共享存储系统结构

Shared Memory Architecture

胡伟武

高等教育出版社
HIGHER EDUCATION PRESS

共享存储系统结构

Shared Memory Architecture

高等教育出版社
HIGHER EDUCATION PRESS

内容提要

本书介绍了作者研究共享存储系统结构中的关键问题。主要内容包括:建立了一个共享存储系统的执行正确性模型,讨论了正确执行的访存次序的条件,提出一个在顺序一致的共享存储系统中实现乱序执行的方案,并对该方案进行了模拟。建立了一个描述存储一致性模型的数学模型,给出了并行程序及系统结构是否满足某种存储一致性的标准。提出了一个基于锁的高速缓存一致性协议,该协议比传统的目录协议具有更好的性能和可伸缩性,实现了一个虚拟共享存储系统 JIAJIA,该系统实现基于锁的一致性协议,并可把多机内存空间组织成大共享空间。本书重点对存储一致性模型、高速缓存一致性协议以及共享虚拟存储等,进行了系统的论述。

图书在版编目(CIP)数据

共享存储系统结构:全国优秀博士学位论文/胡伟武.
—北京:高等教育出版社,2001
ISBN 7-04-009849-0

Ⅰ.共… Ⅱ.胡… Ⅲ.存储器共享-研究-文集
Ⅳ.TP333-53

中国版本图书馆 CIP 数据核字(2001)第 10482 号

共享存储系统结构
胡伟武

出版发行	高等教育出版社		
社　　址	北京市东城区沙滩后街 55 号	邮政编码	100009
电　　话	010-64054588	传　　真	010-64014048
网　　址	http://www.hep.edu.cn http://www.hep.com.cn		
经　　销	新华书店北京发行所		
排　　版	高等教育出版社照排中心		
印　　刷	高等教育出版社印刷厂		
开　　本	787×960　1/16	版　　次	2001 年 7 月第 1 版
印　　张	9.25	印　　次	2001 年 7 月第 1 次印刷
字　　数	160 000	定　　价	15.50 元
插　　页	1		

本书如有缺页、倒页、脱页等质量问题,请到所购图书销售部门联系调换。

作者简介

胡伟武，男，1968 年 11 月出生于浙江永康，1991 年 7 月毕业于中国科技大学计算机系，随后免试进入中国科学院计算技术研究所直接攻读博士学位，师从著名计算机专家夏培肃院士，1996 年 3 月博士毕业并获工学博士学位。现任中国科学院计算技术研究所研究员。主要研究方向为高性能计算机系统结构、并行处理、集成电路设计等。目前已在国内外杂志及会议上发表论文 40 余篇。曾先后荣获“中国科学院科技进步二等奖”、“中国科学院院长奖学金特别奖”、“全国首届优秀博士论文奖”以及“中国科学院盈科优秀青年学者奖”等奖励。

通讯地址：北京 2704 信箱 25 分箱（邮政编码：100080）
中国科学院计算技术研究所
电　　话：010 - 62559641(O)
传　　真：010 - 62567724
E-mail: hww@ ict. ac. cn
http: //www. ict. ac. cn/chpc/hww

导师简介

夏培肃，女，1923 年 7 月出生于重庆。1945 年毕业于中央大学电机系，1950 年获英国爱丁堡大学博士学位。中国科学院计算技术研究所研究员。我国计算技术创始人之一。1956 年参加建立我国计算技术的规划，之后为我国计算技术的起步培养了数百名专业科技人员。1959 年创建中国科学技术大学计算机专业。先后培养了约 40 名博士研究生和硕士研究生。在 20 世纪 50 年代，曾试制成功我国第一台自主设计的通用电子数字计算机。在 20 世纪 60 年代，解决了数字信号在大型高速计算机中传输的某些关键问题。在 20 世纪 80 年代，负责设计研制的高速阵列处理机使石油勘探中的基本地震资料处理速度在原有基础上提高 10 倍以上，并负责研制成功多台不同类型的高性能计算机。曾先后创办《计算机学报》和《Journal of Computer Science and Technology》，并担任第一任主编。1985 年被英国 Heriot - Watt 大学授予荣誉科学博士学位。1991 年当选为中国科学院院士。

前　言

随着单处理机的性能越来越接近物理极限，当前的高性能计算机系统都采用并行处理结构。共享存储多处理机中的存储系统有着不同于其他计算机存储系统的特征，问题集中体现在访存事件及其发生次序上。针对这一问题，本书从访存事件次序的角度系统地研究了共享存储系统中维护数据一致性、提高性能和增加系统的可伸缩性等方面的问题。

全书共分八章。第一章介绍并行处理系统的类型，指出从存储管理及编程界面的角度看，并行处理系统可分为共享存储多处理机系统和消息传递多计算机系统。而且，根据存储器的分布、一致性的维护以及实现方式等特征，将当前常见的共享存储系统的体系结构进行了划分。第二章利用集合论的方法建立了一个共享存储系统的执行正确性模型。在研究冲突访问对执行结果的重要影响的基础上，定义了共享存储系统中执行的概念。并指出判断一个并行执行正确与否的标准是其结果是否等于同一程序在单机多进程环境下的某一执行的结果，决定一个执行结果的关键因素是此执行中冲突访问的执行次序。第三章讨论正确执行的访存事件次序条件。通过把任一访存操作分成若干子操作，建立了一个共享存储访问模型，给出较通用的满足顺序一致性的访存事件次序条件，并在顺序一致性的一个典型实现的正确性的基础上，推出了一种乱序执行的方案。第四章讨论在基于目录的 Cache 一致性协议中访存事件次序条件的实现方法，还建立了一个地址流驱动的模拟模型来评价不同的访存事件次序条件对性能的影响，并指出乱序执行的作用只有在访存冲突不严重的情况下才能充分发挥。第五章建立了存储一致性模型的一个数学模型，以顺序一致性、释放一致性以及域一致性为例给出了证明存储一致性模型正确实现的方法。第六章讨论高速缓存一致性协议中的一些关键问题，从新值的传播方式、新值的传播时机、新值的来源以及新值的传播对象等不同侧面介绍了 Cache 一致性的关键技术。在此基础上，提出了实现域一致性模型的基于锁的新型 Cache 一致性协议。第七章讨论共享虚拟存储系统中的关键问题和关键技术，介绍了我国自主研制的共

享虚拟存储系统 JIAJIA,还就可编程性和性能两方面对软件分布式共享存储系统 DSM 和消息传递并行环境进行了比较。最后,指出利用软件实现共享存储,并由硬件提供必要的支持,是既能改善系统可编程性,又能有效提高性能的方法。第八章引导读者展望了共享存储系统的发展趋势。

本书对共享存储的关键问题,如存储一致性模型、高速缓存一致性协议、共享虚拟存储等,进行了系统的论述。本书可作为研究生教材使用,对相关技术人员也有参考价值,它对于促进共享存储系统研究领域的教学和更进一步的学术探讨有积极的推动作用。书中若有欠妥之处,欢迎广大读者提出宝贵意见和建议。

编 者

2001 年 2 月

目　录

第1章 引 言

1.1 并行处理系统

当前科学技术迅猛发展，人类对高性能计算技术的需求越来越大，高性能计算技术在航空航天、石油勘探和开发、大范围气象预报、核爆炸模拟、材料设计、药物设计、基因信息学、密码学、人工智能、经济模型、数字电影等领域起着重要的作用。高性能计算机可以对所研究的对象进行数值模拟和动态显示，获得实验不易得到甚至得不到的结果，从而产生了理论科学和实验科学之外的第三类科学，即计算科学。可以说，在信息技术高度发达、国际竞争日益激烈的今天，高性能计算机的发展水平是一个国家国力的重要标志。

由于单处理机的性能越来越接近物理极限，当前的高性能计算机系统都采用并行处理结构，从系统结构的角度来看，并行处理系统可以分为以下几类：

1. 并行向量机系统 以 CRAY 系列为代表的向量机在 20 世纪 70 年代和 20 世纪 80 年代前期曾经是高性能计算机发展的主流，在商业、金融、科学计算等领域发挥了重要作用，积累了大量的应用软件及系统软件，现在仍然被广泛使用。并行向量机系统的缺点是难以达到很高的并行度，因此，以 CRAY 公司被 SGI 公司兼并为标志，向量机已不再是高性能计算机发展的主流。

2. SMP(Symmetric MultiProcessor)系统 又称对称多处理机系统，指若干处理机通过共享总线互连而成的多处理机系统。由于多个处理机通过总线与存储器相连，可以较容易地支持共享存储。同时，由于总线是一种独占性资源，这种系统的可伸缩性是有限的。SMP 系统常被作为一个结点以构成更大的并行系统。

3. MPP(Massive Parallel Processing)系统 指在同一地点由大量处理单元构成的并行计算机系统。各个处理单元可以是单机，也可以是多机系统(如

SMP 系统)。处理单元之间通常由可伸缩的互联网络(如 MESH、交叉开关网络等)相连。

4. 机群系统 将大量同一品种的工作站或微机通过高速网络互相连接,以构成廉价的高性能计算机系统。由于机群系统可以充分利用现有的计算、内存、文件等资源,用较少的投资实现高性能计算,因此,越来越成为普通高性能计算用户青睐的对象。

5. 异构计算机系统 系统中的多台计算机可以有不同的体系结构,它们可以是 MPP 系统,也可能是工作站或其他类型的计算机。格网计算(Grid Computing)系统和元计算(Metacomputing)系统都属于异构计算机系统。

在上述系统中,并行向量机系统及 SMP 系统的技术已经成熟,其中,并行向量机系统正逐步被其他系统所取代。未来的高性能计算系统可能采用后 3 种结构。其中,MPP 系统的研究已有较长的历史,且由于研制费用高,主要由大公司或研究机构研制生产,尤其是超大规模的 MPP 系统(如峰值运算速度在每秒一万亿次以上浮点运算的系统)的研制,通常体现为政府行为,如美国的 ASCI 计划(Accelerated Strategic Computing Initiative)和 CIC 计划(Computing, Information, and Communication Program)等。机群系统和异构计算机系统是网络计算的不同分支,都是当前的研究热点。通常,机群计算是指在局域网内的网络计算,而异构计算是在广域网内的网络计算。因此,机群计算的研究比较现实、具体,而异构计算的研究还比较遥远。

目前,高性能计算机系统所面临的主要问题包括:

1. 实际性能差。并行计算机的实际性能通常远远低于其峰值性能。

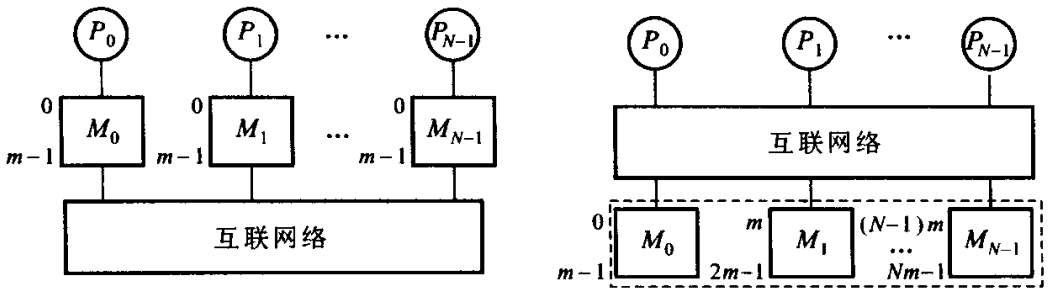
2. 可编程性差。并行程序的开发比较困难,串行程序向并行程序的自动转换效果不好,且不同平台间并行程序的有效移植也有一定的难度。

这些都是 MPP 系统长期没有很好解决的问题,尤其在机群系统中,由于网络带宽有限、缺少有效的并行计算环境等原因,上述问题更加突出。

1.2 消息传递与共享存储

从存储管理及编程界面的角度看,并行处理系统可分为共享存储多处理机系统和消息传递多计算机系统两类。在共享存储多处理机系统中,所有处理机共享主存储器,每个处理机都可以把信息存入主存储器,或从主存储器中取出信息,处理机之间的通信通过访问共享存储器来实现。而在消息传递多计算机系统中,每个处理机都有一个只有它自己才能访问的局部存储器,处理机之间的通信必须通过显式的消息传递来进行。消息传递多计算机和共享存储多处理机的结构示意图如图 1.1 所示。从图中可以看出,在消息传递多计算机系统中,每个

处理机的存储器是单独编址的;而在共享存储多处理机系统中,所有存储器统一编址。



(a) 消息传递多计算机 (b) 共享存储多处理机

图 1.1 消息传递多计算机与共享存储多处理机结构

与消息传递系统相比,共享存储系统由于支持传统的单地址编程空间,减轻了程序员的编程负担,因此,共享存储系统具有较强的通用性,且可以方便地移植已有的应用软件。然而,在共享存储系统中,多个处理机对同一地址空间的共享也带来了一些问题。共享必然会引起冲突,从而使共享存储器成为系统运行中的瓶颈。为此,在规模较大的共享存储系统中,把共享存储器分成许多模块并分布于各处理机之中(这类系统称为分布式共享存储系统)。此外,共享存储系统都采用 Cache 来缓和由共享引起的冲突以及由存储器分布造成的长延迟对性能的影响。然而,存储器的分布引起非一致的访存结构 NUMA(Non-Uniform Memory Access),即不同处理机访问同一存储单元可能有不同的延迟。而 Cache 的使用又带来了 Cache 一致性问题,即如何保证同一单元在不同 Cache 中备份的数据一致。

访存时间的不一致、同一单元的多个备份都破坏了存储访问的不可分割性(atomicity),使得同一单元内容的变化在不同的时刻被不同的处理机所认识,影响了系统的正确性。为了保证正确性,需要对访存操作的发生次序进行严格的限制,许多在单处理机中行之有效的提高性能的技术,如流水、多发射、预取、缓存等,不能在共享存储系统中仿效,这不利于提高系统性能。同时,维持 Cache 一致性需要复杂的硬件构件,影响了共享存储系统的可伸缩性。

可见,共享存储多处理机系统中的存储系统有着不同于其他计算机存储系统的特征,带来了一些新问题。目前,国际业界同仁在这些问题上尚无保证系统的正确性、可伸缩性以及系统性能的圆满解决方案。我国研制高性能计算机也深受这些问题的困扰。因此,必须对分布式共享存储系统的体系结构进行深入研究,在维护分布式共享存储系统的数据一致性、提高系统的性能和增加系统的可伸缩性等方面提出创新的解决方案。

在前述 5 类并行处理系统中,前两类系统都是共享存储系统,后两类网络计

算系统都是消息传递系统。大多数 MPP 系统是消息传递系统。共享存储 MPP 系统的典型代表是 SGI 公司的 Origin 2000,但与同期的消息传递产品相比,Origin 2000 由于硬件的复杂性,其可伸缩性也是有限的。此外,CRAY-T3D 等系统也提供了共享空间,但硬件不负责维护 Cache 一致性。

1.3 常见的共享存储系统

根据共享存储器的分布,共享存储系统可分为集中式和分布式两类。在集中式共享存储系统中,多个处理机通过总线、交叉开关或多级互联网络等与共享存储器相连,所有处理机访问存储器时都有相同的延迟。随着处理机数量的增加,集中式共享存储器很容易成为系统运行中的瓶颈。在分布式共享存储(Distributed Shared Memory,简称 DSM)系统中,共享存储器分布于各结点之间(一个结点可能有一个或多个处理机),即每个结点包含共享存储器的一部分。结点之间通过伸缩性好的互联网络(如 MESH)相连。分布式的存储器和可伸缩的互联网络增加了访存带宽,但导致非一致的访存结构 NUMA。集中式和分布式的共享存储系统又可以分别分成若干类。

根据存储器的分布、一致性的维护以及实现方式等特征,当前常见的共享存储系统的体系结构有以下几种:

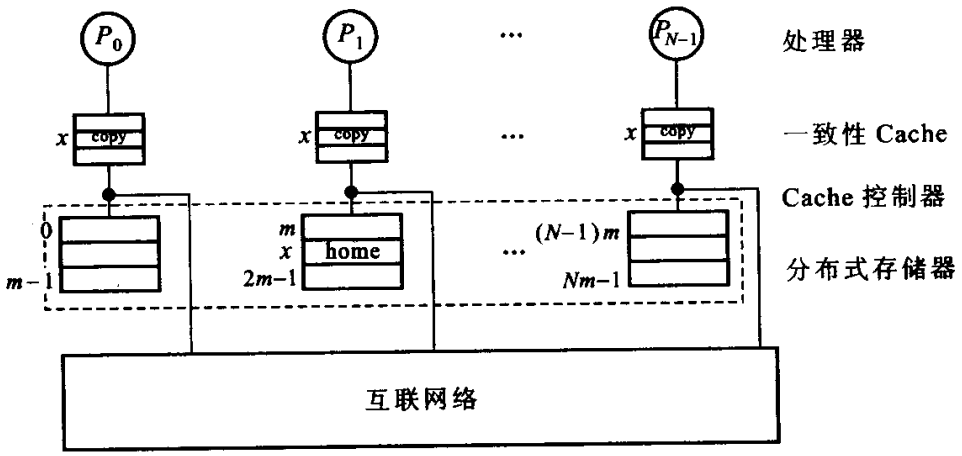
1. 无 Cache 结构 这种系统的处理机没有 Cache,多个处理机通过交叉开关或多级互联网络等直接访问共享存储器。由于任一存储单元在系统中只有一个备份,这类系统不存在 Cache 一致性问题,系统的可伸缩性受限于交叉开关或多级互联网络的带宽。采用这种结构的典型例子是并行向量机及一些大型机,如 CRAY-XMP、YMP-C90 等。此外,无 Cache 的结构还见于早期的分布式共享存储系统中,如 CMU 的 Cm^*、BBN 公司的 Butterfly 和 Illinois 州的 CEDAR 等。

2. 共享总线结构 即 SMP 系统所采用的结构。在这类系统中,每个处理机都有 Cache,多个处理机通过总线与存储器相连。每个处理机的 Cache 通过侦听总线来维持数据一致性。由于总线是独占性资源,这类系统的伸缩性是有限的。这种结构常见于服务器和工作站中,如 DEC 公司、SUN 公司、Sequent 公司以及 SGI 公司等的多机工作站产品均属此类。

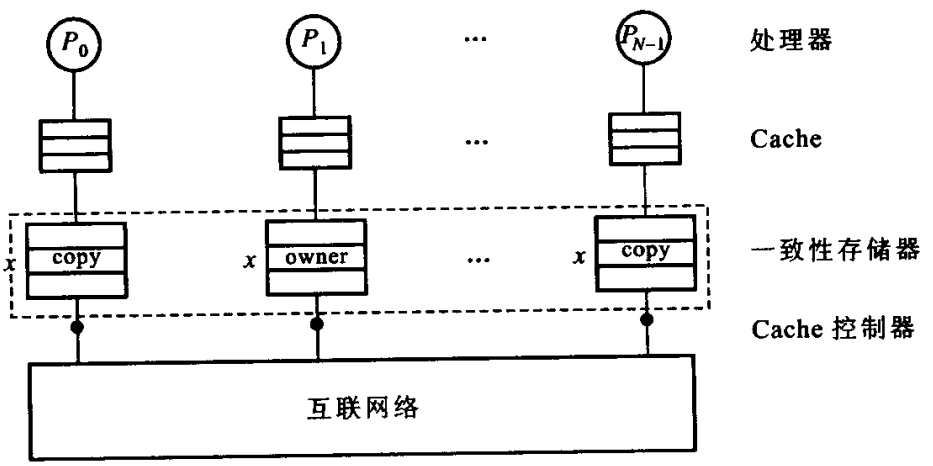
3. CC-NUMA(Cache-Coherent Non-Uniform Memory Access)结构 即 Cache 一致的分布式共享存储系统。这类系统的共享存储器分布于各结点之间,结点通过伸缩性好的互联网络相连,每个处理机都能缓存共享单元,通常采用基于目录的方法来维持处理机之间的 Cache 一致性。Cache 一致性的维护是这类系统的关键,决定着系统的可伸缩性。这类系统的例子有美国斯坦福大学

的 DASH 系统和 FLASH 系统，美国麻省理工学院的 Alewife 系统，以及 SGI 公司的 Origin 2000，等等。

4. COMA(Cache-Only Memory Architecture)结构 即唯 Cache 结构。这类系统的共享存储器的地址是活动的。存储单元与物理地址分离，数据可以根据访存模式动态地在各结点的存储器间移动和复制。每个结点的存储器相当于一个大容量 Cache，数据一致性也在这一级维护。这类系统的优点是当处理机的访问不在 Cache 命中时，在本地共享存储器命中的概率较高。其缺点是当处理机的访问不在本结点命中时，由于存储器的地址是活动的，需要一种机制来查找被访问单元的当前位置，因此延迟时间很长。目前采用唯 Cache 结构的系统有 Kendall Square Research 的 KSR1 和瑞典计算机研究院的 DDM(Distributed Data Management，分布式数据管理系统)。此外，COMA 结构常用于软件 DSM 系统中。图 1.2 描述了 CC-NUMA 和 COMA 结构的主要区别。



(a) CC-NUMA结构



(b) COMA结构

图 1.2 CC-NUMA 与 COMA 结构

5. NCC-NUMA(Non-Cache-Coherent Non-Uniform Memory Access)结构 即 Cache 不一致的分布式共享存储系统。其典型代表是 CRAY 公司的 T3D 及 T3E 系列产品,这种系统的特点是虽然每个处理机都有 Cache,但硬件不负责维护 Cache 一致性。Cache 一致性由编译器或程序员来维护。在 T3D 和 T3E 产品中,系统为用户提供了一些用于同步的库函数,便于用户通过设置临界区等手段来维护数据一致性。这样做的好处是加强系统伸缩性,高档的 T3D 及 T3E 产品可含上千个处理机。

6. 共享虚拟存储(Shared Virtual Memory,简称 SVM)结构 又称软件 DSM 系统。基于结合共享存储系统的可编程性和消息传递系统的硬件的简单想法,共享虚拟存储系统在基于消息传递的 MPP 或机群系统中,用软件的方法把分布于各结点的多个独立编址的存储器组织成一个统一编址的共享存储空间。其优点是在消息传递的系统上实现共享存储的编程界面,主要问题是难以获得令人满意的性能。与硬件共享存储系统相比,软件 DSM 中较大的通信和共享粒度(通常是存储页)会导致假共享及额外的通信。此外,在基于机群的软件 DSM 中,通信开销很大。与消息传递系统如 MPI(Message Passing Interface,消息传递接口)相比,基于软件 DSM 的并行程序通信量通常比基于消息传递的并行程序的通信量大。然而,最近软件 DSM 技术和网络技术的发展使得软件 DSM 的性能得到了极大提高。一方面,诸如弱一致性协议的懒惰实现技术(lazy release consistency)[64]以及多写(multiple writer)[23]协议等针对软件 DSM 的优化措施的提出,大大减少了软件 DSM 中的假共享及额外通信。另一方面,网络技术的发展降低了系统性能对通信量的敏感程度。研究表明,对于大量的应用程序,软件 DSM 的性能在消息传递系统性能的 80% 以上[76]。此外,软件 DSM 可以有效利用硬件支持。如在 SMP 机群系统中,可以在结点内利用 SMP 硬件提供的共享存储,在结点间由软件实现共享存储。又如,软件 DSM 可充分利用某些互联网络实现的远程 DMA(Direct Memory Access,直接存储器存取)功能提高远程访问的速度。常见的虚拟共享存储系统有 Ivy[75]、Midway[18]、Munin[23]、Treadmarks[65]和 JIAJIA[54]等。

1.4 本书的组织和内容

在共享存储系统中,多个处理机对同一地址空间的共享大大增加了系统复杂性,对系统正确性和系统性能产生了重要影响。首先,传统的执行正确性标准不再适用。在单处理机系统中,只有一个处理机访问存储器,取数操作总是取回"最近"一个对同一单元的存数操作所写的值,而存数操作惟一地确定"此后"对同一单元的取数操作所取回的值。在这种访存模式下,只要程序中的数据相关

性不被破坏,就可以乱序执行指令而不影响执行的正确性。而在共享存储系统中,多个处理机同时访问共享存储器。为了缓解共享引起的访存冲突,分布式共享存储系统把共享存储器分成许多模块并分布于各处理机之中,而且采用Cache来缓和由共享引起的冲突以及由存储器分布引起的长延迟对性能的影响。然而,存储器的分布引起非一致的访存结构NUMA,Cache的使用又使得同一数据在系统中有多个备份。访存时间的不一致以及同一单元的多个备份破坏了存储访问的不可分割性(Atomicity),使得同一单元内容的变化在不同的时刻被不同的处理机所识别,在单机系统中的"最近"、"此后"的概念不复存在。例如,在分布式系统中,处理机 P_i 在 t_1 时刻之前发出的取数操作,完全有可能取回处理机 P_j 在 t_1 之后所写的值。

此外,为了保证正确性,需要对访存操作的发生次序进行严格的限制,许多在单处理机中行之有效的提高性能的技术,如流水、多发射、预取、缓存等,不能在共享存储系统中仿效,这不利于提高系统性能。同时,维持Cache一致性需要复杂的硬件构件,影响了共享存储系统的可伸缩性。

可见,共享存储多处理机中的存储系统有着不同于其他计算机存储系统的特征,带来了一些新问题,集中体现在访存事件及其发生次序上。针对这一问题,本书从访存事件次序的角度系统地研究了共享存储系统的正确性(数据一致性)、提高性能和增加系统的可伸缩性等方面的问题。

第二章利用集合论的方法建立了一个共享存储系统的执行正确性模型。这一章从一个简单的程序模型开始,首先讨论了串行执行的正确性。然后,在研究冲突访问对执行结果的重要影响的基础上,定义了共享存储系统中执行的概念。并根据一个正确的并行执行的结果应等于同一程序的一个串行执行的结果这种顺序一致性的要求,讨论并证明了使一个并行执行正确的充要条件。

第三章讨论正确执行的访存事件次序的条件。它根据在分布式共享存储系统中访存操作可分割的特点,通过把任一访存操作分成若干子操作,建立了一个共享存储访问模型。并在该模型以及第二章建立的执行正确性模型的基础上,给出并证明了一个较通用的保证正确执行的访存次序条件。根据这个条件,一方面证明了顺序一致性的一个典型实现的正确性(该实现要求系统中所有处理机严格根据指令在程序中出现的次序执行指令);另一方面提出了一种在满足顺序一致性要求的系统中实现乱序执行的方案,证明了只要满足一定条件,一个访存操作就可以越过它前面的指令执行而不会破坏顺序一致性。

第四章把前两章对执行正确性模型与访存事件发生次序的研究同一个具体的基于目录的Cache一致性协议结合起来,讨论在基于目录的Cache一致性协议中实现访存事件次序条件的方法,并提出通过猜测执行实现第三章提出的乱序执行的方案。本章还建立了一个地址流驱动的模拟模型来评价不同的访存事

件次序条件对性能的影响。模拟结果表明,本书提出的乱序执行方案能有效地提高系统性能。

第五章建立了存储一致性模型的一个数学模型。该模型可以看做是把前几章建立的执行正确性模型及访存正确性条件由顺序一致性推广到其他一致性模型的情况。与传统的从访存事件次序的角度来刻画存储一致性模型的方法不同,该模型从存储一致性模型体现出的行为的角度来描述存储一致性模型。由于一个共享存储程序的并行执行的行为是由该程序中冲突访问的执行次序决定的,因此本章把存储一致性模型定义为规定不同处理机间访存操作执行次序的一种同步机制。在一个执行中,程序序以及同步操作的执行次序决定该执行的最终行为。此外,本章还从访存事件次序的角度建立了判断并行程序以及系统结构是否满足某种存储一致性模型的标准。

第六章讨论高速缓存一致性协议中的一些关键问题。高速缓存一致性协议的本质是把某个处理机新写的值传播给其他处理机以确保所有处理机看到一致的共享存储内容,是对某种存储一致性模型的具体实现。基于这种认识,本章先从不同的侧面介绍如何实现高效的传播,并在此基础上,提出了一个实现域存储一致性模型的基于锁的新型 Cache 一致性协议并证明其正确性。同传统的基于目录的协议相比较,基于锁的协议通过附带在锁上的一致性信息来维护一致性,从而避免由目录引起的存储开销和系统复杂度。与目录协议相比,基于锁的协议更加简单、有效且可伸缩性好。

第七章讨论共享虚拟存储系统中的关键问题。首先,从实现技术、数据一致性以及编程界面等侧面介绍了共享虚拟存储系统的最新进展。然后,介绍了中国科学院计算技术研究所实现的共享虚拟存储系统 JIAJIA。JIAJIA 实现了第六章提出的基于锁的新型一致性协议,采用类似于 NUMA 的存储器组织方式,把多个机器的存储器组织起来形成一个更大的共享空间。同时,JIAJIA 还实现了若干优化策略来避免假共享、减少通信次数和通信量,以及容忍或隐藏通信延迟。采用一些被广泛使用的测试程序对 JIAJIA 进行的测试表明,同近期实现的共享虚拟存储系统(如 CVM)比较,JIAJIA 不仅具有更高的性能,而且可以解决更大规模的问题;基于 JIAJIA 的并行程序性能与基于消息传递环境 PVM(Parallel Virtual Machine,并行虚拟机)并行程序的性能相当;JIAJIA 的优化措施可以有效地降低系统开销,提高性能。此外,本章还就可编程性和性能两方面对软件 DSM 和消息传递并行环境进行了比较,指出在机群系统中,利用软件实现共享存储,并由硬件提供必要的支持,是既能改善系统可编程性,又能有效提高系统性能的方法。

