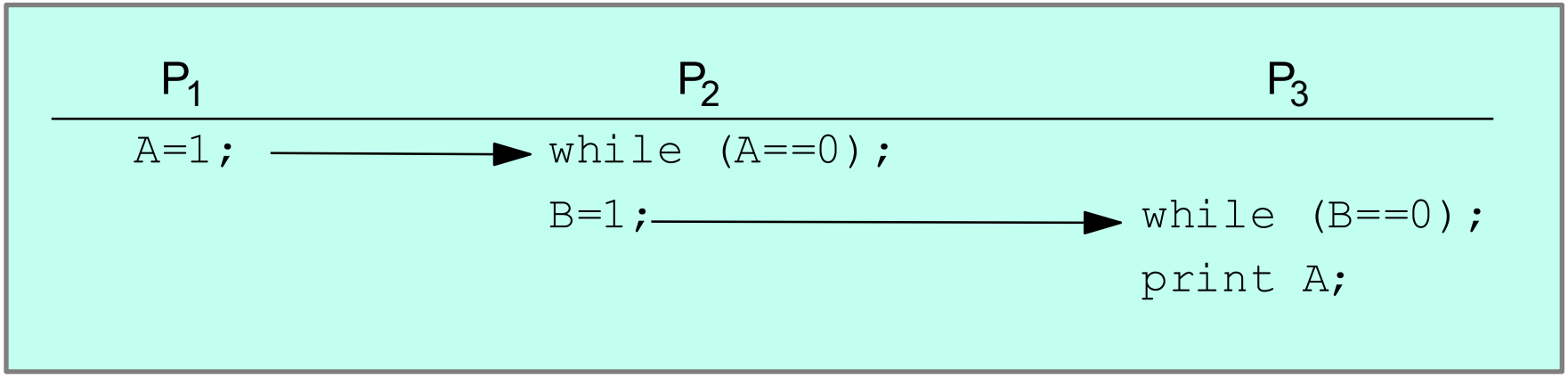
存储操作具体以何种顺序来执行是不重要的，重要的是满足顺序同一性的定义和约束

如何实现顺序同一性（SC）模型

- **要求系统（软件和硬件）满足两个条件：**
  - 程序执行序
    - 一个进程的存储操作被自己或者别的进程可见的顺序必须符合程序执行序
  - 操作的原子性
    - 一个所有进程都能看到的操作的完成，应该在总体顺序中的下一个操作开始执行前
    - 用于确保所有操作的全序，或者说在进程间交叉执行的情况，对所有进程都是一样的
    - 最棘手的部分是解决写操作的原子性

写操作的原子性

- 写操作的原子性（***Write Atomicity***）：一个写操作在所有操作全序中的执行位置应该对所有处理器都是一样的
  - 当一个处理器看到自己执行的一个写操作的效果后，在别的处理器见到这个写操作产生的新值前，这个处理器所做的任何事情（例如另一个写操作），对其他处理器都是不可见的
  - 从效果上看，SC模型扩展了一致性要求的写操作串行化，要求所有写操作在所有处理器看来都是以相同顺序发生



- 根据SC模型，逐个进程传递，A应该为1
- 如果允许P_2越过读A, 直接写B, P_3 就有可能读出 new B but old A

SC模型更加形式化的描述

- 每个进程的程序执行序在所有操作集上强加了一个偏序（部分元素可比较）
- 不同进程操作的交织执行就定义了一个全序（所有元素可比较）
- 因为SC模型并没有定义具体的交织方式，满足每个进程操作偏序的总体程序执行序可能会有多种，因此我们会有如下定义：

- 顺序同一性的执行（*SC Execution*）：如果程序的一次执行产生的结果与前面定义的任何一种可能的顺序产生的结果一样，那么程序的这次执行就称为是顺序同一的
- 顺序同一性的系统（*SC System*）：如果一个系统上任何可能的执行都是顺序同一的，那么该系统就是顺序同一的

保证顺序同一性的充分条件

1. 每个进程按照程序执行序发出存储操作
2. 发出写操作后，进程要等待写的完成，才能发出它的下一操作
3. 发出读操作后，进程不仅要等待读操作的完成，还要等待产生所读数据的那个写操作的完成，才能发出它的下一操作（保证写操作的原子性）

- 注意这只是充分条件→，不是必要条件←
- 很明显，为了SC，编译器不应该改变程序呈现给硬件的操作次序，但事实是他们经常违反这个充分条件
  - 如循环拆分, 寄存器分配等
- 即使编译器没有违反，硬件有时也会违反这个条件来获取更高性能
  - 如写缓冲, 乱序执行等
- 这些优化对串行程序或单处理器是没问题的，只要求对相同存储单元的访问的相关性得到保证
  - 但对于并行多处理器环境，SC的充分条件是一个相当强的要求

在我们这次的课程讨论中，我们假设

- 编译器不改变存储操作的顺序
- 硬件必须满足充分条件，需要有:
  - 检测写操作完成的机制（检测读操作很容易）
  - 保证写操作原子性的机制
- 在我们这次课程讨论的所有协议和实现中我们将看到:
  - 他们如何满足一致性（包括写操作的串行化）
  - 他们如何满足顺序同一性 (检测写完成，保证写原子性)
  - 以及他们如何简化顺序同一性充分条件的处理
- 基于总线的机器，共享总线所带来的必然的操作串行化对存储器操作的定序是很有用的，让问题变简单

例如我们前面提到的写穿透一致性协议

- 不仅仅保证了一致性，还保证了SC
- 我们在一致性的基础上扩充讨论
  - 对所有存储单元的读或写扑空，都在总线上串行化了
  - 当读操作得到了一个写操作的值，那个写操作肯定就已经完成了
    - 因为它引起了一个总线事务
  - 当任何一个处理器进行一个写操作的时候，所有先前的写操作已经以它们访问总线的次序完成了

总线侦听协议的设计空间

- 无需修改处理器、主存、高速缓存
  - 只需要扩展高速缓存控制器并利用总线的性质
- 我们之前基于写穿透的设计不是很高效，为了提高效率，我们下来着重关注基于写返回（**write-back**）的高速缓存一致性协议
- 单处理器下，**cache**块的**Dirty state** 现在会被用来标志一个高速缓存独立的拥有某一存储块的一个拷贝（**exclusive ownership**）
  - Exclusive: only cache with a valid copy (main memory may be too)
  - Owner: responsible for supplying block upon a request for it
- 设计空间
  - 基于作废的协议 vs. 基于更新的协议
  - 状态转换图和相关动作

写返回作废式一致性协议

- **独占（Exclusive）意味着可以对该存储块进行修改而不用通知其他的处理器的高速缓存**
  - 例如，不需要发出总线事务
  - 必须首先使块进入独占态，再修改它
  - 即使做写操作的这个高速缓存块是有效的，也要产生一个总线事务，也叫写扑空，就像要写入一个不能存在或者无效的块
- **写入一个非dirty的块会产生一个特殊的总线事务：排他读（*read-exclusive*）**
  - 告知其他高速缓存即将发生的写, 获得该存储块的独占权
  - 一次只有一个能获得对该块的独占权: 被总线串行化了
- **高速缓存一致性的动作由读和排他读驱动**
  - 写返回事件也可以驱动，但它不是由存储操作引起的（cache替换），对协议存在随机性

写返回更新式一致性协议

- **写操作同时会更新其他cache中的块**
  - 加入一个新的总线事务：更新（Update）
- **优势：**
  - 其他处理器接着访问这一存储块时，直接从cache拿到: 降低延迟
    - 在作废式协议里，他们会cache miss，进而产生更多的总线事务
  - 通过一个总线事务就更新所有相关的cache块，节省总线带宽
    - 仅仅传输所写的数据（word or byte），而不是整个cache块
- **劣势**
  - 多个写操作会引发多次总线上的更新事务
    - 在作废式协议里，第一次写操作就会获得该存储块的独占权，后续写操作都是本地操作
- **还有其他更为细节的权衡，会更加复杂**

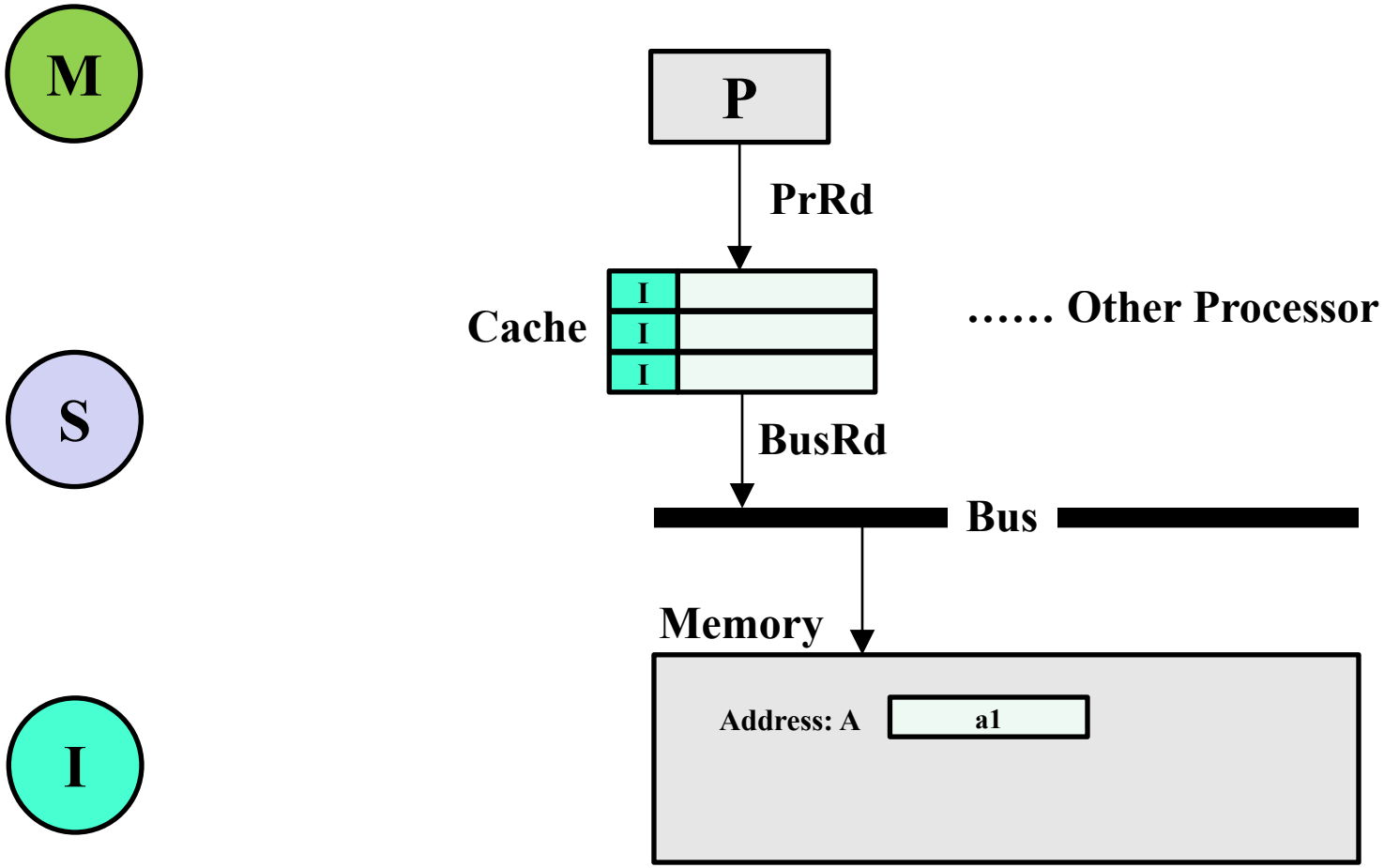
如何在两种协议中选择

- **程序行为的基本问题**
  - 一个被某处理器写过的存储块（cache中），在被该处理器重写之前，是否会被其他处理器读？
- **Invalidation:**
  - Yes => 读数据的处理器产生cache miss（代价）
  - No => 多次写不会产生额外的总线通信数据
    - 而且还清除掉了中间过程的不会被用的数据拷贝
- **Update:**
  - Yes => 读数据的处理器不会产生cache miss
    - 单次总线事务就更新了所有相关的cache块
  - No => 产生多次无用的更新, 甚至对一些已经完全不用块的更新（代价）
- **需要基于程序行为和硬件复杂度来抉择**
- **通常情况，基于作废的协议更加流行(more later)**
  - 有些系统会同时提供两种协议，或两者混合的协议

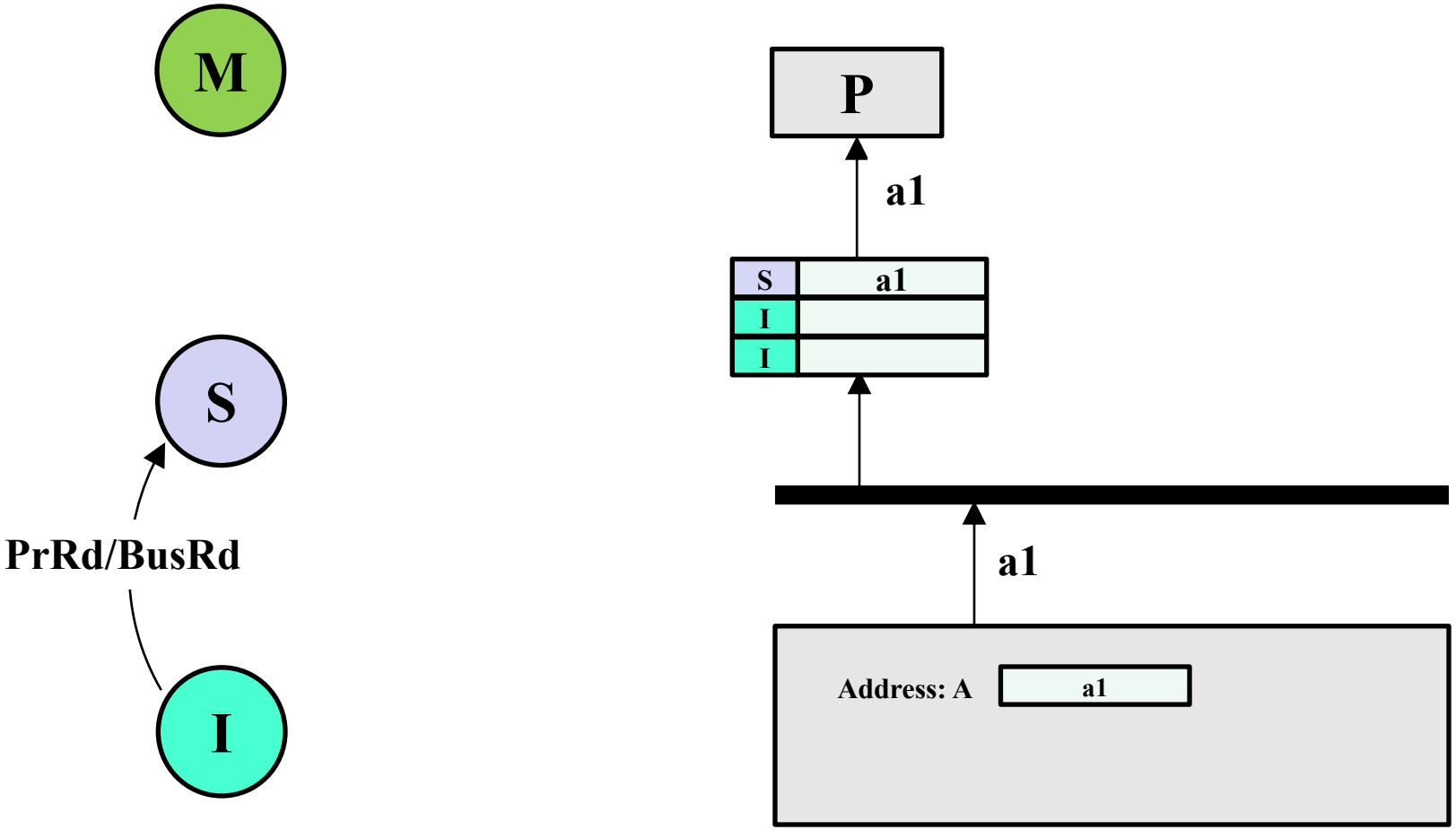
一种三态（MSI）写回作废式协议

- **Cache块的状态（States）**
  - Invalid (I)：无效
  - Shared (S): 被一个或多个共享
  - Dirty or Modified (M): 被独占
- **处理器事件（Events）：**
  - PrRd (read)
  - PrWr (write)
- **总线事务（Bus Transactions）**
  - BusRd: 由处理器Cache扑空的PrRd产生，向存储系统请求一个不会修改的拷贝
  - BusRdX: 由处理器PrWr产生，请求一个要修改的排他拷贝
  - BusWB: 由高速缓存控制器写回时产生，处理器不知道它的发生
- **动作（Actions）**
  - Update state, perform bus transaction, flush value onto bus

