Chapter 1

1、平均无故障时间 MTTF

失效率 $FIT = \frac{1}{MTTF}$

平均修复时间 MTTR

平均故障间隔时间 MTTF+MTTR

模块可用性 $\frac{MTTF}{MTTF+MTTR}$.

2、Amdahl定律

总加速比 = $\frac{原执行时间}{新执行时间} = \frac{1}{1-升级比例+\frac{升级比例}{升级加速比}}$

3、CPU时间 = $\frac{程序CPU时钟周期数}{时钟频率}$.

$CPI = \frac{程序CPU时钟周期数}{指令数}$

CPU时间 = 指令数 × CPI × 时钟周期时长

Chapter 2

更大的 blocksize → 降低 Compulsory miss

$2^{\log_2 set}$ + offset + index = cache size

故已知 set 和 cache size 就可以求出 tag .

VPN 到 TLB 中找到 PPN，组合 offset 得 PA .
(v|D|tag|PPN)

PA 到 cache 里找 data (v|tag|data)

e. virtual indexed and physical tagged cache

TLB 和 Cache 同时访问 . 让 block offset + cache index

控制在 VA 的 offset 内

缺点：限制 cache 的大小.

solution：多路组相联

5. 提升 Cache 性能的方法：

① 提高 DRAM 速度

RAS 上部读行地址 . CAS 上部读列地址 . 读数据 . 而高

I. 一次取多个 word II. DDR 上下降沿分别读 III. Multiple bank 并行读

IV. 低 bit 交叉存储器 (bank , 数量可以是 2^n+1 或 2^n-1)

② 减少 Hit Time

[illegible]

Chapter 3

1. RAW 真相关 WAW 输出/命名相关

WAR 反相关/命名相关

寄存器重命名时 把重名的一条即可

2. scoreboard 寄存器有很多写口 .

四个阶段：Issue + Read Operands + Execution + WB

三个数据结构：

instruction status：指令在以上四阶段的哪个阶段

register result status：哪个 FU 会写入该寄存器 Issue 阶段设 FU，WB 时变成值 .

FU status： busy / op / Fi / Fj / Fk
(Fi 目的寄存器，Fj 源寄存器1，Fk 源寄存器2)

Rj / Rk：源寄存器是否可用 / Qj / Qk：若不可用在哪个 FU 里

注：① WB 阶段 busy 置为 0，下一周期该 FU 才能处理新指令

② Instruction 不能乱序，某一条指令未运行下一条不能执行，但是有可能后一条指令先执行完

③ LD 使用的是 Integer

In-order Issue
Out-of-order EX
out-of-order completion

④ WAW 暂停 issue
WAR 在 WB 时 stall .

3. tomasulo：每个 FU 都有自己的 reservation station

三个阶段：Issue + Exec + WB

三个数据结构：

instruction status：同上

reservation stations： busy / op / Vj / Vk (源寄存器) / Qj / Qk (reservation stations) / A (↓ 地址)

对于 SW 需要

LD/SW 可以单独看作 busy / A / (FU)

其他是 busy / op / Vj / Vk / Qj / Qk

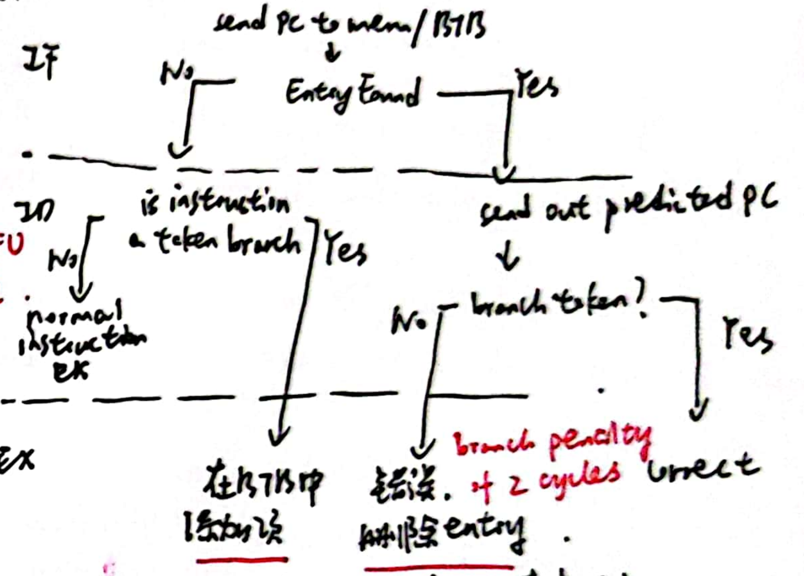
register status：指令结果存放的 FU，(结束时为具体值 (WB))