第八章总结全书内容并展望了共享存储系统的未来发展趋势。

第2章 执行正确性模型

2.1 引　　言

在单机系统中，只有一个处理机读写存储器，其访存事件次序比较简单：取数操作总是取回最近对同一单元的存数操作所写的值，而存数操作惟一地确定此后对同一单元的取数操作所取回的值。这种简单的访存事件模型使单机系统得以高效地运行，目前的大多数单机系统都采用流水、多发射、预取、写缓存等技术来提高性能。只要程序中的数据相关性不被破坏，就可以乱序执行指令而不影响执行的正确性。

而在共享存储系统中，多个处理机都可以读写同一单元，一个处理机所存的数可能被多个处理机所访问，甚至(尤其是在分布式共享存储系统中)一个单元内容的变化可能在不同的时刻被不同的处理机所接受。因此，在共享存储系统中，访存操作的次序比较复杂，为保证正确地执行指令，不仅要考虑单机内的数据相关性，而且要考虑多机之间的数据相关性。

例 2.1　图 2.1 的程序段 PRG_1 是保证只有一个进程进入临界区的一种同步机制。其中，变量 a 是指示进程 P_1 是否进入临界区的标志，$a=0$ 表示 P_1 未进入临界区，$a=1$ 表示 P_1 已进入临界区。变最 b 是进程 P_2 是否进入临界区的标志。当一个进程试图进入临界区时，它首先把本进程的标志置为“1”，然后检查另一进程的标志。若另一进程的标志为“0”，说明另一进程未进入临界区，则本进程进入临界区；否则，本进程等待，直到另一进程退出临界区为止。当 P_1 和 P_2 分别执行完相应的进程时，R_1 和 R_2 的值的正确组合是(0,1)、(1,0)和(1,1)，其中最后一种情况将导致死锁(假设有其他方法来防止死锁)。只有 $R_1=R_2=0$ 的结果是错误的，它将导致 P_1 和 P_2 同时进入临界区。

如果忽略多机间的数据相关而只考虑单机内的数据相关，则 L_{12} 可以先于 L_{11} 执行而不破坏数据相关性。同样，L_{22} 可以先于 L_{21} 执行而不破坏数据相关性。这就会导致 $R_1 = R_2 = 0$ 的错误结果。

可见，在共享存储系统中，为了保证执行指令的正确性，每个处理机都必须根据指令在程序中出现的次序来执行指令。然而，在分布式共享存储系统中，仅仅根据程序序来执行指令还不足以保证执行指令的正确性。

Process P_1	Process P_2
L_{11}: STORE a,1	L_{21}: STORE b,1
L_{12}: LOAD R_1,b	L_{22}: LOAD R_2,a

图 2.1 程序段 PRG_1(初始值 $R_1 = R_2 = a = b = 0$)

例 2.2 在图 2.2 的程序 PRG_2 中，如果仅要求进程 P_1、P_2 及 P_3 根据指令在程序中出现的次序来执行指令，那么这个程序的访存事件可能按如下次序发生：

1. P_1 发出存数操作 L_{11}。
2. L_{11} 到达 P_2，但由于网络堵塞等原因，L_{11} 未到达 P_3。
3. P_2 发出取数操作 L_{21} 取回 a 的新值。
4. P_2 发出存数操作 L_{22}，且其所存的 b 新值到达 P_3。
5. P_3 发出取数操作 L_{31} 取回 b 的新值。
6. P_3 发出取数操作 L_{32}，但由于 L_{11} 未到达 P_3，故 L_{32} 取回 a 的旧值。
7. L_{11} 到达 P_3。

这是一个程序员难以接受的执行，因为从程序员的观点来看，如果 L_{21} 和 L_{31} 分别取回 a 和 b 的新值，则说明存数操作 L_{11} 和 L_{22} 都已完成，L_{32} 必然取回 a 的新值。

Process P_1	Process P_2	Process P_3
L_{11}: STORE a,1	L_{21}: LOAD R_1,a	L_{31}: LOAD R_2,b
	L_{22}: STORE b,1	L_{32}: LOAD R_3,a

图 2.2 程序段 PRG_2(初始值 $R_1 = R_2 = R_3 = a = b = 0$)

在此例中，即使每个处理机都根据指令在程序中出现的次序来执行指令，仍然会导致错误的结果。

可见，在共享存储多处理机系统中不存在单机系统中的全局绝对时钟，难以惟一确定在取数操作之前的“最近”一个对同一单元的存数操作。传统的执行正确性标准已不适用于共享存储系统中复杂的访存行为，因此需要为共享存储系统制定新的执行正确性标准。在参考文献[71]中，Lamport 提出了顺序一致性(Sequential Consistency)模型，这一模型后来被普遍地认为是判断一个并行执行是否正确的标准。该模型指出，如果在共享存储系统中多机并行执行程序的效

果等于把每个处理机所执行的指令流按某种方式顺序地交织在一起后在单机上执行,则该共享存储系统是顺序一致的。

通常,执行正确性是通过对访存事件的发生次序加以限制来保证的。本章利用集合论中序关系的一些基本概念和结果,系统地研究共享存储系统中访存事件的发生次序,并从访存事件次序的角度建立了共享存储系统的执行正确性模型。

本章的第 2 节介绍了集合论中序关系的一些基本概念,第 3 节给出程序模型,第 4 节描述了作为并行执行的正确性标准的串行执行正确性条件,第 5、6 节研究共享存储系统中的执行正确性条件,第 7 节是本章小结。

2.2 序关系的一些基本概念

在本书中,使用参考文献[36]中集合论的基本定义和结果:

1. 设 V 是一个集合,则 $V\times V$ 的一个子集 R 称为集合 V 上的一个(二元)关系。R 中的一个元素 (x,y) 记为 $(x,y)\in R$,或 $x \xrightarrow{R} y$。

2. 如果集合 V 上的关系 R 满足下列条件:

(1) R 是一个传递关系。

(2) 对于集合 V 中的任意两个元素 x 和 y:

$$(x,y)\in R,\ x=y,\ (y,x)\in R$$

有且仅有一个成立,其中"$x=y$"表示 x 和 y 是集合 V 中的相同元素。则称 R 为 V 上的一个全序关系。

对于集合 V 中的任一元素 x:

$$(x,x)\in R,\ x=x,\ (x,x)\in R$$

有且仅有一个成立。由于 $x=x$ 显然成立,因此 $(x,x)\in R$ 不可能成立,满足这个条件的关系称为非自反关系。显然,全序关系是非自反关系。

3. 如果集合 V 上的关系 R 满足下列条件:

(1) R 是一个传递关系。

(2) R 是非自反关系。

则称 R 为 V 上的一个偏序关系。

4. 若 R 是集合 V 上的全序关系或偏序关系,则称二元组 $<V,R>$ 为一个序结构。

5. 关系 R 中如下形式的多个元素:

$$(x_1,x_2),(x_2,x_3),\cdots,(x_{n-1},x_n),(x_n,x_1),\quad n\geqslant 1$$

称为 R 中的一个圈。

6. 若 R 是集合 V 上的一个全序关系或偏序关系，则 R 中无圈。

假设 R 中有一个圈：

$$(x_1, x_2),(x_2, x_3),\cdots,(x_{n-1}, x_n),(x_n, x_1)$$

由 R 的传递性，有 $(x_1, x_1)\in R$，违背了 R 的非自反性。

7. 若 V 是一个有限集合，R 是 V 上的一个无圈关系，则 V 中至少有一个元素 m 满足下述条件：

$$\forall x\in V,(x,m)\notin R \tag{2.1}$$

假设满足式(2.1)的元素 m 不存在，则对 V 中的任一元素 x_0，V 中必然存在元素 x_{-1}，使得 $(x_{-1},x_0)\in R$。同样，对于 x_{-1}，V 中必然存在另一元素 x_{-2}，使得 $(x_{-2},x_{-1})\in R$。依此类推，可得到 R 中如下的元素序列：

$$\cdots,(x_{-i}, x_{-(i-1)}),\cdots,(x_{-2}, x_{-1}),(x_{-1}, x_0)$$

由于 V 是一个有限集合，上述序列中至少有两个元素 x_j 和 x_k 是相同的，即 $x_j = x_k$。而这与 R 中无圈的条件相矛盾。因此，式(2.1)所要求的元素 m 肯定存在。

2.3 程 序 模 型

从本质上说，一个程序包含一个指令集以及在此指令集上的一个序关系。每条指令规定程序要执行的操作，而序关系规定这些操作的发生次序。为了研究访存次序，对程序做如下假设：

1. 一个程序由多个进程组成，进程是指令的有限序列。进程中只有两种指令，即取数指令 LOAD 和存数指令 STORE。LOAD 指令的格式为：

$$\mathrm{L}_i:\mathrm{LOAD}\quad R, v$$

其中，L_i 是指令的标记(程序中每条指令都有惟一的标记)。这条指令的含义是把存储单元 v 中的数取到寄存器 R 中去。

STORE 指令的格式为：

$$\mathrm{L}_i:\ \mathrm{STORE}\quad v, R/C$$

这条指令的含义是把常数 C 或寄存器 R 中的内容存到存储单元 v 中去。

2. 每个寄存器或存储单元都有一个初始值，缺省值为 0。

3. 程序的任一执行的结果由程序中出现的所有寄存器和存储单元的最终值来决定。

在上述假设中，一个进程只包含 LOAD 和 STORE 两种指令。对于描述处理机和存储器之间的交互作用来说，这种假设是合理的。因为从处理机和存储系统的界面上看，任何复杂的进程都表现为存数指令和取数指令的有限序列。

上述程序模型可形式地用定义 2.1 来表述。

定义 2.1 一个进程 P 是一个序结构$\langle V(P), PO(P)\rangle$,其中 $V(P)$是访存指令的集合,$PO(P)$是 $V(P)$上的一个全序关系。

由 N 个进程$P_1, P_2, \cdots, P_N$ 组成的程序$PRG(P_1, P_2, \cdots, P_N)$是一个序结构$\langle V(PRG), PO(PRG)\rangle$,其中,$V(PRG) = V(P_1) \cup V(P_2) \cup \cdots \cup V(P_N)$是程序 PRG 的指令集,$PO(PRG) = PO(P_1) \cup PO(P_2) \cup \cdots \cup PO(P_N)$是 PRG 的程序序。

例 2.3 图 2.1 中的程序 PRG_1 可以用序结构$\langle V(PRG_1), PO(PRG_1)\rangle$来表示,其中:

$$V(PRG_1) = \{L_{11}, L_{12}, L_{21}, L_{22}\}$$

$$PO(PRG_1) = \{(L_{11}, L_{12}), (L_{21}, L_{22})\}$$

$$= L_{11} \xrightarrow{PO} L_{12}, L_{21} \xrightarrow{PO} L_{22}$$

2.4 串行执行的正确性

Lamport 在参考文献[71]中提出的顺序一致的概念给出了在共享存储系统中判断一个执行正确与否的标准。其主要思想是,如果在多处理机环境下的一个并行执行的结果等于同一程序在单处理机多进程环境下的一个执行的结果,则此并行执行正确。可见,串行执行是判断并行执行正确与否的基础,定义串行执行如定义 2.2 所述。

定义 2.2 对任一程序 $PRG = \langle V(PRG), PO(PRG)\rangle$, $V(PRG)$的一个全序称为程序 PRG 的一个串行执行,记为 $SE(PRG)$。

根据上述定义,并非一个程序的所有串行执行都是正确的。只有那些结果符合程序员的期望的串行执行是正确的。程序员总是期望指令按照它们在程序中出现的先后次序被执行。因此,一个串行执行 $SE(PRG)$如果与程序序 $PO(PRG)$一致,那么这个串行执行肯定正确。也就是说,若 $SE(PRG) \cup PO(PRG)$无圈,则 $SE(PRG)$正确。

例 2.4 图 2.1 中的程序 PRG_1 有如下 24 个可能的串行执行:

$$SE_1(PRG_1) = L_{11} \xrightarrow{SE} L_{12} \xrightarrow{SE} L_{21} \xrightarrow{SE} L_{22}$$

$$SE_2(PRG_1) = L_{11} \xrightarrow{SE} L_{21} \xrightarrow{SE} L_{12} \xrightarrow{SE} L_{22}$$

$$SE_3(PRG_1) = L_{11} \xrightarrow{SE} L_{21} \xrightarrow{SE} L_{22} \xrightarrow{SE} L_{12}$$

$$SE_4(PRG_1) = L_{21} \xrightarrow{SE} L_{22} \xrightarrow{SE} L_{11} \xrightarrow{SE} L_{12}$$

$$SE_5(PRG_1)=L_{21}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{12}$$

$$SE_6(PRG_1)=L_{21}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{22}$$

$$SE_7(PRG_1)=L_{12}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{22}$$

$$SE_8(PRG_1)=L_{12}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{22}$$

$$SE_9(PRG_1)=L_{12}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{11}$$

$$SE_{10}(PRG_1)=L_{21}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{11}$$

$$SE_{11}(PRG_1)=L_{21}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{11}$$

$$SE_{12}(PRG_1)=L_{21}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{22}$$

$$SE_{13}(PRG_1)=L_{11}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{21}$$

$$SE_{14}(PRG_1)=L_{11}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{21}$$

$$SE_{15}(PRG_1)=L_{11}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{12}$$

$$SE_{16}(PRG_1)=L_{22}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{12}$$

$$SE_{17}(PRG_1)=L_{22}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{12}$$

$$SE_{18}(PRG_1)=L_{22}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{21}$$

$$SE_{19}(PRG_1)=L_{12}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{21}$$

$$SE_{20}(PRG_1)=L_{12}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{21}$$

$$SE_{21}(PRG_1)=L_{12}\xrightarrow{SE}L_{22}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{11}$$

$$SE_{22}(PRG_1)=L_{22}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{11}$$

$$SE_{23}(PRG_1)=L_{22}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{21}\xrightarrow{SE}L_{11}$$

$$SE_{24}(PRG_1)=L_{22}\xrightarrow{SE}L_{12}\xrightarrow{SE}L_{11}\xrightarrow{SE}L_{21}$$

其中

$$SE_i(PRG_1)\cup PO(PRG_1)=SE_i(PRG_1),\quad i=1,2,\cdots,6$$

无圈，表示前6个串行执行正确。

值得注意的是，并不是所有与程序序的并集有圈的串行执行都是错误的。在例2.4中，虽然 $SE_7(PRG_1)\cup PO(PRG_1)$ 中有一圈 $L_{12}\xrightarrow{SE}L_{11}\xrightarrow{PO}L_{12}$，但 $SE_7(PRG_1)$ 仍产生正确结果 $R_1=0, R_2=a=b=1$。

2.5 并行执行的正确性

顺序一致性提供了判断一个并行执行正确与否的标准。它规定一个正确的并行执行必须得到与串行执行相同的结果。通常,一个执行的正确性是通过对访存事件的发生次序加以限制来保证的。为了找到保证任一执行正确的最松限制,必须研究正确执行区别于错误执行的本质特征。

众所周知,在单处理机中,为了保证执行的正确性,程序中的数据相关性不能被破坏。如果两条访存指令访问的是同一变量且其中至少有一条是存数指令,则这两条指令是数据相关的。在单处理机中,程序执行的结果是由程序中数据相关的指令的执行次序惟一决定的。

在多处理机中,与单处理机中"数据相关"概念相对应的是"冲突访问"[85]的概念,它的定义如定义 2.3 所述。

定义 2.3 在共享存储多处理机系统中,如果两个访存操作访问的是同一单元且其中至少有一个是存数操作,则称这两个访存操作是冲突的。

例 2.5 在图 2.1 的程序 PRG_1 中,L_{11} 和 L_{22} 是冲突访问,因为它们都访问单元 a 且 L_{11} 是存数操作。同样,L_{12} 和 L_{21} 是冲突访问,

不难理解,在并行执行中,冲突访问的执行次序决定执行结果。也就是说,在一个并行执行中,一旦互相冲突的访问的执行次序确定了,那么执行结果也就确定了[85]

定义 2.4 程序 PRG 中冲突访问对的集合记为 $C(PRG)$,$C(PRG)=\{(u, v)\mid(u, v\in V(PRG))\cap(u, v$ 是冲突访问$)\}$。

例 2.6 图 2.2 中的程序 PRG_2 的冲突访问对集合为:

$$C(PRG_2)=\{(L_{11}, L_{21}),(L_{11}, L_{32}),(L_{31}, L_{22})\}$$

定义 2.5 在程序 PRG 中,对冲突访问对集合 $C(PRG)$ 的任一无圈定序称为程序 PRG 的一个执行,记为 $E(PRG)$。

$E(PRG)$ 是 $C(PRG)$ 的一个定序是指,对任意 $(u, v)\in C(PRG)$,$(u, v)\in E(PRG)$ 和 $(v, u)\in E(PRG)$ 有且仅有一个成立。

例 2.7 设 S_1、S_2 和 S_3 是程序 PRG 中访问同一单元的 3 个存数操作,则程序 PRG 的冲突访问对集合为:

$$C(PRG)=\{(S_1, S_2), (S_2, S_3), (S_1, S_3)\}$$

$C(PRG)$ 有如下 8 种可能的定序:

1. $\{(S_1, S_2),(S_2, S_3),(S_1, S_3)\}$
2. $\{(S_1, S_2),(S_2, S_3),(S_3, S_1)\}$

3. $\{(S_1, S_2),(S_3, S_2),(S_1, S_3)\}$

4. $\{(S_1, S_2),(S_3, S_2),(S_3, S_1)\}$

5. $\{(S_2, S_1),(S_2, S_3),(S_1, S_3)\}$

6. $\{(S_2, S_1),(S_2, S_3),(S_3, S_1)\}$

7. $\{(S_2, S_1),(S_3, S_2),(S_1, S_3)\}$

8. $\{(S_2, S_1),(S_3, S_2),(S_3, S_1)\}$

在上述 $C(PRG)$的8个定序中,第2个和第7个含圈,由定义2.5可知,它们不是程序 PRG 的执行。其他6个定序是程序 PRG 的执行。

例 2.8 图2.1中,程序 PRG_1 的冲突访问对集合为:

$$C(PRG_1)=\{(L_{11},L_{22}),(L_{12},L_{21})\}$$

因此,程序 PRG_1 的可能的执行为:

$$E_1(PRG_1)=\{(L_{11},L_{22}),(L_{12},L_{21})\}$$
$$E_2(PRG_1)=\{(L_{22},L_{11}),(L_{12},L_{21})\}$$
$$E_3(PRG_1)=\{(L_{11},L_{22}),(L_{21},L_{12})\}$$
$$E_4(PRG_1)=\{(L_{22},L_{11}),(L_{21},L_{12})\}$$

根据顺序一致性的要求,判断一个执行正确的条件是其运行结果等于一个正确的串行执行的结果,而执行本身不必是串行的。因此,可以把正确的执行定义为定义2.6。

定义 2.6 对于程序 PRG 的一个执行 $E(PRG)$,如果存在 PRG 的一个正确串行执行 $SE(PRG)$,使得 $SE(PRG)\cup PO(PRG)$ 无圈,且 $E(PRG)$ 和 $SE(PRG)$ 的结果相等,则称 $E(PRG)$ 是程序 PRG 的一个正确执行。

引理 2.1 若 $E(PRG)$ 是程序 PRG 的一个执行,且 $E(PRG)\cup PO(PRG)$ 无圈,则存在程序 PRG 的一个串行执行 $SE(PRG)$,使得:

$$E(PRG)\cup PO(PRG)\subseteq SE(PRG)$$

证明 可以按如下方法构造串行执行 $SE(PRG)$:

(1) 令 $SE(PRG)$ 为空集,即 $SE(PRG)=\{\}$。

(2) 由于 $V(PRG)$ 中的关系 $E(PRG)\cup PO(PRG)$ 无圈,$V(PRG)$ 中至少有一个元素 m 满足下述条件:

$$\forall x\in V,(x,m)\notin(E(PRG)\cup PO(PRG))$$

从 $V(PRG)$ 中删除 m,从 $E(PRG)\cup PO(PRG)$ 中删除所有满足 $(m,x)\in(E(PRG)\cup PO(PRG))$ 的 (m,x)。对 $V(PRG)$ 中剩下的所有元素 x,把 (m,x) 加到 $SE(PRG)$ 中去。

(3) 重复步骤(2)直到 $E(PRG)\cup PO(PRG)=\{\}$。

不难看出,根据上述方法构造的 $SE(PRG)$ 是一个全序关系且满足 $(E(PRG)$

$\cup PO(PRG))\subseteq SE(PRG)$。

例 2.9 对于例 2.8 中的执行 $E_1(PRG_1)$：

$$E_1(PRG_1)\cup PO(PRG_1)=\{(L_{11},L_{22}),(L_{12},L_{21})\}\cup\{(L_{11},L_{12}),(L_{21},L_{22})\}$$
$$=\{(L_{11},L_{22}),(L_{12},L_{21}),(L_{11},L_{12}),(L_{21},L_{22})\}$$

无圈，故 $E_1(PRG_1)\cup PO(PRG_1)$能按照如下方法展开成一种全序关系：

(1) 初始时：

$V(PRG_1)=\{L_{11},L_{12},L_{21},L_{22}\}$

$E_1(PRG_1)\cup PO(PRG_1)=\{(L_{11},L_{22}),(L_{12},L_{21}),(L_{11},L_{12}),(L_{21},L_{22})\}$

$SE(PRG_1)=\{\}$

(2) $V(PRG_1)$中的元素 L_{11}满足：

$$\forall x\in V(PRG_1),(x,L_{11})\notin(E_1(PRG_1)\cup PO(PRG_1))$$

从 $V(PRG_1)$中删除 L_{11}，从 $E_1(PRG_1)\cup PO(PRG_1)$中删除(L_{11},L_{22})和(L_{11},L_{12})。把(L_{11},L_{12}),(L_{11},L_{21})和(L_{11},L_{22})加到 $SE(PRG_1)$中去。得到

$V(PRG_1)=\{L_{12},L_{21},L_{22}\}$

$E_1(PRG_1)\cup PO(PRG_1)=\{(L_{12},L_{21}),(L_{21},L_{22})\}$

$SE(PRG_1)=\{(L_{11},L_{12}),(L_{11},L_{21}),(L_{11},L_{22})\}$

(3) $V(PRG_1)$中的元素 L_{12}满足：

$$\forall x\in V(PRG_1),(x,L_{12})\notin(E_1(PRG_1)\cup PO(PRG_1))$$

从 $V(PRG_1)$中删除 L_{12}，从 $E_1(PRG_1)\cup PO(PRG_1)$中删除(L_{12},L_{21})。把(L_{12},L_{21})和(L_{12},L_{22})加到 $SE(PRG_1)$中去。得到

$V(PRG_1)=\{L_{21},L_{22}\}$

$E_1(PRG_1)\cup PO(PRG_1)=\{(L_{21},L_{22})\}$

$SE(PRG_1)=\{(L_{11},L_{12}),(L_{11},L_{21}),(L_{11},L_{22}),(L_{12},L_{21}),(L_{12},L_{22})\}$

(4) $V(PRG_1)$中的元素 L_{21}满足：

$$\forall x\in V(PRG_1),(x,L_{21})\notin(E_1(PRG_1)\cup PO(PRG_1))$$

从 $V(PRG_1)$中删除 L_{21}，从 $E_1(PRG_1)\cup PO(PRG_1)$中删除(L_{21},L_{22})。把(L_{21},L_{22})加到 $SE(PRG_1)$中去。得到

$V(PRG_1)=\{L_{22}\}$

$E_1(PRG_1)\cup PO(PRG_1)=\{\}$

$SE(PRG_1)=\{(L_{11},L_{12}),(L_{11},L_{21}),(L_{11},L_{22}),(L_{12},L_{21}),(L_{12},L_{22}),(L_{21},L_{22})\}$

(5) $E_1(PRG_1)\cup PO(PRG_1)$为空,$SE(PRG_1)$构造结束。

容易看出,上面构造的 $SE(PRG_1)$是程序 PRG_1 的一个串行执行且满足:

$$E_1(PRG_1)\cup PO(PRG_1)\subseteq SE(PRG_1)$$

既然一个执行的结果取决于程序中冲突访问的执行次序,那么此执行的正确性也一定由程序中冲突访问的执行次序来决定。定理 2.1 从访存次序的角度给出了执行正确的充要条件。

定理 2.1 程序 PRG 的执行 $E(PRG)$正确的充要条件是 $E(PRG)\cup PO(PRG)$无圈。

证明: 首先证明充分性。如果 $E(PRG)\cup PO(PRG)$无圈,由引理 2.1 可知,存在 $V(PRG)$上的一个全序(即程序 PRG 的一个串行执行,记为 $SE(PRG)$),使得

$$E(PRG)\cup PO(PRG)\subseteq SE(PRG)$$

由于 $SE(PRG)\cup PO(PRG)=SE(PRG)$无圈,$SE(PRG)$是程序 PRG 的一个正确的串行执行。此外,由于 $E(PRG)\subseteq SE(PRG)$,$SE(PRG)$和 $E(PRG)$对 $C(PRG)$有相同的定序。故 $SE(PRG)$和 $E(PRG)$对程序 PRG 有相同的执行结果 。由定义 2.6 可知,$E(PRG)$是一个正确执行。

然后证明必要性。若 $E(PRG)$是程序 PRG 的一个正确执行,则存在程序 PRG 的一个正确的串行执行 $SE(PRG)$,使得 $SE(PRG)$和 $E(PRG)$的结果相同,即 $SE(PRG)$和 $E(PRG)$对程序 PRG 的冲突访问对集合 $C(PRG)$有相同的定序。故 $E(PRG)\subseteq SE(PRG)$。此外,由于 $SE(PRG)$正确,$SE(PRG)\cup PO(PRG)$无圈 ,因此 $E(PRG)\cup PO(PRG)$也无圈。

例 2.10 在图 2.1 中的程序的 4 个执行中,由于

$$E_1(PRG_1)\cup PO(PRG_1)=\{(L_{11},L_{22}),(L_{12},L_{21})\}\cup\{(L_{11},L_{12}),(L_{21},L_{22})\}$$
$$=\{(L_{11},L_{22}),(L_{12},L_{21}),(L_{11},L_{12}),(L_{21},L_{22})\}$$

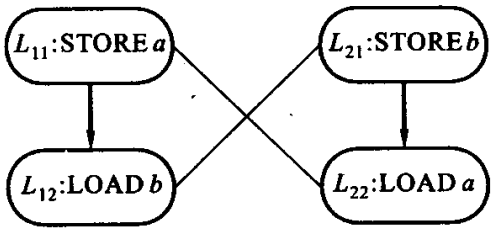
$$E_3(PRG_1)\cup PO(PRG_1)=\{(L_{11},L_{22}),(L_{21},L_{12}),(L_{11},L_{12}),(L_{21},L_{22})\}$$

$$E_4(PRG_1)\cup PO(PRG_1)=\{(L_{22},L_{11}),(L_{21},L_{12}),(L_{11},L_{12}),(L_{21},L_{22})\}$$

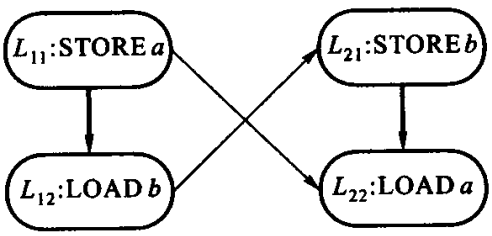
无圈,因而 $E_1(PRG_1)$、$E_3(PRG_1)$和 $E_4(PRG_1)$是正确的执行。而

$$E_2(PRG_1)\cup PO(PRG_1)=\{(L_{22},L_{11}),(L_{12},L_{21})\}\cup\{(L_{11},L_{12}),(L_{21},L_{22})\}$$
$$=\{(L_{11},L_{12}),(L_{12},L_{21}),(L_{21},L_{22}),(L_{22},L_{11})\}$$

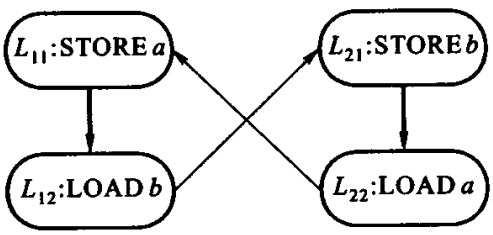
有一个圈 $L_{11}\xrightarrow{PO}L_{12}\xrightarrow{E}L_{21}\xrightarrow{PO}L_{22}\xrightarrow{E}L_{11}$，因此 $E_2(PRG_1)$不正确。