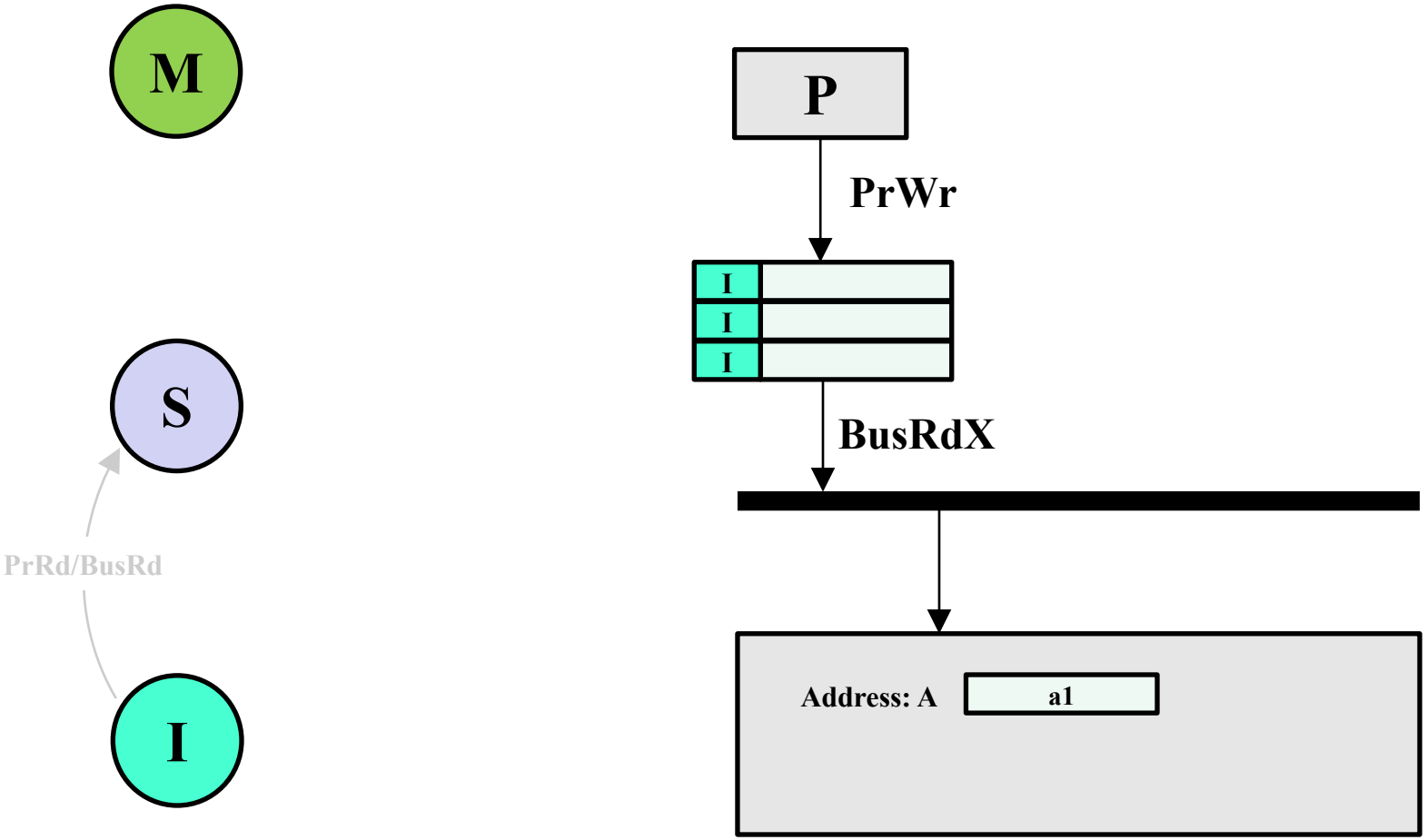
该协议的状态转换图


M
S
I
P
PrRd
Cache
I
I
I
…… Other Processor
BusRd
Bus
Memory
Address: A
a1

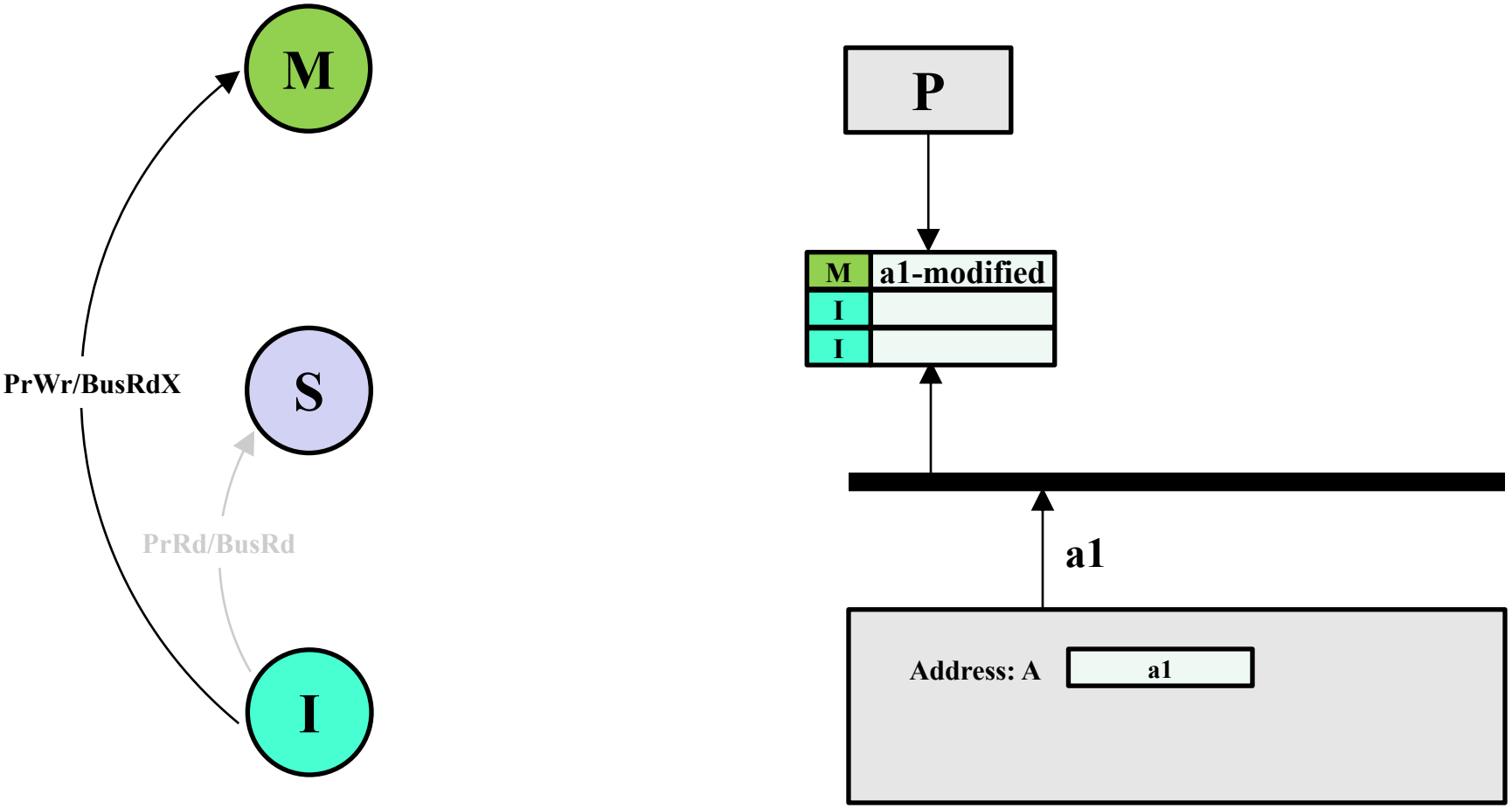
该协议的状态转换图



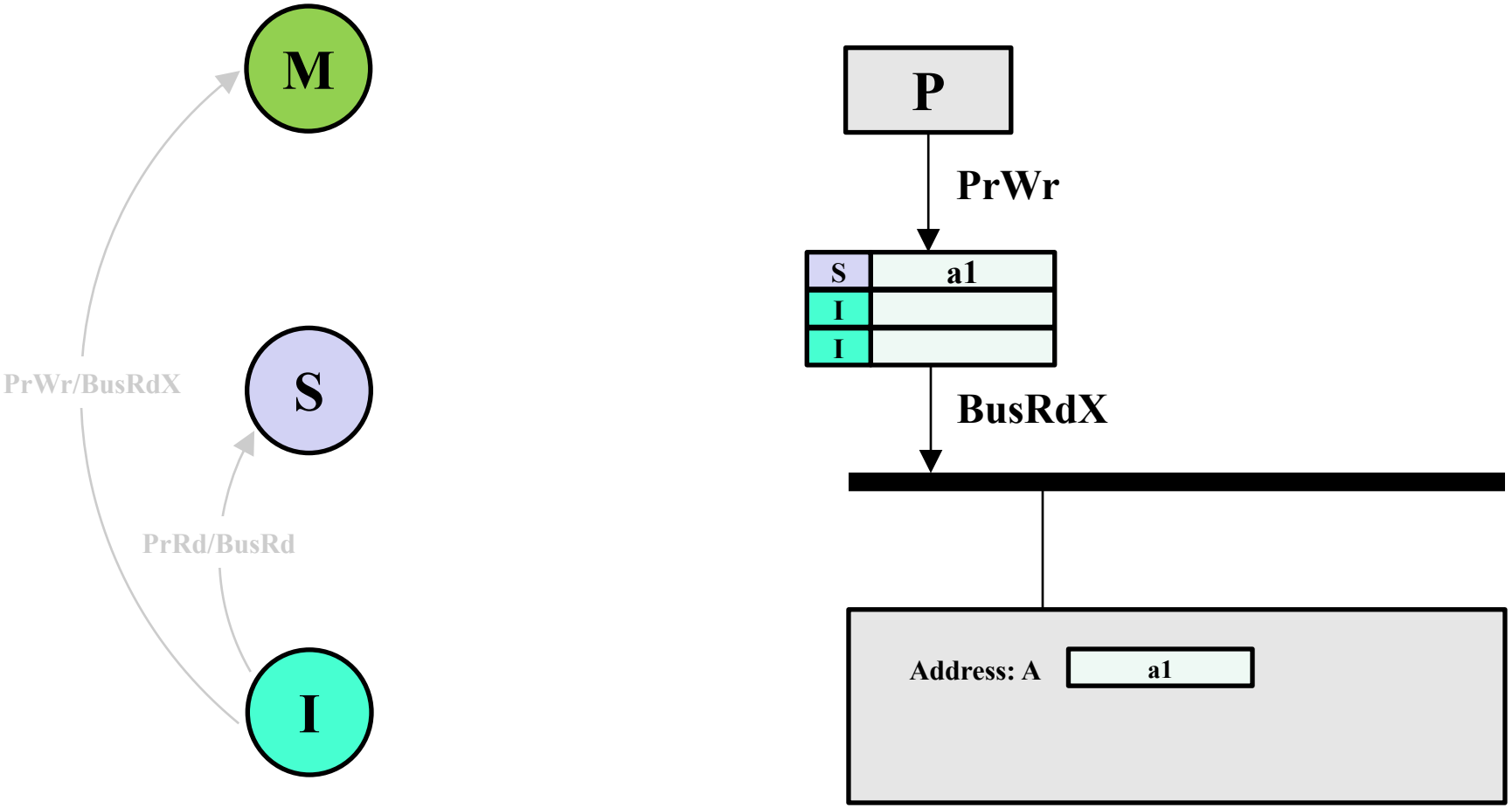
该协议的状态转换图



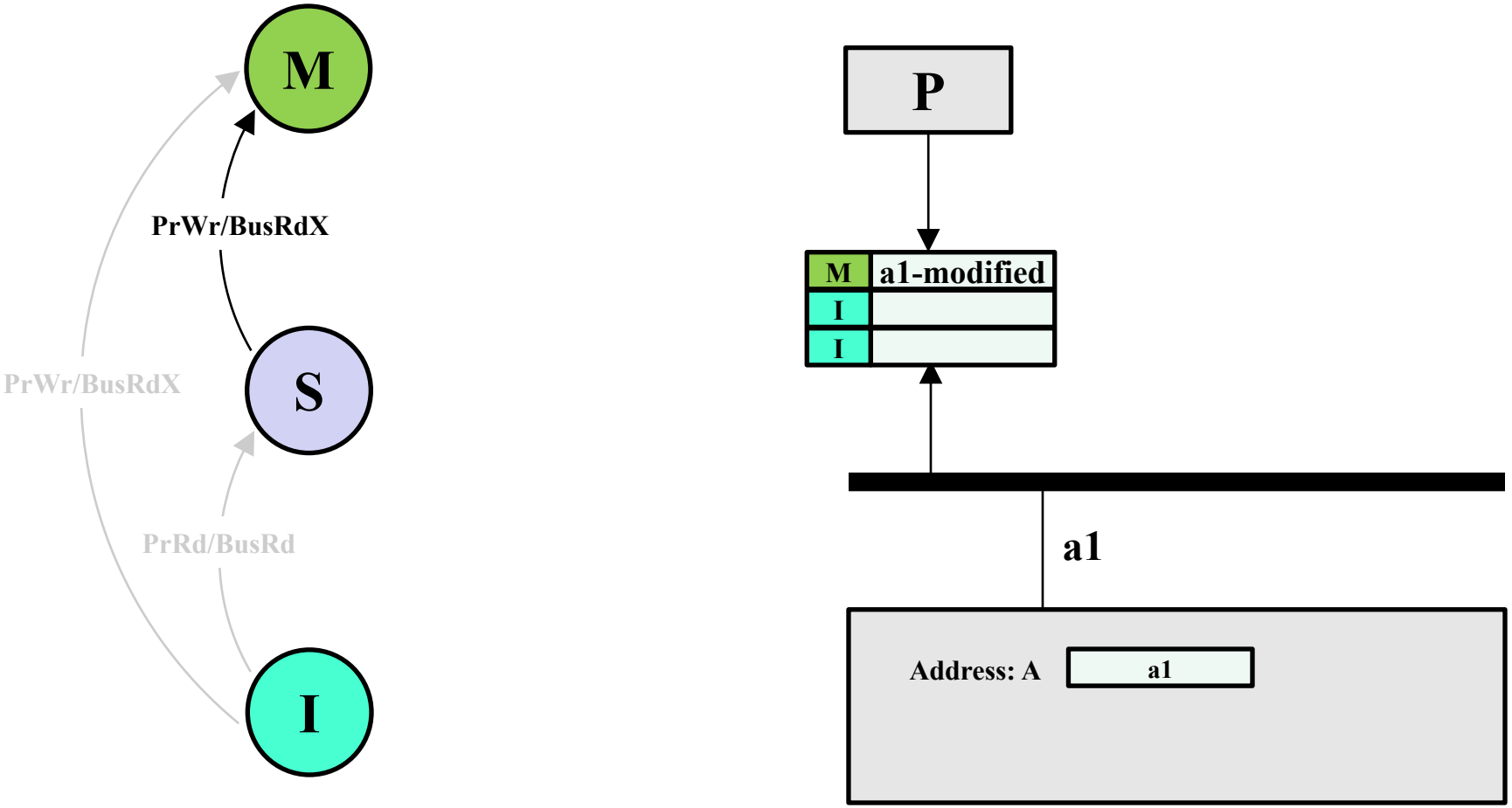
该协议的状态转换图



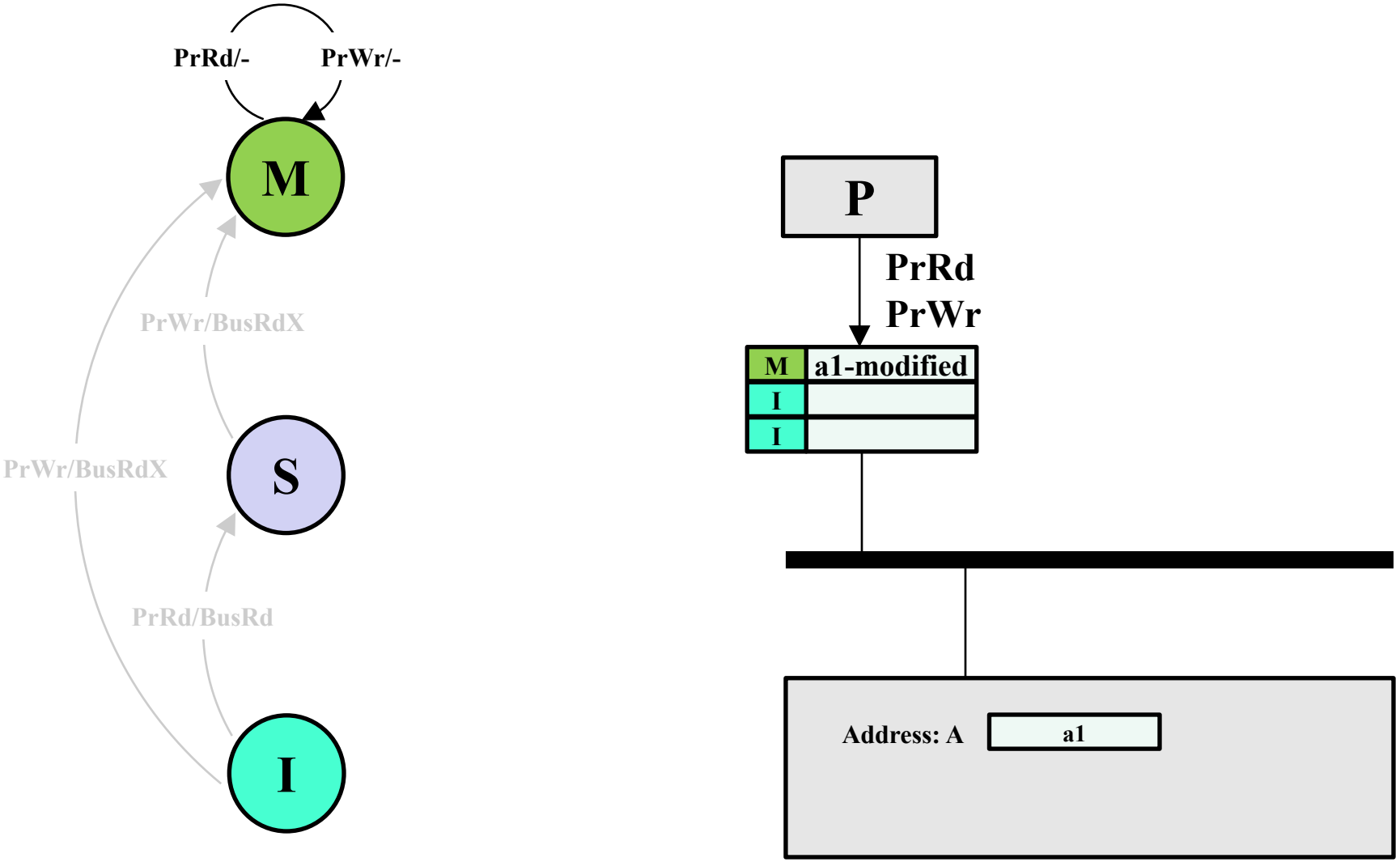
该协议的状态转换图



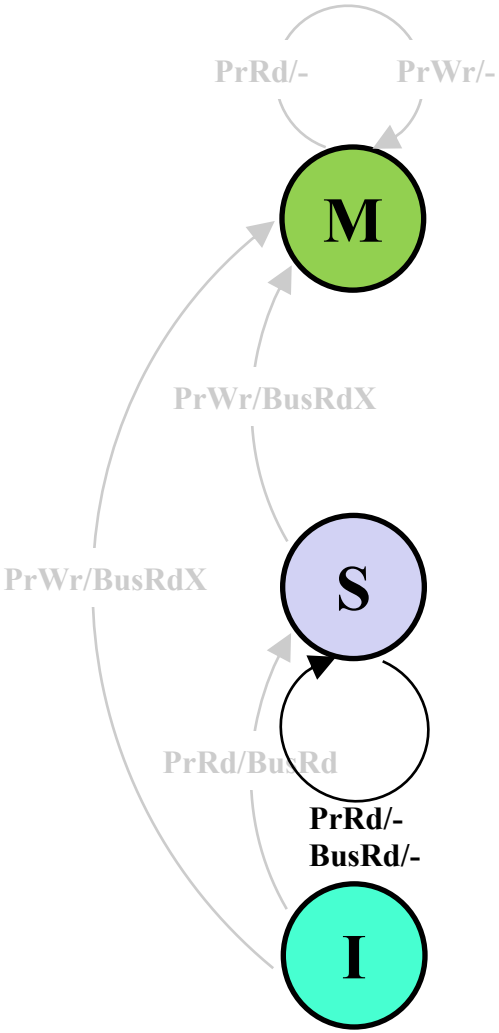
该协议的状态转换图

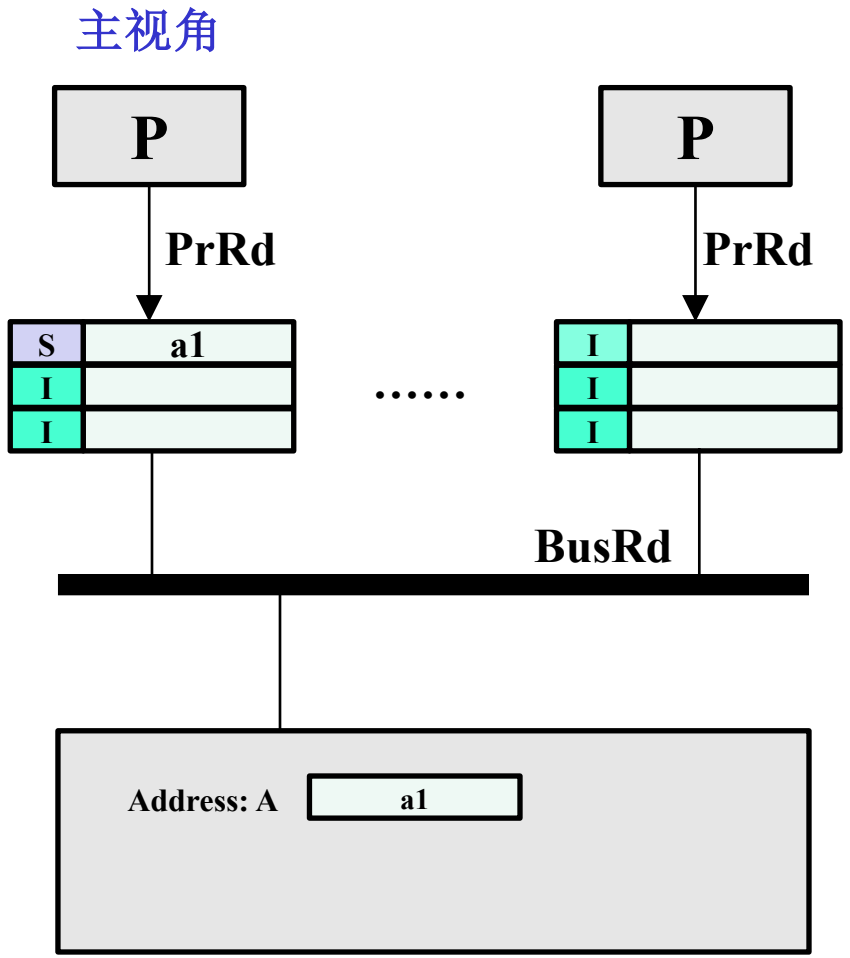

M
PrWr/BusRdX
PrWr/BusRdX
S
PrRd/BusRd
I
P
M
a1-modified
I
I
a1
Address: A
a1

该协议的状态转换图

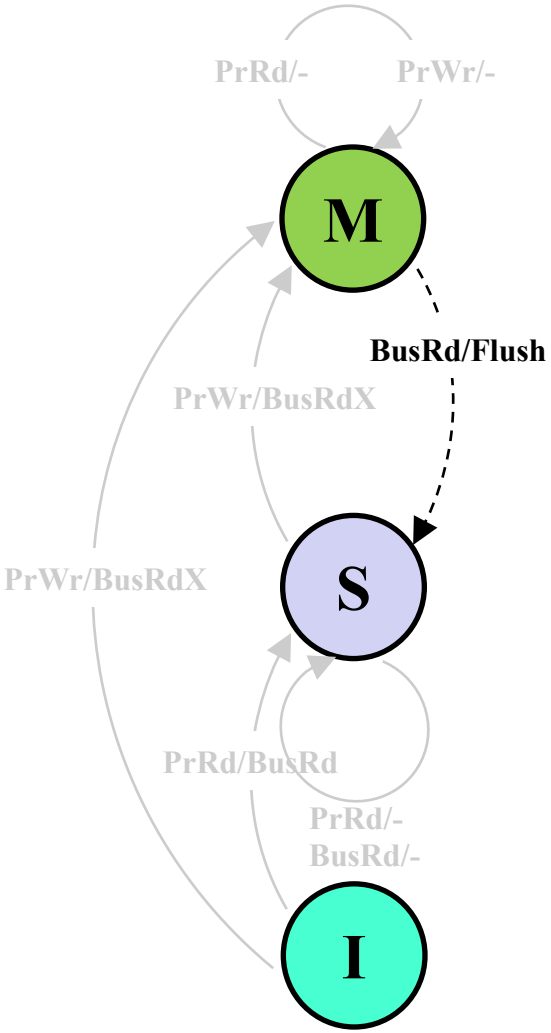

PrRd/-
PrWr/-
M
PrWr/BusRdX
PrWr/BusRdX
S
PrRd/BusRd