注：① in-order issue / out-of-order EX / out-of-order completion ② 不能精确中断

4. scoreboard + explicit renaming .

PPM (Prediction by partial matching)

6. branch target buffer 把 target address 也存起来



7. hardware based speculation 精确中断 .

结果存放到 Reorder Buffer 中

异常或跳转将 ROB 清空 .

阶段四个：Issue + EX + WB + commit

注意：① 对于非 Branch：当到达 head of ROB，更新 Reg 如果是 store 更新 memory ② 如果是错误预测的 branch 就 flush 掉 ROB ③ 正确预测则继续执行

ROB 结构：(一开始未求出)

目的寄存器 | 值 | 指令 | done?

去掉 Reg status，改用 ROB1、ROB2、ROB3 代替 .

存在问题：store 和 load 的 RAW 问题 (store 的数据存在 ROB 中)

Reservation station 结构：

…数的路（bank，数量可以是 2^n+1 或 2^n-1）

㊀ 减少 Hit Time

I. small and simple cache 直接映像

II. way prediction，不管 tag 结果直接先送出，适用于 Icache

III. avoiding address translation　之后再判断 stall

根据 VA 到物理内存的 cache

IV. trace cache，动态执行指令流出时打包放入 cache（包括 prediction info）

提高带宽

I. 流水线（两级，匹配 + 写）

II. Multibanked Cache

III. Nonblocking cache 失配后在 load 的过程中还能接受访问。hit under 1 miss：接受两次 miss

㊁ 减少 Miss Penalty

I. 多级 Cache

II. Give Priority to Read Miss over write

to write buffer，read miss 的时候先去 write buffer 里找（之前是 buffer 先写到内存），然后再去 memory 里找

III. 关键词优先 & early restart：读到就继续

IV. merging write buffer：若某个 word 已经在 buffer 里面，之后的同一个 block 的其他 word 可以和它合并

V. victim cache：被替换放到 victim cache 之后可能会替换回来（miss rate 也降低了）

㊂ 减少 Miss Rate

I. 更大 block size　II. 更大 cache　III. 更高关联度

IV. 编译器优化：

---

out-of-order completion ⑩ 不能精确中断

4. scoreboard + explicit renaming.

逻辑寄存器 F？物理寄存器 P？
（32个）（不止32个）

当指令流入时 F？对应一个 P？，当指令执行完没有异常才生效

维护一个 P 寄存器的 freelist，如果没有 free 的物理寄存器不能发射　（不在 freelist 里的值构成寄存器状态）

存在一个排序结构：→ rd 取一个 freelist 头

new ↓ old

F 寄存器　P 寄存器　指令　done？（且按顺序）

之前对应 P，当指令 done 的时候放入 freelist

注意：① F 寄存器初始化时指定对应 P

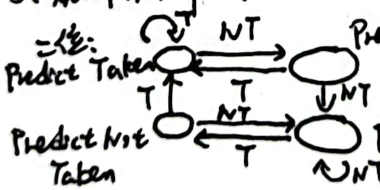
② 每条指令都会取一个 freelist，并修改 register status，但此时对应（F 和 P）没有真实生效即原来的 P 没有返回给 freelist