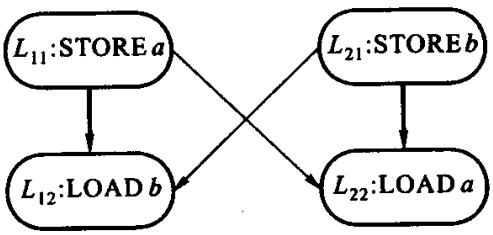
(a) $C(PRG_1)$



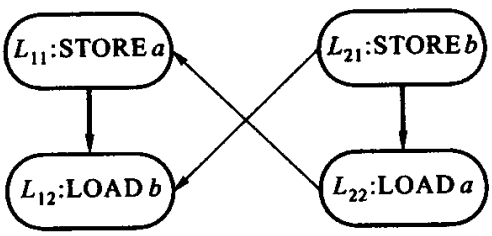
(b) $E_1(PRG_1)(R_1=0,R_2=1)$



(c) $E_2(PRG_1)(R_1=R_2=0)$



(d) $E_3(PRG_1)(R_1=R_2=1)$



(e) $E_4(PRG_1)(R_1=1,R_2=0)$

图 2.3 程序 PRG_1 的冲突访问对集和可能的执行

图 2.3 以图形的方式反映了上述分析。在图 2.3 以及后续的图中，用粗箭头表示程序序，细箭头表示执行次序，细线表示冲突边，椭圆内包含一个访存操作。图 2.3(a)中显示出程序 PRG_1 中的冲突关系，图 2.3(b)～图 2.3(e)分别表示 $C(PRG_1)$的 4 种定序及其执行结果。在图 2.3(c)中有一圈 $L_{11}\xrightarrow{PO}L_{12}\xrightarrow{E}L_{21}\xrightarrow{PO}L_{22}\xrightarrow{E}L_{11}$，表示执行 $E_2(PRG_1)$不正确。

2.6 关 键 圈

错误的执行会引起 $PO(PRG)\cup E(PRG)$中含圈，本节通过关键圈的概念

讨论组成 $PO(PRG)\cup E(PRG)$ 中圈的边和结点的特征。

定义 2.7 若 $\sigma(PRG)$ 是 $PO(PRG)\cup E(PRG)$ 中的一个圈，且 $PO(PRG)\cup E(PRG)$ 中无 $\sigma(PRG)$ 的弦，则称 $\sigma(PRG)$ 是 $PO(PRG)\cup E(PRG)$ 中的一个关键圈。

(u_1,u_n) 是 σ 中的一条弦是指，如果 $(u_1,u_n)\in PO(PRG)\cup E(PRG)$ 且在 σ 中有一条从 u_1 到 u_n 的如下路径：

$$(u_1,u_2),(u_2,u_3),\cdots,(u_{n-1},u_n),\quad n>2$$

例 2.11 图 2.4 中的程序 PRG_3 的冲突访问对集合是：

$$C(PRG_3)=\{(L_{11},L_{32}),(L_{13},L_{21}),(L_{13},L_{31}),(L_{21},L_{31})\}$$

$PO(PRG_3)\cup E(PRG_3)$ 中的一个可能的圈为：

$$\sigma_1(PRG_3)=L_{11}\xrightarrow{PO}L_{12}\xrightarrow{PO}L_{13}\xrightarrow{E}L_{21}\xrightarrow{E}L_{31}\xrightarrow{PO}L_{32}\xrightarrow{E}L_{11}$$

在 $PO(PRG_3)\cup E(PRG_3)$ 中含有 $\sigma_1(PRG_3)$ 的两条弦，即 $L_{11}\xrightarrow{PO}L_{13}$ 和 $L_{13}\xrightarrow{E}L_{31}$。因此 $\sigma_1(PRG_3)$ 不是 $PO(PRG_3)\cup E(PRG_3)$ 中的关键圈。$PO(PRG_3)\cup E(PRG_3)$ 中的一个关键圈为：

$$\sigma_2(PRG_3)=L_{11}\xrightarrow{PO}L_{13}\xrightarrow{E}L_{31}\xrightarrow{PO}L_{32}\xrightarrow{E}L_{11}$$

Process P_1	Process P_2	Process P_3
L_{11}: STORE a,1	L_{21}: LOAD R_2,b	L_{31}: STORE b,2
L_{12}: LOAD R_1,c		L_{32}: LOAD R_3,a
L_{13}: STORE b,1		

图 2.4 程序段 PRG_3

由于 $PO(PRG)$ 和 $E(PRG)$ 都无圈，因此 $PO(PRG)\cup E(PRG)$ 的圈中必然包括 $\xrightarrow{PO}$ 和 $\xrightarrow{E}$ 两种边。定理 2.2 揭示了在 $PO(PRG)\cup E(PRG)$ 的关键圈中 $\xrightarrow{PO}$ 和 $\xrightarrow{E}$ 两种边的关系。

定理 2.2 若 $\sigma(PRG)$ 是 $PO(PRG)\cup E(PRG)$ 中的一个关键圈，p 是 $\sigma(PRG)$ 中由连续的 $\xrightarrow{PO}$ 边组成的路径，e 是 $\sigma(PRG)$ 中由连续的 $\xrightarrow{E}$ 边组成的路径，则 p 的长度为 1，即在 $\sigma(PRG)$ 中不可能存在连续的 $\xrightarrow{PO}$ 边；e 的长度为 1 或 2，且只能以 $w\xrightarrow{E}w$，$w\xrightarrow{E}r$，$r\xrightarrow{E}w$，或 $r\xrightarrow{E}w\xrightarrow{E}r$ 的方式出现，其中 r 是读操作，w 是写操作。

证明 由于 $PO(PRG)$ 的传递性，$u_1\xrightarrow{PO}u_2\xrightarrow{PO}u_3$ 可以归约为 $u_1\xrightarrow{PO}u_3$。同时，由于 $\sigma(PRG)$ 是关键圈，因此在 $\sigma(PRG)$ 中不可能有连续的 $\xrightarrow{PO}$ 边。

假设 $u_1 \xrightarrow{E} u_2 \xrightarrow{E} \cdots \xrightarrow{E} u_n$ 是 $\sigma(PRG)$ 中由连续的 $\xrightarrow{E}$ 边组成的路径，由于 $u_i(i=1,2,\cdots,n)$ 是冲突访问，它们访问相同的单元。(a) 若 u_1 是写操作，u_1 与所有的 $u_i(i=2,\cdots,n)$ 冲突。由于 $E(PRG)$ 无圈，$u_1 \xrightarrow{E} u_n$，因此 $n=2$。否则，$u_1 \xrightarrow{E} u_n$ 构成 $\sigma(PRG)$ 中的一条弦。(b) 若 u_1 是读操作，根据冲突访问的定义，u_2 必须是写操作。根据上面(a)的分析，在 $\sigma(PRG)$ 中 u_2 后最多只能有一个由 $\xrightarrow{E}$ 边相连的操作。此外，若 $u_1 \xrightarrow{E} u_2 \xrightarrow{E} u_3$，则 u_3 肯定是一个读操作，否则 $u_1 \xrightarrow{E} u_3$ 将构成 $\sigma(PRG)$ 中的一条弦。

因此，如果 e 是 $\sigma(PRG)$ 中只由 $\xrightarrow{E}$ 边组成的路径，则 e 只能是 $w \xrightarrow{E} w$，$w \xrightarrow{E} r$，$r \xrightarrow{E} w$，或 $r \xrightarrow{E} w \xrightarrow{E} r$ 之一。

从上述定理可以看出，在 $PO(PRG) \cup E(PRG)$ 的关键圈 $\sigma(PRG)$ 中，任一 $\xrightarrow{PO}$ 边的前后都是 $\xrightarrow{E}$ 边。设 $u_1 \xrightarrow{E} u \xrightarrow{PO} v \xrightarrow{E} v_1$ 是 $\sigma(PRG)$ 中的一条路径，则 u_1 和 u 访问同一单元，v 和 v_1 访问同一单元。从执行 $u \xrightarrow{PO} v$ 操作的处理机的角度看，u“观察到”u_1 对同一单元的访问而 v“被”访问同一单元的 v_1“观察到”。因此，把 u 称做 $\sigma(PRG)$ 上的观察操作，把 v 称做 $\sigma(PRG)$ 上的被观察操作。如果 u 和 v 是 $\sigma(PRG)$ 上的两个观察(或被观察)操作且在 $\sigma(PRG)$ 上从 u 到 v 的路径中没有其他观察(或被观察)操作，则称 u 和 v 为 $\sigma(PRG)$ 上两个相邻的观察(或被观察)操作。

例 2.12 在例 2.11 的关键圈 $\sigma_2(PRG_3) = L_{11} \xrightarrow{PO} L_{13} \xrightarrow{E} L_{31} \xrightarrow{PO} L_{32} \xrightarrow{E} L_{11}$ 中，没有连续的 $\xrightarrow{PO}$ 边，由连续的 $\xrightarrow{E}$ 边组成的路径形式为 $w \xrightarrow{E} w$ 或 $r \xrightarrow{E} w$。

图 2.2 中的程序 PRG_2 的冲突访问对集合是：

$$C(PRG_2) = \{(L_{11}, L_{21}), (L_{11}, L_{32}), (L_{22}, L_{31})\}$$

例 2.2 给出了该程序的一个错误执行，该错误执行可描述为：

$$E(PRG_2) = \{(L_{11}, L_{21}), (L_{22}, L_{31}), (L_{32}, L_{11})\}$$

在 $PO(PRG_2) \cup E(PRG_2)$ 中有一个圈：

$$\sigma(PRG_2) = L_{11} \xrightarrow{E} L_{21} \xrightarrow{PO} L_{22} \xrightarrow{E} L_{31} \xrightarrow{PO} L_{32} \xrightarrow{E} L_{11}$$

该圈是 $PO(PRG_2) \cup E(PRG_2)$ 中的一个关键圈。在 $\sigma(PRG_2)$ 中没有连续的 $\xrightarrow{PO}$ 边，由连续的 $\xrightarrow{E}$ 边组成的路径形式为 $w \xrightarrow{E} r$ 和 $r \xrightarrow{E} w \xrightarrow{E} r$。

定理 2.2 揭示了 $PO(PRG) \cup E(PRG)$ 的关键圈中 $\xrightarrow{PO}$ 和 $\xrightarrow{E}$ 边的特征。下面从访存操作的角度，分析 $PO(PRG) \cup E(PRG)$ 的关键圈的特征。

定义 2.8 若 $\sigma(PRG)$是 $PO(PRG)\cup E(PRG)$中的一个关键圈，w_1 和 w_2 是圈中的两个存数操作，且在圈上从 w_1 到 w_2 的路径中没有其他存数操作，则称 w_1 和 w_2 是 $\sigma(PRG)$上的两个相邻的存数操作。

定理 2.3 指出了 $PO(PRG)\cup E(PRG)$的关键圈中两个相邻存数操作之间的可能路径。

定理 2.3 若 $\sigma(PRG)$是 $PO(PRG)\cup E(PRG)$中的一个关键圈，w_1 和 w_2 是 $\sigma(PRG)$上两个相邻的存数操作，则 $\sigma(PRG)$上从 w_1 到 w_2 的所有可能路径为：

$$\begin{cases} w_1 \xrightarrow{PO} w_2 \\ w_1 \xrightarrow{E} r \xrightarrow{PO} w_2 \\ w_1 \xrightarrow{PO} r \xrightarrow{E} w_2 \\ w_1 \xrightarrow{E} r_1 \xrightarrow{PO} r_2 \xrightarrow{E} w_2 \\ w_1 \xrightarrow{E} w_2 \end{cases} \tag{2.2}$$

其中，r、r_1 和 r_2 是取数操作。

证明 由于 w_1 和 w_2 相邻且取数操作不相冲突，故 $\sigma(PRG)$上从 w_1 到 w_2 的路径只能有如下形式：

$$w_1 \to r_1 \xrightarrow{PO} r_2 \xrightarrow{PO} \cdots \xrightarrow{PO} r_n \to w_2$$

其中，“$\to$”可以是“$\xrightarrow{PO}$”或“$\xrightarrow{E}$”，$r_i(i=1,2,\cdots,n)$是取数操作。

由于 $PO(PRG)$是一个传递的关系，即 $u_1 \xrightarrow{PO} u_2 \xrightarrow{PO} u_3$ 可以推导出 $u_1 \xrightarrow{PO} u_3$，关键圈中不会有两条连续的$\xrightarrow{PO}$边。因此，$\sigma(PRG)$上从 w_1 到 w_2 的路径中只能有 0 个、1 个或 2 个取数操作。

1. 若圈 $\sigma(PRG)$上从 w_1 到 w_2 的路径中没有取数操作，则有 $w_1 \xrightarrow{PO} w_2$ 或 $w_1 \xrightarrow{E} w_2$。

2. 若圈 $\sigma(PRG)$上从 w_1 到 w_2 的路径中有两个取数操作，这两个取数操作肯定由$\xrightarrow{PO}$连接。由于关键圈中不会有两条连续的$\xrightarrow{PO}$边，因此，$\sigma(PRG)$上从 w_1 到 w_2 的路径只能是 $w_1 \xrightarrow{E} r_1 \xrightarrow{PO} r_2 \xrightarrow{E} w_2$。

3. 若圈 $\sigma(PRG)$上从 w_1 到 w_2 的路径中有 1 个取数操作，则圈上从 w_1 到 w_2 的可能路径为：

(a) $w_1 \xrightarrow{PO} r \xrightarrow{PO} w_2$

(b) $w_1 \xrightarrow{E} r \xrightarrow{PO} w_2$

(c) $w_1 \xrightarrow{PO} r \xrightarrow{E} w_2$

(d) $w_1 \xrightarrow{E} r \xrightarrow{E} w_2$

由于关键圈中不会有两条连续的$\xrightarrow{PO}$边，且从 $w_1 \xrightarrow{E} r \xrightarrow{E} w_2$ 可以推出 $w_1 \xrightarrow{E} w_2$(根据 $E(PRG)$无圈且 w_1 和 w_2 冲突)，故上述第 1、4 种路径违背 $\sigma(PRG)$是关键圈的条件。

总结上述分析可以看出，圈 $\sigma(PRG)$上从 w_1 到 w_2 只有定理 2.3 中指出的 5 种可能路径。

例 2.13 在例 2.11 的关键圈 $\sigma_2(PRG_3)$中，L_{11}和 L_{13}是相邻的存数操作，圈中从 L_{11}到 L_{13}的路径是 $L_{11} \xrightarrow{PO} L_{13}$，这是式(2.2)的第 1 种情况。$L_{13}$和 L_{31}是相邻的存数操作，圈中从 L_{13}到 L_{31}的路径是 $L_{13} \xrightarrow{E} L_{31}$，这是式(2.2)的第 5 种情况。$L_{31}$和 L_{11}是相邻的存数操作，圈中从 L_{31}到 L_{11}的路径是 $L_{31} \xrightarrow{PO} L_{32} \xrightarrow{E} L_{11}$，这是式(2.2)的第 3 种情况。

在例 2.12 的关键圈 $\sigma(PRG_2)$中，L_{11}和 L_{22}是相邻的存数操作，从 L_{11}到 L_{22}的路径是 $L_{11} \xrightarrow{E} L_{21} \xrightarrow{PO} L_{22}$，这是式(2.2)的第 2 种情况。同样，$L_{22}$和 L_{11}是相邻的存数操作，$\sigma(PRG_2)$中从 L_{22}到 L_{11}的路径是 $L_{22} \xrightarrow{E} L_{31} \xrightarrow{PO} L_{32} \xrightarrow{E} L_{11}$，这是式(2.2)的第 4 种情况。

2.7 小 结

判断一个并行执行正确与否的标准是看其结果是否等于同一程序在单机多进程环境下的某一执行的结果。决定一个执行结果的关键因素是此执行中冲突访问的执行次序。程序 PRG 的一个并行执行 $E(PRG)$从本质上说，是程序 PRG 的冲突访问对集合 $C(PRG)$的一个无圈定序。$E(PRG)$正确的充要条件是 $E(PRG) \cup PO(PRG)$无圈。在 $E(PRG) \cup PO(PRG)$中无弦的圈称为 $E(PRG) \cup PO(PRG)$中的关键圈。在 $E(PRG) \cup PO(PRG)$中的一个关键圈中不可能有连续的$\xrightarrow{PO}$边；由连续的$\xrightarrow{E}$边构成的路径只能以 $w \xrightarrow{E} w$，$w \xrightarrow{E} r$，$r \xrightarrow{E} w$，或 $r \xrightarrow{E} w \xrightarrow{E} r$ 的方式出现，其中 r 是读操作，w 是写操作。在 $E(PRG) \cup PO(PRG)$中的一个关键圈中，两个相邻的存数操作之间只有 5 种可能路径。

第3章 正确的访存事件次序

3.1 访问模型

在侦听总线的共享存储系统或者没有 Cache 的共享存储系统中，任一存数操作所写的值同时被系统中所有处理机所接受，这种系统叫做写不可分割(Write Atomic)的共享存储系统。在这种系统中，取数操作和存数操作是决定系统行为的最基本事件。

在分布式共享存储系统中，由于不同处理机访问同一单元的不同延迟以及同一单元在 Cache 中的多个备份，导致同一单元内容的变化可能在不同的时刻被不同的处理机所接受，这类系统称为写可分割(Write Nonatomic)的系统。在这种系统中，抽象的存数操作或取数操作已不足以描述系统行为。因此，需要一个更精确的模型来描述共享存储系统的并行执行中复杂的事件次序，此模型必须反映出同一访存操作在不同的时刻到达不同的处理机这个事实。下面所述的共享存储访问模型 SAM(Shared-memory Access Model)可以精确地表示访存操作可分割这个事实。

定义 3.1 在一个有 N 个处理机的共享存储系统中，任一访存操作 u 可以分割成 N 个子操作 $u^1, u^2 \cdots, u^N$，其中 u^i 表示操作 u 相对于处理机 P_i 完成(或被处理机 P_i 所接受)。一个存数操作相对于处理机 P_i 完成指的是，如果没有对同一单元的其他存数操作，P_i 访问该单元将取回该存数操作所写的值。一个取数操作相对于处理机 P_i 完成指的是，P_i 所写的值将不再影响该取数操作的返回值[38]。子操作不可分割。

在 CC-NUMA 结构的共享存储系统中，若处理机 P_i 的 Cache 中有 x 的备

份,则存数子操作 $w^i(x)$相当于更新 x 在 P_i 中的备份(在写更新协议中),或使 x 在 P_i 中的备份无效(在写使无效协议中)。否则,$w^i(x)$相当于更新 x 在主存中的备份或使 x 在主存中的备份无效。对于读单元 x 的取数操作 $r(x)$,若发出读操作的处理机 P_i 的 Cache 中有 x 的备份,则 P_i 在 Cache 中读 x 时所有 $r(x)$的子操作同时完成。否则,在 P_i 向存储器发读数请求的过程中,在该请求到达存储器之前,若某一进程 P_j 对 x 的修改已不可能影响存储器中 x 的值,则 $r^j(x)$完成,在 P_i 的读数请求到达存储器后,$r(x)$的所有子操作才算都完成。

第二章建立的执行正确性模型指出,共享存储系统中一个执行 $E(PRG)$正确的充要条件是 $E(PRG)\cup PO(PRG)$无圈。通常,为了保证执行的正确性,需要对访存事件的发生次序进行限制。也就是说,若 $u \xrightarrow{E} v$ 或 $u \xrightarrow{PO} v$,则应保证 v 的子操作 $v^1, v^2, \cdots, v^N$ 不同程度地在 u 的子操作 $u^1, u^2, \cdots, u^N$ 之后发生,使得 $E(PRG)\cup PO(PRG)$中不存在形成圈的条件。当然,这种限制越严格,越不利于系统性能的提高。

本章讨论保证正确执行的访存次序条件。第 2 节讨论在 $u \xrightarrow{E} v$ 的情况下,u 和 v 的子操作的访存次序限制;第 3 节给出一个较通用的保证正确执行的访存事件次序条件;第 4 节讨论顺序一致性的一个充分条件,该条件要求所有访存操作必须严格按照程序的次序执行;第 5 节提出一种乱序执行的方案,并证明其正确性;第 6 节通过例子说明乱序执行可能带来的系统性能的提高。为了叙述上的方便,表 3.1 列出了在本章以及后续章节中常用的符号及其含义。

表 3.1 常用符号及其含义

符 号	含 义
$w, w(x)$	写(单元 x)操作
$r, r(x)$	读(单元 x)操作
$u, v, u(x), v(x)$	访问(即读或写)(单元 x)操作
u^i	访存操作 u 被处理机 P_i 所接受
$A \Rightarrow B$	若 A,则有 B
$a < b$	事件 a 发生在事件 b 之前

3.2 写一致条件

在第二章,基于并行程序中冲突访问的执行次序决定执行结果的这种想法,把并行程序在共享存储系统中的一个执行抽象地定义为该程序中所有冲突访问

对的一个定序，而并没有赋予该执行具体的含义。在实际的执行中，若 $u \xrightarrow{E} v$，则根据 u 和 v 是读操作还是写操作，可以分为以下 3 种情况。

若 u 是写操作而 v 是读操作，则 $u \xrightarrow{E} v$ 的含义是 v 读回 u 所写的值(或 v 读回 u 后面的写操作所写的值)。因此，若 P_c 是发出 v 的处理机，则 u^c 在 v^c 之前发生(可以理解为在 P_c 开始执行取数操作 v 之前，u 所写的值已经到达 P_c)，即 $u^c < v^c$。

若 u 是读操作而 v 是写操作，则 $u \xrightarrow{E} v$ 的含义是 P_c 发出的取数操作 u 不能读回写数操作 v 所写的值，即 v 所写的内容对 u 取回的值没有影响。因此，u^c 在 v^c 之前发生(可以理解为在 P_c 执行完取数操作 u 之后，v 所写的值才到达 P_c)，即 $u^c < v^c$。

若 u 和 v 都是写操作，则 $u \xrightarrow{E} v$ 的含义是写操作 u 发生在写操作 v 之前。在分布式共享存储系统中，每个处理机都可能拥有共享存储器的一个备份，一个正确的执行应使同一单元对所有处理机显示出一致的值。这就要求对同一单元的所有存数操作以相同的次序到达所有处理机，即 $u^i < v^i (i = 1,2,\cdots,N)$。该条件称为写一致(Write Consistency，简称 WC)条件[26]。

一个执行 $E(PRG)$ 满足写一致条件是 $E(PRG)$ 正确的前提。在第二章，把一个执行定义为冲突访问对的一个无圈定序，事实上已经隐含了该执行满足写一致条件。否则，若一个执行不满足写一致条件，则对同一单元 x 的两个存数操作 w_1 和 w_2，有可能 w_1 先于 w_2 到达处理机 P_i；而 w_2 先于 w_1 到达处理机 P_j。这样，在执行结束后，x 在 P_i 中的备份为 w_2 所写的值，而 x 在 P_j 中的备份为 w_1 所写的值。上述过程可以描述为 $w_1^i < w_2^i \cap w_2^j < w_1^j$，这就难以确定是 $w_1 \xrightarrow{E} w_2$ 还是 $w_2 \xrightarrow{E} w_1$。因此，在第二章建立的执行正确性模型中，不满足写一致条件的访存事件不能作为一个执行。在参考文献[47]中，建立了一个更细致的执行正确性模型，可以描述不满足写一致条件的访存事件。

上述关于 $u \xrightarrow{E} v$ 的访存事件次序的条件可以表示为：

$$\begin{cases} w_1 \xrightarrow{E} w_2 \Rightarrow w_1^i < w_2^i \\ r \xrightarrow{E} w \Rightarrow r^c < w^c \\ w \xrightarrow{E} r \Rightarrow w^c < r^c \end{cases} \tag{3.1}$$

其中，w、w_1 和 w_2 是写操作，r 是读操作，c 是发出相应取数操作的处理机序号，$i = 1,2,\cdots,N$，N 是系统中处理机的个数。在式(3.1)中，第 2、3 项是自然满足的。因此，为了简便，把上式统称为写一致条件。

值得指出的是，在实际的系统中，对冲突访问的访存次序限制往往比式(3.1)严格得多。一种典型的实现是，让所有对同一单元的访问串行地执行。具体的做法是，在存储器响应一个访存请求的过程中(包括存储器向一些处理机发出写回请求或无效请求等)，其他对同一个存储单元的访问必须等待，即 $u \xrightarrow{E} v \Rightarrow u^i < v^j (i,j=1,2,\cdots,N)$。

3.3 正确执行的访存次序条件

第二章定理 2.1 从访存事件发生次序的角度，给出了共享存储系统中的一个执行正确的充要条件(即 $PO(PRG) \cup E(PRG)$无圈)。然而，由于一个处理机难以知道其他处理机执行访存操作的次序，所以这个条件的实现需要对处理机间的通信和协调付出昂贵的代价。因此，这个充要条件的理论价值比它的实际应用价值更多一些。

参考文献[83]给出了一个可行的共享存储系统中保证执行正确的访存次序条件：在一个访存操作允许被发出之前，同一进程中所有先于它的访存操作都已经“彻底完成”。其中，一个存数操作“彻底完成”是指该操作已相对于所有处理机完成，一个取数操作“彻底完成”是指该取数操作已相对于所有处理机完成(即这个取数操作取回的值已经确定)且写此值的存数操作已彻底完成。根据这个条件，例 2.1 和例 2.2 中的错误执行都不会发生。

定理 3.1 给出了一个更加通用的正确执行的访存次序条件。

定理 3.1 如果程序 PRG 的一个执行 $E(PRG)$满足如下的访存次序条件，则该执行正确。

$$\begin{cases} w_1 \xrightarrow{E} w_2 \xrightarrow{PO} v \Rightarrow w_1^c < v^i \\ w \xrightarrow{E} r \xrightarrow{PO} v \Rightarrow w^c < v^i \\ r \xrightarrow{E} w \xrightarrow{PO} v \Rightarrow r^c < v^i \\ r_1 \xrightarrow{E} w \xrightarrow{E} r_2 \xrightarrow{PO} v \Rightarrow r_1^c < v^i \end{cases} \tag{3.2}$$

其中，w、w_1 和 w_2 是写操作，r、r_1 和 r_2 是读操作，v 可以是写操作或读操作，$i=1,2,\cdots,N$，c 值介于 1 和 N 之间。

证明 用反证法。假设 $E(PRG)$不正确，由定理 2.1 可知，$PO(PRG) \cup E(PRG)$中有圈。

设 $\sigma(PRG)$是 $PO(PRG) \cup E(PRG)$中的一个关键圈，u 和 v 是 $\sigma(PRG)$中两个相邻的被观察操作。根据定理 2.2，在 $\sigma(PRG)$中从 u 到 v 有如下可能的路径：

1. $w_1(=u)\xrightarrow{E}w_2\xrightarrow{PO}v$

2. $w(=u)\xrightarrow{E}r\xrightarrow{PO}v$

3. $r(=u)\xrightarrow{E}w\xrightarrow{PO}v$

4. $r_1(=u)\xrightarrow{E}w\xrightarrow{E}r_2\xrightarrow{PO}v$

根据式(3.2),对于从 u 到 v 的所有可能路径,都有 $u^c<v^i$ 成立,其中 $i=1,2,\cdots,N$,c 值介于 1 和 N 之间。

假设 v_1 是 $\sigma(PRG)$中在 v 之后且与 v 相邻的被观察操作,根据上述过程同样可以推出 $v^c<v_1^i$ 对 $i=1,2,\cdots,N$ 以及 1、N 之间的某个 c 成立。根据"<"的传递性,$u^c<v_1^i$ 同样对 $i=1,2,\cdots,N$ 以及 1、N 之间的某个 c 成立。

继续上述过程可以得到,$u^c<u^i$ 也对 $i=1,2,\cdots,N$ 以及 1、N 之间的某个 c 成立。但这是不可能的。

定理 3.2 如果程序 PRG 的一个执行 $E(PRG)$满足如下的访存次序条件,则该执行正确。

$$\begin{cases} u\xrightarrow{PO}w_1\xrightarrow{E}w_2\Rightarrow u^i<w_2^c \\ u\xrightarrow{PO}w\xrightarrow{E}r\Rightarrow u^i<r^c \\ u\xrightarrow{PO}r\xrightarrow{E}w\Rightarrow u^i<w^c \\ u\xrightarrow{PO}r_1\xrightarrow{E}w\xrightarrow{E}r_2\Rightarrow u^i<r_2^c \end{cases} \tag{3.3}$$

其中,w、w_1 和 w_2 是写操作,r、r_1 和 r_2 是读操作,u 可以是写操作或读操作,$i=1,2,\cdots,N$,c 值介于 1 和 N 之间。

证明 用反证法。假设 $E(PRG)$不正确,由定理 2.1 可知,$PO(PRG)\cup E(PRG)$中有圈。

设 $\sigma(PRG)$是 $PO(PRG)\cup E(PRG)$中的一个关键圈,u 和 v 是 $\sigma(PRG)$中两个相邻的观察操作。根据定理 2.2,在 $\sigma(PRG)$中从 u 到 v 有如下可能的路径:

1. $u\xrightarrow{PO}w_1\xrightarrow{E}w_2(=v)$

2. $u\xrightarrow{PO}w\xrightarrow{E}r(=v)$

3. $u\xrightarrow{PO}r\xrightarrow{E}w(=v)$

4. $u\xrightarrow{PO}r_1\xrightarrow{E}w\xrightarrow{E}r_2(=v)$

根据式(3.3),对于从 u 到 v 的所有可能路径,都有 $u^i<v^c$ 成立,其中 $i=1,2,\cdots,N$,c 值介于 1 和 N 之间。

假设 v_1 是 $\sigma(PRG)$中在 v 之后且与 v 相邻的观察操作，根据上述过程同样可以推出 $v^i < v_1^c$ 对 $i=1,2,\cdots,N$ 以及 1、N 之间的某个 c 成立。根据“<”的传递性，$u^i < v_1^c$ 同样对 $i=1,2,\cdots,N$ 以及 1、N 之间的某个 c 成立。