I
P
PrRd
PrWr
M
a1-modified
I
I
Address: A
a1

该协议的状态转换图

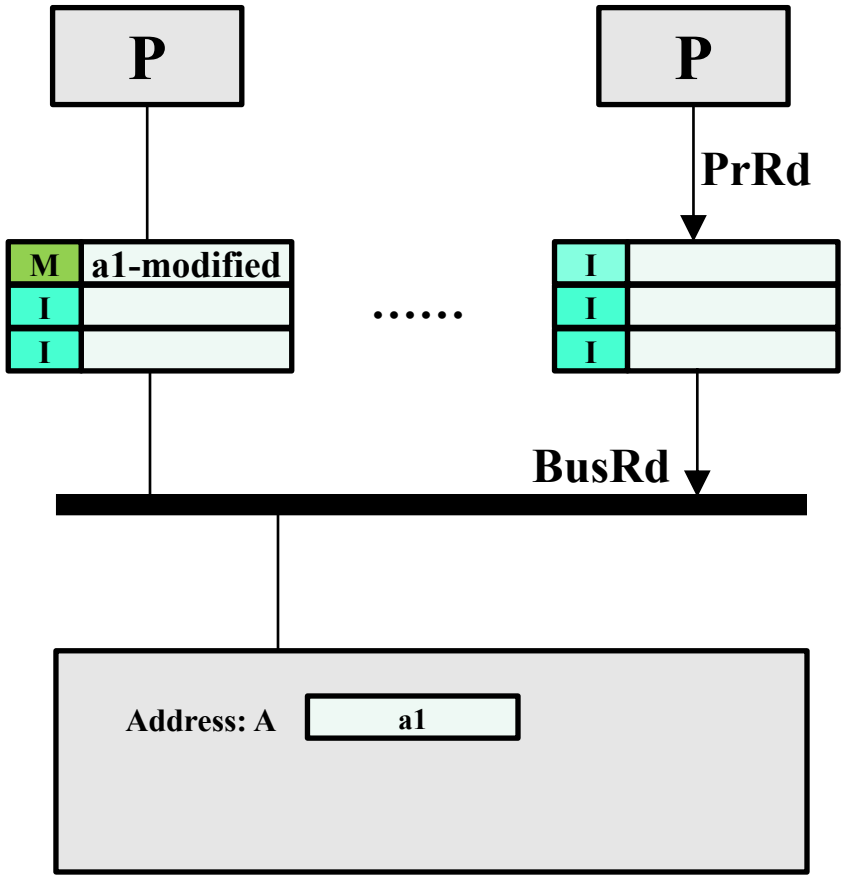

PrRd/-
PrWr/-
M
PrWr/BusRdX
PrWr/BusRdX
S
PrRd/BusRd
PrRd/-
BusRd/-
I


主视角
P
P
PrRd
PrRd
S
a1
I
I
I
I
I
I
.....
BusRd
Address: A
a1

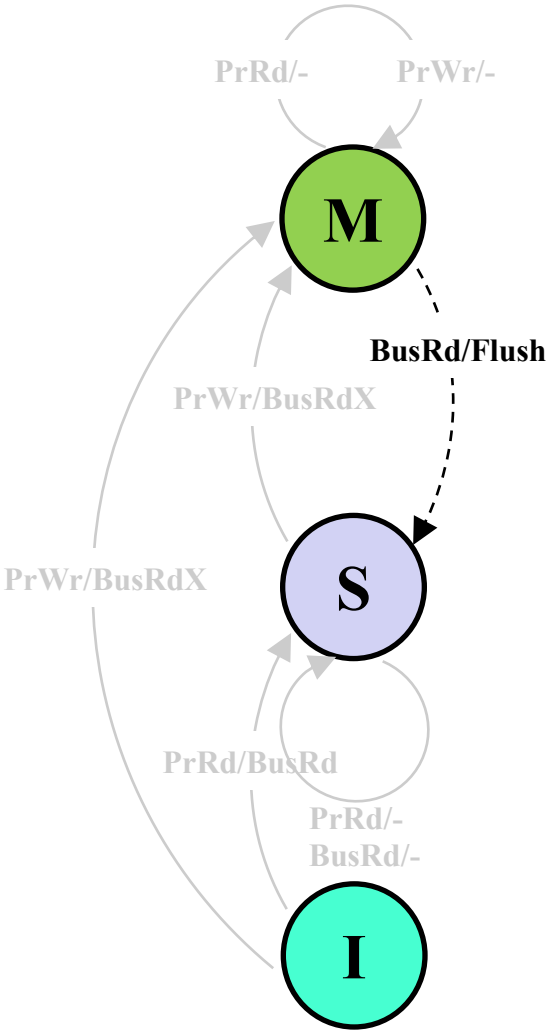
该协议的状态转换图



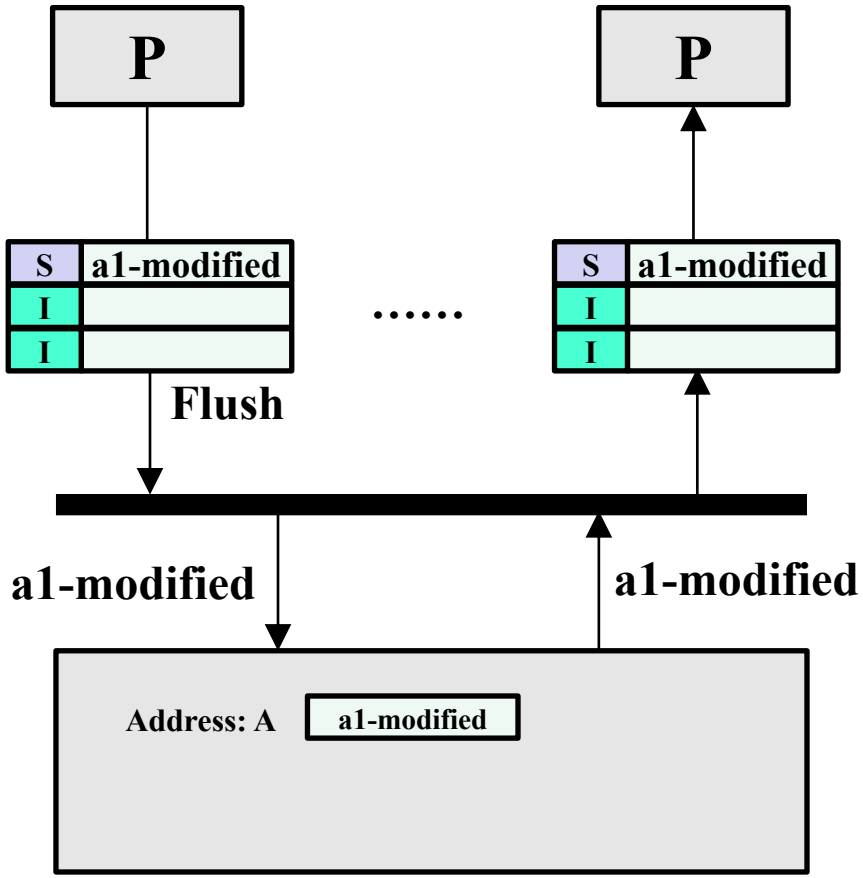
主视角



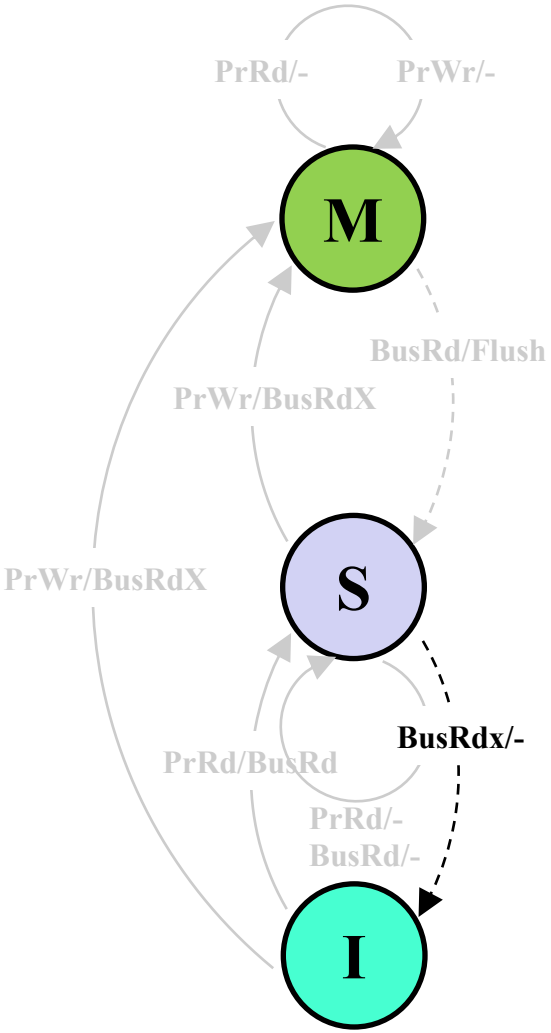
该协议的状态转换图

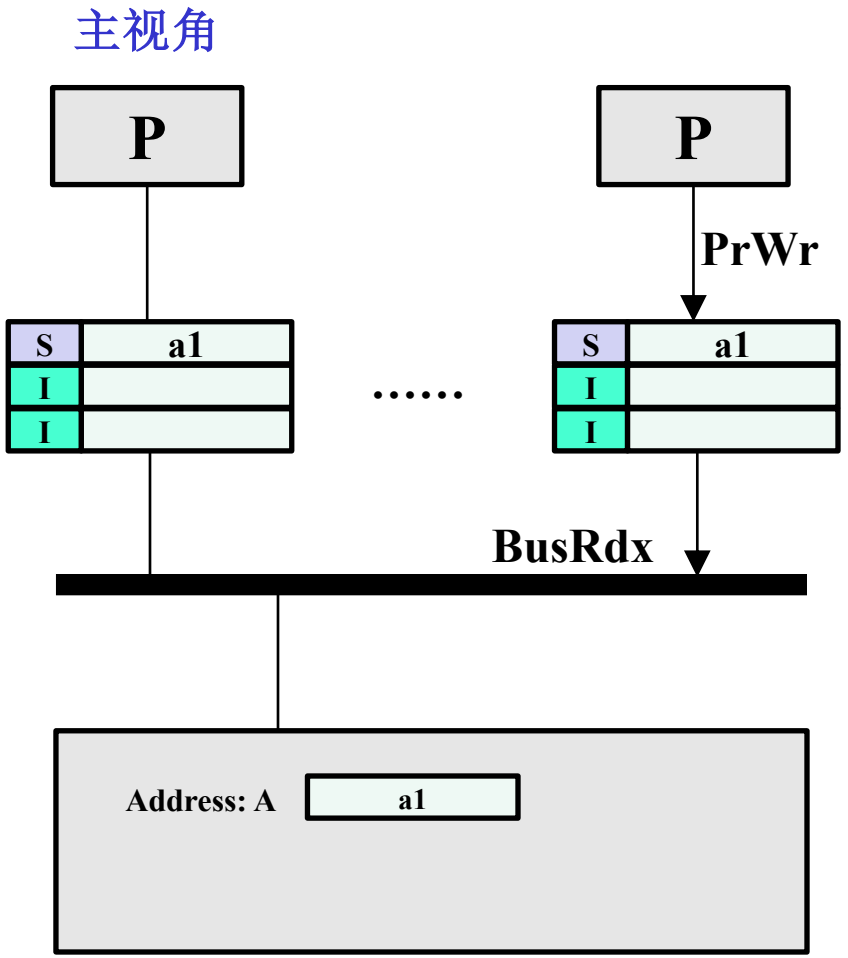


主视角

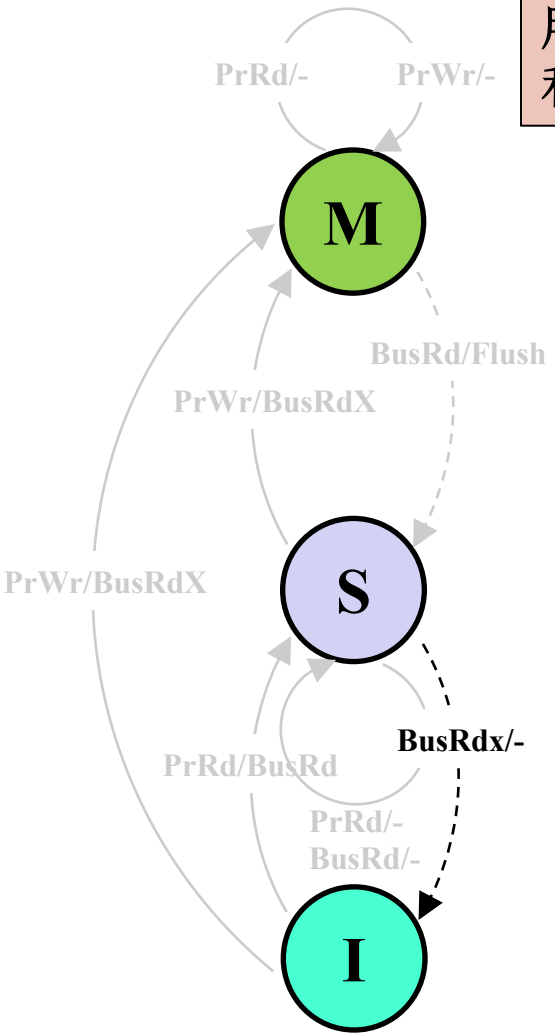


该协议的状态转换图

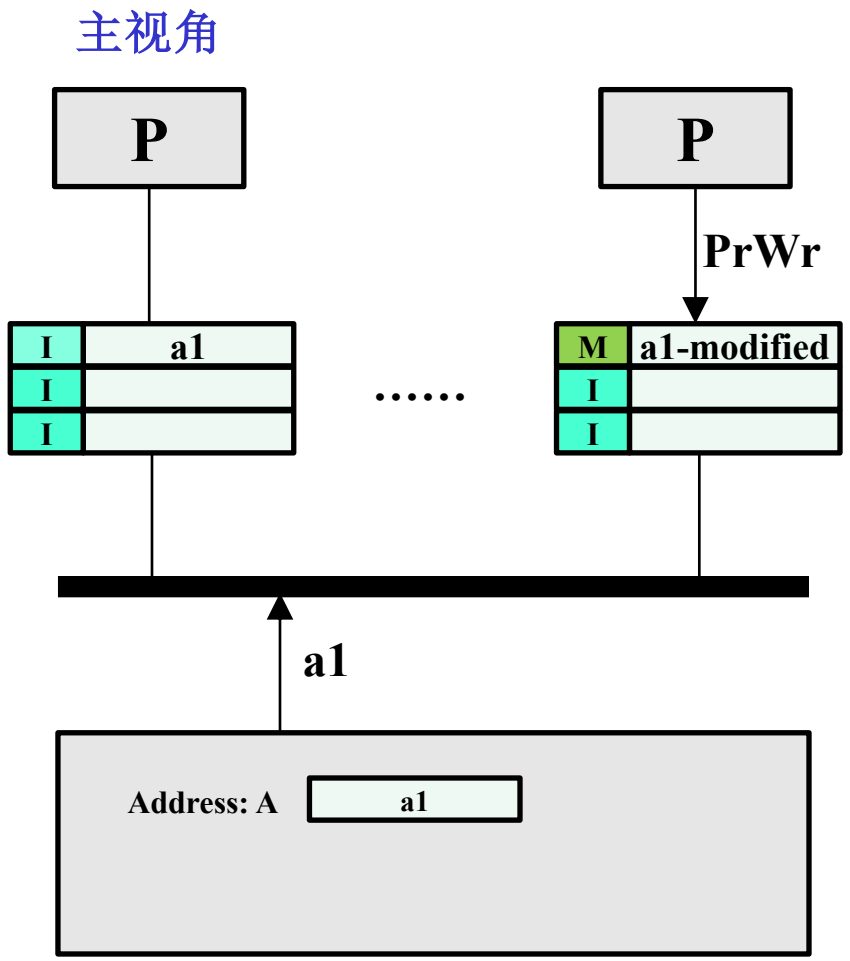

PrRd/-
PrWr/-
M
BusRd/Flush
PrWr/BusRdX
PrWr/BusRdX
S
BusRdx/-
PrRd/BusRd
PrRd/-
BusRd/-
I