③ 只有 RAW 11（读操作时会 stall（和 tomasulo 一样）

④ 存在 store 的 undo 问题

5. 硬件预测

二位：



相关预测器：

（1,1）上次为 NT 就看第一位，上次为 T 就看第二位　（实际上太多了，所以一般有限，根据指令后四位来 index）

每一个 branch 都有自己的预测器，上一 branch 是 T 或 NT
如 T/NT　（1,1）中前一

在 ROB 中）

Reservation station 结构：

Dest / op / 源寄存器或源 ROB
↓
ROB 序号　（对于 load 是地址）
不考虑 WAW

8. 多发射（同时 issue）superscalar.

static：按序执行

dynamic：取指数大于发射数，乱序执行

speculation：可以发射 branch 之后的指令

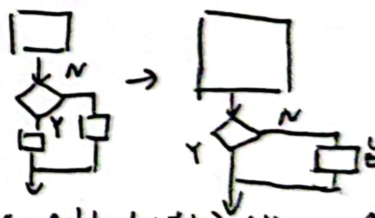
9. 软件流水线

→ 这一段可以同时执行

例子：
循环展开，可以每个循环执行不同指令

10. trace scheduling.

路径选择 + 跳转压缩　多个片段组织起来调度
（预测器）



11. superblock 对上面优化，No 全部到一个 handler 里统一解决

12. 硬件加指令（条件转移指令）

13. 多线程：

细粒度：每个时钟周期切换

粗粒度：发生 FU data hazard 或 L1/L2 miss 切换

…Miss Penalty … [illegible]

I. 多级Cache

II. Give Priority to Read Miss over write
如write buffer, read miss的时候先去write buffer里找(之前是buffer先写到内存)
然后再去memory里找

III. 关键词优先 & early restart: 读到后继续

IV. merging write buffer: 若某个word已经在buffer里面, 之后的同一个block的某些word可以和它合并

V. victim cache: 被替换放到victim cache之后可能会替换回来 (miss rate也降低了)

③ 减少Miss Rate

I. 更大block size II. 更大cache III. 更高关联度

IV. 编译器优化:
merging arrays 多个数组合成一个struct的数组
loop interchange 内外循环交换
loop fusion 多个loop变成一个loop
分块化

V. 伪关联cache + way prediction
↳ 直接映像miss rate高, 于是指向一个对应块 cache index + 最高位取0和取1分别对应一个块
增加预测器.

④ 利用并行性降低Miss Rate/Miss Penalty.

I. 硬件预取: memory空闲预取. 指令预取/数据预取

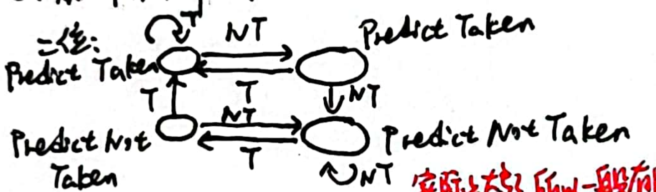
II. 编译控制预取

III. 高带宽 HBM 扩展存储层次

注意: ① F寄存器初始化时指定对应P
② 每条指令都会取一个freelist, 并修改register status, 但此时对应(F→P)没有真实生效即本来的P 没有返回给freelist
③ 只有RAW/结构竞争时会stall (和tomasulo一样)
④ 存在store的undo问题

5. 分支预测

二位:



相关预测器:
(1,1)上次为NT就看第一位, 上次为T就看第二位
每一个branch都有自己的预测器. 上一branch是T或NT 如T/NT —— (1,1)中前个1
决定当前branch的预测器选哪一位, 如果正确不修改否则修改这个值. —— (1,1)中后一个1

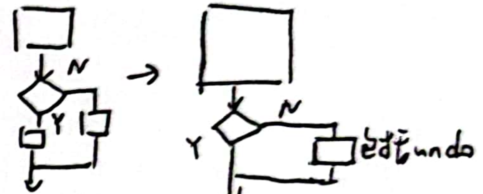
(m,n):
对于每个branch:
AAA…A / BBB…B / … / XXX…XXX
(n位; 2^m种)