继续上述过程可以得到，$u^i < u^c$ 也对 $i=1,2,\cdots,N$ 以及 1、N 之间的某个 c 成立。但这是不可能的。

不难看出，如果一个执行 $E(PRG)$的访存事件次序满足如下条件，则该执行的访存事件次序也满足定理 3.1 要求的条件。

$$\begin{cases} u \xrightarrow{E} u_1 \xrightarrow{PO} v \Rightarrow u^c < v^i \\ r \xrightarrow{E} w \Rightarrow r^c < w^i \end{cases} \tag{3.4}$$

其中，w 和 r 分别代表写操作和读操作，u、u_1 和 v 是任意访存操作，$i=1,2,\cdots,N$，c 值在 1 和 N 之间。

同样，如果一个执行 $E(PRG)$的访存事件次序满足如下条件，则该执行的访存事件次序也满足定理 3.2 要求的条件。

$$\begin{cases} u \xrightarrow{PO} u_1 \xrightarrow{E} v \Rightarrow u^i < v^c \\ w \xrightarrow{E} r \Rightarrow w^i < r^c \end{cases} \tag{3.5}$$

在一般的共享存储系统中，对同一共享单元的访存操作通常是顺序执行的。例如，在常见的基于目录的写使无效(Write-Invalidate)一致性协议中(见第四章)，从存储器收到一个处理机的访问请求，一直到存储器对这个请求的服务已经完成的这段时间内，需锁住相应行的目录项。在此期间，来自其他处理机的对同一行的访问请求必须等待。在这种情况下，式(3.4)和式(3.5)中的 $r \xrightarrow{E} w \Rightarrow r^c < w^i$ 和 $w \xrightarrow{E} r \Rightarrow w^i < r^c$ 条件总能得到满足。例如，如果处理器 P_j 的取数操作 r 取回 P_i 的存数操作 w 对单元 a 所写的值，则当 P_j 发出操作请求 r 时，要么 a 不在 P_j 的 Cache 中，要么 a 在 P_j 的 Cache 中的备份已经由于 P_i 发出的存数操作 w 被置为无效。在这两种情况下，P_j 都会向存储器发出取数请求 read(a)，而存储器必须在 P_i 发出的存数请求 write(a)完成后才能对该取数请求做出响应。这就保证了 $w \xrightarrow{E} r \Rightarrow w^i < r^c$ 能够得到满足。

3.4 正确执行的充分条件

参考文献[83]给出了一个共享存储系统中保证执行正确的访存次序条件。它要求处理机等待一个访存操作“彻底完成”后才能发出下一个访存操作请求。该访存次序条件可以表示为：

$$\begin{cases} u \xrightarrow{PO} v \Rightarrow u^i < v^j \\ w \xrightarrow{E} r \xrightarrow{PO} v \Rightarrow w^i < v^j \end{cases} \tag{3.6}$$

其中，w 是写操作，r 是读操作，u、v 是任意访存操作，$i,j=1,2,\cdots,N$，N 是系统中处理机的个数。式(3.6)中的条件称为 GPPO(Globally Performed in Program Order)条件。

不难看出，如果一个执行 $E(PRG)$ 的访存事件次序满足式(3.1)和式(3.6)中的条件，则它也满足式(3.2)中的条件。因此，GPPO 条件是顺序一致的共享存储系统中正确执行的一个充分条件。

上述结论也可以用定理 2.3 来证明，详见参考文献 [50]。

值得注意的是，GPPO 条件是正确执行的充分条件，但它们不是正确执行的必要条件，它们对访存事件发生次序所施加的限制比定理 2.1 中的正确执行的充要条件严格。

例 3.1 在第二章图 2.2 的程序中，3 个进程 P_1、P_2 及 P_3 共享 a 和 b 两个变量。在对这个程序的访存事件发生次序不做任何限制的情况下，如果存数操作 L_{11} 到达处理机 P_3 比到达 P_2 要花更多的时间，那么这个程序的访存事件可能按 $L_{11}^2 < L_{21}^2 < L_{22}^2 < L_{22}^3 < L_{31}^3 < L_{32}^3 < L_{11}^3$ 的次序发生。这是一个错误的执行，其结果($R_1=1, R_2=1, R_3=0$)是程序员所不能接受的。事实上，这个执行序列导致相应的 $PO(PRG)\cup E(PRG)$ 中含有一个圈 $L_{11}\rightarrow L_{21}\rightarrow L_{22}\rightarrow L_{31}\rightarrow L_{32}\rightarrow L_{11}$。

如果这个程序的访存事件发生次序符合 GPPO 条件，那么在存数操作 L_{11} 到达处理机 P_3 之前，处理机 P_2 发出的取数操作 L_{21} 就没有“彻底完成”，因而根据 GPPO 条件的要求，P_2 不得开始执行 L_{22}。这样，如果 L_{21} 和 L_{31} 分别取回 a 和 b 的新值，L_{32} 不可能取回 a 的旧值。

而根据定理 2.1 给出的正确执行的充要条件，P_2 在取数操作 L_{21} 取回 L_{11} 所存的值后，可以马上发出指令 L_{22}。在 P_3 发出的 L_{31} 取回 L_{22} 所写的 b 的值后，$PO(PRG)\cup E(PRG)$ 中有一条从 L_{11} 到 L_{32} 的路径 $L_{11}\xrightarrow{E}L_{21}\xrightarrow{PO}L_{22}\xrightarrow{E}L_{31}\xrightarrow{PO}L_{32}$。若 P_3 在 L_{11} 到达 P_3 之前继续发出指令 L_{32} 并取回 a 的旧值，那么，$L_{32}\xrightarrow{E}L_{11}$。这就在 $PO(PRG)\cup E(PRG)$ 中形成一个圈 $L_{11}\xrightarrow{E}L_{21}\xrightarrow{PO}L_{22}\xrightarrow{E}L_{31}\xrightarrow{PO}L_{32}\xrightarrow{E}L_{11}$。因此，为了保证执行的正确性，$L_{32}$ 必须等待 L_{11} 到达 P_3 后才能执行。

可见，在例 3.1 中，GPPO 条件要求处理机 P_2 在取数操作 L_{21} 取回 L_{11} 所存

的值后，必须等待 L_{11} 到达 P_3 后才能继续执行 L_{22}。而定理 2.1 给出的正确执行的充要条件允许 L_{11} 的传播同 L_{22} 及 L_{31} 的执行并行进行。然而，由于一个处理机在执行过程中并不知道其他处理机执行访存操作的次序，因此定理 2.1 给出的正确执行的充要条件难以实现。在具体实现中，人们通常采用 GPPO 条件。

3.5 乱序执行

GPPO 条件要求每个处理机根据指令在程序中出现的次序来执行它们。这不利于提高系统性能，尤其是在分布式共享存储系统中，一旦所访问的单元不在 Cache 中，处理机等待的时间很长。

不允许指令重叠执行的原因是为了防止错误执行的发生。然而，并非指令的所有重叠执行都会导致错误。在绝大多数情况下，即使对一个程序的访存事件发生次序不做限制，也会产生正确的结果[39]。因此，只要对那些容易引起错误的访存操作的执行次序加以限制，绝大多数访存操作就可以重叠执行而不影响执行的正确性。

事实上，定理 3.1 和定理 3.2 所给出的正确执行的访存事件次序并没有规定每个处理机必须严格按照指令在程序中出现的次序来执行它们。式(3.5)指出，在满足 $w \xrightarrow{E} r \Rightarrow w^i < r^c$ 的前提下(如前所述，在一般的共享存储系统中，对同一共享单元的访存操作通常是顺序执行的，此条件自动得到满足)，如果一个执行 $E(PRG)$ 的访存事件次序符合 $u \xrightarrow{PO} u_1 \xrightarrow{E} v \Rightarrow u^i < v^c$ 的要求，则该执行正确。

上述 $u \xrightarrow{PO} u_1 \xrightarrow{E} v \Rightarrow u^i < v^c$ 条件的具体含义是，如果 u 和 u_1 是进程 P_c 中的两个访存操作，且根据程序序 u_1 在 u 之后(即 $u \xrightarrow{PO} u_1$)、v(v 是 u_1 的冲突访问)在 P_c 发出指令 u_1 后才相对于 P_c 完成(即 $u_1 \xrightarrow{E} v$)，那么，v 必须在 u 的所有子操作都完成后才能相对于 P_c 完成。因此，如果指令 u_1 在 u 之前发出，并且在 u_1 发出之后而 u“彻底完成”之前的这段时间内，u_1 的冲突访问 v 不会相对于 P_c 完成，则 u 和 u_1 的乱序执行并不破坏式(3.5)所要求的访存次序条件 $u \xrightarrow{PO} u_1 \xrightarrow{E} v \Rightarrow u^i < v^c$。

根据上述分析可以得出结论，如果 u 和 u_1 是同一进程 P_c 的两个访存操作且 u 出现在 u_1 之前，在如下条件下 u_1 可以先于 u 执行而不影响执行的正确性：在 u_1 发出之后而 u“彻底完成”之前的这段时间内，任何其他与 u_1 冲突的

访存操作不会相对于 P_c 完成。这个条件称为 GPOO(Globally Performed Out of Order)条件。

上述结论也可以用定理 2.3 来证明,详见参考文献[50]。

由于 GPOO 条件在一定的条件下允许取数操作先于它前面的操作被执行,因而在访存延迟较大的分布式共享存储系统中,它可望比 GPPO 条件带来更高的系统性能。

例 3.2 例 2.1 表明,在程序 PRG_1 中,L_{12}必须等 L_{11}到达处理机 P_2 后才能开始执行,L_{22}必须等 L_{21}到达处理机 P_1 后才能开始执行,这正是 GPPO 条件的要求。可以看出,只要满足 GPPO 条件,就不会产生例 2.1 中指出的错误结果(即例 2.10 中错误执行 $E_2(PRG_1)$的结果)。

而根据 GPOO 条件,只要在 L_{12}发出之后而 L_{11}"彻底完成"之前的这段时间内,单元 b 在处理机P_1 中的备份不被更新,取数操作 L_{12}可以在存数操作 L_{11}完成之前开始执行。同样,只要在 L_{22}发出之后而 L_{21}"彻底完成"之前的这段时间内,单元 a 在处理机P_2 中的备份不被更新,取数操作 L_{22}可以在存数操作 L_{21}完成之前开始执行。可以看出,只要满足上述条件,同样不会产生例 2.1 中指出的错误结果(即例 2.10 中错误执行 $E_2(PRG_1)$的结果)。

3.6 一个乱序执行的例子

本节通过一个例子说明乱序执行所能带来的系统性能的提高。

假设有一个采用写使无效 Cache 一致性协议的共享存储系统。在此系统中,每个 Cache 的每一行都有 3 种状态,即无效状态(INV)、共享状态(SHD)及独占状态(EXC)。若一个存储行被多个处理机所共享,则此存储行处于 CLEAN 状态;若一个存储行被某个处理机所独占,则此存储行处于 DIRTY 状态。表 3.2给出了此系统的一些基本操作的延迟,这些延迟是美国斯坦福大学的 DASH 系统[73]的典型延迟。

表 3.2 一些基本操作的延迟

访存操作	被访问单元的位置	延迟(时钟周期)
LOAD	在 Cache 命中(Cache 行处于共享或独占状态)	1
	不在 Cache 行命中,存储行处于 CLEAN 状态	61
	不在 Cache 行命中,存储行处于 DIRTY 状态	80
STORE	在 Cache 命中(Cache 行处于独占状态)	3
	不在 Cache 行命中,存储行处于 CLEAN 状态	57
	不在 Cache 行命中,存储行处于 DIRTY 状态	86

以图 3.1 中的程序为例，假设初始时 a、c 和 e 在 P_2 中处于 EXC 状态，b 和 d 在 P_1 中处于 SHD 状态。由此可知，L_{11}、L_{13}和 L_{15}不在 P_1 的 Cache 中命中，而 L_{12}和 L_{14}在 P_1 的 Cache 中命中。考虑此程序的如下执行序列：

1. P_1 发出存数操作 L_{11}，由于它所访问的单元 a 被 P_2 所独占，L_{11}不在 P_1 的 Cache 命中。

2. 在 L_{11}完成之前，P_1 发出指令 L_{12}从 Cache 中取回单元 b 的旧值。

3. P_2 执行存数操作 L_{21}。

4. P_2 发出指令 L_{22}，并在 P_1 发出的指令 L_{11}到达 P_2 之前在 P_2 的 Cache 中完成。

5. L_{11}到达 P_2。

6. P_1 执行 L_{13}。

7. P_1 执行 L_{14}。

8. P_1 执行 L_{15}。

上述执行序列对应于如下执行：

$$E(PRG)=\{(L_{12},\ L_{21}),(L_{22},\ L_{11})\}$$

在 $PO(PRG)\cup E(PRG)$中有一个圈

$$\sigma(PRG)=L_{11}\xrightarrow{PO}L_{12}\xrightarrow{E}L_{21}\xrightarrow{PO}L_{22}\xrightarrow{E}L_{11}$$

表明这是一个不正确的执行。

Process P_1			Process P_2		
L_{11}:	STORE	a,1	L_{21}:	STORE	b, 1
L_{12}:	LOAD	R_1, b	L_{22}:	LOAD	R_5, a
L_{13}:	LOAD	R_2, c			
L_{14}:	LOAD	R_3, d			
L_{15}:	LOAD	R_4, e			

图 3.1 乱序执行的例子

如果一个写可分割执行满足 GPPO 条件，则每个访存操作都必须等待它前面的访存操作结束后才能执行。因此，取数操作 L_{12}在 L_{11}到达 P_2 之前不能被发出，而取数操作 L_{22}在 L_{21}到达 P_1 之前不能被发出。在这些限制之下，上述不正确的执行不可能发生。由于 L_{11}、L_{13}和 L_{15}不在 P_1 的 Cache 中命中，而 L_{12}和 L_{14}在 Cache 中命中，P_1 的 5 个访存操作一共需要：

$$86+1+80+1+80=248(\text{个时钟周期})$$

如果一个写可分割执行满足 GPOO 条件，则取数操作 L_{13}、L_{14}和 L_{15}可以在存数操作 L_{11}完成之前开始执行。而且只要在指令 L_{12}发出之后而 L_{11}“彻底

完成"之前的这段时间内,单元 b 在处理机 P_i 中的备份不被更新,取数操作 L_{12} 可以在存数操作 L_{11} 完成之前开始执行。此外,取数操作 L_{14} 和 L_{15} 可以在取数操作 L_{13} 完成之前开始执行。

因此,只要在 P_1 发出的指令 L_{11} 执行期间,P_2 发出的 L_{21} 不更新 b 在 P_1 中备份的值,取数操作 L_{12}、L_{13}、L_{14} 和 L_{15} 就能在存数操作 L_{11} 完成之前开始执行。假设在一个时钟周期内每个处理机只能发出 1 条指令,P_1 的 5 个访存操作一共需要:

$$\text{Max}\{86,2+80,4+80\}=86(\text{个时钟周期})$$

如果 P_2 发出的 L_{21} 在 P_1 发出的 L_{11} 执行期间到达 P_1,取数操作 L_{13}、L_{14} 和 L_{15} 可以在存数操作 L_{11} 完成之前开始执行,而取数操作 L_{12} 必须等到存数操作 L_{11} 完成之后才能开始执行。在这种情况下,P_1 的 5 个访存操作一共需要:

$$86+80=166(\text{个时钟周期})$$

在现实系统中,P_1 在发出指令 L_{11} 时难以预知在 L_{11} 执行过程中是否会收到来自其他处理机的对单元 b 的存数操作,因此,P_1 难以决定是否在 L_{11} 完成之前发出指令 L_{12} 及其后的取数操作。这一问题将在后续章节讨论。

3.7 小 结

本章通过把取数操作和存数操作进一步分割成若干子操作,讨论了保证正确执行的访存次序条件。首先,说明了写一致条件是共享存储系统正确执行的前提。然后,给出一个较通用的访存事件次序条件。根据该条件,证明了传统的正确执行的充分条件(GPPO 条件)的正确性。同时推出一种乱序执行的方案,以克服 GPPO 条件对访存事件发生次序限制过多、不利于提高系统性能的缺点。最后,通过例子说明乱序执行所带来的系统性能的提高。

第4章 访存事件次序的实现

本章通过一个具体的 Cache 一致性协议来讨论访存事件次序在 Cache 一致性协议中的实现。第 1 节介绍基本协议，第 2 节介绍 GPPO 条件在基本协议中的实现，第 3 节介绍乱序执行的实现策略，第 4 节介绍模拟模型，第 5 节给出模拟结果，第 6 节是本章小结。

4.1 基本协议

在此以一个写使无效的位向量目录协议作为基本协议来讨论 Cache 一致性协议中的访存事件发生次序。通常，一个 Cache 一致性协议应包括 3 个方面的内容，即 Cache 行状态、存储行状态以及保持 Cache 一致性的状态转化规则。

4.1.1 Cache 行状态和存储行状态

在基本协议中，Cache 的每一行都有 3 种状态，即无效状态(INV)、共享状态(SHD)及独占状态(EXC)。若 Cache 的某一行处于无效状态，处理机对这一行的取数或存数访问都不命中。若 Cache 的某一行处于共享状态，说明可能还有其他处理机持有这一行的有效备份，处理机对这一行的取数访问可以在 Cache 中完成。若 Cache 的某一行处于独占状态，说明这是此存储行的惟一有效备份，处理机对这一行的取数或存数访问都可以在 Cache 中完成。

在存储器中，每行都有一个相应的目录项。每个目录项有一个 N 位的向量，其中，N 是系统中处理机的个数。位向量中若第 i 位为“1”，表示此存储行在第 i 个处理机 P_i 中有备份。此外，每个目录项有一个改写位，当改写位为“1”时，表示某处理机独占并已改写此存储行，相应的存储行处于 DIRTY 状态。否

则,相应的存储行处于 CLEAN 状态。

例 4.1 图 4.1 显示了基本协议中的 Cache 行状态和存储行状态。在图4.1中,单元 x 在 P_j 和 P_k 的 Cache 中处于共享状态,在 P_i 的 Cache 中处于无效状态,并在存储器中处于 CLEAN 状态。

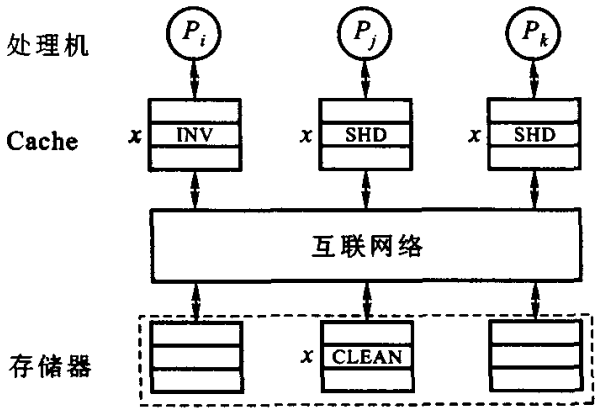


图 4.1 基本协议中的
Cache 行状态和存储行状态

4.1.2 取数操作

当处理机 P_i 发出取数操作"LOAD x"时,根据 x 在 Cache 和存储器中的不同状态采取如下不同的操作:

1. 若 x 在 P_i 的 Cache 中处于共享状态或独占状态,则取数操作"LOAD x"在 Cache 命中。

2. 若 x 在 P_i 的 Cache 中处于无效状态,那么这个处理机向存储器发出一个读数请求 read(x)。存储器在收到这个 read(x)请求后查找与单元 x 相对应的目录项。

(1) 如果目录项的内容显示出 x 所在的存储行处于 CLEAN 状态(改写位为"0"),即 x 在存储器的内容是有效的,那么存储器向发出请求的处理机 P_i 发出读数应答 rdack(x)以提供 x 所在行的一个有效备份,并把目录项中位向量的第 i 位置为"1"。

(2) 如果目录项的内容显示出 x 所在的存储行已被某个处理机 P_k 改写(改写位为"1"),那么存储器向 P_k 发出一个写回请求 wtbk(x)。P_k 在收到 wtbk(x)请求后,把 x 在 Cache 中的备份从独占状态(EXC)改为共享状态(SHD),并向存储器发出写回应答 wback(x)以提供 x 所在行的一个有效备份。存储器收到来自 P_k 的 wback(x)后,向发出请求的处理机 P_i 发出读数应答 rdack(x)以提供 x 所在行的一个有效备份,把目录项中的改写位置为"0",并把

位向量的第 i 位置为“1”。

3. 如果 x 不在 P_i 的 Cache 中，那么 P_i 先从 Cache 中替换掉 1 行再向存储器发出一个读数请求 read(x)。

4.1.3 存数操作

当处理机 P_i 发出存数操作“STORE x”时，根据 x 在 Cache 和存储器中的不同状态采取如下不同的操作：

1. 若 x 在 P_i 的 Cache 中处于独占状态，则存数操作“STORE x”在 Cache 命中。

2. 若 x 在 P_i 的 Cache 中处于共享状态，那么这个处理机向存储器发出一个写数请求 write(x)。存储器在收到这个 write(x)请求后查找与单元 x 相对应的目录项。

(1) 如果目录项的内容显示出 x 所在的存储行处于 CLEAN 状态(改写位为“0”)，并没有被其他处理机所共享(位向量中所有位都为“0”)，那么存储器向发出请求的处理机 P_i 发出写数应答 wtack(x)表示允许 P_i 独占 x 所在行，把目录项中的改写位置为“1”，并把位向量的第 i 位置为“1”。

(2) 如果目录项的内容显示出 x 所在的存储行处于 CLEAN 状态(改写位为“0”)，并且在其他处理机中有共享备份(位向量中有些位为“1”)，那么存储器根据位向量的内容向所有持有 x 的共享备份的处理机发出一个使无效信号 invld(x)。持有 x 的有效备份的处理机在收到 invld(x)信号后，把 x 在 Cache 中的备份从共享状态(SHD)改为无效状态(INV)，并向存储器发出使无效应答 invack(x)。存储器收到所有 invack(x)后，向发出请求的处理机 P_i 发出写数应答 wtack(x)，把目录项中的改写位置为“1”，并把位向量的第 i 位置为“1”，其他位清“0”。

3. 若 x 在 P_i 的 Cache 中处于无效状态，那么这个处理机向存储器发出一个写数请求 write(x)。存储器在收到这个 write(x)请求后查找与单元 x 相对应的目录项。

(1) 如果目录项的内容显示出 x 所在的存储行处于 CLEAN 状态(改写位为“0”)，并没有被其他处理机所共享(位向量中所有位都为“0”)，那么存储器向发出请求的处理机 P_i 发出写数应答 wtack(x)提供 x 所在行的一个有效备份，把目录项中的改写位置为“1”，并把位向量的第 i 位置为“1”。

(2) 如果目录项的内容显示出 x 所在的存储行处于 CLEAN 状态(改写位为“0”)，并且在其他处理机中有共享备份(位向量中有些位为“1”)，那么存储器根据位向量的内容向所有持有 x 的共享备份的处理机发出一个使无效信号 inv-

ld(x)。持有 x 的有效备份的处理机在收到 invld(x)信号后,把 x 在 Cache 的备份从共享状态(SHD)改为无效状态(INV),并向存储器发出使无效应答 invack(x)。存储器收到所有 invack(x)后,向发出请求的处理机 P_i 发出写数应答 wtack(x)以提供 x 所在行的一个有效备份,把目录项中的改写位置为"1",并把位向量的第 i 位置为"1",其他位清"0"。

(3) 如果目录项的内容显示出 x 所在的存储行已被某个处理机 P_k 改写(改写位为"1",位向量第 k 位为"1"),那么存储器向 P_k 发出一个使无效并写回请求 invwb(x)。P_k 在收到 invwb(x)请求后,把 x 在 Cache 的备份从独占状态 EXC 改为无效状态 INV,并向存储器发出使无效并写回应答 invwback(x)以提供 x 所在行的有效备份。存储器收到来自 P_k 的 invwback(x)后,向发出请求的处理机 P_i 发出写数应答 wtack(x)以提供 x 所在行的一个有效备份,把目录项中的改写位置为"1",并把位向量的第 i 位置为"1",其他位清"0"。

4. 如果 x 不在 P_i 的 Cache 中,那么 P_i 先从 Cache 中替换掉一行再向存储器发出一个写数请求 write(x)。

4.1.4 替换操作

如果某处理机要替换一个 Cache 行且被替换行处在 EXC 状态,那么这个处理机需要向存储器发出一个替换请求 rep(x)以把被替换掉的行写回存储器。

4.1.5 例子

例 4.2 假设单元 x 初始时在存储器中处于 CLEAN 状态(改写位为"0"),并被处理机 P_j 和 P_k 所共享(在 P_j 和 P_k 的 Cache 中处于 SHD 状态),如图 4.2(a)所示。接着,x 被多个处理机按如下次序访问:

1. 处理机 P_i 发出存数操作"STORE x"。
2. 处理机 P_k 发出存数操作"STORE x"。
3. 处理机 P_i 发出取数操作"LOAD x"。
4. 处理机 P_j 发出取数操作"LOAD x"。

图 4.2(b)~(f)显示出上述访问序列引起的一系列消息传递,以及 x 在 Cache 中的状态及在存储器中的状态的转化过程。

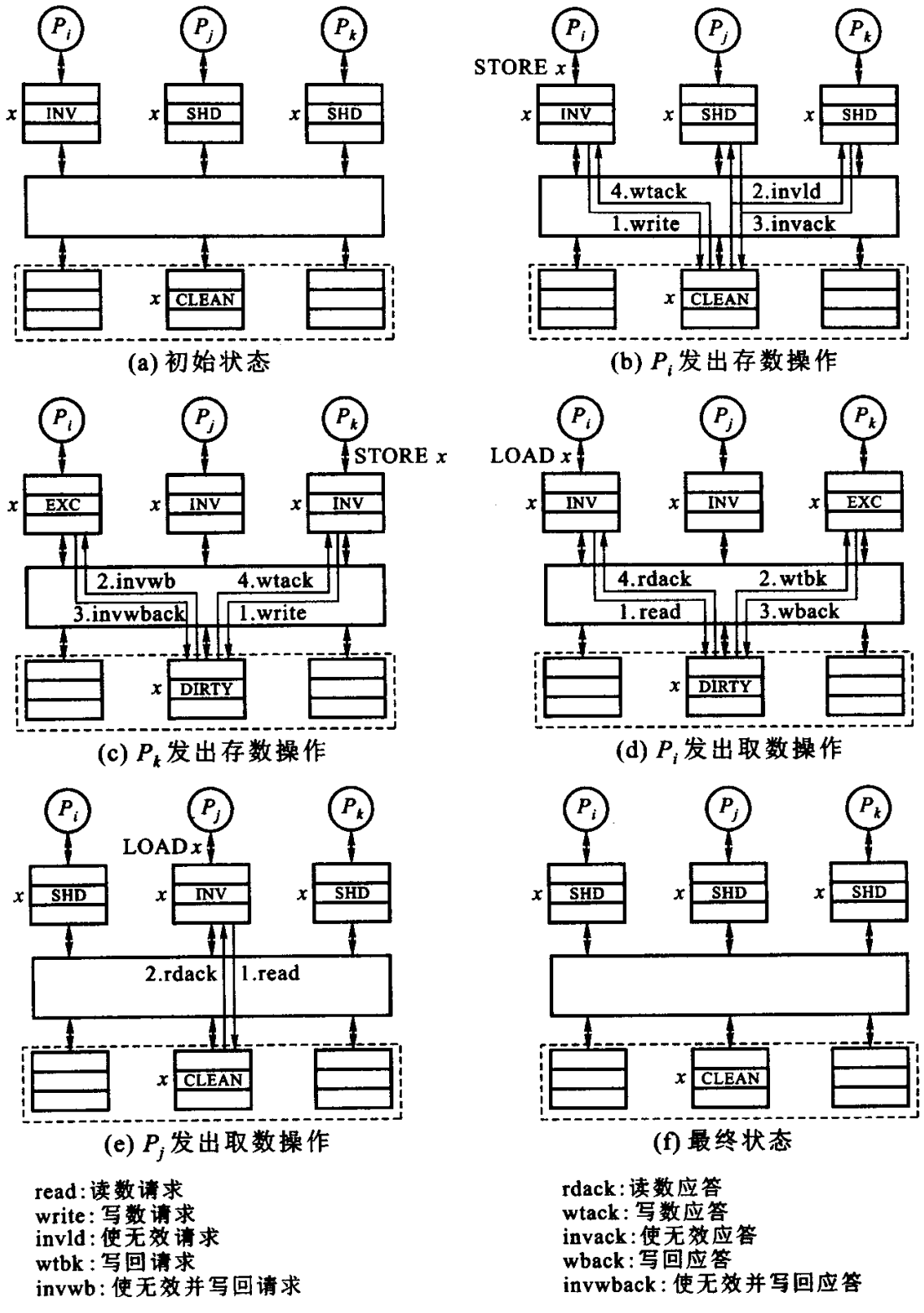

P_i
P_j
P_k
x INV
x SHD
x SHD
x CLEAN
(a) 初始状态
STORE x
4.wtack
2.invld
1.write
3.invack
(b) P_i 发出存数操作
STORE x
x EXC
x INV
x INV
2.invwb
4.wtack
3.invwback
1.write
x DIRTY
(c) P_k 发出存数操作
LOAD x
x INV
x INV
x EXC
4.rdack
2.wtbk
1.read
3.wback
x DIRTY
(d) P_i 发出取数操作
LOAD x
x SHD
x INV
x SHD
2.rdack
1.read
x CLEAN
(e) P_j 发出取数操作
x SHD
x SHD
x SHD
x CLEAN
(f) 最终状态
read:读数请求
write:写数请求
invld:使无效请求
wtbk:写回请求
invwb:使无效并写回请求
rdack:读数应答
wtack:写数应答
invack:使无效应答
wback:写回应答
invwback:使无效并写回应答