主视角
P
P
PrWr
S
a1
I
I
.....
S
a1
I
I
BusRdx
Address: A
a1

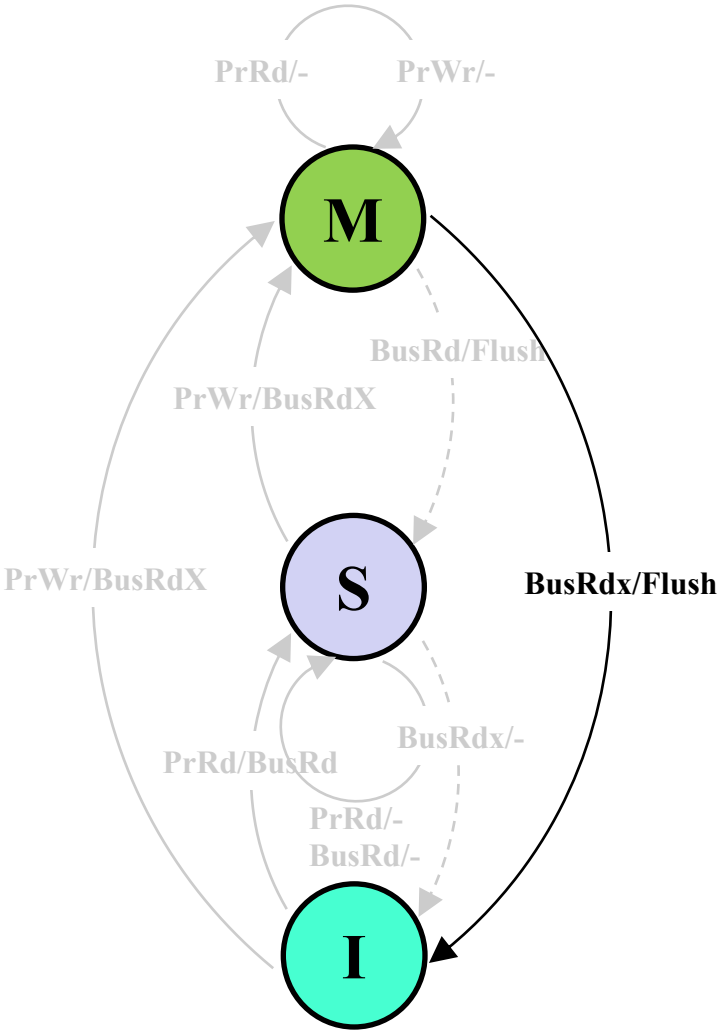
该协议的状态转换图

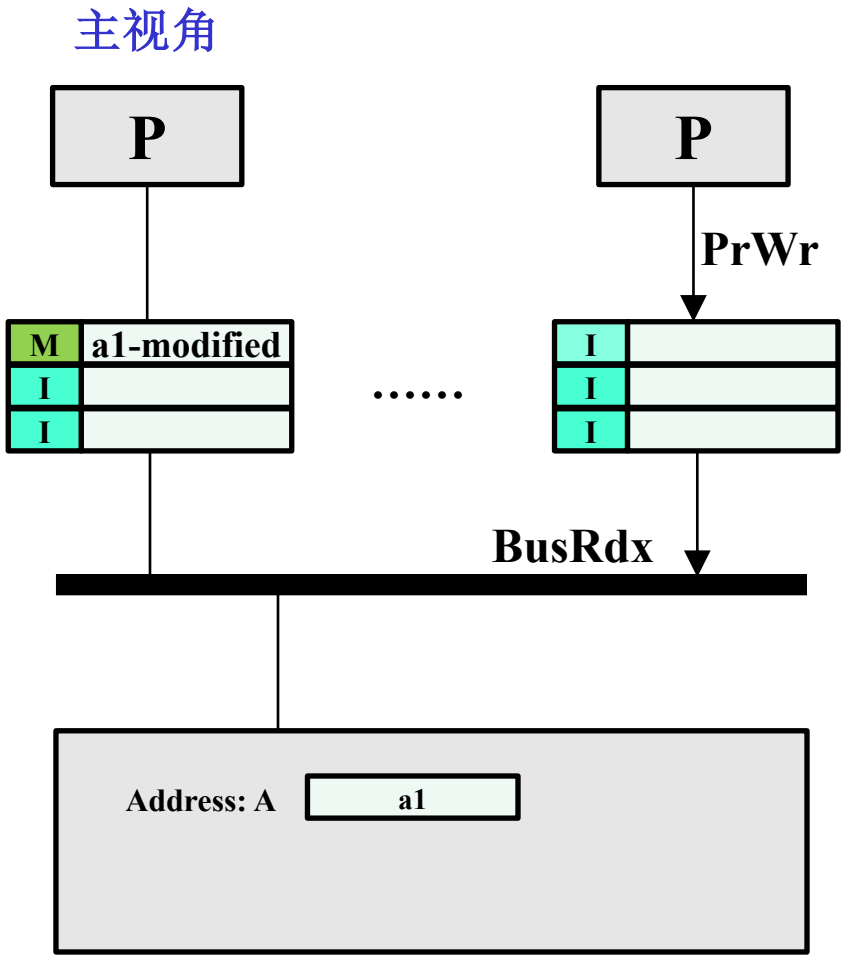


用BusUpgr替代BusRdX能够防止内存提供数据，节省带宽，和后面的更新式协议有点类似

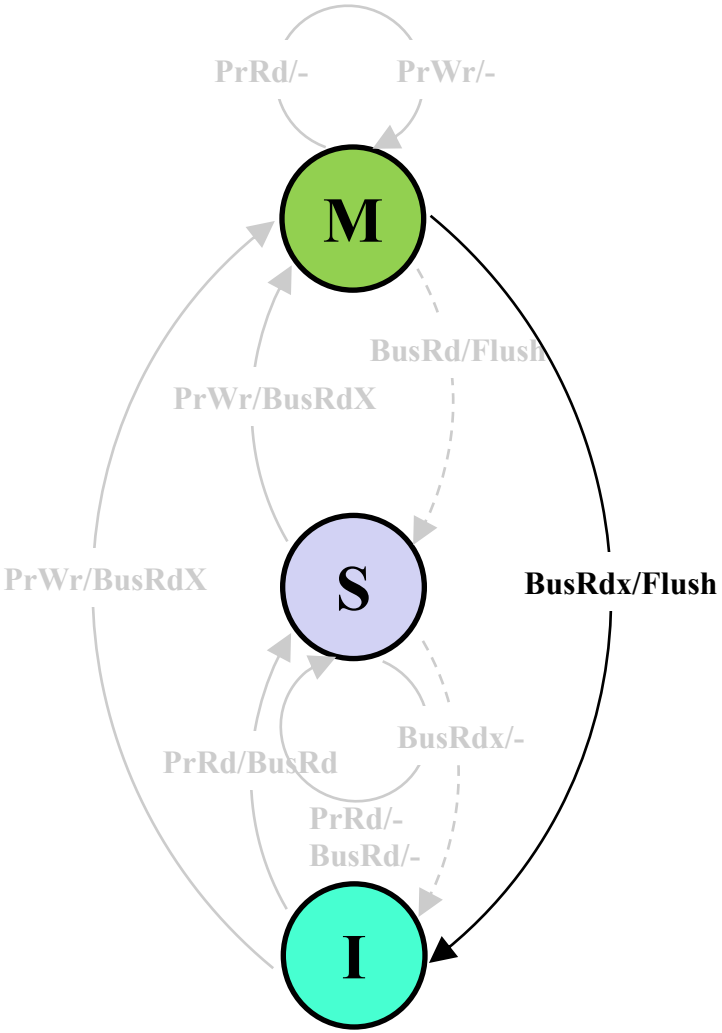


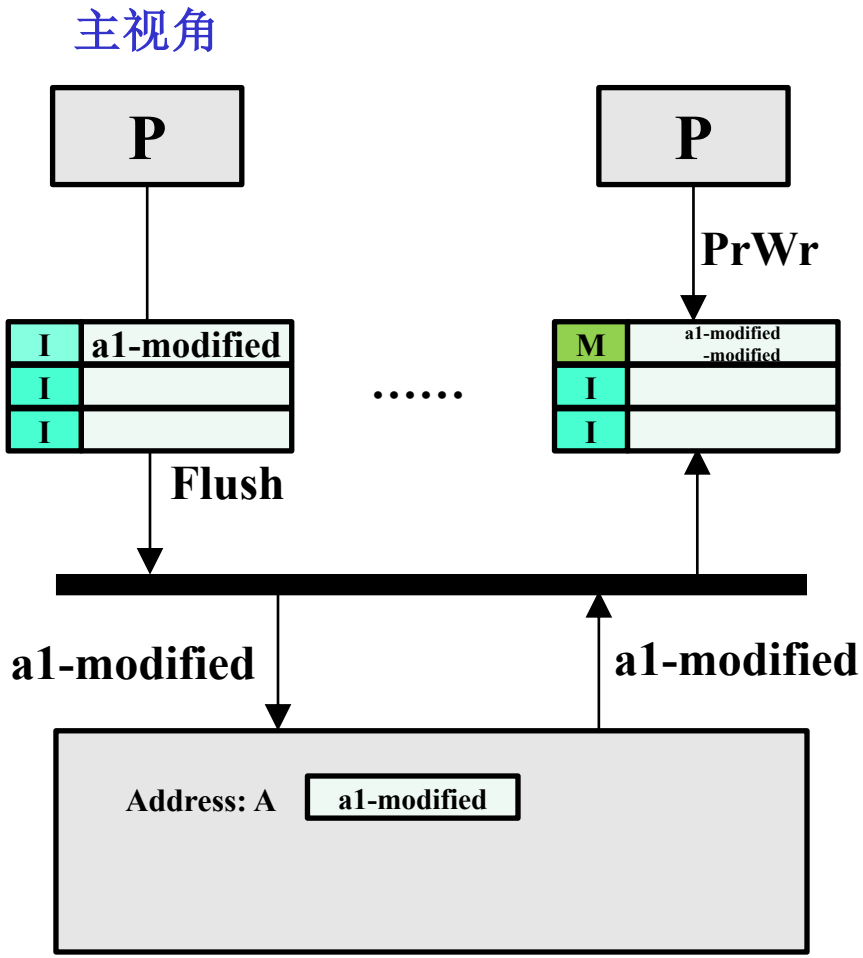
该协议的状态转换图


PrRd/-
PrWr/-
M
BusRd/Flush
PrWr/BusRdX
PrWr/BusRdX
S
BusRdx/Flush
BusRdx/-
PrRd/BusRd
PrRd/-
BusRd/-
I


主视角
P
P
PrWr
M
a1-modified
I
I
I
I
I
I
.....
BusRdx
Address: A
a1

该协议的状态转换图


PrRd/-
PrWr/-
M
BusRd/Flush
PrWr/BusRdX
PrWr/BusRdX
S
BusRdx/Flush
BusRdx/-
PrRd/BusRd
PrRd/-
BusRd/-
I


主视角
P
P
PrWr
I
a1-modified
I
I
.....
M
a1-modified
-modified
I
I
Flush
a1-modified
a1-modified
Address: A
a1-modified

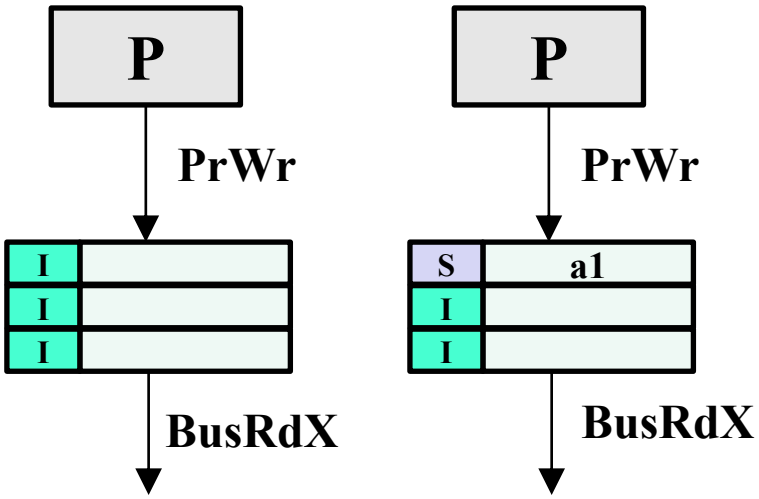
三态写回作废式协议对一致性的保证

- 回顾一致性的形式化定义
  - 写传播（*Write propagation*）：写操作的效果对其他进程可见
  - 写串行化（*Write serialization*）：对于某个单元的所有写操作（来自相同或者不同的进程）而言，所有进程都是以相同顺序看到这些操作

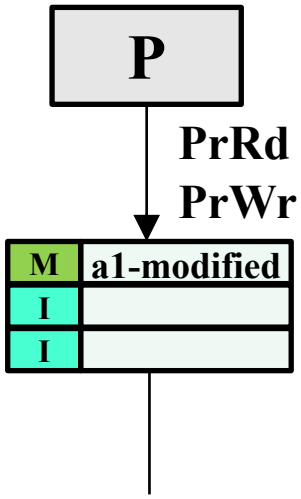
- 写传播特性的满足是显而易见的（**why？**）
  - 对Invalid和Shared态的Wr：产生BusRdX
  - 对Modified态的Wr：不产生总线事务，但是其他处理器的读会产生Flush或BusWB事务

三态写回作废式协议对一致性的保证

- 我们主要考察写串行化是否满足?



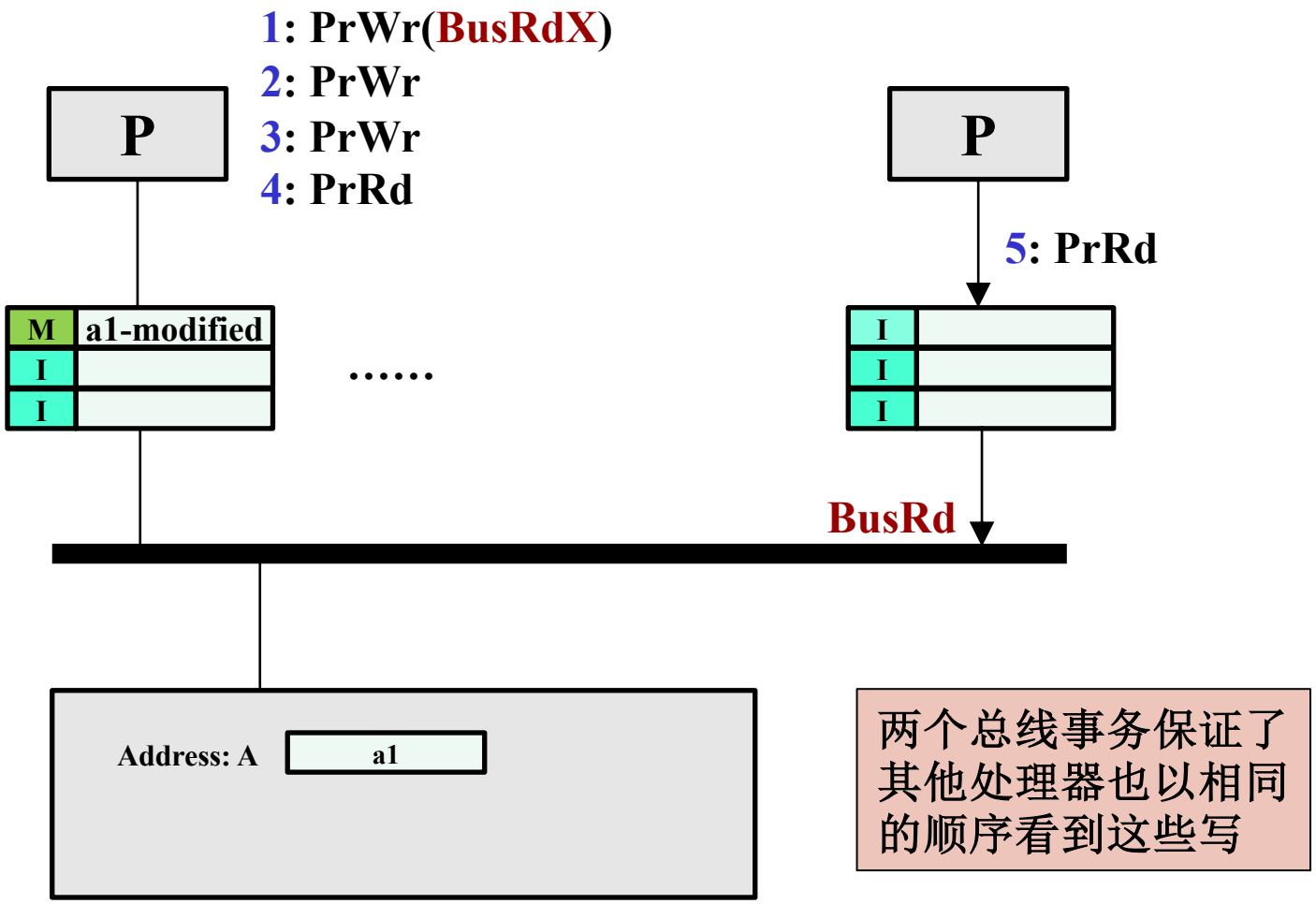
显而易见的写串行化



不那么显而易见

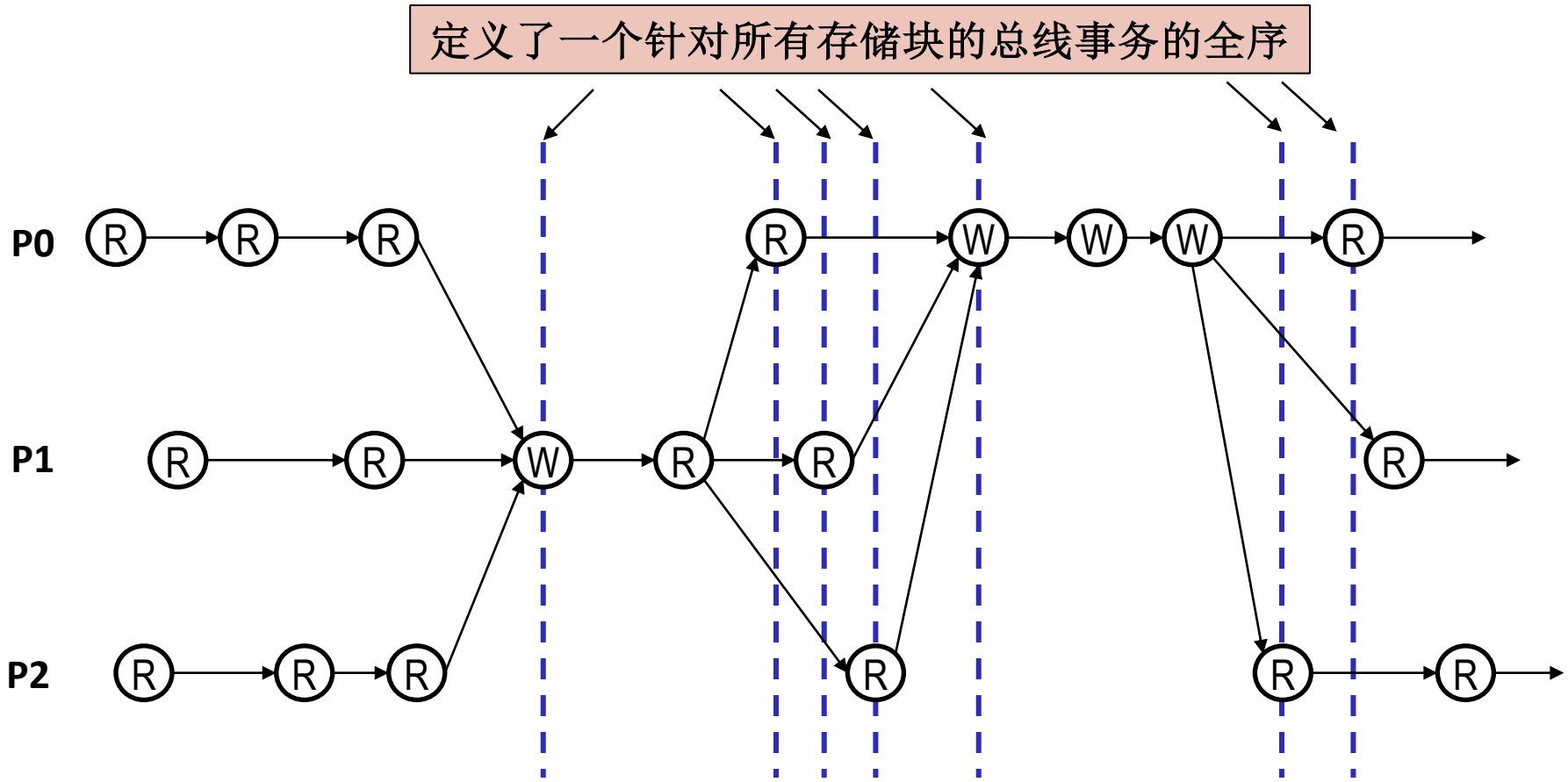
三态写回作废式协议对一致性的保证

- 我们主要考察写串行化是否满足?