实际上太多了, 所以一般有限, 根据指令后四位来index

锦标赛预测器:
branch history → global predictor → 结果
branch addr → local predictor / selector → 结果

gshare: 将上面history和addr异或

10. trace scheduling.
跳转选择 + 跳转后补偿 多个片段组织起来调度 (预测器)



11. superblock 对上面优化, No全部到一个handler里统一解决

12. 硬件加指令 (条件转移指令)

13. 多线程:
细粒度: 每个时钟周期切换
粗粒度: 发生FU data hazard或L1/L2 miss切换

Chapter4

1. chaining —— 有chaining就可以
2. convoy 打包指令组: 不包含结构冒险 / RAW
3. 钟声 chime. 执行一次convoy需要一次chime
4. 多车道: 同时多个FU
5. 处理多余循环
6. 循环依赖
如果存在相关, GCD(c,a) 是否整除 (d-b)

(SMP) / UMA

- centralized shared memory
- distributed memory
  - distributed shared memory (DSM) / NUMA — shared-memory
  - multiple computer — message passing — synchronized / asynchronized

- cache coherence (value, same location)

更新问题:

memory → disk: 最新数据可能在cache里 (write through时候)

disk → memory: 如果 write through 那么一直在 cache 里 hit 不从 disk 拿新数据

解决方法: 硬件: 从disk写入读取要前 cache

软件: 新建 buffer 里面存最新数据且不会到 cache 中去

coherence miss: 无效化导致的 miss

两种情况: true/false sharing miss

步骤:

① Snoopy protocols

write invalidate:

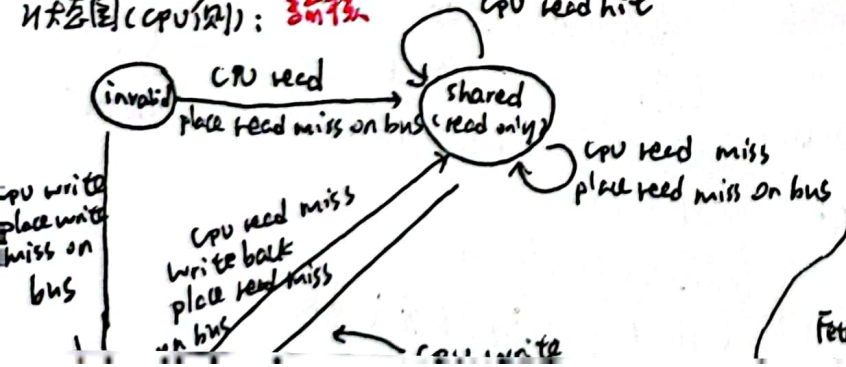
写的时候在自己 cache 里写, 其他人 invalidate;

cache miss 的时候最近修改的人 write back 然后容错的人从 memory 里读

write broadcast:

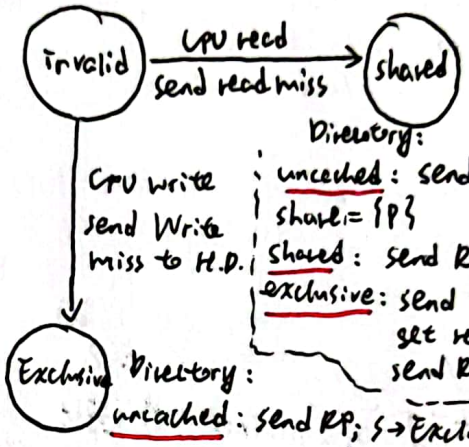
写的时候将值广播到总线上, 其他核直接更新这个值

状态图 (CPU视角): 写核



状态图 (CPU): 注: RP = response; R.N. = remote node

状态 invalid:



Directory:

- uncached: send RP; S → shared, share = {p}
- shared: send RP; share += {p}
- exclusive: send Fetch to R.N. get reply from R.N. send RP → shared += {p}