图 4.2 基本协议

4.2 充分条件的实现策略

在第三章中,证明了一个确保正确执行的充分条件,该条件包括两个部分,即写一致条件和 GPPO 条件。

4.2.1 基本协议中的访存事件

在 SAM(Shared-memory Access Model,共享存储访问模型)中,任一访存操作 u 都被分割成 N 个子操作 $u^1, u^2, \cdots, u^N$,其中 u^i 表示 u 相对于处理机 P_i 完成(或被处理机 P_i 所接受)。一个存数操作相对于处理机 P_i 完成指的是如果没有对同一单元的其他存数操作,P_i 访问该单元将取回该存数操作所写的值。一个取数操作相对于处理机 P_i 完成指的是 P_i 所写的值将不再影响该取数操作的返回值[38]。在不同的 Cache 一致性协议中,一个访存操作相对于一个处理机完成这一概念有着不同的表现方式。表 4.1 给出了在 4.1 节描述的基本协议中 P_i 发出的存数操作"STORE x""到达" P_j 的具体含义。表 4.2 给出了 P_i 发出的取数操作"LOAD x""到达" P_j 的具体含义。

表 4.1 基本协议中 P_i 发出的"STORE x"指令"到达" P_j 的时刻

x 在 P_i 中的状态	x 在存储器中的状态	x 在 P_j 中的状态	P_i 发出的"STORE x"指令到达 P_j 的时刻
INV(或 NON)	CLEAN	INV(或 NON)	write(x)到达存储器
		SHD	invld(x)到达 P_j
	DIRTY	INV(或 NON)	write(x)到达存储器
		EXC	invwb(x)到达 P_j
SHD	CLEAN	INV(或 NON)	write(x)到达存储器
		SHD	invld(x)到达 P_j
EXC	DIRTY	INV(或 NON)	P_i 发出"STORE x"

注:NON:x 不在 Cache 中

INV:x 在 Cache 中处于无效状态

SHD:x 在 Cache 中处于共享状态

EXC:x 在 Cache 中处于独占状态

表 4.2 基本协议中 P_i 发出的"LOAD x"指令"到达" P_j 的时刻

x 在 P_i 中的状态	x 在存储器中的状态	x 在 P_j 中的状态	P_i 发出的"STORE x"指令到达 P_j 的时刻
INV(或 NON)	CLEAN	INV(或 NON)	read(x)到达存储器
		SHD	read(x)到达存储器
	DIRTY	INV(或 NON)	read(x)到达存储器
		EXC	wtbk(x)到达 P_j
SHD	CLEAN	INV(或 NON)	P_i 发出"LOAD x"
		SHD	P_i 发出"LOAD x"
EXC	DIRTY	INV(或 NON)	P_i 发出"LOAD x"

注:NON:x 不在 Cache 中

INV:x 在 Cache 中处于无效状态

SHD:x 在 Cache 中处于共享状态

EXC:x 在 Cache 中处于独占状态

4.2.2 WC 条件的实现

如前所述,在分布式共享存储系统中,写一致条件是使一个写可分割执行正确的前提。它要求对同一单元的存数操作以相同的次序到达所有处理机。也就是说,若 u 和 v 是对同一单元的两个存数操作且 u 先于 v 到达处理机 P_i,那么,u 先于 v 到达所有其他处理机。

确保写一致条件不被破坏的最简单的方法是:从存储器收到一个处理机的访问请求,一直到存储器对这个请求的服务完成的这段时间内,锁住相应行的目录项。在此期间,来自其他处理机的对同一行的访问请求必须等待。

在上述锁机制的保护下,所有对同一行的存数操作都是串行的,即在一个存数操作正在进行的过程中,对同一行的其他存数操作不得进行。这实际上比写一致条件的要求严格,但这确实是一个比较可行的方法。同样,如果一个取数操作取回一个存数操作所存的值,那么,这个取数操作必须等到相应的存数操作到达所有处理机后才能进行。

有些情况并不破坏写一致条件。第一,P_i 拥有 x 的独占备份,在 P_j 对 x 的访问操作正在进行的过程中(但 invwb(x)或 wtbk(x)未到 P_i),P_i 对 x 进行存数或取数访问并在 Cache 中完成。这可以看成 P_i 的访存操作先于 P_j 的访存操作被所有处理机所接受的情况。第二,P_i 拥有 x 的共享备份,在 P_j 对 x 的存数操作正在进行的过程中(但 invld(x)未到 P_i),P_i 对 x 进行取数访问并在 Cache

中完成。这种情况可以看成 P_i 的取数操作先于 P_j 的存数操作被所有处理机接受。

上述的锁机制可以表示为：

$$u \xrightarrow{E} v \Rightarrow u^i < v^j, \quad i,j=1,2,\cdots,N \tag{4.1}$$

其中，u 和 v 是对同一单元的访存操作，u^i 和 v^j 分别是 u 和 v 的子操作。

值得一提的是，上述的锁机制使目录项成为所有对同一单元的存储访问的串行点，因而成为潜在的瓶颈。在一个存储单元的目录项被锁住期间，对此单元的其他存储访问必须等待。缓解这一问题的方法有链表目录协议(linked list directory protocols)[42,95]、瞬态协议(transient states protocols)[46]以及向前传递技术(forwarding technique)[73]等。

4.2.3 GPPO 条件的实现

GPPO 条件要求系统中所有处理机根据指令在程序中出现的次序执行它们，且在当前访存指令彻底完成之前不能开始执行下一条访存指令。

GPPO 条件的优点是它只对每个处理机内部的访存事件发生次序做出要求，因而避免了昂贵的多个处理机之间的协调。

因此，在基本协议中可以如此实现 GPPO 条件：

1. 所有处理机根据指令在程序中出现的次序执行它们。

2. 当处理机 P_i 发出存数操作“STORE x”后，它必须等这个存数操作到达所有处理机后才能继续执行其他指令。也就是说，若 P_i 持有 x 的独占备份，那么这一存数操作在 Cache 中完成。否则，P_i 向存储器发出 write(x)请求，在收到应答信号 wtack(x)之后才能继续执行后续指令。

3. 当处理机 P_i 发出取数操作“LOAD x”之后，它必须等这个取数操作取回的值已确定且写此值的存数操作已到达所有处理机后才能继续执行其他指令。也就是说，若 P_i 持有 x 的共享或独占备份，那么这个取数操作在 Cache 中完成。否则，P_i 向存储器发出 read(x)请求后等待，在收到应答信号 rdack(x)之后才能继续执行后续指令。锁目录机制可以保证在 P_i 收到应答信号 rdack(x)时，写 P_i 所取回的值的存数操作已到达所有处理机。

上述 GPPO 条件的实现对访存事件发生次序的限制可表示为：

$$u \xrightarrow{PO} v \Rightarrow u^i < v^j, \quad i,j=1,2,\cdots,N \tag{4.2}$$

其中，u 和 v 是取数操作或存数操作。

4.2.4 实现策略的正确性

定理 4.1 证明了上述实现策略的正确性。

定理 4.1 若写可分割执行 $E(PRG)$的访存事件发生次序满足式(4.1)和式(4.2)的要求,则 $E(PRG)$正确。

证明 如果一个执行 $E(PRG)$的访存事件次序满足式(4.1)和式(4.2)的要求,即

$$\begin{cases} u \xrightarrow{E} v \Rightarrow u^i < v^j, & i,j=1,2,\cdots,N \\ u \xrightarrow{PO} v \Rightarrow u^i < v^j, & i,j=1,2,\cdots,N \end{cases} \tag{4.3}$$

则该执行的访存事件次序显然满足定理 3.1 中正确执行的访存事件次序的要求:

$$\begin{cases} w_1 \xrightarrow{E} w_2 \xrightarrow{PO} v \Rightarrow w_1^c < v^i \\ w \xrightarrow{E} r \xrightarrow{PO} v \Rightarrow w^c < v^i \\ r \xrightarrow{E} w \xrightarrow{PO} v \Rightarrow r^c < v^i \\ r_1 \xrightarrow{E} w \xrightarrow{E} r_2 \xrightarrow{PO} v \Rightarrow r_1^c < v^i \end{cases} \tag{4.4}$$

其中,w、w_1 和 w_2 是写操作,r、r_1 和 r_2 是读操作,v 可以是写操作或读操作,$i=1,2,\cdots,N$,c 值介于 1 和 N 之间。

4.3 乱序执行的实现策略

GPPO 条件要求任何指令必须等它的前一条指令完成后才能开始执行,即由同一处理机发出的指令不能重叠执行。这不利于提高系统性能,尤其是在分布式共享存储系统中,一旦所访问的单元不在 Cache 中,处理机等待的时间会很长。

不允许指令重叠执行的原因是为了防止错误执行的发生。然而,并非指令的所有重叠执行都会导致错误 。在绝大多数情况下,即使对一个程序的访存事件发生次序不做限制,也会产生正确的结果[39]。因此,只要对容易引起错误的少数访存操作的执行次序加以限制,那么绝大多数访存操作就可以重叠执行而不影响执行的正确性。

在第三章证明了:若在进程 P_i 中 u 是先于 u_1 的访存操作,则在下述条件下访存操作 u_1 可以越过它前面的 u 执行而不影响执行的正确性:在 u_1 发出之后而 u"彻底完成"之前的这段时间内,没有与 u_1 冲突的访问相对于 P_i 发生。这个条件称为 GPOO 条件。

然而,由于 P_i 难以预知在 u 执行过程中是否会收到来自其他处理机的对 u_1 所访问单元的访存操作,P_i 无法决定是否在 u 完成之前发出 u_1 及其后的访

存操作。因此,GPOO 条件的实现策略通常是在 u 执行过程中,允许 u_1 猜测性地执行,而在随后的执行过程中检测并纠正那些会导致错误的猜测执行。

由于对写操作的猜测执行在猜测错误的情况下难以纠正错误,因此如果 u_1 是写操作,对 u_1 的猜测执行一般不彻底进行,而是把 u_1 所访问的单元预取到 Cache 中并获得对该单元的可写权即可。下面先给出 u_1 是读操作时猜测执行的实现方法,再说明 u_1 是写操作时预取的实现方法。

在猜测执行技术中,如果 u 是一个延迟较长的操作而 $r(x)$是 u 之后的一个取数操作,那么处理机在等待 u 完成的过程中,可以把 $r(x)$要访问的单元 x 的值预取到 Cache 中来,把预取的值返回到处理机,并继续后续指令的执行。如果在 u 结束之前,x 在 P_i 中的备份不被更新,则猜测执行是成功的。在 u 结束的同时 $r(x)$也结束了。这样,在 u 结束之前执行 $r(x)$不会影响执行的正确性,因为即使 $r(x)$在 u 结束之后执行,所取回的值也是一样的。否则,猜测执行有可能是错误的,必须采取措施来预防或纠正错误。

防止由于猜测执行引起错误的一个方法是:如果 $r(x)$已被猜测执行,但在 u 结束之前 P_i 检测到对 x 的存数操作,则放弃猜测执行的结果。例如,在写使无效的基于目录的 Cache 一致性协议中,可以如此判断猜测执行的正确性:如果 P_i 在 u 结束之前收到使单元 x 无效的请求信号 invld(x)(由于其他处理机对 x 进行存数操作),那么处理机对 $r(x)$的猜测执行所取回的 x 值就是错误的,处理机必须抛弃对 $r(x)$的猜测执行所取回的值,重新执行 $r(x)$。否则,猜测执行就成功。这与 GPOO 条件的要求是一致的。为了简便,把这种猜测执行的技术称为检测并纠正猜测执行技术,或 IC-SPEC(Invalidation-Correct-SPECulation)技术。

防止由于猜测执行引起错误的另一个方法是:在处理机 P_i 对 $r(x)$进行猜测执行之后,就不允许其他处理机对 x 在 P_i 中的备份进行存数操作,直到 u"彻底完成"。在写使无效的基于目录的 Cache 一致性协议中,在处理机对一个取数指令进行猜测执行之后,就把这个取数操作所访问的 Cache 行锁住,不允许使这个 Cache 行中的数无效或被替换掉,直到这个取数操作前面的所有指令都已"彻底完成"。这种猜测执行的技术称为延迟无效猜测执行技术,或 ID-SPEC(Invalidation-Delayed-SPECulation)技术。

例 4.3 当图 4.3 中的程序在具有 IC-SPEC 功能的系统中执行时,可能产生如下执行序列:

1. P_1 发出 L_{11}但不在 Cache 命中。

2. 在等待 L_{11}结束的过程中,P_1 猜测执行 L_{12}并在 Cache 命中,于是 P_1 从 Cache 中取回 b 的旧值 0。

3. P_2 发出 L_{21} 但不在 Cache 命中，导致存储器向 P_1 发使无效信号 invld(b)。

4. P_1 收到 invld(b)信号后，认为对 L_{12} 的猜测执行所取回的 b 值是错误的，于是抛弃猜测执行所取回的值并重新执行 L_{12}。

上述执行导致($R_1 = a = b = 1$)的正确结果，这个结果与串行执行 $L_{21} < L_{11} < L_{12}$ 的结果一致。

当这个程序在具有 ID-SPEC 功能的系统中执行时，处理机 P_1 发出 L_{11} 后，猜测执行 L_{12} 并把单元 b 在 Cache 中的备份锁住。当由 L_{21} 引起的 invld(b)到达 P_1 时，b 在 P_1 的备份已被锁住。在 L_{11} 执行结束后，invld(b)得以进行。因此，P_1 对 L_{12} 的猜测执行被认为是正确的，不需进行纠正。这样得到的执行结果($a = b = 1, R_1 = 0$)也是正确的。

Process P_1	Process P_2
L_{11}: STORE a, 1	L_{21}: STORE b, 1
L_{12}: LOAD R_1, b	

图 4.3 猜测执行的例子(初始值 $R_1 = a = b = 0$)

从例 4.3 中可以看出，ID-SPEC 判断猜测执行正确与否比 IC-SPEC 更加精确，减少了一些不必要的纠正。然而，把 Cache 行锁住容易引起互等资源死锁。

例 4.4 在图 4.4 的程序中，如果采用 ID-SPEC 来防止错误猜测执行的发生，就可能产生如下执行序列：

1. P_1 发出 L_{11} 不在 Cache 命中，于是 P_1 向存储器发出存数请求 write(a)。P_2 发出 L_{21} 不在 Cache 命中，于是 P_2 向存储器发出存数请求 write(b)。

2. 在等待存数应答 wtack(a)的过程中，P_1 猜测执行完 L_{12}，并把 b 在 Cache 的备份锁住。在等待存数应答 wtack(b)的过程中，P_2 猜测执行完 L_{22}，并把 a 在 Cache 的备份锁住。

3. 存储器在收到来自 P_1 的 write(a)请求和来自 P_2 的 write(b)请求后，向 P_2 发 invld(a)信号，并向 P_1 发 invld(b)信号。

4. Invld(b)到达处理机 P_1，由于 b 在 P_1 中的备份被锁住，invld(b)要等到 L_{11} 完成后才能执行。同样，到达 P_2 的 invld(a)必须等到 L_{21} 完成后才能执行。因此，L_{21} 的完成依赖于 L_{11} 的完成，而 L_{11} 的完成反过来依赖于 L_{21} 的完成，这就导致了互等资源死锁的发生。

Process P_1	Process P_2
L_{11}: STORE a, 1	L_{21}: STORE b, 1
L_{12}: LOAD R_1, b	L_{22}: LOAD R_2, a

图 4.4 猜测执行的例子(初始值 $R_1 = R_2 = a = b = 0$)

从例 4.4 可以看出,在 ID-SPEC 技术中,如果 invld(x)到达处理机 P_i 并发现在 P_i 的预取指令栈中有 1 条对 x 的取数指令 $r(x)$已被猜测执行,那么 invld(x)就要等待 $r(x)$及 $r(x)$前面的所有指令结束后才能执行,这就有可能导致死锁的发生。如果 $r(x)$是指令栈中的第 k 条指令,且 $r(x)$前面的$(k-1)$条指令都不必等待来自存储器的应答就可以完成(如对私有变量的访问或可在 Cache 命中的共享访问),那么在预取指令栈中 $r(x)$前面的指令(包括 $r(x)$)最终肯定能结束,由猜测执行 $r(x)$引起的对 x 在 P_i 中备份的锁肯定会被释放,因而 invld(x)也肯定能被应答。在这种情况下对 $r(x)$的猜测执行不会引起死锁。此外,对于这种肯定不会引起死锁的情况,只要 P_i 保证在执行完 $r(x)$后使 x 无效,invld(x)就可以在 x 真正无效之前被应答而不影响执行的正确性(只限于不会引起死锁的情形)。

如果 $r(x)$前面的$(k-1)$条指令中有不在 Cache 命中的访问共享存储器的指令,这些指令的完成依赖于来自存储器的应答信号。在这种情况下,如果收到 invld(x),就应停止对 $r(x)$的猜测执行并进行纠正。

在具体应用中大多数访存操作都是对私有变量的访问,且共享存储访问的 Cache 命中率也很高,上述分析中 $r(x)$前面的$(k-1)$条指令中有不在 Cache 命中的共享存储访问的概率是很小的。因此,在大多数情况下即使收到 invld(x),也可以继续对 $r(x)$猜测执行而不用纠正。

上述猜测执行描述了将取数操作提前进行的情况。由于对存数操作的错误猜测执行难以纠正,因此一般不对存数操作进行彻底的猜测执行,而是采用预取技术。

设 u 和 $w(x)$是由同一处理机 P_i 执行的两个连续的访存操作。通常写数操作 $w(x)$必须等 u"彻底完成"后才能开始执行。如果 u 所访问的单元不在 P_i 的 Cache 中,那么 P_i 等待的时间就会很长。P_i 可以向存储器发出写数请求 write(x)把 $w(x)$所访问的单元 x 预取到 Cache 中来并处于独占状态。如果在 x 预取进来后,而 u 完成之前的这段时间内,其他处理机对 x 进行取数或存数操作,那么根据协议的要求,P_i 会收到来自存储器的 wtbk(x)或 invwb(x)信号,预取进来的值就会变成共享或无效状态。在这种情况下,当 $w(x)$真正被执行时,还得向存储器发出对 x 的写数请求。

同样,对于存数操作的预取,也可以采用类似于 ID-SPEC 中的技术。如果在 x 预取进来后,而 u 完成之前的这段时间内,P_i 收到来自存储器的 wtbk(x)或 invwb(x)信号时,不把预取的值变成共享或无效状态,除非在预取指令栈内 $w(x)$前面的$(k-1)$条指令中有不在 Cache 命中的访问共享存储器的指令,因为这些指令的完成依赖于来自存储器的应答信号。

4.4 模拟模型

笔者和同事建立了一个地址流驱动(Trace-Driven)模拟模型来评估本书提出的乱序执行技术对性能的影响。此模拟模型模拟了一个基于位向量目录的 Cache 一致性协议的行为。它主要包含 4 个模块,即地址流生成模块、处理机及 Cache 模块、存储器模块以及互联模块。

模拟模型用 C 语言编写,在模型中除了模拟多处理机系统所必需的数据结构(如指令寄存器、预取指令栈、Cache、目录等)之外,每个模块都有相应的状态信息。在每个时钟周期,每个模块检查自己的当前状态,完成一定的动作,并生成新的状态。

4.4.1 地址流的生成

地址流的生成主要有两种方法。一种是把一个实际执行的地址流收集起来作为模拟模型的输入,另一种是在对共享存储访问的特征做一定假设的基础上通过合成的方法产生地址流作为模拟模型的输入。前者的优点是它的真实性;而后者的优点是它比较灵活,可以通过适当改变表征程序访问模式的参数来生成反映不同应用、不同系统配置的地址流。由于没有合适的真实地址流,笔者和同事选用了在参考文献[30]中提出并在参考文献[13]和[96]得到改进的合成地址流生成方法来产生地址流。

地址流生成模块为每个处理机产生 1 个独立的地址流。在每个地址流中共享存储访问出现的概率是 P_{shd},私有访问出现的概率是 $1-P_{shd}$。读访问出现的概率是 P_{rd},写访问出现的概率是 $1-P_{rd}$。

私有访问不影响 Cache 一致性,与单处理机中的访问没有区别。因此,可以用概率的方法产生私有访问地址流。一个私有访问在 Cache 命中的概率是 P_{hit}。当一个私有变量被替换掉时,它因被改写过而需要写回的概率是 P_{wtbk}。

由于共享存储访问的特征难以用简单的概率模型来表示,地址流生成模块为每个共享存储访问产生一个具体的行地址。每次访问的行地址由 1 个最近最少使用的 LRU(Least Recently Used)栈来实现。每个处理机都有 1 个 LRU 栈与之对应,每个栈有 N_{shd}层,每层表示 1 个存储行,其中 N_{shd}是共享存储行的数

目。为了反映共享存储访问的时间和空间的局部性，离栈顶越近的存储行被访问的概率越高。且在每次访问后，刚被访问的存储行就被放在栈顶。一个共享存储访问访问第 i 层存储行的概率是：

$$p_i = g\left(\frac{1}{i+loc} - \frac{1}{i+loc+1}\right), \quad i = 1,2,\cdots,N_{shd}$$

其中，g 是一个归一化因子，它的作用是使 $\sum_{i=1}^{N_{shd}} p_i = 1$，常数 loc 可以用来调整共享存储访问的局部性，loc 越小，访问的局部性越强。

4.4.2 处理机及 Cache 模块

处理机及 Cache 模块模拟 N_{proc} 个处理机的行为。它调用地址流生成模块来生成地址流。对于地址流中的每个访存操作，处理机首先检查此操作是否在 Cache 命中，对于不命中的操作，处理机向存储模块发读数请求 read(x)或写数请求 write(x)。如果需要替换掉 Cache 中的某一行，则处理机向存储模块发替换请求 replace(x)。

此外，处理机及 Cache 模块还处理来自存储模块的 invld(x)、wtbk(x)及 invwb(x)等请求。其中，invld(x)要求处理机把其 Cache 中原来处于共享状态的 x 所在行置为无效状态；wtbk(x)要求处理机把其 Cache 中原来处于独占状态的 x 所在行写回存储器，并置为共享状态；invwb(x)要求处理机把其 Cache 中原来处于独占状态的 x 所在行写回存储器，并置为无效状态。处理机在收到这些请求后，在 Cache 中查询相应行，改变此行在 Cache 中的状态，并向存储器发 invack(x)、wback(x)及 invwback(x)等应答信号。

Cache 的每一行都有 4 种状态，即私有状态(PRV)、无效状态(INV)、共享状态(SHD)及独占状态(EXC)。若 Cache 的某一行处于私有状态，表示相应行为此处理机所私有；若 Cache 的某一行处于无效状态，处理机对这一行的取数或存数访问都不命中；若 Cache 的某一行处于共享状态，说明可能还有其他处理机持有这一行的有效备份，处理机对这一行的取数访问可以在 Cache 中完成；若 Cache 的某一行处于独占状态，说明这是此存储行的惟一有效备份，处理机对这一行的取数或存数访问都可以在 Cache 中完成。模拟开始时 Cache 中所有行都处于 PRV 状态。

为了评估乱序执行对系统性能的影响，在此模拟了 3 种不同的执行方案，即一般方案、IC-SPEC 方案及 ID-SPEC 方案。对于不同的方案，处理机及 Cache 模块的行为也是不同的。

1. 在一般方案中，每个处理机在执行完当前指令后才继续下一条指令的执行。

2. 在 IC-SPEC 方案中，每个处理机都有一个预取指令栈 ILB(Instruction

Lookahead Buffer)。指令以程序序进入 ILB。ILB 中的存数指令必须以 FIFO 的次序结束,而取数指令则可以越过它前面的取数或存数指令而猜测性地执行。对存数指令的预取操作也可以在它前面的指令执行完之前执行。

当 ILB 中的第 1 条指令结束后,在它之后已被猜测执行完毕的所有取数指令也随之结束。然而,如果在 ILB 中的第 1 条指令执行过程中,在它之后已被猜测执行的某一取数指令所访问的单元 x 被其他处理机所更新(处理机收到 invld(x)或 invwb(x)请求),那么已进行或完成的猜测执行就要抛弃掉,这正是 IC-SPEC 方案所要求的。

3. ID-SPEC 方案则更进一步。若处理机在猜测执行对单元 x 的取数操作 $r(x)$后收到 invld(x)或 invwb(x)请求,并不立即认为对 $r(x)$的猜测执行是错误的,而是检查 ILB 中 $r(x)$前面的指令中是否有不在 Cache 命中的访问共享存储器的指令。若有这样的指令,则认为对 $r(x)$的猜测执行是错误的并对其结果予以抛弃。否则,认为对 $r(x)$的猜测执行是正确的,并把 invld(x)或 invwb(x)延迟到 $r(x)$结束后再执行。

4.4.3 存储器模块

在存储器中,每行都有一个相应的目录项。每个目录项有一个 N_{proc}位的向量,其中 N_{proc}是系统中处理机的个数。位向量中第 i 位为"1"表示此存储行在第 i 个处理机 P_i 中有备份。此外,每个目录项有一个改写位,当改写位为"1"时,表示某处理机独占并已改写此行。

存储器模块的主要功能是根据目录的内容处理来自处理机及 Cache 模块的 read(x)、write(x)及 replace(x)等请求。在处理这些请求的过程中,根据需要向处理机及 Cache 模块发出如下请求:

1. 若 x 被多个处理机所共享,并收到 write(x)请求,则存储器模块向当前共享 x 的所有处理机发出 invld(x)请求。

2. 若 x 被某个处理机所独占,并收到 read(x)请求,则存储器模块向当前独占 x 的处理机发出 wtbk(x)请求。

3. 若 x 被某个处理机所独占,并收到 write(x)请求,则存储器模块向当前独占 x 的处理机发出 invwb(x)请求。

在收到相应的应答(invack(x)、wback(x)或 invwback(x))后向处理机及 Cache 模块发应答信号 rdack(x)或 wtack(x)。

为了保证正确性,从存储器收到一个访问请求(read(x)或 write(x)),直到存储器对这个请求的服务已经完成的这段时间内,锁住相应行的目录项。在此期间,来自其他处理机的对同一行的访问请求必须等待。

4.4.4 互联模块

互联模块为处理机、Cache 模块和存储器模块传递信息。为了防止发生死锁，请求信号和应答信号分别在不同的“互联网络”上传输。此外，笔者和同事还采取了一定的措施保证有相同的源和目的的信号根据被发出的先后次序传输。

4.4.5 模拟参数和模拟输出

采用合成地址流使得模拟参数可以灵活地改变，表 4.3 列出了主要参数的取值范围。

表 4.3 模 拟 参 数

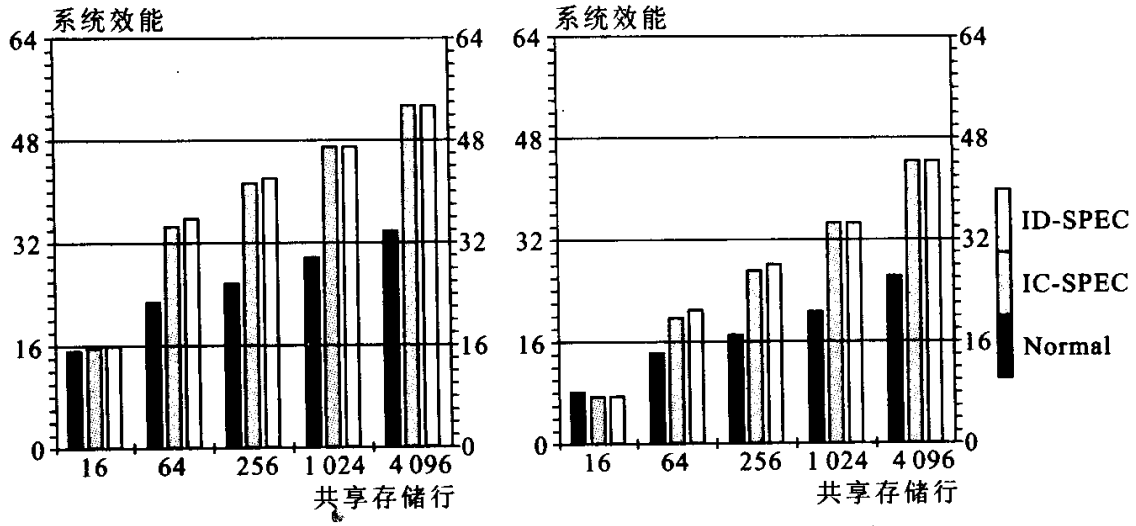
参数	范围	参数的含义
N_{proc}	64	处理机个数
N_{win}	16～64	预取的指令数
N_{mod}	$=N_{proc}$	存储模块数
N_{shd}	16～4 096	共享存储行的数目
N_{Cache}	256～2 048	每个 Cache 的行数
D_{mem}	4	访存延迟
D_{inv}	1	使一 Cache 行无效所需的周期数
D_{wtbk}	4	写回一 Cache 行所需的周期数
D_{trans}	$=\sqrt{N_{proc}}$	网络传输延迟
P_{shd}	0.05～0.10	任一访问是共享存储访问的概率
P_{rd}	0.80	任一访问是读访问的概率
P_{hit}	0.95	私有访问的 Cache 命中率