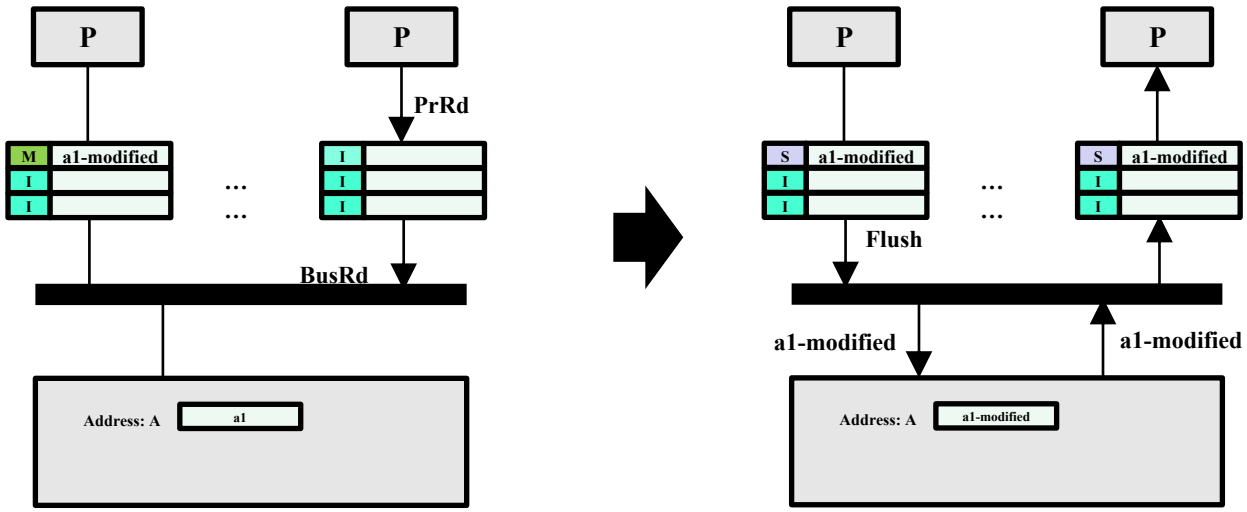
三态写回式作废协议对顺序同一性的保证

- 基于原单处理器上的序，构造一套全局交叉序
- 写操作的完成检测，写操作的原子性保证？



如果一个读（非local处理器）返回一个写的值，那么该写已经对其他所有处理器可见了

上述协议在实现时（低层）的设计抉择

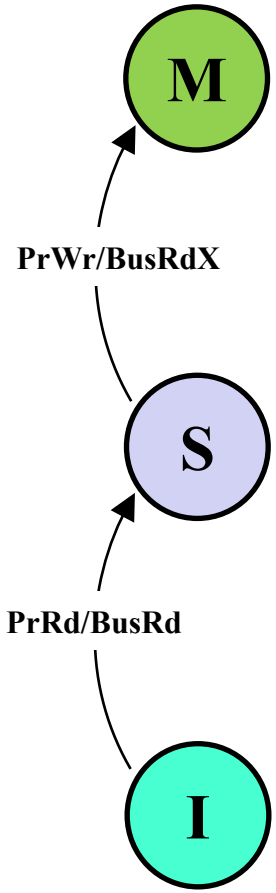


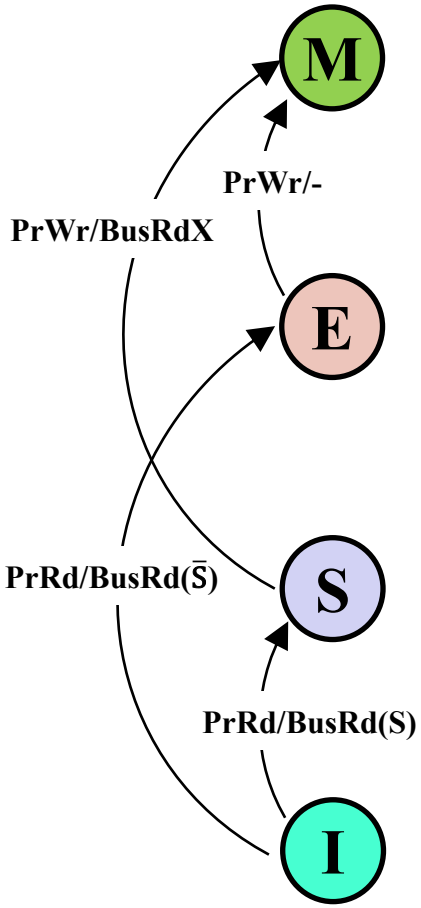
- **选择转移到S态: 持有该块的处理器更可能继续读这个块**
  - 对于频繁读这个数据的访问模式是好的
  - 但对于进程之间来回传递信息的一个标记或缓冲区，就不好
    - 一个处理器写入、另一个处理器读出且修改，然后第一个处理器读并修改...
    - 这样的访问模式下，更好的选择是迁移到I态，另一个处理器修改时，第一个处理器就不用再Invalidate
  - 某些机器（Sequent Symmetry and MIT Alewife）会在不同的访存模式被观察到时对协议做动态调整

一种四态（MESI）写回作废式协议

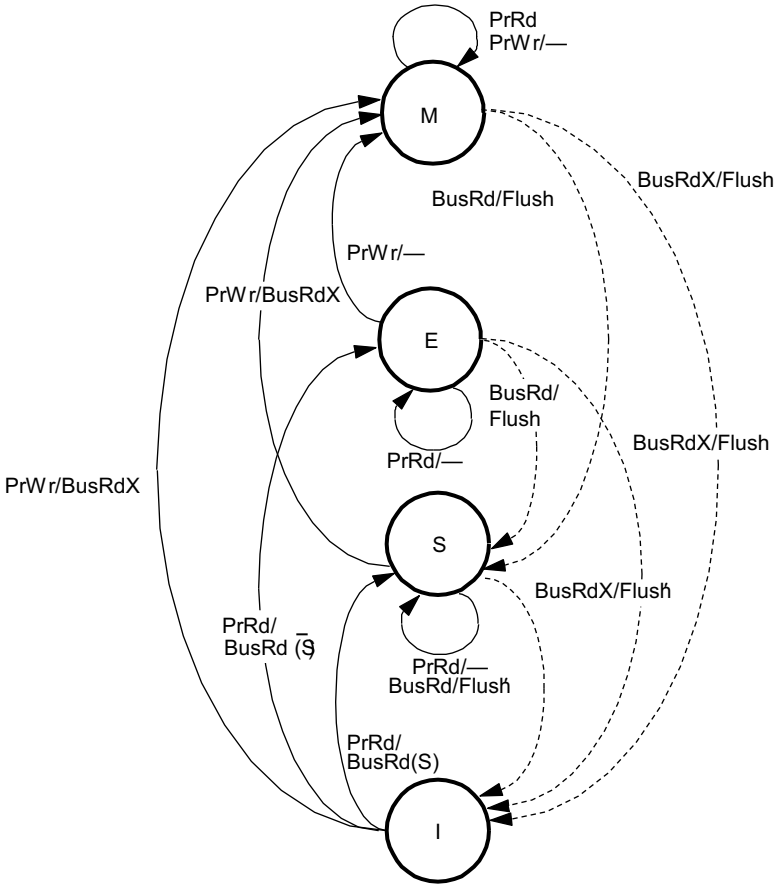
问题：即使没有其他处理器来共享这个数据，也要产生两个总线事务

解决方法：
（1）增加非拥有的独占态exclusive state
（2）在总线上增加一个共享信号（S）





MESI 协议的状态转换图



- BusRd(S) 表示 BusRd时共享信号线拉高了
- Flush’: 如果cache之间共享数据, 只需要相关的cache flush数据
- MOESI 协议: Owned state: exclusive but memory not valid， 需要在观察到相关总线事务时提供数据

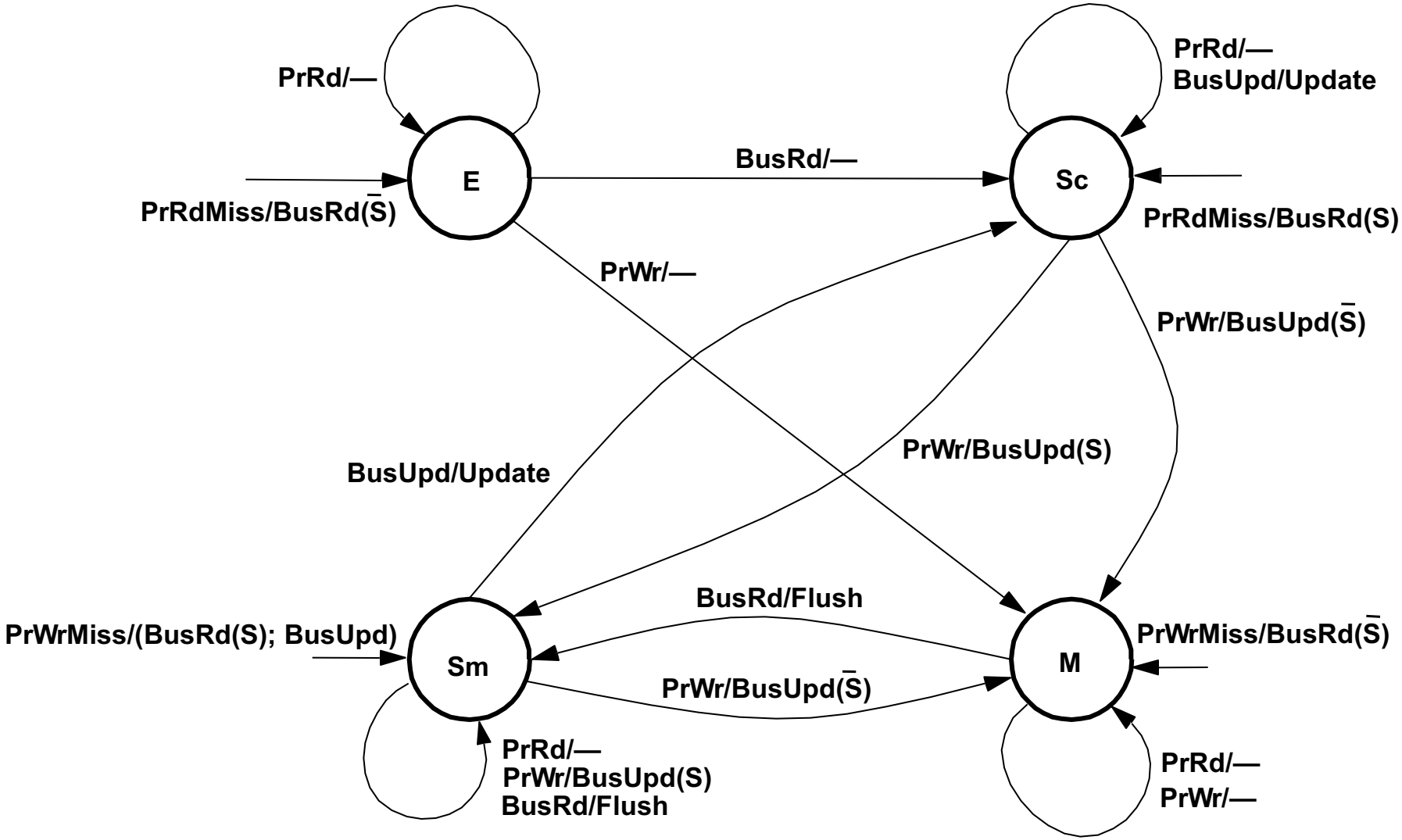
MESI协议在实现时（低层）的设计抉择

- 当发生一个**BusRd**事务时，如果存储器和某个缓存都有拷贝，谁应该提供数据？
- 需要根据情况选择
  - 由缓存提供会增加复杂度：需要缓存到缓存共享的技术（Cache-to-cache sharing），增加了总线分复杂性
    - 主存必须等待，直到它能肯定没有缓存提供数据后才能驱动总线
    - 如果多个缓存都有这个数据，就要一个选择算法
    - 在最原始版本（*Illinois* MESI）中: 由缓存提供
  - 但是缓存提供适配更多场景：
    - 这个技术对于物理上分布存储的多处理器系统是有用的
    - 因为从近处的缓存获取数据要比从远处的存储器快得多
    - 尤其是由SMP节点构成的网络型机器中(Stanford DASH)

一种四态（Dragon）写回更新式协议

- 最初由Xerox PARC的研究人员提出，用在他们的Dragon多处理器系统中，增强版本用在Sun SparcServer多处理器系统中
- 4 states
  - Exclusive-clean or exclusive (E): I and memory have it
  - Shared clean (Sc): I, others, and maybe memory, but I'm not owner
  - Shared modified (Sm): I and others but not memory, and I'm the owner
    - 一个存储块一个时间只能在一个缓存里是Sm状态，可以在其他缓存块中是Sc状态
  - Modified or dirty (D): I and, none else
- 没有无效状态（总是保持缓存中块的内容是最新的）
  - 如果在cache中，直接使用该数据
  - 如果不在cache中，可以被想象为一种特别的无效或者不存在状态
- 由于没有无效状态，需要加两种处理器请求: PrRdMiss, PrWrMiss
  - 说明当前cache匹配不成功时的动作
- 增加新的总线事务: BusUpd 总线更新
  - 将处理器P写的特定的字或者字节广播到总线上，从而所有其他处理器的缓存能更新它们自己

Dragon协议的状态转换图



上述协议在实现时（低层）的设计抉择

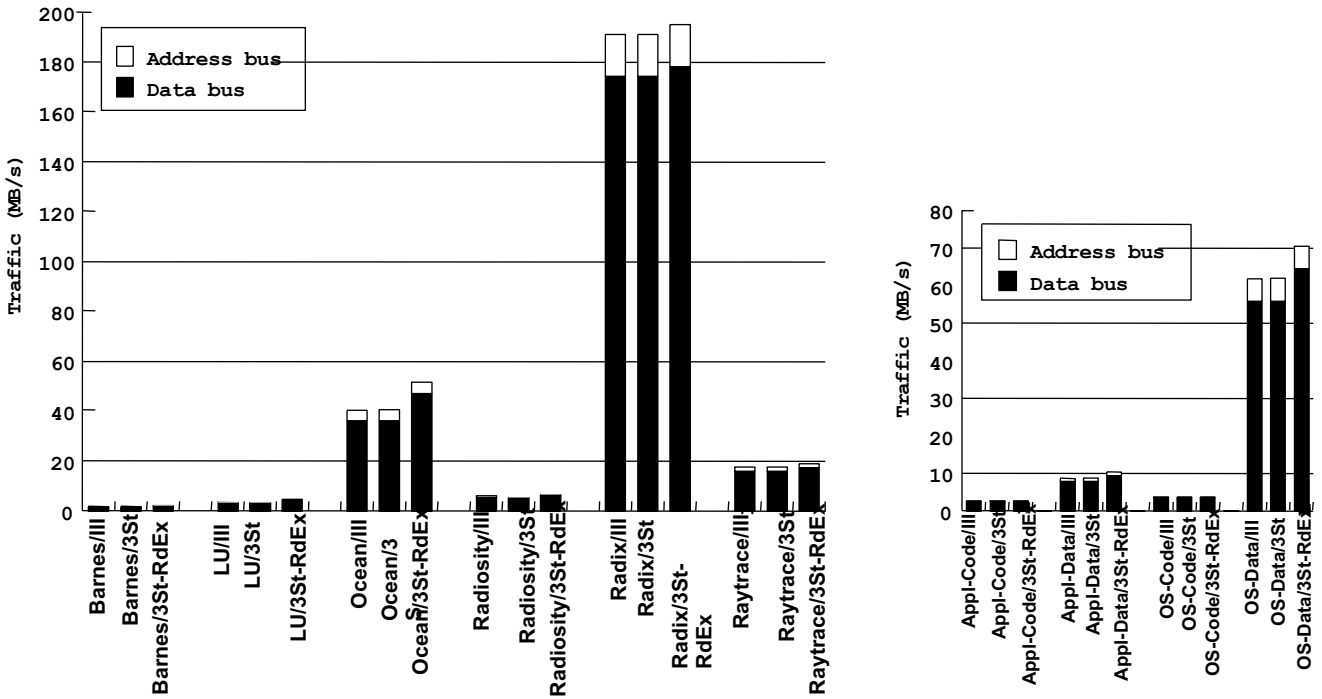
- 共享已修改状态是否可以去掉？
  - 每当BusUpd事务发生时，和其他持有该存储块的缓存一道，主存也可以更新它的内容 (DEC Firefly机器就是这么做的)
  - 而Dragon协议是基于这样的假设：主存更慢，等待主存更新不合适
- 一个干净的共享存储块被替换时，是否应该通过一个总线事务让其他缓存得知这个替换？
  - 是的理由：如果只有一个缓存持有拷贝，就可以将其状态修改为独享或者已修改的
  - 好处是这个由替换引起的总线事务可能不在一次存储操作的关键路径上，而它所省下来的后来的总线事务可能就在关键路径上
- 在缓存控制器获得总线操作权之前，本地的写命中不应该更新本地的拷贝，否则会破坏串行化
- 一致性、顺序同一性与写穿透情况类似

不同协议之间的评估

- 协议特性的权衡取决于性能和机器体系结构参数的要求
- 在真实系统设计中做出决定，部分是艺术部分是科学
  - 艺术: 经验、直觉和设计者的审美观
  - 科学: 基于工作负载驱动的评估
    - 设计目标通常是满足代价和性能指标，达到一个平衡的系统
- 方法学:
  - 使用模拟器; 选择一组参数 (default 1MB, 4-way cache, 64-byte block, 16 processors; 64K cache for some)
  - 重要的是分析各种事件发生的频率，而不是它们所画的绝对时间
    - 不依赖于具体的系统实现和工艺假设
  - 使用理想化的存储系统性能模型，以避免跨处理器的引用交织与机器参数的变化
    - Cheap simulation: 不涉及竞争，用简单的PRAM模型，所有存储操作的完成时间都假定是相同的，无论它们是命中还是不命中缓存。

评估举例：协议优化的效果

- **MSI vs. MESI，E状态对带宽的节省效果，使用BusUpgr 替代 BusRdX**
  - 计算状态转换过程中的通信开销
  - 左边是并行程序的数据，右边是Multiprog模拟得到的数据
  - 每一组中最左边是MESI，中间是MSI BusUpgr，最右边是MSI使用BusRdX