Directory:

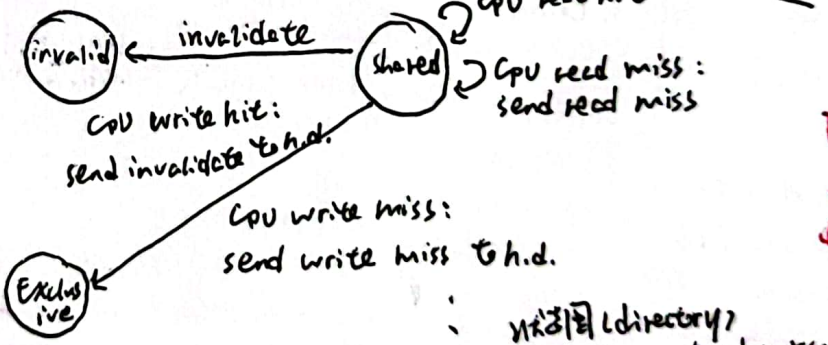
- uncached: send RP; S → Exclusive; share = {p}
- shared: send invalidate; send RP; S → Exclusive, share = {p}
- exclusive: send Fetch/Invalidate to R.N. get reply from R.N. send RP to P; S → Exclusive, shared = {p}

关于 true/false sharing miss

first time to write shared data = true

如果 block size = 1 word, miss 是否存在. 如存在就是 true

状态 shared:



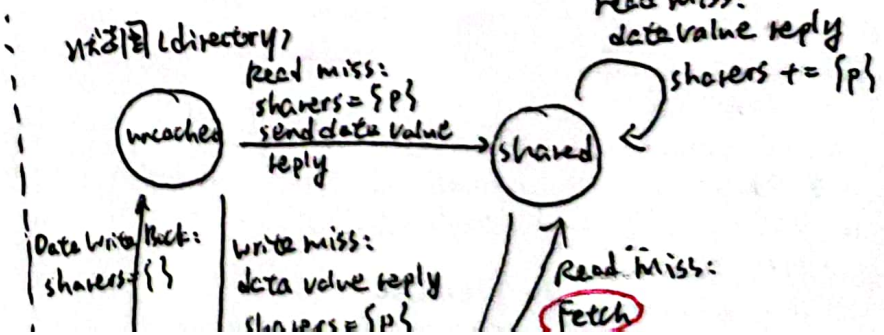
Fetch: h.d. → R.N.

获取 data

data WB: R.N. → h.d.

状态 Exclusive:

状态 (directory):