一次模拟运行 20 000 个时钟周期。每次模拟都对一些重要的性能指标进行统计，其中最重要的性能指标是一个称为系统效能(system power)[13,96]的量，它是系统中所有处理机的利用率之和。

4.5 模拟结果及分析

图 4.5 给出了在不同的参数下 3 种执行方案的模拟结果。这些结果是在下列参数下得到的：处理机数为 64，每个处理机的 Cache 大小为 256 行，共享存储行的数目为 16 至 4 096。图 4.5(a)的输入地址流中有 5%的访问是共享存储访

问,而图 4.5(b)的输入地址流中有 10% 的访问是共享存储访问。



(a) 5% 共享存储访问　　(b) 10% 共享存储访问

图 4.5　不同执行方案的系统效能(64 个处理机,ILB=16,256Cache 行)

4.5.1　访存冲突的影响

如前所述,为了保证执行的正确性,在一个存储模块对某个访存请求进行服务期间,不接收其他处理机对这个存储模块的访存请求。这就可能导致冲突。频繁的冲突对系统性能有两方面不利的影响。一是增加延迟;二是不同处理机对同一行的存数访问使得多个处理机互相剥夺访问权,从而降低 Cache 命中率。

而在猜测执行方案中,一个处理机可以并行执行多条指令,这就使得每个存储模块的负担增加,访存冲突的概率也随之增加。

模拟结果证实了上述分析。从图 4.5 中可以明显地看出,系统效能随着共享存储行数目的增加而显著增加,在猜测执行的情况下尤其如此。

表 4.4 给出了图 4.5 中 Cache 大小为 256 行时的不同参数下共享存储器冲突的概率。从表 4.4 中可以看出,冲突概率随着共享存储行数目的增加而减少。另外,猜测执行加剧了存储冲突。

表 4.4　访存冲突概率(64 个处理机、256Cache 行、5% 共享存储访问)

存储行数	一般方案	IC-SPEC 方案	ID-SPEC 方案
16	0.427	0.706	0.708
64	0.046	0.152	0.147
256	0.032	0.100	0.098
1 024	0.022	0.065	0.062
4 096	0.010	0.031	0.030

4.5.2 乱序执行的效果

从图 4.5 中可以看出,在绝大多数情况下,乱序执行的系统效能比一般方案的系统效能要高。例如,在共享存储行的数目为 4 096 的情况下,采用乱序执行的系统效能比不采用乱序执行的系统效能高出 50% 左右。这充分显示了乱序执行的有利效果。

图 4.5 还显示出,随着共享存储行数目的增加,乱序执行的效果变得更加明显。这是由于在乱序执行的情况下,一个处理机可以并行执行多条指令,在同一时间内到达一个存储模块的访存请求数目大大增加,访存冲突比没有乱序执行的时候严重得多。因此,在乱序执行的情况下增加共享存储行的数目比在没有乱序执行的情况下增加共享存储行的数目能更有效地缓解访存冲突对性能的影响。

从图 4.5(b)中还可以看出,当地址流中有 10% 的共享存储访问而共享存储器只有 16 行时,采用乱序执行反而降低了系统效能。这是因为在访存冲突相当严重的情况下,系统性能受到很大影响,而乱序执行大大地加剧了冲突。此时,并行执行所带来的性能的提高还不足以补偿冲突所带来的性能的下降。进一步的模拟表明,在这种访存冲突严重的情况下,绝大多数猜测执行都是失败的。

从上述分析可以看出,在访存冲突不严重的情况下,乱序执行能有效地提高系统性能。而在访存冲突严重的情况下,乱序执行加剧了冲突,有时反而会降低系统性能。

4.5.3 两种乱序执行方案的比较

从图 4.5 中可以看出,ID-SPEC 方案总比 IC-SPEC 方案略优。如前所述,这是由于 ID-SPEC 比 IC-SPEC 能更精确地判断猜测执行的正确性。

从图 4.5 中还可以看出,随着共享存储行数的增加,ID-SPEC 和 IC-SPEC 的系统效能又趋于接近。当共享存储行的数目为 16、64 或 256 时,ID-SPEC 的系统效能比 IC-SPEC 的系统效能增加得明显。而当共享存储行的数目增加到 1 024或 4 096 时,IC-SPEC 和 ID-SPEC 的系统效能几乎没有差别。这可以从系统的共享模式上得到解释。当 64 个处理机共享 64 或 256 存储行时,系统处于紧密共享(Tight Sharing)模式,即多个处理机在一段时间内频繁地访问同一变量,导致处理机间频繁地互相剥夺对同一行的访问权。在这种情况下,ID-SPEC 由于能合理地延迟使无效信号 invld(x)的执行而使 Cache 的命中率有较大提高。而在有 1 024 或 4 096 存储行的情况下,系统处于顺序共享(Sequential Sharing)状态,即在较长时间内只有一个处理机访问一个变量。在这种情况下,处理机互相剥夺访问权对 Cache 的命中率的影响不大。因此,延迟使某一 Cache

行无效对于 Cache 的命中率没有明显的作用。

4.5.4 ILB 大小的影响

为了研究不同的预取指令数对乱序执行效果的影响，可以对不同 ILB 大小的情况进行模拟。图 4.6 给出了 ILB 大小分别为 16、32、48 及 64 时的模拟结果。

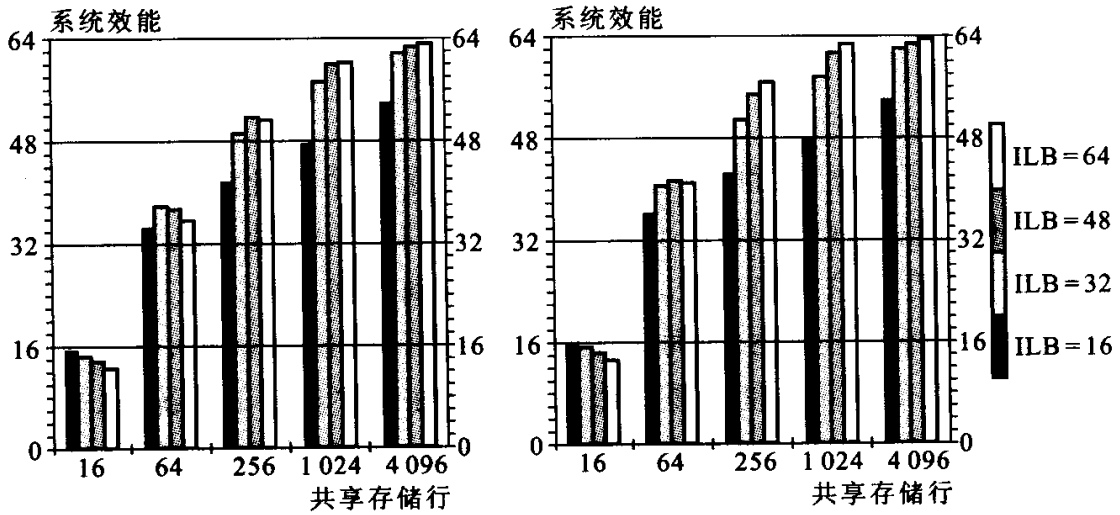


图 4.6 不同 ILB 大小的系统效能(64 个处理机，256Cache 行，5% 共享存储访问)

通过对图 4.6 的分析可以看出：

1. 当访存冲突不严重时，乱序执行的系统效能随着预取指令数由 16 增加到 32 而显著增加，但随着预取指令数的进一步增加，系统效能的增加趋于缓慢。一个重要的原因是，增加预取指令数加剧了访存冲突。另一个原因是，在模拟模型中，一个处理机每个时钟周期最多只能结束或猜测性地结束一条指令(虽然它可以并行执行多条指令)。这是维持顺序一致性所要求的。因此，当预取的指令足够多时，处理机的效能趋于饱和，减少了进一步提高的余地。如在 4 096 个共享存储行、预取 32 条指令的情况下，IC-SPEC 和 ID-SPEC 的系统效能分别为 61.51 和 61.71，已十分接近最大值 64。

2. 当访存冲突严重时，增加预取指令数反而会降低系统性能。从图 4.6 中可以看出，当共享存储行的数目为 16 或 64 时，都出现了系统效能随预取指令数的增加而降低的现象。这是因为增加预取指令数就增加了并行执行的访存操作数，从而加剧了访存冲突。

3. 随着预取指令数的增加，ID-SPEC 对于 IC-SPEC 的优势更加明显。这是由于随着预取指令数的增加，被猜测执行的指令数随之增加。这样，当一个 invld(x)请求到达处理机 P_i 时，P_i 已猜测执行某“LOAD x”或“STORE x”指令

的可能性也就增大了。因此，当 ILB 增加时，延迟对 invld(x)的执行所带来的效果更加明显。

4.6 小 结

本章首先讨论了一个具体的 Cache 一致性协议中的访存事件发生次序。它把前几章对访存事件发生次序的研究与一个基于目录的 Cache 一致性协议结合起来，讨论了在具体协议中实现顺序一致性的方法，并利用执行正确性模型证明了实现策略的正确性。

本章还讨论了乱序执行的实现策略，并建立了一个地址流驱动模拟模型来评估乱序执行对系统性能的影响。从模拟结果可以看出，乱序执行能有效地提高顺序一致共享存储系统的性能，ID-SPEC 的实现方案由于比 IC-SPEC 能更精确地判断猜测执行的正确性，因而其性能也优于 IC-SPEC。此外，在本章所做的模拟中，访存冲突对系统性能有重要影响。由于在乱序执行的情况下，一个处理机可以并行执行多条指令，加剧了访存冲突，因此，乱序执行的作用只有在访存冲突不严重的情况下才能充分发挥出来。而在访存冲突严重的情况下，乱序执行加剧了冲突，有时反而会降低系统性能。

第5章 存储一致性模型

5.1 引　　言

前几章以顺序一致性模型作为正确执行的标准，建立了执行正确性模型，讨论了实现正确执行的访存事件次序，并在此基础上讨论了访存事件次序的具体实现。在共享存储系统中，为了实现正确的执行，需要对访存事件次序施加严格的限制。在单处理机中的若干提高性能的技术，如流水线技术、超标量技术等，在共享存储系统中都难以有效地使用。

为了放松对访存事件次序的限制，人们提出了一系列弱存储一致性模型。这些弱存储一致性模型的基本思想是：在顺序一致性模型中，虽然为了保证正确执行而对访存事件次序施加了严格的限制，但在大多数不会引起访存冲突的情况下，这些限制是多余的。因此，可以让程序员承担部分执行正确性的责任，即在程序中指出需要维护一致性的访存操作，系统只保证在用户指出的需要保持一致性的部分维护数据一致性，而对用户未加说明的部分，可以不考虑处理机之间的数据相关。

目前常见的弱存储一致性模型包括弱一致性(Weak Consistency)模型、释放一致性(Release Consistency)模型、急切更新释放一致性(Eager Release Consistency)模型、懒惰更新释放一致性(Lazy Release Consistency)模型、域一致性(Scope Consistency)模型以及单项一致性(Entry Consistency)模型等。这些存储一致性模型对访存事件次序的限制不同，因而对程序员的要求以及所能得到的系统性能也不一样。存储一致性模型对访存事件次序施加的限制越弱越有利于提高性能，但编程工作越难。

值得指出的是,在所有存储一致性模型中,执行正确性的标准是一致的,即遵循顺序一致性模型所规定的正确性标准。在其他一致性模型中,系统结构对于满足要求的程序体现出顺序一致性的行为。即只要一个程序满足某种弱一致性模型的要求,则该程序在实现这种弱一致性模型的系统中执行的结果与在实现顺序一致性模型的系统中执行的结果是一样的。但如果一个程序不满足这种弱一致性模型的要求,则该程序在实现这种弱一致性模型的系统中执行的结果可能是错误的,即与顺序一致的系统中执行的结果不一致。

本章建立一个存储一致性模型的框架模型。传统的存储一致性模型都是通过其对访存事件次序的限制来描述的,因而是面向硬件的。这是因为一种新一致性模型的提出主要通过放松对访存事件次序的限制来提高系统性能。这就要求程序员在编程时考虑访存操作的硬件执行次序,增加了本来就不容易的并行编程的负担。此外,规定一个存储一致性模型的访存事件次序也限制了进一步的硬件优化。基于存储一致性模型是系统设计者和程序员之间的一个约定的认识,本章从访存操作所体现出的行为的角度建立了一个存储一致性模型的新框架模型。由于一个共享存储程序的并行执行的行为是由该程序中冲突访问的执行次序决定的,因此可以把存储一致性模型定义为规定不同处理机间访存操作执行次序的一种同步机制。在一个执行中,程序序以及同步操作的执行次序决定该执行的最终行为。

此外,本章还从访存事件次序的角度建立了判断并行程序以及系统结构是否满足某种存储一致性模型的标准。

本章的第2节回顾常见的一些存储一致性模型,第3节介绍一种新的用来描述存储一致性模型的框架模型并给出了并行程序、系统结构是否满足某种存储一致性模型的标准,第4节以顺序一致性、释放一致性以及域一致性为例给出证明一个系统结构满足某种存储一致性模型的方法,第5节是本章小结。

5.2 有关的存储一致性模型

5.2.1 顺序一致性模型

顺序一致性(Sequential Consistency,简称SC)是程序员最乐于接受的存储一致性模型。对于满足顺序一致性的多处理机中的任一执行,总可以找到同一程序在单机多进程环境下的一个执行与之对应,使得二者结果相等。在参考文献[83]中,Scheurich和Dubois给出了以下满足顺序一致性的条件:

在共享存储多处理机中,若任一处理机都严格按照访存指令在进程中出现的次序执行访存指令,且在当前访存指令彻底完成之前不能开始执行下一条访

存指令,则此共享存储系统是顺序一致的。

在上述条件中,一个存数操作"彻底完成"是指它所引起的值的变化已被所有处理机所接受,一个取数操作"彻底完成"是指它取回的值已确定且写此值的存数操作已"彻底完成"。上述条件就是前面介绍的 GPPO 条件。

5.2.2 处理机一致性模型

由 Goodman[44] 提出的处理机一致性(Processor Consistency,简称 PC)比顺序一致性弱,故对于某些在顺序一致条件下能正确执行的程序,在处理机一致条件下执行时可能会导致错误结果。处理机一致性对访存事件发生次序施加的限制是[38]:

1. 在任一取数操作 LOAD 允许被执行之前,所有在同一处理机中先于取数操作 LOAD 的其他取数操作都已完成。

2. 在任一存数操作 STORE 允许被执行之前,所有在同一处理机中先于存数操作 STORE 的访存操作(包括 LOAD 和 STORE)都已完成。

上述条件允许 STORE 操作之后的 LOAD 操作越过 STORE 操作而执行,放松了顺序一致性模型对访存次序的限制。

5.2.3 弱一致性模型

为了进一步放松对访存事件发生次序的限制,M. Dubois 等[31] 提出了弱一致性(Weak Consistency,简称 WC)模型。其主要思想是通过在硬件和程序员之间建立某种约定来让程序员负担一些维持数据一致性的责任,从而放松硬件对访存事件发生次序的限制。具体做法是把同步操作和普通访存操作区分开来,程序员必须用硬件可识别的同步操作把对于写共享单元的访问保护起来,以保证多个处理机对于写共享单元的访问是互斥的。弱一致性对访存事件发生次序做如下限制[38]:

1. 同步操作的执行满足顺序一致性条件。

2. 在任一普通访存操作允许被执行之前,所有在同一处理机中先于这一访存操作的同步操作都已完成。

3. 在任一同步操作允许被执行之前,所有在同一处理机中先于这一同步操作的普通访存操作都已完成。

上述条件允许在同步操作之间的普通访存操作以任意次序执行,使多个访问的重叠和流水成为可能。虽然弱一致性模型增加了程序员的负担,但它能有效地提高性能。

值得注意的是,即使是在顺序一致的共享存储并行程序中,同步操作仍是不可避免的,否则程序的行为难以确定。因此,在弱一致性模型的程序中,专门为

保持数据一致性而增加的同步操作不多。

5.2.4 释放一致性模型

释放一致性(Release Consistency,简称 RC)模型[38]是对弱一致模型的改进,它把同步操作进一步分成获取操作 acquire 和释放操作 release。acquire 用于获取对某些共享存储单元的独占性访问权,而 release 则用于释放这种访问权。释放一致性模型对访存事件发生次序做如下限制:

1. 同步操作的执行满足顺序一致性条件。

2. 在任一普通访存操作允许被执行之前,所有在同一处理机中先于这一访存操作的 acquire 操作都已完成。

3. 在任一 release 操作允许被执行之前,所有在同一处理机中先于这一 release 操作的普通访存操作都已完成。

由硬件维护数据一致性的系统(如 DASH)实现了上述释放一致性模型。在 DASH 系统中,对共享单元的存数操作是及时进行的,当一个处理机执行到一条存数指令时,它就根据 Cache 一致性协议向相应的单元发出存数请求。这样,在两个同步操作之间,多个访存操作可以并行地进行。

5.2.5 急切更新释放一致性模型

在共享虚似存储系统或在由软件维护数据一致性的共享存储系统中,由于通信和数据交换的开销很大,有必要减少通信和数据交换的次数。为此,人们在释放一致性模型的基础上提出了急切更新释放一致性模型(Eager Release Consistency,简称 ERC)[23]和懒惰更新释放一致性模型(Lazy Release Consistency,简称 LRC)[64]。

在急切更新的释放一致性模型中,在临界区内的多个存数操作不是及时进行的,而是在执行 release 操作之前(即退出临界区之前)集中进行。这样,通过把多个存数操作合并在一起统一执行,减少了数据通信次数,这对于由软件实现的共享存储系统是十分必要的。采用急切更新的释放一致性模型的典型系统是建立在多机互联网络上的 Munin[23]系统。

5.2.6 懒惰更新释放一致性模型

懒惰更新的释放一致性模型则在减少通信和数据交换的次数方面更进一步。在懒惰更新的释放一致性模型中,由一个处理机对某单元的存数操作并不是由此处理机主动地传播到所有共享该单元的其他处理机,而是在其他处理机要用到此处理机所写的数据时(即其他处理机执行 acquire 操作时)再向此处理机索取该单元的最新备份。这样可以进一步减少通信量。采用懒惰更新的释放

一致性模型的典型系统是建立在多机互联网络上的 Treadmarks[65] 系统。

5.2.7 域一致性模型

域一致性(Scope Consistency,简称 ScC)模型[59]对访存事件次序的要求比懒惰更新释放一致性更宽松。在 LRC 中,当处理机 P 从处理机 Q 获得锁 l 时,处理机 Q 所看到(visible)的修改操作都被传给处理机 P。但在 ScC 模型中,只有用锁 l 保护起来的区域中所做的修改才会传送给 P。ScC 对事件次序的规定如下:

1. 在处理机 P 执行获得锁 l 的 acquire 操作之前,所有已执行的相对于锁 l 的访存操作必须相对于处理机 P 执行完。

2. 在处理机 P 执行访存操作之前,所有在此之前的 acquire 操作都已经完成。

一个访存操作相对于一个锁已执行完当且仅当该访存操作在由该锁保护的临界区内发出且该锁已被释放。

5.2.8 单项一致性模型

单项一致性(Entry Consistency,简称 EC)模型[18]通过同步变量和共享对象的紧密联系来进一步放松对访存事件次序的限制。它要求每个共享对象都和一个同步变量相关联,对任一共享变量的访问必须由与该变量关联的同步变量的同步操作来保护。这样,在对某一同步变量执行获取访问时,只有那些与该同步变量相关联的共享数据的修改信息被传送。显然,单项一致性模型进一步放松了对访存事件次序的限制,减少了处理机间的数据传送量,从而有利于系统性能的提高。但是,要求程序设计者为每个共享对象都指定一个同步变量与之关联,大大增加了程序设计的复杂性。

5.3 存储一致性模型的框架模型

5.3.1 从面向硬件设计到面向程序设计

由于最初弱一致性模型的提出是为了放松对访存事件次序的限制,因此上述存储一致性模型的定义大多是面向硬件的,即通过其对共享存储系统中访存事件次序的限制来描述一个存储一致性模型。这些限制通常体现为要求一个处理机发出的访存操作在什么时刻相对于其他处理机执行完毕(或被其他处理机所接受)。这种面向硬件的存储一致性模型的定义有利于硬件实现,但增加了程序设计的复杂度。事实上,程序员对一个访存操作到达其他处理机的时刻并不

感兴趣，因为从程序设计的角度来说，一个访存操作应该同时被所有处理机所接受。人们没有理由要求程序设计者考虑访存事件的可分割性（即一个访存操作在不同时刻被不同的处理机所接受），增加本来就不容易的并行程序设计的负担。此外，规定了存储一致性模型的访存事件次序也限制了硬件的进一步优化，因为某些硬件优化其实并不会改变存储一致性模型的真正语义。

因此，作为系统设计者和程序设计者之间界面的存储一致性模型应该表示的是共享存储系统的行为而不是对访存事件次序的规定。当然，要求共享存储系统体现出什么行为从某种意义上说也规定了该模型所允许的（最宽松）访存事件发生次序。

以图 5.1 中的程序为例，当这个程序在分别实现 RC、ERC 以及 LRC 的机器中执行时，处理机 P_i 所写的 y 的新值到达处理机 P_j 的时刻是不一样的。在 RC 模型中，P_i 执行操作“$y=1$”时就开始把 y 的新值传播到 P_j，而在 ERC 模型中，P_i 在执行操作“rel($l1$)”时才开始传播 y 的新值。LRC 模型则更进一步，它不要求 P_i 主动传播 y 的新值，只有在 P_j 执行“acq($l1$)”时才向 P_i 索取 y 的新值。

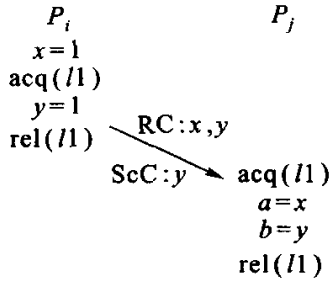


图 5.1 释放一致性与域一致性中数据的传递（初始值 $x=y=0$）

虽然在实现 RC 模型、ERC 模型和 LRC 模型时处理机 P_i 所写的 y 的新值在不同的时刻“到达”P_j，但 P_j 却在相同的时刻“观察到”y 的新值，即 P_j 通过赋值操作“$b=y$”读取 y 的新值。因此，该程序在执行 RC 模型、ERC 模型和 LRC 模型时都得到相同的结果“$a=b=1$”。然而，当该程序在实现 ScC 模型的系统中执行时却可能得到“$a=0, b=1$”的不同结果。因为根据 ScC 模型的要求，P_j 在执行“acq($l1$)”操作时只需看到由锁 $l1$ 保护的内容，而不是根据“happen-before-1”关系[64]在此操作之前的所有同步区间（Synchronization Interval）修改的内容。

图 5.1 中的程序在 RC 模型、ERC 模型和 LRC 模型中执行时都得到相同的结果，这表明 RC 模型、ERC 模型和 LRC 模型对同一程序体现出相同的行为，即它们在程序行为一级有相同的语义，因而对程序员有相同的实现正确程序设计的要求。它们不是不同的存储一致性模型，而是同一存储一致性模型的不同实

现方式。

对访存操作执行次序的限制只是实现存储一致性模型的手段，而不是一个存储一致性模型的本质特征。真正体现存储一致性模型本质的是决定程序执行结果以及程序的正确编程界面的某种机制。处理机之间的同步方式就是这样一种机制。

5.3.2 同步在并行程序中的作用

同步在共享存储系统并行程序中起着重要作用。共享存储系统中的多个处理机在执行并行程序时，虽然可以通过共享存储器进行通信，但还需要进行必要的同步操作，以得到确定的结果。可以通过两个简单的程序说明同步机制在共享存储系统并行程序中的作用。图 5.2 和图 5.3 分别给出了软件 DSM(Distrib-

```
#define N 1024
double (*a)[N],(*b)[N],(*c)[N];
main (int argc,char** argv)
{int i,j,k,begin,end;

a=(double (*)[N])malloc(N*N*8);
b=(double (*)[N])malloc(N*N*8);
c=(double (*)[N])malloc(N*N*8);

for (i=0;i<N;i++) for (j=0;j<N;j++){
  a[i][j]=1.0; b[i][j]=1.0;
}

begin=0;
end=N;
for (i=begin;i<end;i++)
  for (j=0,j<N;j++){
    c[i][j]=0.0;
    for (k=0;k<N;k++)
      c[i][j]+=a[i][k]*b[k][j];
  }

printmatrix(c)
jia_exit();
}
```

(a) 串行程序

```
#include <jia.h>
#define N 1024
double (*a)[N],(*b)[N],(*c)[N];
main(int argc,char** argv)
{int i,j,k,begin,end;
jia_init(argc,argv);
a=(double (*)[N])jia_alloc(N*N*8);
b=(double (*)[N])jia_alloc(N*N*8);
c=(double (*)[N])jia_alloc(N*N*8);
if (jiapid==0)
for (i=0;i<N;i++) for (j=0;j<N;j++){
  a[i][j]=1.0;  b[i][j]=1.0;
}
jia_barrier();
begin=N*jiapid/jiahosts;
end=N*(jiapid+1)/jiahosts;
for (i=begin;i<end;i++)
  for (j=0;j<N;j++){
    c[i][j]=0.0;
    for (k=0;k<N;k++)
      c[i][j]+=a[i][k]*b[k][j];
  }
jia_barrier();
if (jiapid==0) printmatrix(c);
exit(0);
}
```

(b) 并行程序

图 5.2 矩阵乘法程序中的同步操作

uted Shared Memory,分布式共享存储)系统 JIAJIA 的矩阵乘法并行程序和积分求 π 的并行程序。

在图 5.2 的矩阵乘法并行程序中,有两个用于全局同步机制的 barrier 操作(barrier 操作要求所有处理机都就绪后才能继续执行后续指令)。在分配完共享空间之后,由 0 号处理机对 a 和 b 两个矩阵进行初始化,其他处理机则在第 1 个 barrier 指令处等待,等全齐后才能开始进行并行的矩阵乘法。所有处理机都完成自己的任务后,在第 2 个 barrier 指令处等全齐,再由 0 号处理机打印结果矩阵。在这个程序中,两个 barrier 操作都是不可少的。如果没有第 1 个 barrier 指令,则除了 0 号处理机外的其他处理机都可能对未初始化的矩阵进行矩阵乘法;如果没有第 2 个 barrier 指令,则 0 号处理机可能在其他处理机没有完成相应部分的矩阵乘法前就打印结果矩阵。

```
#define n 1000000
void main (int argc,char ** argv)
{int i,begin,end;
double sum,x,h, * pa;

pa=(double * )malloc(sizeof(double));



h=1.0/(double)n; sum=0.0;
begin=1;
end=n;
for (i=begin;i<=end; i++ ){
  x=h * ((double)i-0.5)
  sum+ =4.0/(1.0+x * x);
}
 * pa=h * sum;




printf(" %.16f \n ", * pa);
exit(0);

}
```

(a) 串行程序

```
#include <jia.h>
#define  n  1000000
void main (int argc,char  ** argv)
{int i,begin,end;
double sum,x,h,mypi, * pa;
jia_init(argc,argv);
pa=(double * )jia_alloc(sizeof(double));
if (jiapid==0)  * pa=0.0;
jia_barrier();
h=1.0/(double)n; sum=0.0;
begin=n/jiahosts * jiapid+1;
end=n/jiahosts * (jiapid+1);
for (i=begin;i<=end;i++ ){
  x=h * ((double)i-0.5)
  sum+ =4.0/(1.0+x * x);
}
mypi=h * sum;
jia_lock(1);
 * pa= * pa+mypi;
jia_unlock(1);
jia_barrier();
if (jiapid==0) printf(" %.16f \n ", * pa);

jia_exit();
}
```

(b) 并行程序

图 5.3 积分求 π 程序中的同步操作

在图 5.3 的积分求 π 并行程序中，每个处理机计算一部分和，最终把每个处理机的部分和累加到一起形成正确 π 值。在相加过程中使用了 lock 和 unlock (即 acquire 和 release)操作，以保证每个处理机通过临界区顺序地把部分和累加到共享变量中。如果不使用临界区，除非硬件支持不可分割(Atomic)的累加操作，否则结果值就有可能出错。

可见，在共享存储系统并行程序中，为了得到确切的结果，同步操作是不可避免的。各种弱一致性模型正是利用了共享存储系统并行程序的这种特点，使用专门的同步操作并在同步点维护数据一致性，以放松对访存事件次序的限制，并且在大多数情况下不用专为数据一致性而增加同步操作。在顺序一致的系统中，可以利用普通访存操作实现进程间同步(如图 2.1 中的程序就是一种简单的同步机制)，也可以使用专门的同步操作。