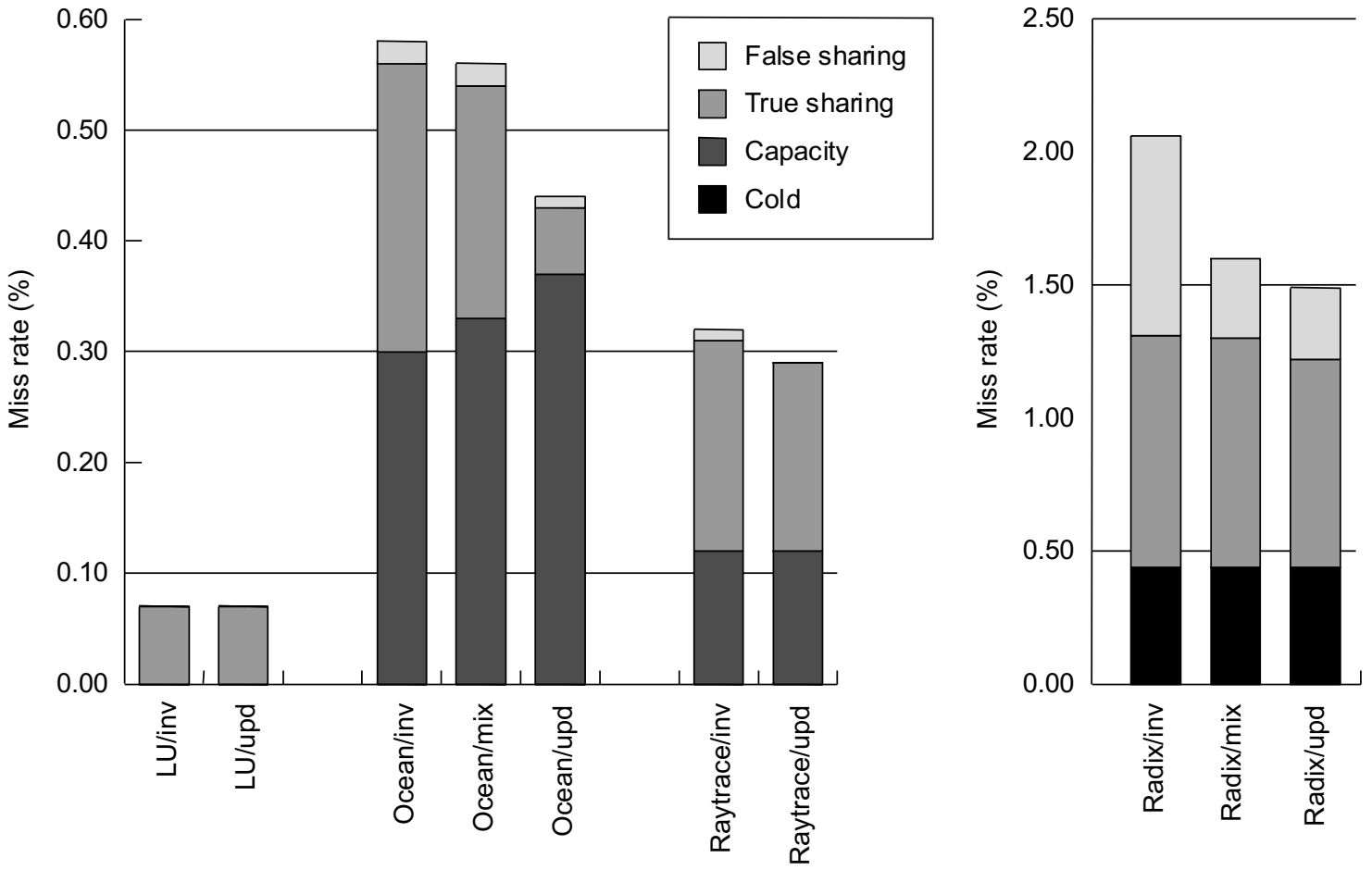


- 对这些负载来说，由于E→M转换很少，所以两种协议带宽差别不大
- 使用总线更新事务替代独占读，能够减少带宽占用
- Ocean, Radix, Raytrace三种负载类型下，总线带宽不同，系统可支持的处理器数量也不同

基于更新和基于作废协议的比较

- 是一个争论不休的话题，很大程度上取决于应用负载表现出来的共享模式
- 直观上:
  - 如果处理器在更新之前使用数据，并可能希望要在将来看到新的值，更新就会比作废表现出更好的性能
    - e.g. 生产者-消费者模式 producer-consumer pattern
  - 如果持有老数据的处理器再也不会用它了，更新就是不必要的
    - 收集鼠现象（“pack rat”），串行程序在操作系统的控制下在处理器之间迁移情况下，这种现象最为
- 可以混合使用两种方案
  - E.g. competitive: observe patterns at runtime and change protocol
- 下面我们就在真实负载下对这两种方案进行评估

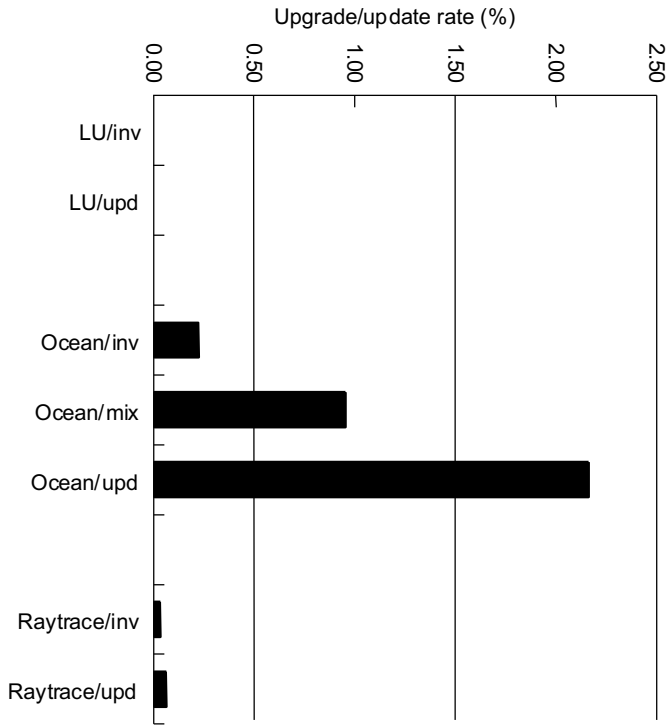
基于更新和基于作废协议的Cache Miss率评估

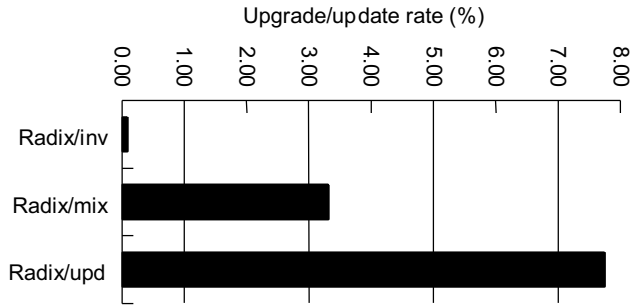


- 对于伪共享或真共享类的cache miss: updates help
- 对于容量型cache miss: updates hurt (keep data in cache uselessly)
- 这里没有考虑updates带来的总线带宽占用

Upgrade and Update Rates (Traffic)

- 和更新相联的流量是相当大的
- 主要原因是处理器P在其他处理器读取之前进行了多次写操作
  - 多次总线更新事务 vs. 一次作废事务
  - 可以采用延迟更新和合并操作来优化
- 基于更新的协议在业界使用的越来越少
  - 带宽, 复杂度, 缓存块变大的趋势, 多道程序的收集鼠现象等
- 更新协议对于可扩展缓存一致性系统还有一些其他问题（自学）





共享存储多处理器架构下的 OPENMP编程

An Overview of OpenMP

- A Guided Tour of OpenMP
- Case Study
- Wrap-Up

File Edit View History Bookmarks Tools Help

http://openmp.org/wp/

Google

Getting Started Latest Headlines Developer Guide

OpenMP.org

OpenMP

THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

RSS

What's Here:

» API Specs

- »About OpenMP.org
- »OpenMP Compilers
- »OpenMP Resources
- »OpenMP Forum

Input Register

Alert the OpenMP.org webmaster about new products or updates and we'll post it here.

»webmaster@openmp.org

Search OpenMP.org

Google™ Custom Search

Search

Archives

- June 2008
- May 2008
- April 2008

Admin

- Log in

OpenMP News

»Christian's First Experiments with Tasking in OpenMP 3.0

From Christian Terboven's blog:

> OpenMP 3.0 is out, maybe a bit later than we hoped for, but I think that we got a solid standard document. At IWOMP 2008 a couple of weeks ago, there was an OpenMP tutorial which included a talk by Alex Duran (from UPC in Barcelona, Spain) on what is new in OpenMP 3.0 - which is really worth a look! My talk was on some OpenMP application experiences, including a case study on Windows, and I really think that many of our codes can profit from Tasks. Motivated by Alex' talk I tried the updated Nanos compiler and prepared a couple of examples for my lectures on Parallel Programming in Maastricht and Aachen. In this post I am walking through the simplest one: Computing the Fibonacci number in parallel.

Read more…

Posted on June 6, 2008

»New Forum Created

The **OpenMP 3.0 API Specifications forum** is now open for discussing the specs document itself.

Posted on May 31, 2008

»New Links

New links and information have been added to the **OpenMP Compilers** and the **OpenMP Resources** pages.

Posted on May 23, 2008

»Recent Forum Posts

- **strange behavior of C function strcmp() With OPENMP**
- **virtual destructor not called with first private clause**

OpenMP.org

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

»Read about OpenMP

Get It

»OpenMP specs

Use It

»OpenMP Compilers

Learn It

Done

What is OpenMP?

- 基于C, C++, and Fortran语言编写共享内存并行应用程序的事实标准API
- 包含:
  - 编译器制导 Compiler directives
  - 运行时库 Run time routines
  - 环境变量 Environment variables
- 规范由 OpenMP 架构审查委员会维护 (http://www.openmp.org)
- Version 5.0 has been released May 2018

什么时候使用OpenMP?

- 当编译器无法按照你希望的方式执行并行化时：
  - 它发现不了并行性
    - ✓数据依赖性分析无法确定并行化是否安全
  - 编译器自动分析的粒度不够细
    - ✓编译器缺乏信息，无法在尽可能细的级别上进行并行化
- 这时，就可以通过OpenMP相关指令显式并行化

OpenMP的优势

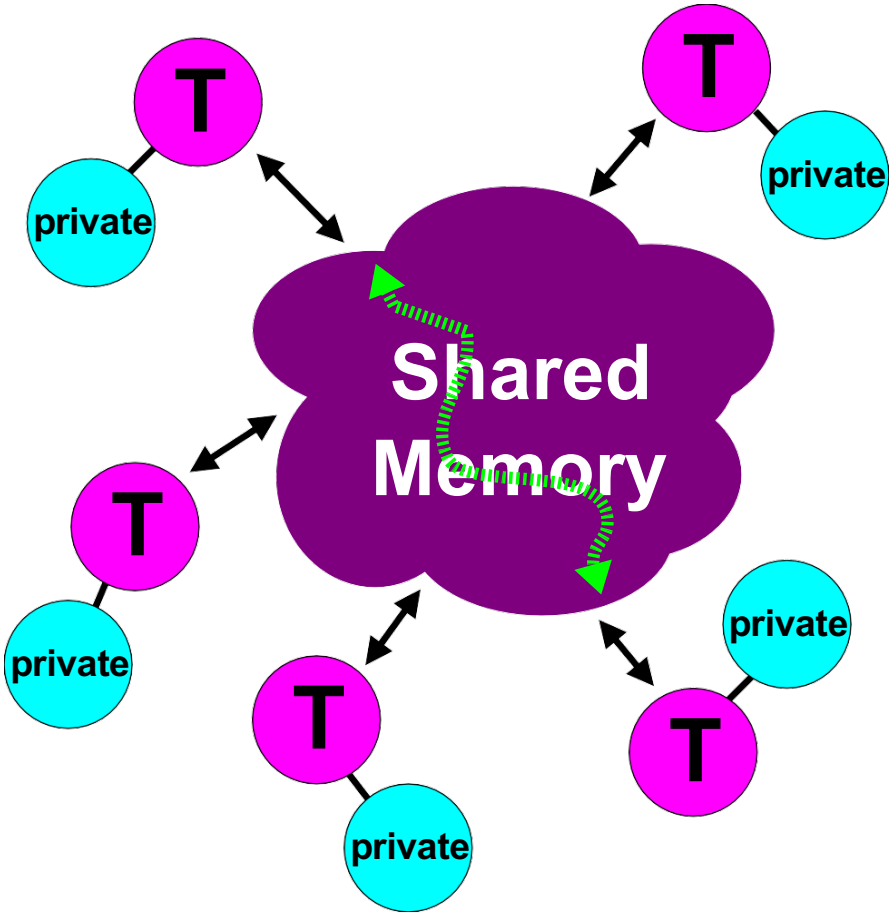
- 良好的性能和可扩展性
  - 如果使用得到的话 ....
- 事实上的和成熟的标准
- OpenMP程序可移植
  - 大量编译器都支持
- 只需要少量的编程工作
- 允许渐进地并行化程序

OpenMP 和多核体系结构

OpenMP is ideally suited for multicore architectures

Memory and threading model map naturally
Lightweight Mature
Widely available and used

OpenMP的存储模型



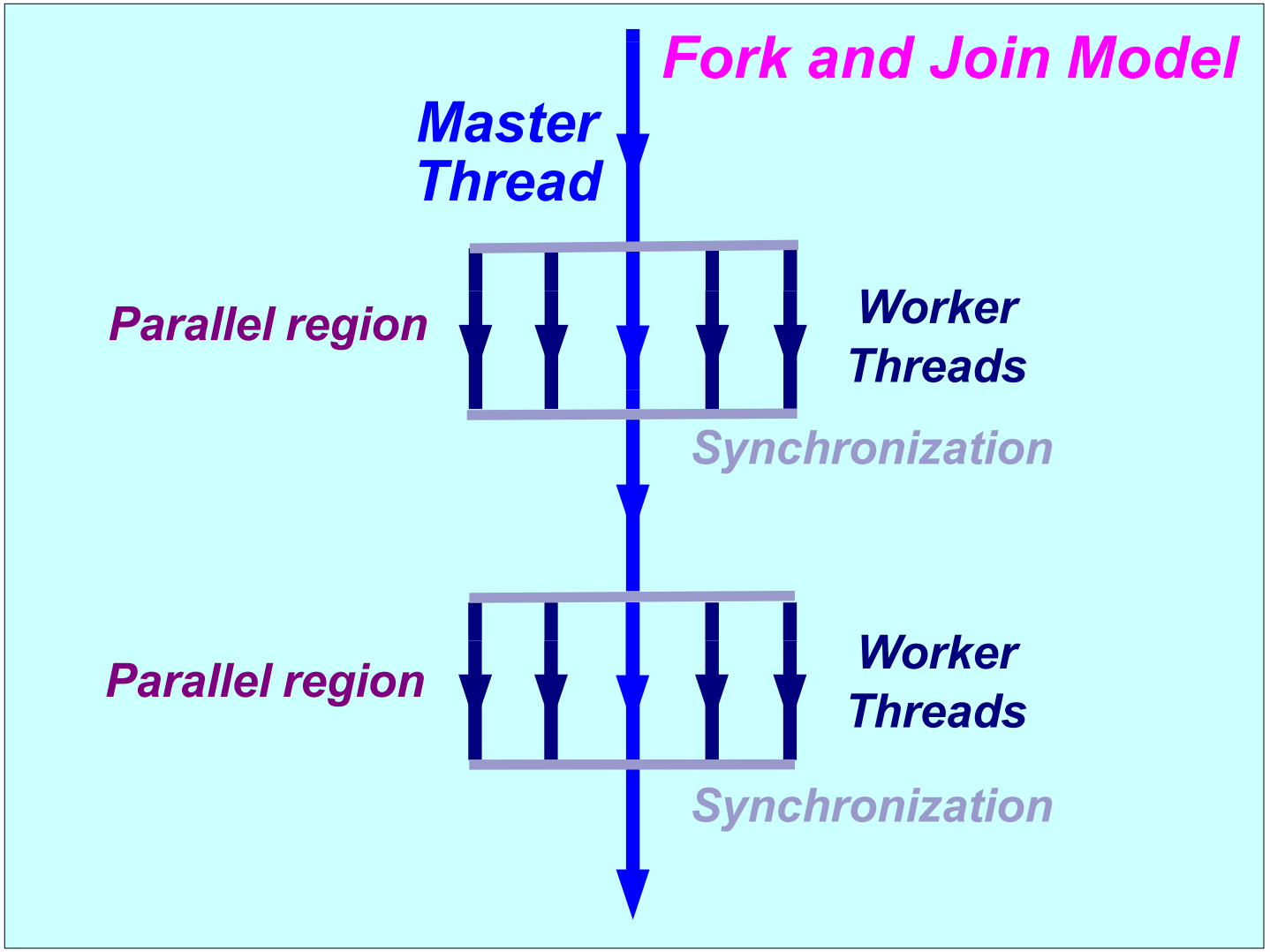
- ✓*所有线程都可以访问一块全局的共享存储*
- ✓*数据可以是共享的也可以是私有的*
- ✓*共享数据可以被所有线程访问*
- ✓*私有数据只能被拥有它的线程访问*
- ✓*数据传递对程序员是透明的（通过共享存储区域）*
- ✓*一般情况下，同步都是隐式的进行*

数据共享属性Data-Sharing Attributes

- 在OpenMP程序中，数据需要打上标签
- 有两种基本标签类型:
  - Shared
    - ✓ 只有一个数据的实例
    - ✓ 所有线程都可以同时读写该数据，除非通过特定的 OpenMP 结构对它进行了保护
    - ✓ 所做的所有更改都对所有线程可见

      但不一定是立竿见影的，除非强制执行.....
  - Private
    - ✓ 每个线程都有一个数据的副本
    - ✓ 其他线程不可以访问此数据
    - ✓ 只有拥有数据的线程可以对其进行修改

OpenMP的处理模型



简单的OpenMP程序样例

For-loop with independent iterations

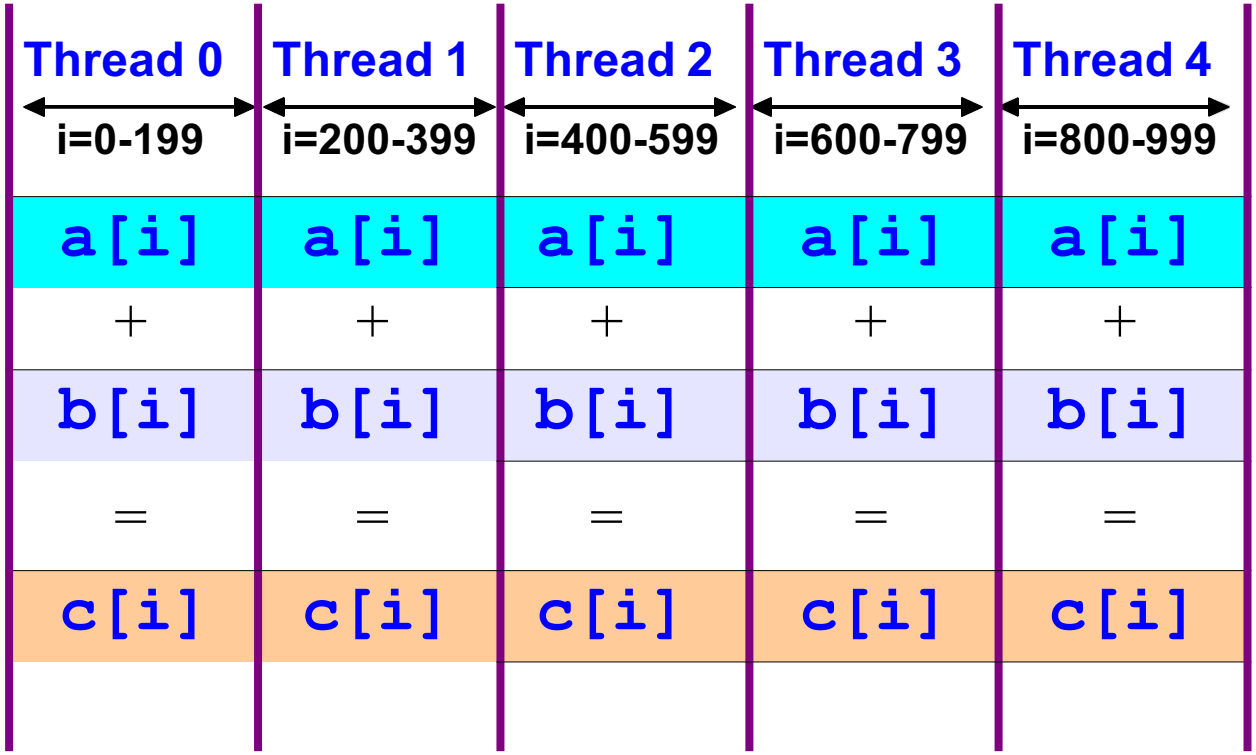
```
for (int i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 5
% a.out
```