软件：新进来 buffer 里面存最新数据且不会到 cache 中去

coherence miss：无效性导致的 miss

两种情况：true/false sharing miss

写核：

① Snoopy protocols

write invalidate：

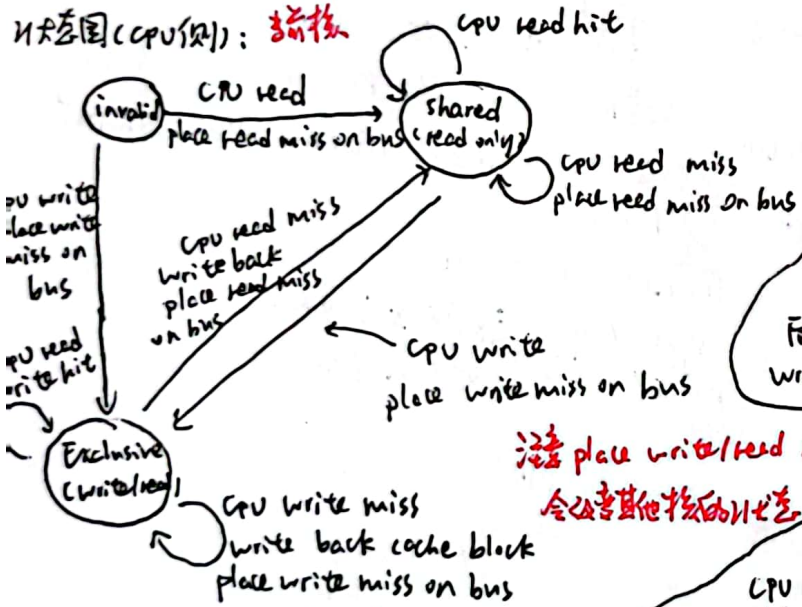
写的时候在自己 cache 里写，其他人 invalidate；

cache miss 的时候最近修改的人 write back 然后需要的人从 memory 里读

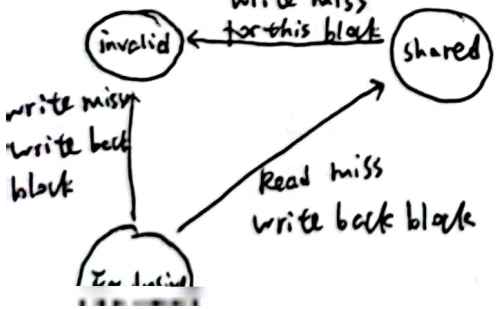
write broadcast：

写的时候特值广播到总线上，其他核直接更新这个值

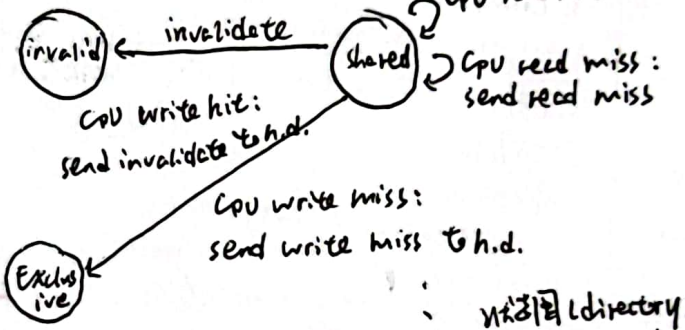
状态图（CPU 视角）：当前核



状态图（总线视角）：其他核



shared:



shared:

invalid: send invalidate; send Rp; S→Exclusive

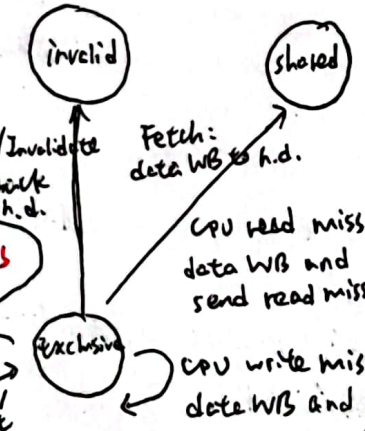
share = {p}

exclusive: send Fetch/Invalidate to R.N.

get reply from R.N.

send Rp to P; S→Exclusive, shared={p}

Exclusive:



注意 place write/read miss

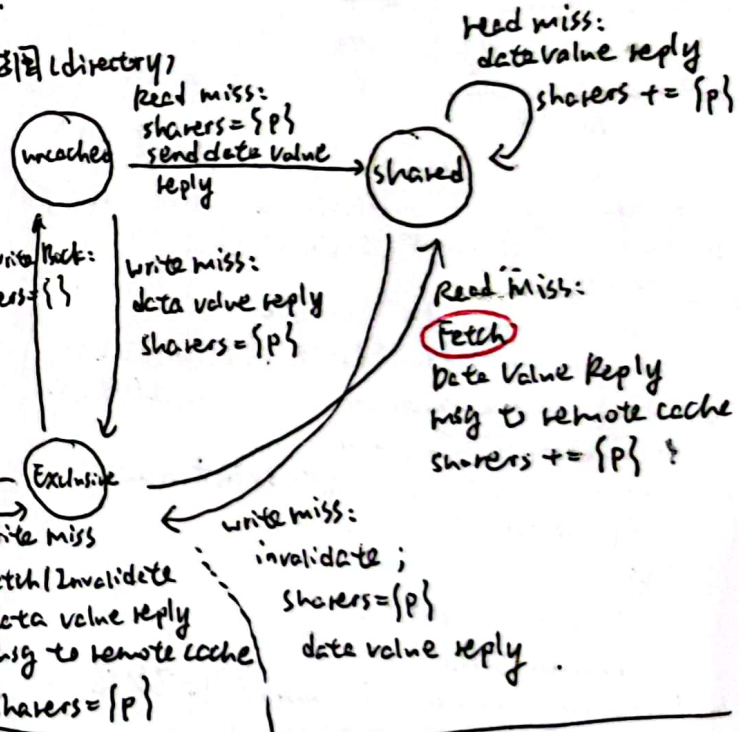
会改变其他核的状态

Fetch: h.d. → R.N.

获取 data

data WB: R.N → h.d.

状态图（directory）



3. 同步

原子交换：如果交换得到 1 说明在锁的

test-and-set：读取目标地址，如果是 0 就写入

fetch-and-increment

① spinlock：将 lock 读入 cache，拥有者一修改

（会让他人发送 invalidate，然后大家中某个拿到

性能：i 个核 read miss 和 lock=0，i 个核 write miss，

1 个 write 成功，2b·sum(2i+1)

② barrier：

```
local_sense = !local_sense;
lock(counterlock)
counter = counter + 1
if (counter == total) {
    counter = 0
```

$\sum_{i=1}^{n-1}(3i+4) = \frac{3n^2+11n}{2} - 1$

sort：指数后退 / 合同树

排队锁 3n-1

```
local_sense = !local_sense;
fetch_and_inc(count)
```

place write miss on bus

CPU read/
write hit

data WB and send
write miss to h.d.

Fetch/Invalidate
data value reply
msg to remote cache
Sharers={P}

Sharers={P}
data value reply.

状态图（总结例） 其他核

例子：

	P1 state	P1 addr	P1 val	P2 state	P2 addr	P2 val	bus action	bus proc	bus addr	bus val	memory addr	memory val
P1 write 10 to A1	EX	A1	10				WM	P1	A1			
P1 read A1	EX	A1	10									
P2 read A1				SH	A1		RM	P2	A2			
	SH	A1	10				WB	P1	A1	10	A1	10
				SH	A1	10	RD	P2	A1	10	A1	10
P2 write 20 to A1	Invalid			EX	A1	20	WM	P2	A1		A1	10
P2 write 20 to A2							WM	P2	A2		A1	10
				EX	A2	20	WB	P2	A1	20	A1	20

① directory-based schemes

三种状态：shared / uncached / exclusive

简化：write hit/miss 都视作 write miss

核心思想：读变成 shared. 写的话先向有copy的处理器发 invalidate 信号

等响应后才能继续修改 .

三种节点 local 发起的请求 / home: memory所在 / remote: 有copy（不管EX或SH）

3. 同步

原子交换：如果交换得到1说明不成功

test-and-set：读取目标地址，如果是0就写入

fetch-and-increment

① spinlock：将lock读入cache，拥有者一修改会给其他人发送 invalidate，然后大家中某个拿到

性能：i个核 read miss 知 lock=0，i个核 write miss，1个 write 成功. sum(2i+1)

$= n^2+n$

② barrier:

```
local_sense = ! local_sense ;
lock(counterlock)
counter = counter + 1
if (counter == total) {
    counter = 0
    release = local_sense
}
unlock(counterlock)
spin(release == local_sense)
```

$\Sigma(3i+4)-1 = \frac{3n^2+11n}{2}-1$

改进：指数后退 / 合并树

排队锁 3n-1

```
local_sense = ! local_sense ;
fetch_and_inc(count)
if (count == total) { ... }
else { spin ... }
```

3n

4. consistency (Order)

sequential consistency → total store order → partial store order → weak ordering → release consistency

R→R, R→W, W→R, W→W　　R→R, R→W, W→W　　R→R, R→W