在共享存储系统并行程序中，处理机间的同步机制决定了处理机间冲突访问的执行次序。例如，在矩阵乘法并行程序中，初始化时 0 号处理机写初始值至 a、b 矩阵和并行计算时其他处理机读 a、b 矩阵中的值是冲突访问，而程序中的第 1 个 barrier 操作决定了 0 号处理机的写数操作发生在其他处理机的读数操作之前。因此，处理机间的同步不仅直接影响程序员能否编写正确的程序，也决定了并行程序的执行结果。

5.3.3 框架模型的定义

作为程序设计者和系统设计者间的一个界面，存储一致性模型应该以一种双方都能接受的方式来描述。当程序员写程序时，可以根据存储一致性模型的要求知道程序中每个访存操作所产生的行为，从而决定如何设计满足存储一致性模型的正确程序。另一方面，存储一致性模型应该为系统设计者进行优化设计而留有余地。因此，存储一致性模型的定义不应对访存事件次序做出具体的限制，而应指出满足该模型的系统结构在执行并行程序时应体现的行为。这些行为可以通过对访存事件发生次序进行限制来实现。

在共享存储系统中，并行程序的执行结果由程序中冲突访问的执行次序确定。如果两个访存操作访问同一单元，且其中至少有一个是存数操作，则称这两个访存操作是冲突的。在同一进程中冲突访问的执行次序由程序序规定，在后面的讨论中假设同一进程中的数据相关性不被破坏，即同一进程中冲突访问的执行次序总是与程序序一致。如何决定不同进程间冲突访问的执行次序正是存储一致性模型要解决的问题。存储一致性模型通过进程间的同步来确定进程间冲突访问的执行次序。进程间的同步规定一个处理机所写的值在何时通过何种方式传播到其他处理机。基于这种认识，可以对存储一致性模型做如下定义。

定义 5.1 一个存储一致性模型 M 是一个二元组$\langle C_M, SYN_M\rangle$，其中 C_M

是对访存操作的一个分类(或对同步操作的一种描述),SYN_M 是确定处理机间访存操作执行次序的一种进程间的同步机制。

根据定义 5.1,不同的存储一致性模型有着独立的用以确定处理机间访存事件执行次序的同步机制。定义 5.1 是面向程序设计者的,因为程序设计者为了设计正确的程序,必须考虑并行程序的进程间同步。同时,定义 5.1 也为系统设计者留有提高系统性能的余地,因为它没有直接规定一个存储一致性模型的访存事件次序。因此,在一定程度内,同种同步机制的访存事件次序限制可以不同。相应地,系统性能和实现复杂度也不一样。例如,RC 模型、ERC 模型和 LRC 模型就是同种存储一致性模型的不同实现。它们有相同的访存操作分类和相同的同步机制。在 RC 模型、ERC 模型和 LRC 模型中,处理机 P_i 发出的访存操作 u 都要通过"$u_i \longrightarrow \mathrm{rel}_i(l) \longrightarrow \mathrm{acq}_j(l) \longrightarrow v_j$"的执行序列才能被处理机 P_j 所接受,其中,u_i 和 v_j 是冲突访问(本章用下标表示执行某种操作的处理机号)。

共享存储系统中并行程序的执行结果由程序中冲突访问的执行次序决定,而冲突访问的执行次序又是由同步操作的执行次序决定的,因此可以把同步操作的执行次序定义为并行程序的一个执行。

定义 5.2 程序 PRG 在存储一致性模型 M 中的一个执行,记为 $E_M(PRG)$,是该程序中同步操作的一个定序。

例 5.1 以图 5.1 中的程序为例,该程序在 RC 模型中两个可能的执行序列为:

$$E^1_{\mathrm{RC}}(PRG) = \{(\mathrm{rel}_i(l1), \mathrm{acq}_j(l1))\} = \mathrm{rel}_i(l1) \xrightarrow{E} \mathrm{acq}_j(l1)$$

和

$$E^2_{\mathrm{RC}}(PRG) = \{(\mathrm{rel}_j(l1), \mathrm{acq}_i(l1))\} = \mathrm{rel}_j(l1) \xrightarrow{E} \mathrm{acq}_i(l1)$$

在一个并行执行中,一旦同步操作的执行次序被确定了,其他有关的访存操作执行次序也就确定了。

定义 5.3 在存储一致性模型 M 中的一个执行 $E_M(PRG)$ 的同步序,记为 $SO_M(E_M(PRG))$,是在该执行中被 M 的同步机制 SYN_M 所定序的普通访存操作对的集合。即

$$SO_M(E_M(PRG)) = \{(u, v) \mid u \text{ 在 } E_M(PRG) \text{ 中被 } SYN_M \text{ 定序在 } v \text{ 之前执行}\}$$

存储一致性模型的本质是一种确定处理机间访存操作次序的同步机制,因此一个存储一致性模型就可以用定义 5.3 中的同步序来描述。下面以顺序一致性模型、释放一致性模型以及域一致性模型为例来说明上述概念的具体应用。

1. 顺序一致性。顺序一致性模型并不区分普通访存操作和同步操作,所有

访存操作都可以看做是同步操作。根据定义 5.2,程序 PRG 在 SC 中的一个执行是该程序中冲突访问的一个定序,即如果 u_i 和 v_j 是程序 PRG 中的冲突访问,则$(u_i, v_j) \in E_{SC}(PRG)$或者$(v_j, u_i) \in E_{SC}(PRG)$。

根据 SC 的语义,如果"$u_i \xrightarrow{E} v_j$",则可以认为 SC 的同步机制把 u_i 定序在 v_j 之前执行,即"$u_i \xrightarrow{SO_{SC}(E_{SC}(PRG))} v_j$"。因此,可以这样表示顺序一致性模型:

$$\begin{cases} C_{SC} = \{r(x), w(x)\} \\ SO_{SC}(E_{SC}(PRG)) = \{(u_i, v_j) \mid u_i \xrightarrow{E} v_j\} \end{cases} \tag{5.1}$$

2. 释放一致性。根据定义 5.2 程序 PRG 在 RC 中的一个执行可以用 release-acquire 对的集合来表示。根据 RC 的语义,如果"$u_i \xrightarrow{PO} \mathrm{rel}_i(l) \xrightarrow{E} \mathrm{acq}_j(l) \xrightarrow{PO} v_j$)",则 u_i 在 v_j 之前执行。因此,可以这样描述释放一致性模型:

$$\begin{cases} C_{RC} = \{r(x), w(x), \mathrm{acq}(l), \mathrm{rel}(l)\} \\ SO_{RC}(E(PRG)) = \{(u_i, v_j) \mid u_i \xrightarrow{PO} \mathrm{rel}_i(l) \xrightarrow{E} \mathrm{acq}_j(l) \xrightarrow{PO} v_j\}^+ \end{cases} \tag{5.2}$$

其中,$\{\cdots\}^+$是$\{\cdots\}$的传递闭包。

3. 域一致性。域一致性与释放一致性的不同之处在于,如果 u_i 被"$\mathrm{rel}_i(l) \xrightarrow{E} \mathrm{acq}_j(l)$"定序在 v_j 之前执行,则 u_i 必须在锁 l 所保护的临界区内。域一致性模型可这样表示:

$$\begin{cases} C_{ScC} = \{r(x), w(x), \mathrm{acq}(l), \mathrm{rel}(l)\} \\ SO_{ScC}(E(PRG)) = \{(u_i, v_j) \mid \mathrm{acq}_i(l) \xrightarrow{PO} u_i \xrightarrow{PO} \mathrm{rel}_i(l) \xrightarrow{E} \mathrm{acq}_j(l) \xrightarrow{PO} v_j\}^+ \end{cases} \tag{5.3}$$

5.3.4 程序的正确性

作为程序设计者与结构设计者之间的一个界面,存储一致性模型应该给出正确程序设计和正确结构设计的标准。

一个并行执行中,进程内部的冲突访问由程序序来定序,进程间的冲突访问

由同步序来定序，把程序序和同步序的并集称为该并行执行的发生序。

定义 5.4 若 $E_M(PRG)$是程序 PRG 在一致性模型 M 中的一个执行，该执行的同步序与程序序的并集称为该执行的发生序(heppen-before order)，记为 $HB_M(E_M(PRG))$，即

$$HB_M(E_M(PRG)) = PO(PRG) \cup SO_M(E_M(PRG)) \tag{5.4}$$

对于一个程序来说，正确的存储一致性模型应该为该程序中的所有冲突访问定序。反之，判断一个程序是否符合存储一致性模型 M 的标准是该程序中的所有冲突访问是否都能被 M 定序。基于这种认识，可以给出判断一个程序是否满足存储一致性模型 M 的标准。

定义 5.5 程序 PRG 符合存储一致性模型 M 的要求(即程序 PRG 是 M 中的正确程序)当且仅当对于该程序在 M 中的任一执行 $E_M(PRG)$，程序中所有冲突访问对都被 $HB_M(E_M(PRG))$定序。

例 5.2 以图 5.1 中的程序为例。如前所述，该程序在释放一致性模型中一个正确的执行是：

$$E_{RC}(PRG) = \{(\mathrm{rel}_i(l1), \mathrm{acq}_j(l1))\} = \mathrm{rel}_i(l1) \xrightarrow{E} \mathrm{acq}_j(l1)$$

该执行的同步序是：

$$SO_{RC}((E_{RC}PRG)) = \{(w_i(x), r_j(x)), (w_i(x), r_j(y)), (w_i(y), r_j(x)), (w_i(y), r_j(y))\}$$

因此，该执行的发生序是：

$$\begin{aligned} HB_{RC}(E_{RC}(PRG)) &= PO(PRG) \cup SO_{RC}(E_{RC}(PRC)) \\ &= \{(w_i(x), r_j(x)), (w_i(x), r_j(y)), (w_i(y), r_j(x)), (w_i(y), \\ &\quad r_j(y)), (w_i(x), w_i(y)), (r_j(x), r_j(y))\} \end{aligned}$$

然而，当这个程序在域一致性模型中执行时，所得到的发生序是与此不同的。假设域一致性模型中同步操作的执行次序与在释放一致性模型中一致，即

$$E_{ScC}(PRG) = \{(\mathrm{rel}_i(l1), \mathrm{acq}_j(l1))\} = \mathrm{rel}_i(l1) \xrightarrow{E} \mathrm{acq}_j(l1)$$

该执行的同步序与发生序为：

$$SO_{ScC}(E_{ScC}(PRG)) = \{(w_i(y), r_j(x)), (w_i(y), r_j(y))\}$$

$$HB_{ScC}(E_{ScC}(PRG)) = \{(w_i(y), r_j(x)), (w_i(y), r_j(y)), (w_i(x), w_i(y)), (r_j(x), r_j(y))\}$$

从例 5.2 可以看出，在执行 $E_{RC}(PRG)$中，所有冲突访问都被定序。而在执行 $E_{ScC}(PRG)$中，只有冲突访问 $w_i(y)$和 $r_j(y)$被定序，冲突访问 $w_i(x)$和 $r_j(x)$没有被定序。如果上述 $E_{RC}(PRG)$和 $E_{ScC}(PRG)$是该程序的惟一执行(即程序中有其他机制保证执行"$\mathrm{rel}_j(l1) \xrightarrow{E} \mathrm{acq}_i(l1)$"不可能发生)，则图 5.1 中的程序 PRG 是 RC 模型中的正确程序，但不是 ScC 模型中的正确程序。事实上，在 ScC 模型中，取数操作"$a = x$"的返回值不确定。

5.3.5 系统设计的正确性

一个存储一致性模型 M 若能称做正确实现，必须保证所实现的系统正确执行所有满足该一致性模型的程序。一个执行的结果由冲突访问的执行次序决定，而冲突访问执行次序又是由程序序和同步操作的执行次序决定的，因此一个正确的系统应该对同步操作的执行次序以及同一进程内操作的执行次序施以合理的限制，以确保错误的执行不会发生。

顺序一致性提供了判断一个并行执行正确与否的标准，它规定一个正确的并行执行必须得到与串行执行相同的结果。在 SC 模型中一个并行执行 $E(PEG)$ 正确的条件是 $E(PRG)$ 所确定的冲突访问执行次序与程序序 $PO(PRG)$ 一致，即 $E(PRG)\cup PO(PRG)$ 无圈。其他存储一致性模型也采用了顺序一致性模型所规定的正确性标准，即只要一个程序满足某种弱一致性模型的要求，则该程序在实现这种弱一致性模型的系统中执行的结果与在实现顺序一致性模型的系统中执行的结果是一样的。因此，上述对 SC 模型中的执行正确性条件对其他存储一致性模型也适用，即存储一致性模型 M 中的执行 $E_M(PRG)$ 正确的条件是它所确定的访存操作发生次序与程序序一致。

根据上述分析，可以按照定义 5.6 判断一个系统是否满足存储一致性模型的要求。

定义 5.6 一个系统满足存储一致性模型 M 的要求当且仅当该系统对 M 中正确程序的任一执行 $E_M(PRG)$ 都有 $HB_M(E_M(PRG))$ 无圈。

例 5.3 以图 5.4 中的程序为例。根据定义 5.2，该程序段在 RC 模型中有 4 个可能的执行：

$$E^1_{RC}(PRG)=\{(\mathrm{rel}_i(l1),\mathrm{acq}_j(l1)),(\mathrm{rel}_i(l2),\mathrm{acq}_j(l2))\}$$
$$E^2_{RC}(PRG)=\{(\mathrm{rel}_i(l1),\mathrm{acq}_j(l1)),(\mathrm{rel}_j(l2),\mathrm{acq}_i(l2))\}$$
$$E^3_{RC}(PRG)=\{(\mathrm{rel}_j(l1),\mathrm{acq}_i(l1)),(\mathrm{rel}_i(l2),\mathrm{acq}_j(l2))\}$$
$$E^4_{RC}(PRG)=\{(\mathrm{rel}_j(l1),\mathrm{acq}_i(l1)),(\mathrm{rel}_j(l2),\mathrm{acq}_i(l2))\}$$

由式(5.2)及式(5.4)，这 4 个执行的发生序为：

$$HB_{RC}(E^1)=PO(PRG)\cup\{(w_i(y),r_j(y)),(r_i(x),w_j(x)),(r_i(x),r_j(y)),(w_i(y),w_j(x))\}$$
$$HB_{RC}(E^2)=PO(PRG)\cup\{(w_i(y),r_j(y)),(w_j(x),r_i(x))\}$$
$$HB_{RC}(E^3)=PO(PRG)\cup\{(r_j(y),w_i(y)),(r_i(x),w_j(x)),\cdots\}$$
$$HB_{RC}(E^4)=PO(PRG)\cup\{(r_j(y),w_i(y)),(w_j(x),r_i(x)),(r_j(y),r_i(x)),(w_j(x),w_i(y))\}$$

P_i	P_j
acq ($l1$)	acq ($l2$)
$y=1$	$x=1$
rel ($l1$)	rel($l2$)
acq($l2$)	acq ($l1$)
$a=x$	$b=y$
rel ($l2$)	rel ($l1$)

图 5.4 释放一致性的程序例子(初始值 $x=y=0$)

显然,在 $HB_{RC}(E^3)$中有一个圈 $w_i(y)\xrightarrow{PO}r_i(x)\xrightarrow{E}w_j(x)\xrightarrow{PO}r_j(y)\xrightarrow{E}w_i(y)$,表明这是 RC 模型中的一个错误执行。因此,RC 模型的正确实现应当保证执行 $E^3_{RC}(PRG)$不会发生。

5.4 系统设计正确性的证明

5.4.1 基本概念

存储一致性模型通常是通过对访存事件次序的限制来实现的。因此,可以把为了实现某种存储一致性模型而对访存事件次序施加的一组限制称为该模型的一个实现。

存储一致性模型对访存事件次序的限制通常就是把执行序和程序序赋予物理上的先后发生的意义,即把发生序具体化。例如,"在一个访存操作允许被发出之前,同一进程中所有先于它的访存操作都已经彻底完成"就是对程序序赋予物理上先后发生的意义。

如果一个系统不满足一致性模型 M,则存在 M 中正确程序 PRG 的一个执行 $E_M(PRG)$,使得 $HB_M(E_M(PRG))$中包含圈。这里引进 $HB_M(E_M(PRG))$中关键圈的概念来实现正确性的证明。

定义 5.7 若 $\sigma(PRG)$是 $HB_M(E_M(PRG))$中的一个圈且 $HB_M(E_M(PRG))$中无 $\sigma(PRG)$的弦,则称 $\sigma(PRG)$是 $HB_M(E_M(PRG))$中的一个关键圈。

(u_1,u_n)称做 $\sigma(PRG)$中的一条弦如果满足$(u_1,u_n)\in HB_M(E_M(PRG))$且在 σ 中有一条从 u_1 到 u_n 的路径如下:

$$(u_1,u_2),(u_2,u_3),\cdots,(u_{n-1},u_n)\quad n>2$$

5.4.2 顺序一致性的正确实现

前几章证明了 WC 条件以及 GPPO 条件是实现顺序一致性模型的充分条

件，为了保持本章的完整性，用本章建立的数学模型重新证明该结论的正确性。

WC条件要求对同一单元的所有存数操作以相同的次序到达所有处理机，即若 u 和 v 都是写操作且 $u \xrightarrow{E} v$，则 $u^i < v^i (i=1,2,\cdots,N)$。前面已经指出，一个执行满足写一致条件是该执行正确的前提。

参考文献[83]给出了一个可行的共享存储系统中正确执行的访存次序条件：在一个访存操作允许被发出之前，同一进程中所有先于它的访存操作都已经“彻底完成”。其中，一个存数操作“彻底完成”是指该操作已相对于所有处理机完成；一个取数操作“彻底完成”是指该取数操作已相对于所有处理机完成(即这个取数操作取回的值已经确定)且写此值的存数操作已彻底完成。这个条件称为 GPPO(Globally Performed in Program Order)条件。

WC条件和GPPO条件对访存事件次序的限制如下：

$$\begin{cases} u \xrightarrow{PO} v \Rightarrow u^i < v^j \\ w \xrightarrow{E} r \xrightarrow{PO} v \Rightarrow w^i < v^j \\ w_1 \xrightarrow{E} w_2 \Rightarrow w_1^i < w_2^i \\ r \xrightarrow{E} w \Rightarrow r^c < w^c \\ w \xrightarrow{E} r \Rightarrow w^c < r^c \end{cases} \tag{5.5}$$

其中，u、v 是任意访存操作，w、w_1 和 w_2 是写操作，r 是读操作，c 是发出相应存取操作的处理机号，$i,j=1,2,\cdots,N$(N 是系统中处理机的个数)。

定理 5.1 WC条件和GPPO条件对访存事件次序的限制是顺序一致性的一个正确实现。

证明 根据定义5.6，如果对某系统中的任一执行 $E_{SC}(PRG)$，$HB_{SC}(E_{SC}(PRG))$都无圈，则该系统是SC模型的正确实现。

假设在 $HB_{SC}(E_{SC}(PRG))$中有圈。设 σ 是 $HB_{SC}(E_{SC}(PRG))$中的一个关键圈，u 和 v 是 σ 中两个相邻的操作，即 $u \xrightarrow{HB} v$。

若 $u \xrightarrow{PO} v$，则由式(5.5)可得 $u^i < v^j (i,j=1,2,\cdots,N)$。

若 $u \xrightarrow{SO} v$，则由式(5.1)可知 $u \xrightarrow{E} v$。

1. 若 u 和 v 都是存数操作，则由式(5.5)可得 $u^i < v^i (i=1,2,\cdots,N)$。

2. 若 u 是存数操作，v 是取数操作，设 x 是关键圈 σ 中操作 v 的下一个操作。

(1) 若 $u \xrightarrow{E} v \xrightarrow{PO} x$,则由式(5.5)可得 $u^i < x^j (i,j=1,2,\cdots,N)$。

(2) 若 $u \xrightarrow{E} v \xrightarrow{E} x$,则 x 肯定是一个存数操作(因为 v 和 x 冲突且 v 是取数操作)。因此,u 和 x 都是存数操作且 $u \xrightarrow{E} x$,则构成了 σ 中的一条弦,即这种情况是不可能的。

3. 若 u 是由处理机 P_c 发出的取数操作且 v 是存数操作,而由式(5.5)可得 $u^c < v^c$。设 x 是关键圈 σ 中 v 的下一个操作。

(1) 若 $u \xrightarrow{E} v \xrightarrow{PO} x$,则由式(5.5)中的第 1 种情况可得 $v^i < x^j (i,j=1,2,\cdots,N)$。因此,$u^c < x^j (j=1,2,\cdots,N)$。

(2) 若 $u \xrightarrow{E} v \xrightarrow{E} x$,则 x 肯定是一个取数操作(因为若 x 是存数操作则 u 和 x 冲突,且不难证明 $u \xrightarrow{E} x$,构成了关键圈 σ 中的一条弦)。设 y 是关键圈 σ 中操作 x 的下一个操作,即 $v \xrightarrow{E} x \xrightarrow{HB} y$,根据上述推导,有 $v^i < y^j (i,j=1,2,\cdots,N)$。因此,$u^c < y^j (j=1,2,\cdots,N)$。

由此可知,若 u、v、x、y 是关键圈 σ 中 4 个相邻的操作,即 $u \xrightarrow{HB} v \xrightarrow{HB} x \xrightarrow{HB} y$,则有 $u^i < v^j$,或 $u^i < v^i$,或 $u^i < x^j$,或 $u^c < x^j$,或 $u^c < y^j$ 成立,其中 c 是发出相应取数操作的处理机号,$i,j=1,2,\cdots,N$。上述结论可以简化为,若 $u \xrightarrow{HB} v \xrightarrow{HB} x \xrightarrow{HB} y$,则 $u^c < v^j$ 或 $u^j < v^j$ 或 $u^c < x^j$ 或 $u^c < y^j$ 必有一个成立。

在关键圈 σ 中继续上述过程,不难推出 $u^j < u^j$ 或 $u^c < u^j$ 或 $v^c < v^j$ 或 $x^c < x^j$,其中 c 是一个特定值,$j=1,2,\cdots,N$。但是这在关键圈中是不可能的。

5.4.3 释放一致性的正确实现

传统的 RC 模型(如在 DASH 系统中实现的 RC 模型)、ERC 模型以及 LRC 模型都是释放一致性模型的不同实现,下面证明传统 RC 模型实现的正确性。

由式(5.2)可知,在 RC 模型中,$u \xrightarrow{SO_{RC}} v$ 的含义是:

$$u \xrightarrow{PO} \text{rel} \xrightarrow{E} \cdots \xrightarrow{E} \text{acq} \xrightarrow{PO} v$$

其中,省略部分表示一些连续的 $\text{acq} \xrightarrow{PO} \text{rel}$ 对。值得指出的是,根据 RC 模型的要求,这些同步操作不一定访问同一把锁。

可以把 rel 看做写操作,把 acq 看做读操作。由于同步操作是顺序一致的,假设同步操作的访存事件次序满足式(5.5)的要求,则有 $\text{rel}(l1) \xrightarrow{E} \text{acq}(l1) \xrightarrow{PO} \text{rel}(l2) \Rightarrow \text{rel}^i(l1) < \text{rel}^j(l2)$ 以及 $\text{rel} \xrightarrow{E} \text{acq} \Rightarrow \text{rel}^c < \text{acq}^c$,其中 $i,j=1,2,$

$\cdots, N, c$ 是发出 acq 操作的处理机号。因此，$\text{rel} \xrightarrow{E} \cdots \xrightarrow{E} \text{acq} \Rightarrow \text{rel}^i < \text{acq}^c$ $(i = 1,2,\cdots,N)$。

根据传统 RC 模型实现的要求，在任一普通访存操作允许被执行之前，所有在同一处理机中先于这一访存操作的 acquire 操作都已完成；在任一 release 操作允许被执行之前，所有在同一处理机中先于这个 release 操作的普通访存操作都已完成。因此，$u \xrightarrow{PO} \text{rel} \xrightarrow{E} \cdots \xrightarrow{E} \text{acq} \xrightarrow{PO} v \Rightarrow u^i < v^j$ $(i,j = 1,2,\cdots,N)$。

根据上述分析，传统 RC 模型实现对访存事件次序的限制可表述为：

$$\begin{cases} u \xrightarrow{PO} \text{rel} \xrightarrow{PO} v \Rightarrow u^i < v^j \\ u \xrightarrow{PO} \text{acq} \xrightarrow{PO} v \Rightarrow u^c < v^j \\ u \xrightarrow{SO_{RC}} v \Rightarrow u^i < v^j \end{cases} \tag{5.6}$$

其中，u 和 v 是访存操作，c 是发出 u 操作的处理机号，$i,j = 1,2,\cdots,N$。

定理 5.2 传统 RC 模型实现对访存事件次序的限制是释放一致性模型的正确实现。

证明 根据定义 5.6，释放一致性模型正确实现的条件是，该实现对于 RC 模型中正确程序的任一执行 $E_{RC}(PRG)$，都有 $HB_{RC}(E_{RC}(PRG))$ 无圈。

假设 $HB_{RC}(E_{RC}(PRG))$ 中有圈，σ 是 $HB_{RC}(E_{RC}(PRG))$ 中的一个关键圈，u 和 v 是 σ 中任意两个相邻的操作，即 $u \xrightarrow{PO} v$ 或 $u \xrightarrow{SO_{RC}} v$。

1. 若 $u \xrightarrow{PO} v$，设 x 是关键圈 σ 中操作 v 的下一个操作，即 $u \xrightarrow{PO} v \xrightarrow{HB} x$，则或者 $u \xrightarrow{PO} v \xrightarrow{PO} x$，或者 $u \xrightarrow{PO} v \xrightarrow{SO_{RC}} x$。显然从 $u \xrightarrow{PO} v \xrightarrow{PO} x$ 可以推出 $u \xrightarrow{PO} x$，而根据式(5.2)从 $u \xrightarrow{PO} v \xrightarrow{SO_{RC}} x$ 可得 $u \xrightarrow{SO_{RC}} x$。这两种情况都违背 σ 是关键圈的假设，因而是不可能发生的。

2. 若 $u \xrightarrow{SO_{RC}} v$，由式(5.6)可得 $u^i < v^j$ $(i,j = 1,2,\cdots,N)$。

由此可知，若 $u \xrightarrow{HB} v$，则必有 $u^i < v^j$ $(i,j = 1,2,\cdots,N)$。

在关键圈 σ 中继续上述过程不难推出，对任意的 $i,j = 1,2,\cdots,N$，都有 $u^i < u^j$。但这是不可能发生的。

5.4.4 域一致性的正确实现

参考文献[59]中给出的 ScC 模型对访存事件次序的规定如下：

1. 在处理机 P 执行获得锁 l 的 acquire 操作之前，所有相对于锁 l 已执行

的访存操作必须相对于处理机 P 执行完。

2. 在处理机 P 执行访存操作时，所有在此之前的 acquire 操作都已经完成。

一个访存操作相对于一个锁已执行完当且仅当该访存操作在该锁保护的临界区内发出且该锁已被释放。

上述 ScC 模型对访存事件次序的规定可以表示如下：

$$\begin{cases} u \xrightarrow{PO} \text{acq} \xrightarrow{PO} v \Rightarrow u^c < v^j \\ \text{acq}(l) \xrightarrow{PO} u \xrightarrow{PO} \text{rel}(l) \xrightarrow{E} \cdots \xrightarrow{E} \text{acq}(l) \xrightarrow{PO} v \Rightarrow u^c < v^j \end{cases} \tag{5.7}$$

其中，u 和 v 是任意访存操作，$j=1,2,\cdots,N$，c 是发出访存操作 v 的处理机号，省略部分表示一串 $\text{acq}(l)\xrightarrow{PO}\text{rel}(l)$ 对。需要指出的是，根据 ScC 模型的要求，上述所有同步操作访问同一把锁。

由式(5.3)可以看出，$\text{acq}(l)\xrightarrow{PO}u\xrightarrow{PO}\text{rel}(l)\xrightarrow{E}\cdots\xrightarrow{E}\text{acq}(l)\xrightarrow{PO}v$ 相当于 $u\xrightarrow{SO_{ScC}}v$。因此式(5.7)可以简化为：

$$\begin{cases} u \xrightarrow{PO} \text{acq} \xrightarrow{PO} v \Rightarrow u^c < v^j \\ u \xrightarrow{SO_{ScC}} v \Rightarrow u^c < v^j \end{cases} \tag{5.8}$$

定理 5.3 ScC 模型对访存事件次序的规定是域一致性模型的正确实现。

证明 根据定义 5.6，域一致性模型正确实现的条件是该实现对 ScC 模型中正确程序的任一执行 $E_{ScC}(PRG)$，都有 $HB_{ScC}(E_{ScC}(PRG))$无圈。

假设 $HB_{ScC}(E_{ScC}(PRG))$中有圈，σ 是$HB_{ScC}(E_{ScC}(PRG))$中的一个关键圈，u 和 v 是 σ 中两个相邻的操作，即 $u\xrightarrow{PO}v$ 或 $u\xrightarrow{SO_{ScC}}v$。

1. 若 $u\xrightarrow{PO}v$，有两种可能的情况。

(1) u 和 v 不在同一个临界区内，u 和 v 至少被一个 acq 操作隔开。由式(5.8)可得，$u^c<v^j$ 对任意的 $j=1,2,\cdots,N$ 成立。