上述程序的实际执行过程



OpenMP 2.5版本的相关组件

Directives

- Parallel region
- Worksharing
- Synchronization
- Data-sharing attributes
  - private
  - firstprivate
  - lastprivate
  - shared
  - reduction
- Orphaning

Runtime environment

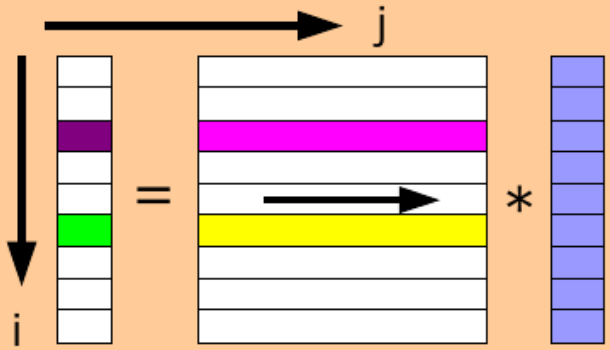
- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Wallclock timer
- Locking

Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism

举例 – 矩阵乘向量

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

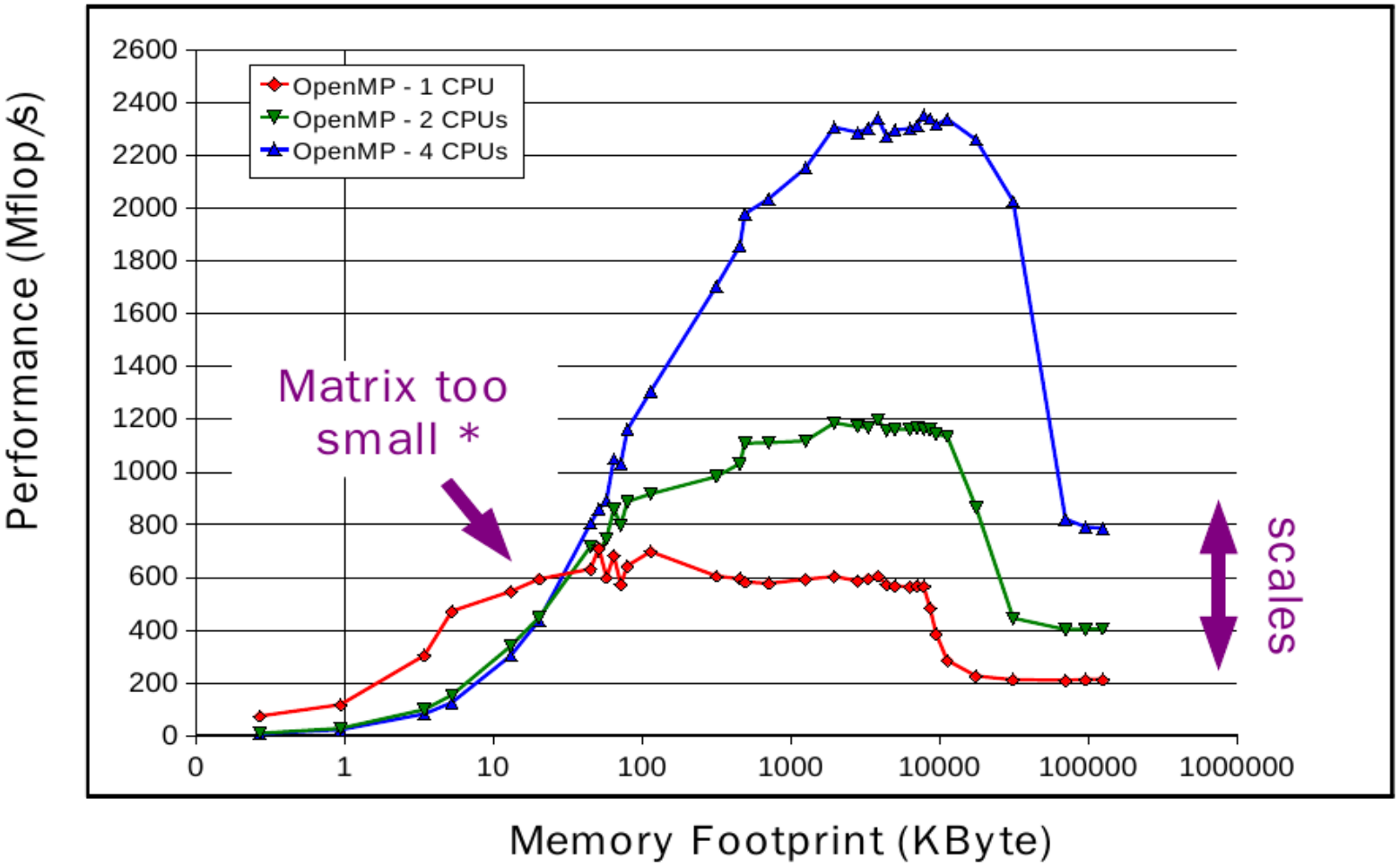
```
for (i=0,1,2,3,4)
 i = 0
sum = Σ  b[i=0][j]*c[j]
   a[0] = sum
 i = 1
sum = Σ  b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
 i = 5
sum = Σ  b[i=5][j]*c[j]
   a[5] = sum
 i = 6
sum = Σ  b[i=6][j]*c[j]
   a[6] = sum
```

... etc ...

矩阵乘向量OpenMP程序的性能



*) With the IF-clause in OpenMP this performance degradation can be avoided

一个更复杂的例子

```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];

#pragma omp for nowait

    for (i=0; i<n; i++)
       a[i] = b[i] + c[i];

#pragma omp barrier

          ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
          ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop
(work is distributed)

parallel loop
(work is distributed)

synchronization

Statement is executed by all threads

parallel region

OpenMP In Some More Detail

术语和行为

- ***OpenMP Team := Master + Workers***
- 一个并行区域（**Parallel Region**） 是一个被所有线程同时执行的代码块
  - **master**线程的线程 **ID**始终是 **0**
  - 线程调整（如果启用了）只能在进入并行区域前完成
  - 并行区域可以嵌套，但是对嵌套的支持是依赖于实现的
  - 可以使用“**if**”子句来保护并行区域；如果条件的计算结果为“ **false**”，则代码将连续执行
- 共享工作构造（***work-sharing construct***）将它作用的代码段拆分到进入此区域的线程**team**的成员执行

if/private/shared 子句

if (scalar expression)

- ✓ *只有当 **expression** 计算结果为 **true** 时才并行执行*
- ✓ *否则串行执行*

private (list)

- ✓ *与原始对象没有存储关联*
- ✓ *所有引用都指向本地对象*
- ✓ *值在进入和退出时未定义*

shared (list)

- ✓ *数据可以被**team**中的所有线程访问*
- ✓ *所有线程访问同一地址空间*

```
#pragma omp parallel if (n > threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

Barrier/1

Suppose we run each of these two loops in parallel over i:

```
for (i=0; i < N; i++)
    a[i] = b[i] + c[i];
```

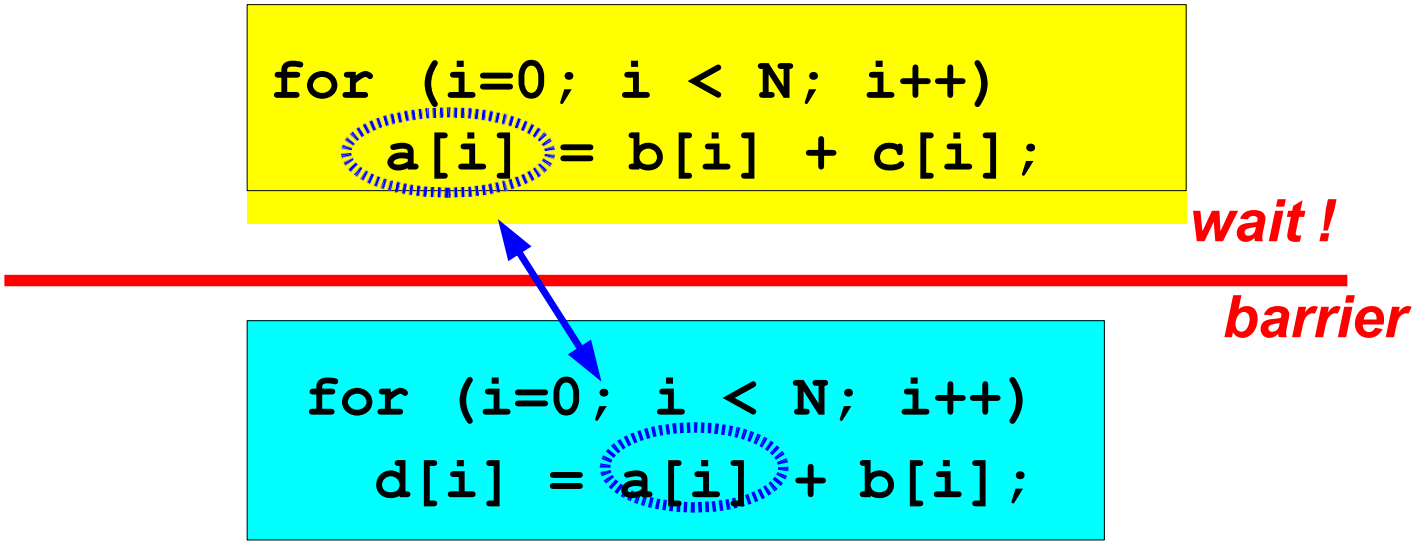
```
for (i=0; i < N; i++)
    d[i] = a[i] + b[i];
```

This may give us a wrong answer (one day)

Why ?

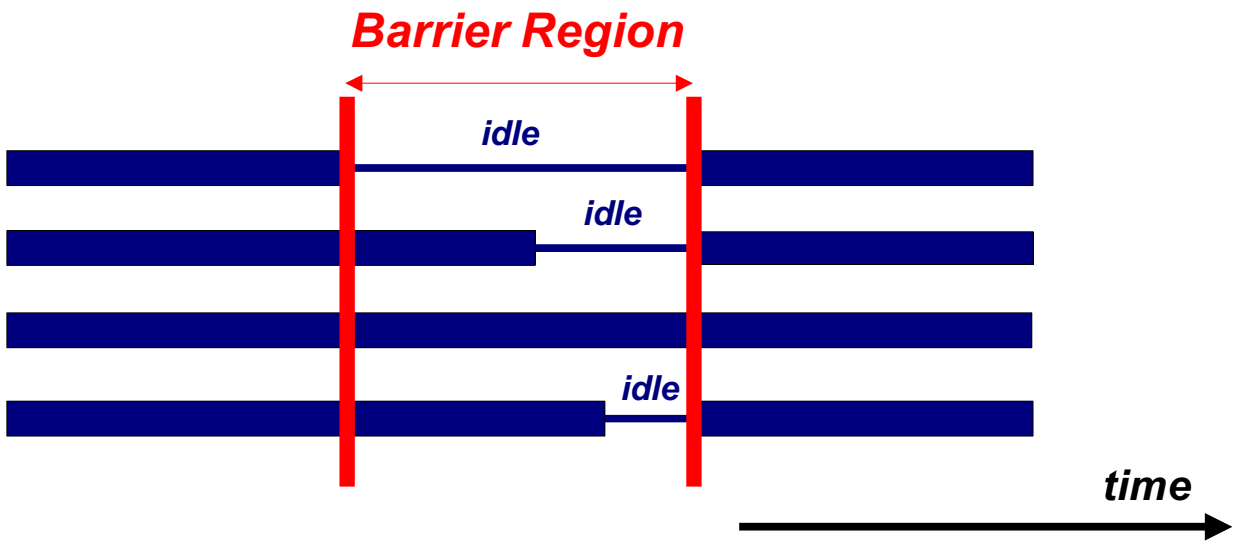
Barrier/2

We need to have updated all of a[ ] first, before using a[ ]



All threads wait at the barrier point and only continue when all threads have reached the barrier point

Barrier/3



Barrier syntax in OpenMP:

```
#pragma omp barrier
```

```
!$omp barrier
```

nowait子句

- 为了最小化同步开销，OpenMP 制导/pragmas 支持可选的nowait子句
- 线程不会在该特定构造的结尾处进行同步/等待
- 在 Fortran编程中，nowait 子句附加在构造的结尾部分
- 在C编程中，该子句是pragma的一部分

```
#pragma omp for nowait
{
        :
}
```

```
!$omp do
        :
        :
!$omp end do nowait
```

并行区域

并行区域是由多个线程同时执行的代码块

```
!$omp parallel [clause[[,] clause] ...]
    "this is executed in parallel"
!$omp end parallel (implied barrier)
```

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this is executed in parallel"
} (implied barrier)
```

共享工作构造Work-sharing constructs

The OpenMP work-sharing constructs

```
#pragma omp for
{
    ....
}

!$OMP DO
    ....
!$OMP END DO
```

```
#pragma omp sections
{
        ....
}

!$OMP SECTIONS
        ....
!$OMP END SECTIONS
```

```
#pragma omp single
{
        ....
}

!$OMP SINGLE
        ....
!$OMP END SINGLE
```

- 共享工作构造内的工作在所有线程上分配执行
- 必须包含在并行区域内
- 必须由team中所有的线程执行，或一个都不执行
- 进入时没有隐含的barrier; 出口时有隐含的barrier(除非指定nowait)
- 共享工作构造不会启动任何新的线程

for/do制导语句

作用：将for循环的不同迭代分布到不同的线程上执行

```
#pragma omp for [clause[[,] clause] ...]
   <original for-loop>
```

```
!$omp do [clause[[,] clause] ...]
        <original do-loop>
!$omp end do [nowait]
```

Clauses supported:

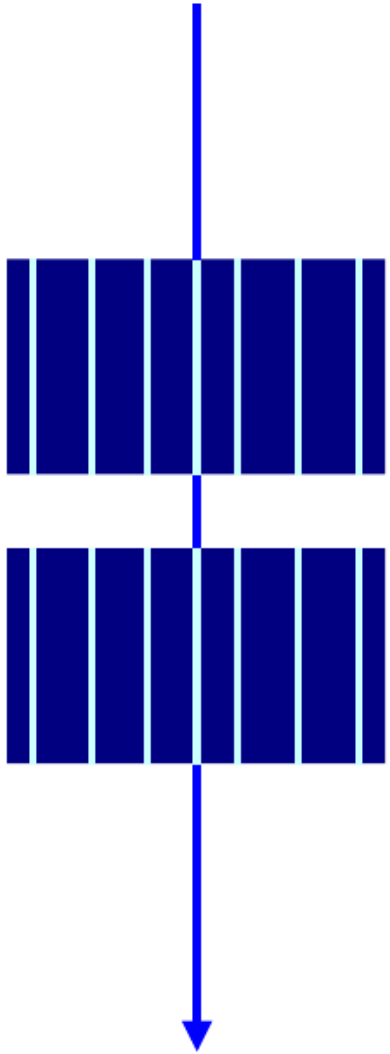
private	firstprivate
lastprivate	reduction
*ordered**	*schedule* ← *covered later*
nowait	

**) Required if ordered sections are in the dynamic extent of this construct*

omp指导语句的使用样例- Example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)



Section制导语句

作用：将section内的代码块分布在线程上执行

```
#pragma omp sections [clause(s)]
{
#pragma omp section
          <code block1>
#pragma omp section
          <code block2>
#pragma omp section
            :
}
```

```
!$omp sections [clause(s)]
!$omp section
          <code block1>
!$omp section
          <code block2>
!$omp section
            :
!$omp end sections [nowait]
```

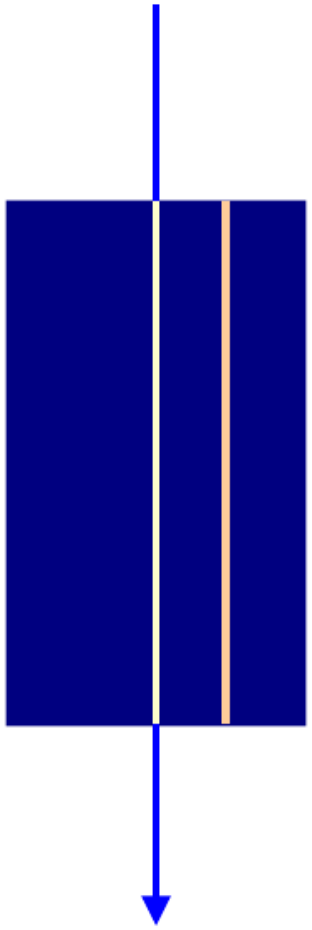
Clauses supported:

private **firstprivate**
lastprivate **reduction**
nowait

Note: The SECTION directive must be within the lexical extent of the SECTIONS/END SECTIONS pair

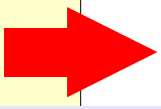
The sections directive - Example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```



共享工作构造的组合

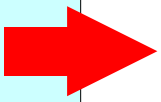
```
#pragma omp parallel
#pragma omp for
   for (...)
```



```
#pragma omp parallel
for  for (....)
```

Single PARALLEL loop

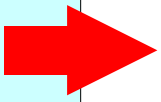
```
!$omp parallel
!$omp do
        ...
!$omp end do
!$omp end parallel
```



```
!$omp parallel do
        ...
!$omp end parallel do
```

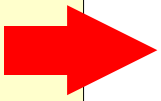
```
!$omp parallel
!$omp workshare
        ...
!$omp end workshare
!$omp end parallel
```

Single WORKSHARE loop



```
!$omp parallel workshare
        ...
!$omp end parallel workshare
```

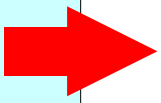
```
#pragma omp parallel
#pragma omp sections
{ ...}
```



```
#pragma omp parallel sections
{ ... }
```

Single PARALLEL sections

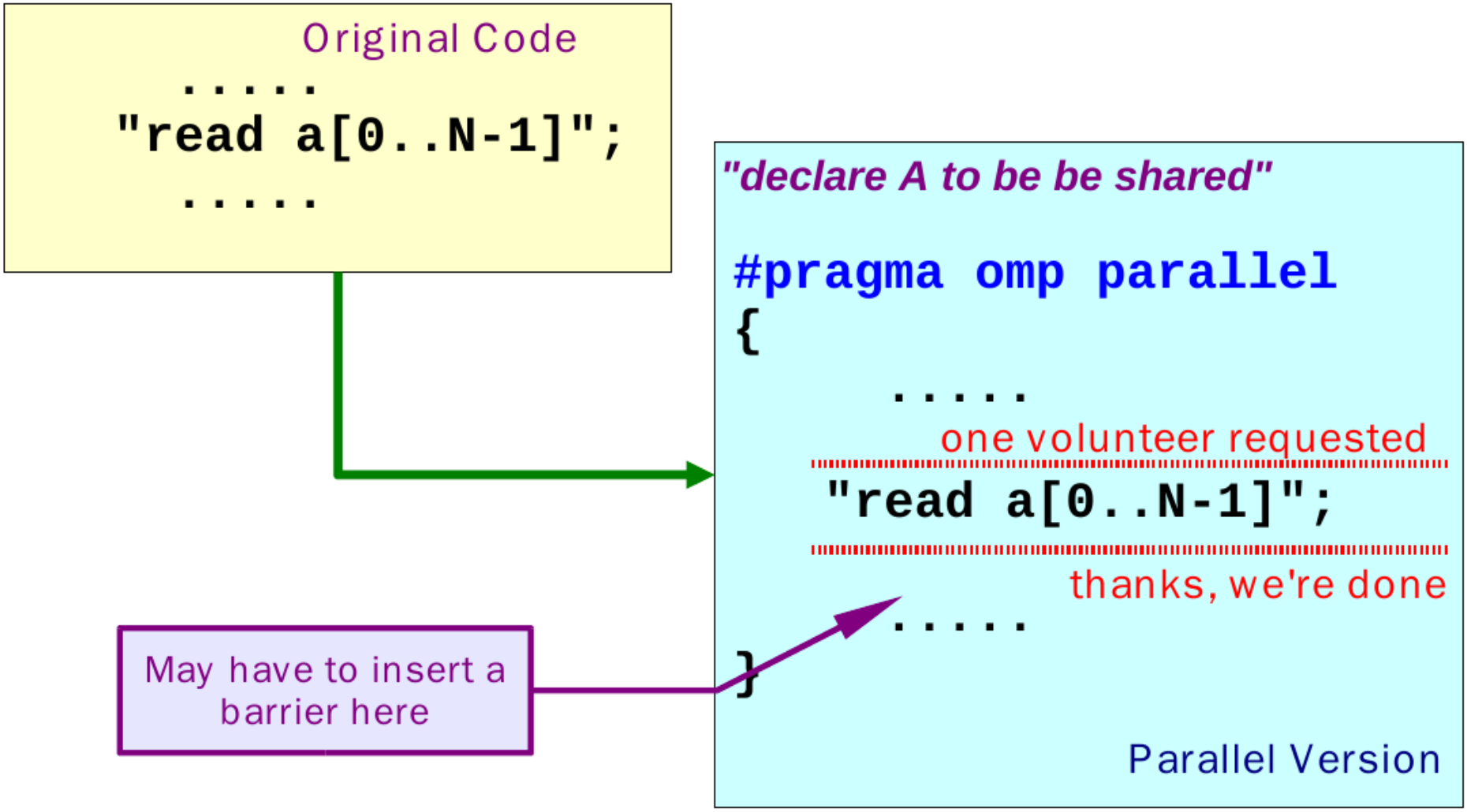
```
!$omp parallel
!$omp sections
        ...
!$omp end sections
!$omp end parallel
```



```
!$omp parallel sections
        ...
!$omp end parallel sections
```

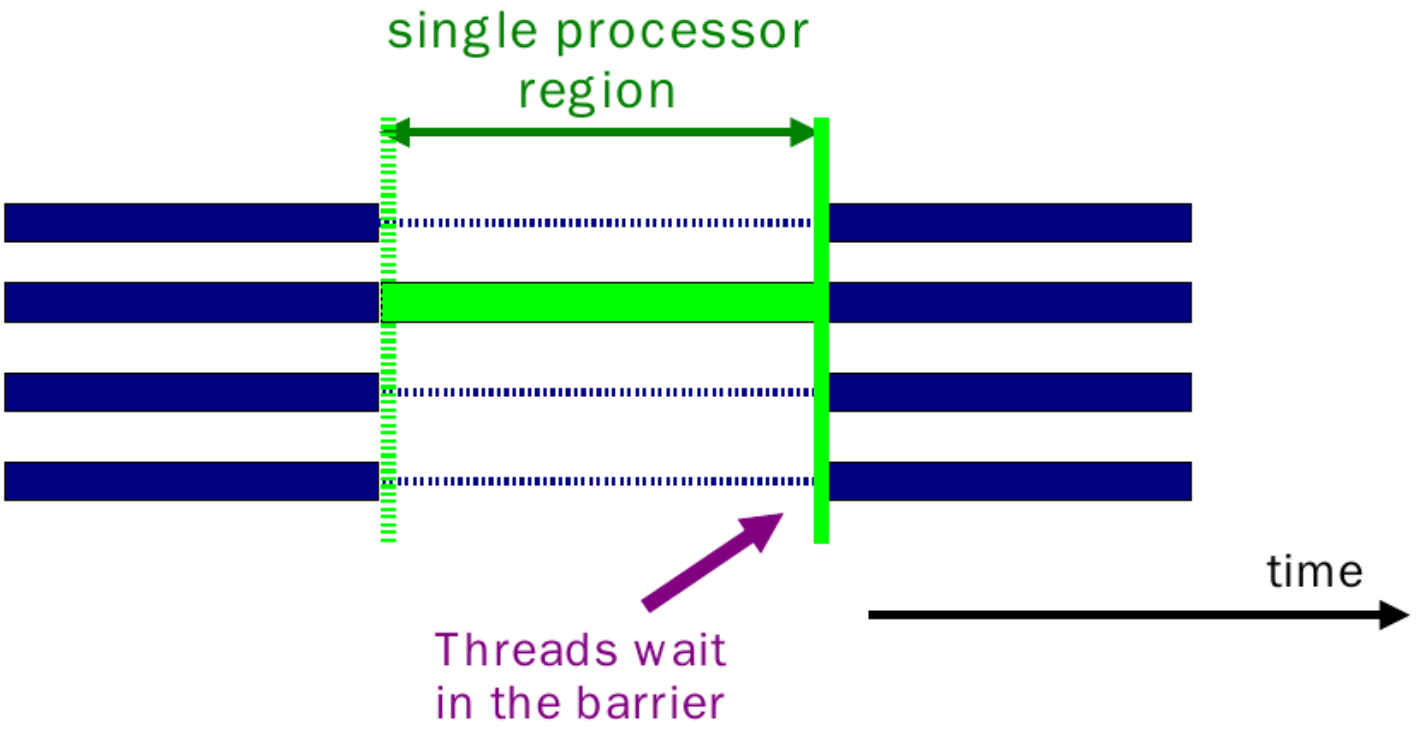
Single processor region/1

This construct is ideally suited for I/O or initializations



Single processor region/2

- Usually, there is a barrier at the end of the region
- Might therefore be a scalability bottleneck (Amdahl's law)



SINGLE and MASTER construct

Only one thread in the team executes the code enclosed

```
#pragma omp single [private][firstprivate] \
                          [copyprivate][nowait]
{
        <code-block>
}
```

```
!$omp single [private][firstprivate]
   <code-block>
!$omp end single [copyprivate][nowait]
```

Only the master thread executes the code block:

```
#pragma omp master
{<code-block>}
```

```
!$omp master
        <code-block>
!$omp end master
```

There is no implied barrier on entry or exit !

Critical Region/1

If sum is a shared variable, this loop can not run in parallel

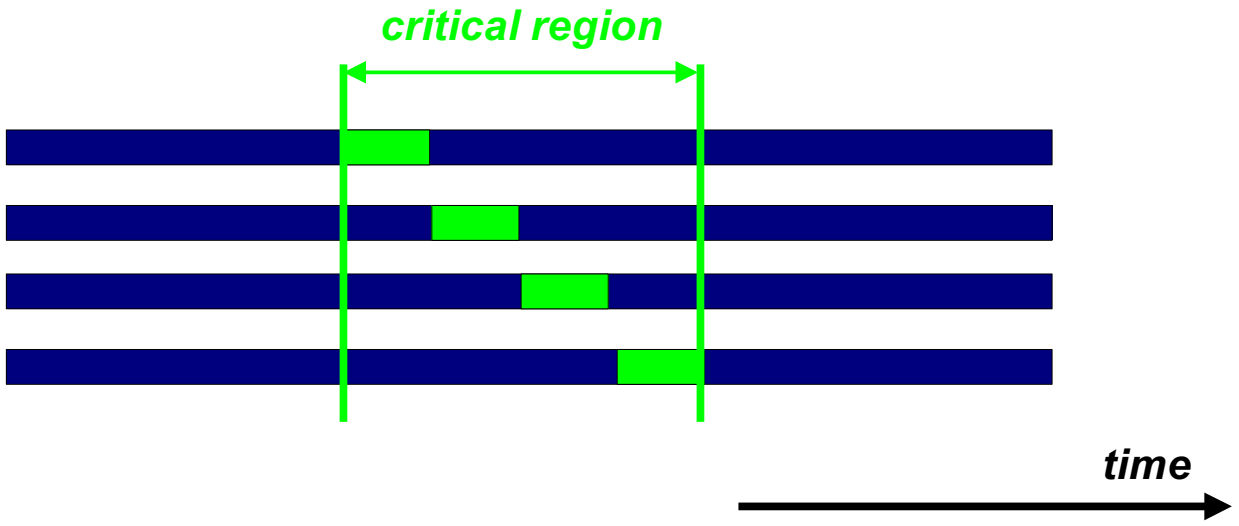
```
for (i=0; i < N; i++){
      .....
    sum += a[i];
      .....
}
```

We can use a critical region for this:

```
for (i=0; i < N; i++){
      .....                    one at a time can proceed
    sum += a[i];
      .....                    next in line, please
}
```

Critical Region/2

- *Useful to avoid a race condition, or to perform I/O (but that still has random order)*
- *Be aware that there is a cost associated with a critical region*



Critical and Atomic constructs

Critical: All threads execute the code, but only one at a time:

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

There is no implied barrier on entry or exit !

Atomic: only the loads and store are atomic ....

```
#pragma omp atomic
   <statement>
```

```
!$omp atomic
   <statement>
```

This is a lightweight, special form of a critical section

```
#pragma omp atomic
    a[indx[i]] += b[i];
```

Why The Excitement About OpenMP 3.0 ?

Support for TASKS !

With this new feature, a wider range of applications can now be parallelized

Example - A Linked List

```
        ........

while(my_pointer) {


        (void) do_independent_work (my_pointer);

  my_pointer = my_pointer->next ;
} // End of while loop

        ........
```

Hard to do before OpenMP 3.0:
First count number of iterations, then convert while loop to for loop

Example - A Linked List With Tasking

```
my_pointer = listhead;

#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
          #pragma omp task firstprivate(my_pointer)
          {
              (void) do_independent_work (my_pointer);
          }
          my_pointer = my_pointer->next ;
        }
    } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

OpenMP Task is specified here
(executed in parallel)

Case Study A Neural Network

Neural Network application

Performance Analyzer Output

Excl. User CPU sec.	%	Incl. User CPU sec.	Excl. Wall sec.	Name
120.710	100.0	120.710	128.310	<Total>
116.960	96.9	116.960	122.610	calc_r_loop_on_neighbours
0.900	0.7	118.630	0.920	calc_r
0.590	0.5	1.380	0.590	_doprnt
0.410	0.3	1.030	0.430	init_visual_input_on_V1
0.280	0.2	0.280	1.900	_write
0.200	0.2	0.200	0.200	round_coord_cyclic
0.130	0.1	0.130	0.140	__arint_set_n
0.130	0.1	0.550	0.140	__k_double_to_decimal
0.090	0.1	1.180	0.090	fprintf

Callers-callees fragment:

Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
116.960	0.900	118.630	calc_r
116.960	116.960	116.960	*calc_r_loop_on_neighbours

Source line information

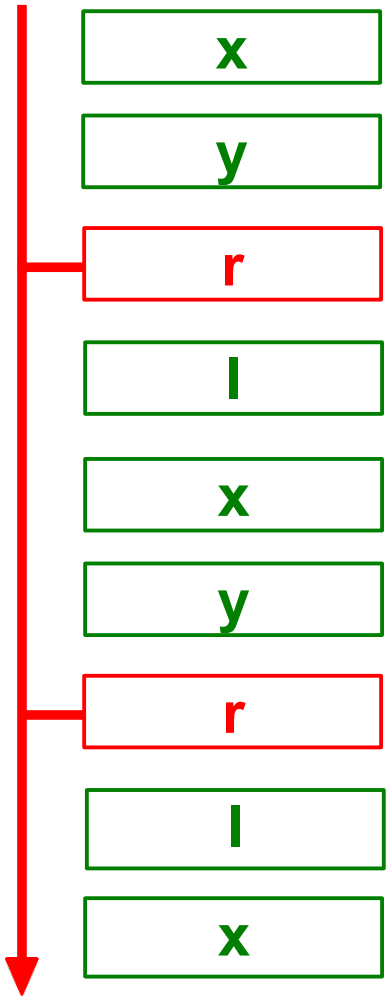
```
    struct cell{
      double x; double y; double r; double I;
     };
                        ......

     struct cell V1[NPOSITIONS_Y][NPOSITIONS_X];
     double       h[NPOSITIONS][NPOSITIONS];
                        ......
   Excl. User CPU  Excl. Wall
     sec.     %       sec.
                              1040. void
                              1041. calc_r_loop_on_neighbours
                                          (int y1, int x1)
     0.080    0.1     0.080   1042. {
                              1043. struct interaction_structure *next_p;
                              1044.
     0.130    0.1     0.130   1045. for (next_p = JJ[y1][x1].next;
     0.460    0.4     0.470   1046.      next_p != NULL;
                              1047.      next_p = next_p->next) {
##  116.290  96.3    121.930  1048.     h[y1][x1] += next_p->strength *
                               V1[next_p->y][next_p->x].r;
                              1049.
                              1052.   }
                              1053. }
```

What is the problem ?

96% of the time spent in this single statement

Data structure problem

- *We only use 1/4 of a cache line !*
- *For sufficiently large problems this will:*
  - *Generate additional memory traffic*
    - ✓*Higher interconnect pressure*
  - *Waste data cache capacity*
    - ✓*Reduces temporal locality*
- *The above negatively affects both serial and parallel performance*
- *Fix: split the structure into two parts*
  - *One contains the "r" values only*
  - *The other one contains the {x,y,l} sets*



Fragment of modified code

```
double V1_R[NPOSITIONS_Y][NPOSITIONS_X];

void
calc_r_loop_on_neighbours(int y1, int x1)
{
  struct interaction_structure *next_p;

  double sum = h[y1][x1];

  for (next_p = JJ[y1][x1].next;
       next_p != NULL;
       next_p = next_p->next) {
       sum += next_p->strength * V1_R[next_p->y][next_p->x];
  }
  h[y1][x1] = sum;
}
```

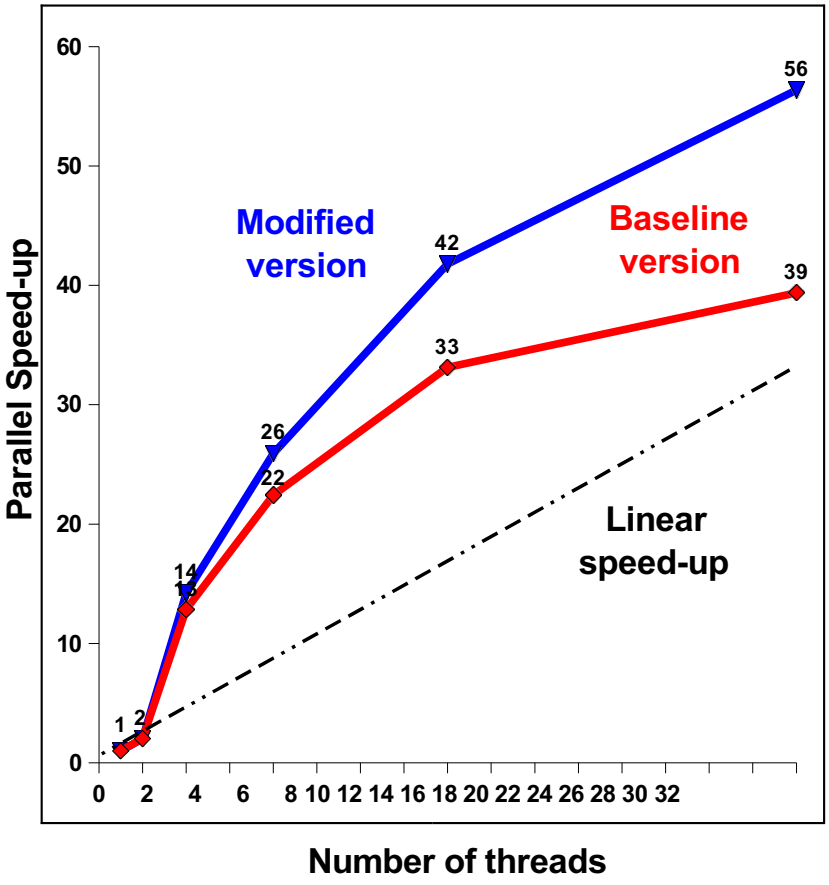
Parallelization with OpenMP

```
void calc_r(int t)
{

#include <omp.h>

#pragma omp parallel for default(none)  \
        private(y1,x1) shared(h,V1,g,T,beta_inv,beta)
  for (y1 = 0; y1 < NPOSITIONS_Y; y1++) {
    for (x1 = 0; x1 < NPOSITIONS_X; x1++) {

      calc_r_loop_on_neighbours(y1,x1);
      h[y1][x1] += V1[y1][x1].I;

      <statements deleted>

    }
  }

/*-- End of OpenMP parallel for --*/
```

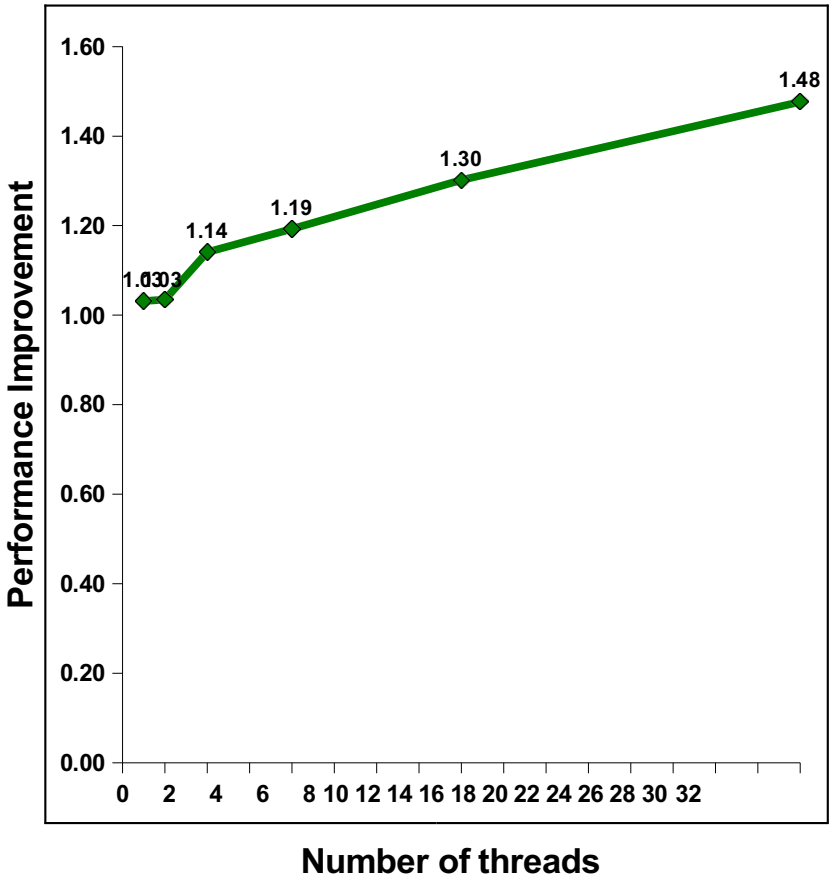
Can be executed in parallel

Scalability results

T(One proc)/T(P procs)



T(baseline)/T(modified)



Note:

Single processor run time is 5001 seconds for the baseline version (4847 for the modified version)

Thanks
Q&A