(2) u 和 v 在同一个临界区内，设 x 是关键圈 σ 中操作 v 的下一个操作，即 $u\xrightarrow{PO}v\xrightarrow{HB}x$，则或者 $u\xrightarrow{PO}v\xrightarrow{PO}x$，或者 $u\xrightarrow{PO}v\xrightarrow{SO_{ScC}}x$。显然从 $u\xrightarrow{PO}v\xrightarrow{PO}x$ 可以推出 $u\xrightarrow{PO}x$，而根据式(5.3)从 $u\xrightarrow{PO}v\xrightarrow{SO_{ScC}}x$ 可得 $u\xrightarrow{SO_{ScC}}x$。这两种情况都违背 σ 是关键圈的假设，因而是不可能的。

2. 若 $u \xrightarrow{SO_{SC}} v$，由式(5.8)可得 $u^c < v^j$，$(j=1,2,\cdots,N)$。

根据上述分析，若 $u \xrightarrow{HB} v$，则必有 $u^c < v^j (j=1,2,\cdots,N)$。

在关键圈 σ 中继续上述过程后则不难推出，对任意的 $j=1,2,\cdots,N$，都有 $u^c < u^j$。但这是不可能发生的。

5.5 小 结

本章介绍了一个存储一致性模型的数学模型。与传统的从访存事件次序的角度来刻画存储一致性模型的方法不同，该模型从存储一致性模型体现出的行为的角度来描述存储一致性模型。

基于并行执行的行为由该执行中冲突访问的执行次序确定这种认识，可以把存储一致性模型看做是一种处理机间的同步机制，该同步机制确定处理机间冲突访问的执行次序。并行执行的结果由同步序与程序序共同确定。存储一致性模型 M 中的一个执行本质上是对程序中同步操作的一个定序。

存储一致性模型 M 中正确程序的条件是，对该程序在 M 中的任一执行，程序中所有冲突访问都被程序序或同步序定序。一个系统满足存储一致性模型 M 的条件是，对于该系统中所有可能的执行，该执行的同步序与程序序一致。

存储一致性模型实现的本质是该模型对访存事件次序的一组限制。本章基于这个认识，以顺序一致性、释放一致性以及域一致性为例给出了证明存储一致性模型正确实现的方法。

不难看出，本章建立的存储一致性模型的数学模型是前几章建立的执行正确性模型以及访存事件正确性条件的推广。

第6章 高速缓存一致性协议

6.1 引　　言

高速缓冲存储器(Cache)技术在共享存储系统中占有重要地位。它不仅可以削弱由处理机和存储器的空间分布引起的大延迟,而且可以减少多个处理机访问共享存储器时的冲突。然而,Cache 在多处理机中会引起 Cache 一致性问题,即如何使同一单元在不同 Cache 以及主存中的多个备份保持数据一致。Cache 一致性问题的解决不仅决定系统的正确性,而且对系统性能有重要影响。人们已经提出了若干 Cache 一致性协议来解决这个问题。

本章讨论 Cache 一致性协议中的一些关键问题。Cache 一致性协议的本质是,把某个处理机新写的值传播给其他处理机以确保所有处理机看到一致的共享存储内容。基于这种认识,本章先从不同的侧面介绍如何实现高效的传播。在此基础上,提出了一个实现域存储一致性模型的基于锁的新型 Cache 一致性协议,并证明其正确性。同传统的基于目录的协议相比较,基于锁的协议通过附带在锁上的一致性信息来维护一致性,从而避免了由目录引起的存储开销和系统复杂度。与目录协议相比,基于锁的协议更加简单、有效且可伸缩性好。

6.2 Cache 一致性协议回顾

如何解决 Cache 一致性问题,即如何保持数据在多个 Cache 和主存中的多个备份的一致性,是实现共享存储系统的关键。Cache 一致性问题的解决不仅决定系统的正确性,而且对系统性能有重要影响。

Cache一致性协议都是为实现某种存储一致性模型而设计的。存储一致性模型对Cache一致性协议提出一致性要求,即Cache一致性协议应该实现怎样的"一致性"。例如,在释放一致性中,一个处理机对共享变量写的新值,其他处理机只有等到该处理机释放锁后才能看到;而在顺序一致性中,一个处理机写的值会立刻传播给所有处理机。因此,顺序一致性(Sequential Consistency,简称SC)和释放一致性(Release Consistency,简称RC)所描述的"一致性观点"(Coherent View)不同,实现SC的Cache一致性协议与实现RC的Cache一致性协议也就不一样。

人们已经提出了若干Cache一致性协议来解决Cache一致性问题。Cache一致性协议的实质是把一个处理机新写的值传播给其他处理机的一种机制。为了实现高效的传播,一致性协议通常需要考虑以下几个方面:

1. 如何传播新值:写使无效与写更新。
2. 怎样产生新值:单写协议与多写协议。
3. 何时传播新值:及时传播与延迟传播。
4. 新值将传播到何处:侦听协议与目录协议。

在具体实现一致性协议时经常需要在系统复杂性与性能之间进行取舍。通常,系统性能的提高是以协议复杂性的增加为代价的。

6.2.1 写使无效与写更新

根据所采取的写传播策略,Cache一致性协议可分为写使无效(Write Invalidate)和写更新(Write Update)两种。在写使无效协议中,当根据一致性要求企图把一个处理机对某一单元所写的值传播给其他处理机时,就使其他处理机中该单元的备份无效。当其他处理机随后要用到该单元中的值时,再获得该单元的新值。在写更新协议中,当根据一致性要求企图把一个处理机对某一单元所写的值传播给其他处理机时,就把该单元的新值传播给所有拥有该单元备份的处理机,从而对相应的备份进行更新。

写使无效协议的优点是,一旦某处理机使某一变量在所有其他Cache中的备份无效后,它就取得了对此变量的独占权,随后它可以随意地更新此变量而不必通知其他处理机,直到其他处理机请求访问此变量而导致独占权被剥夺。写使无效协议的缺点是,当某变量在某一处理机中的备份变为无效后,此处理机再读此变量时会引起Cache不命中,在一个共享块被多个处理机频繁访问的情况下会引起所谓的"乒乓"效应,即处理机之间频繁地互相剥夺对一个共享块的访问权而导致系统的性能严重下降。写更新协议的优点是,一旦某Cache缓存了某一变量,它就一直持有此变量的最新备份,除非此变量被替换掉。写更新协议的缺点是,写数的处理机每次都要把所写的值传播给其他处理机,即使其他处理

机不再使用所写的共享块。因此,写使无效协议适用于顺序共享(Sequential Sharing)的程序,即在较长时间内只有一个处理机访问一个变量;而写更新协议适用于紧密共享(Tight Sharing)的程序,即多个处理机在一段时间内频繁地访问同一变量。在实际应用中,也有一些把写使无效协议和写更新协议结合起来的协议[61]被提出来。

6.2.2 侦听协议与目录协议

侦听协议的基本思想是:当一个处理机对共享变量的访问不在 Cache 命中或可能引起数据不一致时,它就把这一事件广播到所有处理机,系统中所有处理机的 Cache 都侦听广播。当拥有广播中涉及的共享变量的 Cache 侦听到广播后,就采取相应的维持一致性的行动(例如,使本 Cache 中变量的备份无效、向总线提供数据等)。侦听协议适用于多个处理机通过总线互相连接的集中式共享存储系统,这是因为总线是一种方便而快捷的广播媒介。

根据执行写操作时所采取的一致性策略,侦听协议可进一步分为写更新和写使无效两种。在写使无效协议中,当一个 Cache 侦听到其他处理机欲写某一单元且自己持有此单元的备份时,就使自身当中的备份无效以保持数据一致性。在写更新协议中,当一个 Cache 侦听到自己持有备份的某一共享单元的内容被某一处理机所更新时,就根据侦听到的内容更新此备份的值。写使无效协议的例子有 Write-Once 协议[43] Berkeley 协议[63] 以及 Illinois 协议[78] 等,而 Firefly 协议[94] 和 Dragon 协议[13] 等都是写更新协议的例子。

侦听协议虽然简单有效,但由于它需要广播因而只适用于可伸缩性差的共享总线结构中。在采用通用互联网络的分布式系统中,通常使用基于目录的 Cache 一致性协议。其主要思想是:为每个存储行维持一个目录项,该目录项记录所有当前持有此行备份的处理机号以及此行是否已被改写等信息。当一个处理机欲往某一存储行写数且可能引起数据值的不一致时,它就可以根据目录的内容只向持有此行备份的那些处理机发出使无效或更新信号,从而避免了向所有处理机广播。

根据目录的不同组织方式,目录协议可分为位向量目录、有限指针目录、链表目录等几种。

位向量目录中的每个目录项有一个 N 位的向量,其中 N 是系统中处理机的个数。位向量中第 i 位为"1"表示此存储行在第 i 个处理机中有备份。另外,每个目录项有一个改写位,当改写位为"1"时,表示某处理机独占此存储行并已将其改写。美国斯坦福大学的 DASH 系统就采用了位向量目录。

位向量目录所需的目录存储器的容量随处理机数 N、共享存储容量 M 的增加而以 $M \times N$ 的速度增加,当 N 很大时实现目录协议的开销很大。在大多

数情况下，一个变量每次只被少数的几个处理机所共享，因此可以用有限个指针指向当前持有此变量的那些处理机。每个指针需 $\log_2^N$ 位，整个目录所需的存储容量为 $O(M \times \log_2^N)$。由于每个存储行只有有限个指针，当共享某存储行的处理机个数多于目录项中的指针个数时，就导致指针溢出(Pointer Overflow)。处理指针溢出的方法有指针替换[9]、广播[9]、软件支持[25]等。美国麻省理工学院的 Alewife 多处理机就采用软件中断的方法来处理指针溢出的情况。

另一种减少目录存储容量的方法是，把目录信息分布到各个 Cache 中。具体做法是，把所有持有同一存储行的 Cache 行用链表链接起来，链表头在存储行处。当某一存储行被缓存到某个 Cache 中时，就把相应的 Cache 行链接到此存储行的链表头所指的链表中。当某一存储行在某个 Cache 中的备份变无效或被替换时，把相应的 Cache 行从此存储行的链表头所指的链表中删去。链表可以是单向链表，如 SDD(Singly-linked Distributed Directory protocol)协议[95]；也可以是双向链表，如 IEEE SCI 标准协议[42]。

在上述目录组织方案中，每个目录都有一个固定的宿主结点(home)，处理机访存失效时，可以直接根据失效地址查找相应目录项。上述目录组织方式适用于 NUMA 结构的系统。这种目录组织方式的缺点是，每次访存失效时都需进行远程访问目录。可能属主(Probowner)[75]。目录提供了一个解决方案。在 Probowner 目录中，每个共享块都有一个持有该共享块最新备份的属主结点(Owner)。对于每个共享块，每个处理机都保存一个可能属主指针。所谓“可能属主指针”，指的是这个指针“很可能”指向该共享块的属主结点。如果可能属主指针没有指向属主结点，则通过该指针所指的处理机中的可能属主指针往后找，最终总能找到属主结点。当处理机收到关于某个共享块的无效信号，或者释放该共享块的属主权，或者转发该共享块的取数请求时，就修改该共享块的可能属主指针。

在大多数情况下，可能属主目录可以减少远程目录查找的次数。因此，可能属主目录适用于远程访问延迟很长的系统或没有宿主结点的 COMA 结构的系统。软件共享存储系统(如 Ivy[75]和 Munin[23])都采用了可能属主目录。

6.2.3 单写协议与多写协议

传统的共享存储系统通常采用单写协议，即每次只允许一个处理机写某个共享单位。每当一个处理机要写某个存储块时，它必须取得独占的写权限。

在共享粒度较大的共享存储系统中，尤其是共享虚拟存储系统中，共享粒度很大(共享虚拟存储系统的共享粒度通常为一页)，假共享会严重地影响系统的性能。所谓假共享指的是，几个处理机虽然共享某个存储块，但并没有真正共享数据。在几个处理机同时访问同一共享块的不同部分时，就会发生假共享现象。

在传统的单写协议中，假共享会严重地降低系统性能。在写使无效协议中，当一个处理机修改一个共享块中的某个字时，就会使该共享块在其他处理机中的备份都无效，即使其他处理机根本就不会访问被修改的那个字。假共享严重时会引起所谓的“乒乓”效应，即虽然不同的处理机访问不同的共享字，但由于这些字恰好在同一个共享块内，导致处理机间频繁地争夺对该共享块的访问权，引起大量无谓的失效访问。在写更新协议中，当一个处理机修改一个共享块中的某个字时，就会把修改的内容传播给所有持有该共享块备份的处理机，也就使其他处理机根本不会访问被修改的字。

多写(Multiple Writer)协议[23]可以有效地解决假共享问题，它允许多个处理机同时读或写分布在同一个共享单位中的不同单元。一个处理机对某个单元的修改对其他处理机来说可以暂时不可见，直到进行同步操作时才把新值传播给其他处理机。程序员必须保证在一个同步区间内不会有对同一单元的冲突访问。图 6.1 表示多写协议的原理。在图 6.1 中，处理机 P_0、P_1 和 P_{N-1} 分别写共享块 B 中的 x、y 和 z 单元。

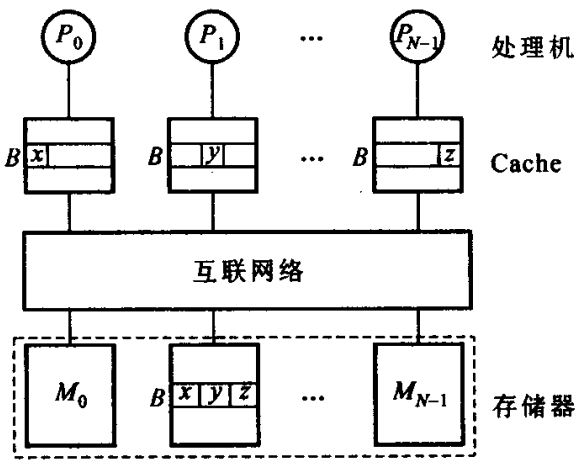


图 6.1 多写协议原理

多写协议会引起额外的存储开销和计算开销。为了把不同处理机对同一共享块中不同部分的修改内容“合并”在一起，需要识别各个处理机修改了该共享块中的哪些内容。通常的做法是，在处理机第一次写一个共享块之前，为该共享块做一个备份，称为该共享块的块备份(twin)。而当一致性协议要求对不同处理机的修改内容进行合并时，每个修改过该共享块的处理机把该块当前的内容与它的 twin 进行比较，得出本处理机的修改内容，称为块差(diff)。这样，就可以把不同处理机关于同一个共享块的 diff 合并在一起，得到该共享块的新内容。

在共享虚拟存储系统中，假共享的问题更加突出。一方面，在软件的共享存储系统中由于实现手段的限制，共享单位通常为一页，大大提高了假共享的概率。另一方面，软件共享存储系统的通信网络通常比硬件共享存储系统的通信

网络慢。因此,多写协议的提出,是近年来软件共享存储系统获得迅速发展的重要原因之一。目前,已经实现多写协议的系统有 TreadMarks[65]、Munin[23]、JIAJIA[54]和 Brazos[88]等。

6.2.4 及时传播与延迟传播

硬件共享存储系统一般采取及时更新的策略,即一个处理机更新某一共享块时,及时向其他处理机发出写无效信号或写更新信号。延迟更新策略主要是针对软件共享存储系统通信开销大的特点提出来的。

软件 DSM(Distributed Shared Memory,分布式共享存储)系统不同于硬件 DSM 系统的主要特征是通信开销大,所以减少通信次数是提高系统性能的重要途径。因此,在 Munin[23]的实现中提出了急切更新释放一致性(Eager Release Consistency,简称 ERC)的概念,即对共享单元的修改在释放锁操作时才传播到各个处理机,这样就可以把多个消息合并成一个消息来传播。假设处理机 P_1 和 P_2 共享单元 x、y 和 z,急切更新释放一致性模型对同一临界区内存数操作的合并过程如图 6.2 所示。

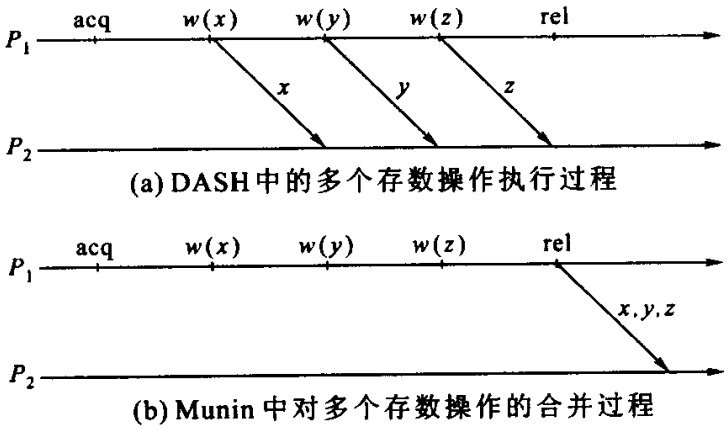
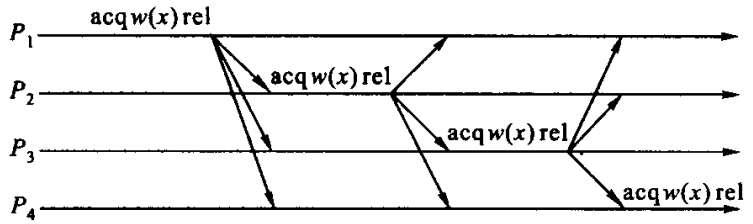


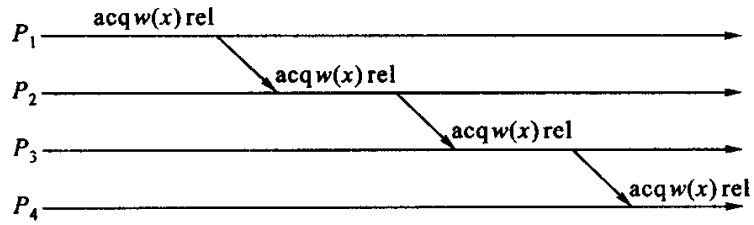
图 6.2 急切更新释放一致性模型对存数操作的合并过程

TreadMarks 的懒惰更新释放一致性模型(Lazy Release Consistency,简称 LRC)[64]则更进一步,它不是在释放锁时由写数的处理机把对共享单元的修改传播给其他处理机,而是把传播新值的操作延迟到获取锁时,而且只传送给需要此数据的处理机。假设处理机 P_1、P_2、P_3 和 P_4 共享单元 x、y 和 z,急切更新释放一致性模型和懒惰更新释放一致性模型的通信量比较如图 6.3 所示。

多写协议与延迟传播策略是软件共享存储系统的两大技术支柱。



(a) 急切更新释放一致性中的通信量



(b) 懒惰更新释放一致性中的通信量

图 6.3 急切更新和懒惰更新的释放一致性模型的通信量比较

6.3 基于锁的一致性协议

基于目录的协议必须在存储器中维护大量的目录信息，当被处理结点增加时，目录的存储容量也随之增加。此外，维护目录信息的复杂性也严重限制了系统的可扩展性。在此提出一个基于锁的 Cache 一致性协议，该协议通过附带在锁上的一致性信息来维护一致性，避免了由目录引起的存储开销大和系统复杂度高等问题。

6.3.1 设计考虑

Cache 一致性协议是把一个处理机新写的值传播给其他处理机的一种机制。在基于目录的协议中，写数的处理机负责把新写的值传播给所有持有被写存储块的处理机。因此，需要为每个共享块维护一个目录项，用以记录持有该共享块的处理机。

另一种传播新值的方法是，要求使用新值的处理机在访问被修改共享块时自己主动去取。这种方法可以避免由写数处理机主动传播新值而带来的盲目性，减少一些不必要的一致性维护，但这就需要一种机制通知使用一个共享块的处理机何时何地去取其他处理机修改过的该块的内容。弱一致性模型的同步机制可以被用来传递上述信息。例如，在释放一致性中，由于程序员负责把处理机间的冲突访问用 release-acquire 对隔开，执行 acquire 操作的处理机可以从执行 release 操作的处理机那里知道哪些存储块被修改过。懒惰更新释放一致性 LRC 就采用了这种想法。

但在 TreadMarks 以及其他采用 COMA 结构的共享虚拟存储系统中，还是需要一种机制来记录有哪些处理机修改了一个共享块，以便一个处理机在访问失效时找这些处理机去取修改过的内容。在 TreadMarks 中使用了一些复杂的数据结构[如在缺页时用于找到该页属主的可能属主(Probowner)目录、近似复制集(Approximate Copyset)等]来解决这个问题。因此，基于锁的一致性协议采用 NUMA 结构，即每个共享块都有一个固定的宿主(home)结点，任一处理机在访问失效时可以直接到失效共享块的 home 结点去取数。

基于锁的协议中新值的传播比 LRC 协议早，但比 ERC 协议晚。在 ERC 协议中，处理机在执行释放锁的操作时把新值传播给有关处理机；在 LRC 协议中，每个处理机都一直保留自己新写的值(存储开销很大)等待其他处理机在执行获取锁操作时来取。在基于锁的协议中，处理机在执行释放锁操作时把新值送回到它的 home 结点，其他处理机在访存失效时到相应的 home 结点去取。记录哪些共享块被写过的一致性信息则保存在锁处，通过释放锁和获取锁的操作来传播。

6.3.2 支持域一致性模型

Cache 一致性协议都是为实现某种存储一致性模型而设计的。存储一致性模型对 Cache 一致性协议提出一致性要求，而 Cache 一致性协议是实现这种一致性要求的一种机制。

新协议实现的是域存储一致性模型。域一致性模型是介于单项一致性和释放一致性之间的存储一致性模型。它比 LRC 协议更加“懒惰”。在 LRC 协议中，当处理机 P 从处理机 Q 获得锁 l 时，处理机 Q 所看到(visible)的修改操作都被传给处理机 P。但在 ScC 协议中，只有用锁 l 保护起来的区域中所做的修改才会传送给 P。ScC 协议对访存事件次序的规定如下：

1. 在处理机 P 执行获得锁 l 的 acquire 操作之前，所有相对于锁 l 已执行的访存操作必须相对于处理机 P 执行完。

2. 在处理机 P 执行访存操作之前，所有在此之前的 acquire 操作都已经完成。

一个访存操作相对于一个锁已执行完，当且仅当该访存操作在由该锁保护的临界区内发出且该锁已被释放。

为了保证所有对于锁 l 已执行的访存操作必须相对于获得该锁的处理机执行完，需满足条件：

(1) 执行 release 的处理机必须等待在此之前的所有访存操作执行完后才能释放锁。

(2) 释放锁时，释放锁的处理机把所有在相应的临界区内修改过共享块的

标号保存在锁中，获得锁的处理机根据相应锁中所保存的共享块标号使本地Cache中的备份无效。

上述条件(1)保证在处理机获得一个锁之前，所有相对于锁执行的读操作都已经执行完，即在此之后其他处理机的任何写操作对release之前的读操作都不会有任何影响。否则，在极端的情况下，在release之前的读操作可能会取回在下一个acquire之后的写操作所写的值。上述条件(2)可以保证在处理机获得一个锁之前，所有相对于锁执行的写操作都已经执行完，即所有在acquire之后的读操作都能取回在release之前的临界区内所写的内容。

采用ScC协议大大简化了基于锁的Cache一致性协议。在实现LRC协议的TreadMarks系统中，为了记录LRC的“happen-before-1”关系，需要使用诸如矢量时间戳(vector timestamp)和区间(interval)等复杂数据结构。ScC协议不需要建立完全的“happen-before-1”关系，一个处理机在acquire操作时只需看到由同一个锁保护的区间所修改的内容，而不是根据“happen-before-1”关系看到在此之前的所有区间中被修改的内容。

6.3.3 基本协议

在基本协议中，每个共享块都有固定的home，不在home中的共享块可以备份到Cache中。在Cache中的共享块可以处于3种状态，即无效(INV)、只读(RO)和可读写(RW)。由于采用了多写一致性协议，同一共享块可以同时在不同Cache中处于不同的状态。锁实际上是一种特殊的共享变量，因此每个锁也有固定的管理器。

执行release操作时，在release之前的所有访存操作都完成后，释放锁的处理机通过比较临界区中已修改的共享块和它们的twin来产生diff，并送到共享块所在的home结点。同时，还要给相应锁的管理器发送消息，通知该锁已被释放，用于记录临界区中被修改共享块信息的写标志(write-notice，一个write-notice实际上是一个被修改共享块的地址)也随之被发送给锁管理器。

执行acquire操作时，请求锁的处理机向相应的锁管理器发出请求后便一直处于等待状态，只有获得锁后才能继续进程的执行。锁管理器把所有获取锁的请求按发出的顺序排成队列。当锁被一个处理机释放时，锁管理器发消息给请求队列中的第一个结点以允许它获得锁，同时把与这个锁相关的写标志write-notice也附加在应答消息中。发出请求的处理机在得到响应后，根据应答消息中的write-notice把本地Cache中的相应共享块置为无效。对处于RW状态的共享块，先为它产生diff并送回home结点，然后才使共享块无效。

一个barrier操作可以看做先释放锁再获得锁。到达barrier意味着一个临界区的结束，离开barrier意味着新临界区的开始。因此，两个barrier操作之间

是一个完整的临界区。此外,在执行 barrier 操作时,全部与锁相关的 write-notice 都被清除。

当处理机发生读不命中时,直接到相应的 home 结点取来该共享块放入本地的 Cache 中,并置为 RO 状态。

当处理机发生写不命中时,若失效的共享块不在本地的 Cache 中,或处于 INV 状态,则将它从相应 home 结点取来并置为 RW 状态;若失效的共享块在 Cache 中处于 RO 状态,则直接把状态变为 RW。在开始写数之前,要产生此共享块的 twin,并记录 write-notice。

如果由于本地 Cache 容量不够进行块替换,被替换的共享块目前处于 RW 状态,则需要为被替换共享块计算 diff 并写回它的 home。

6.3.4 协议的优点和不足

图 6.4 表示基于锁的协议中共享块的状态转换图。其中,在 release 操作或 acquire 操作时页状态从 RW 转为 RO 是在软件 DSM 中写识别的需要,并非协议本身所要求(后续章节将介绍在软件 DSM 中避免这种状态转换的优化方法)。从图 6.4 可以看出,与基于目录的协议相比较,基于锁的协议中所有的一致性相关操作都是在同步点执行的,普通读写操作几乎没有与一致性有关的额外开销。表 6.1 比较了懒惰释放协议和基于锁的协议在不命中、获得锁、释放锁以及 barrier 时传递消息的个数。从表 6.1 可以看出,基于锁的协议在普通的访存操作和 acquire 操作时,消息的数目少于懒惰释放协议;但在 release 或 barrier 操作时,需要额外的消息把产生的 diff 写回 home 结点。此外,由于基于锁的协议采用了 NUMA 结构,处理机在对本地的 home 进行读写操作时,可以直接访问,避免了产生 twin 和 diff 等开销。

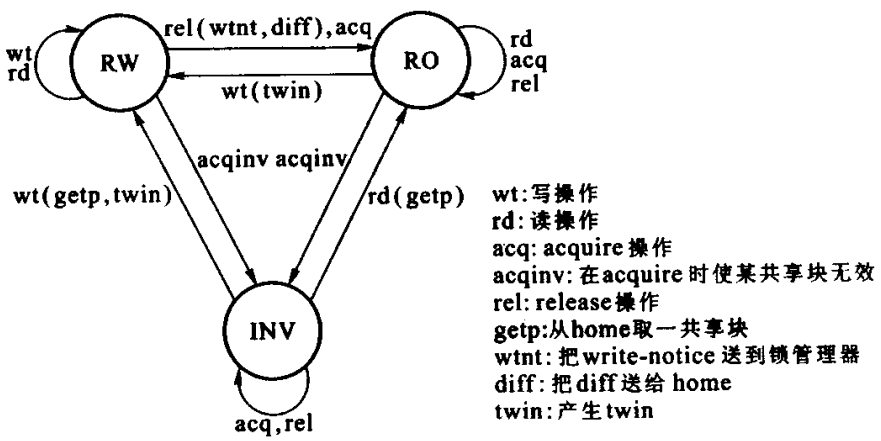


图 6.4 基于锁的一致性协议状态转换图

表 6.1 共享存储操作中的消息数目

协议	不命中	获取锁	释放锁	barrier
LRC	$2m$	3	0	$2(n-1)$
lock－based	2	2	$f+1$	$2(n-1)+F$

注：m：对不命中共享块进行并发修改的处理机数

f：发送 diff 所需消息数

n：系统中的处理机数

$F=\sum_{i=1}^{n}$(处理机 i 发送 diff 所需消息数)

基于锁的协议比 LRC 协议的存储开销也小得多。一方面，基于锁的协议在 release 时及时地把 diff 写回到 home，避免了 LRC 协议中由于 diff 的积累而引起的内存开销。另一方面，LRC 协议比基于锁的协议多出额外的目录开销。

6.3.5 协议的优化

基于锁的协议通过附带在锁上的 write-notice 来维护一致性。为了减少假共享，应尽量减少在执行 release 操作时送往锁的 write-notice 和在执行 acquire 操作时从锁获得的 write-notice。为此，可以采取以下措施：

1. 在执行 acquire 操作时，如果发现保存在锁中的某一 write-notice 在该处理机以往申请该锁时曾送给过该处理机，即其他处理机写的值已经传播到申请锁的处理机，则此次没必要再把该 write-notice 送往申请锁的处理机。

2. 在执行 acquire 操作时，如果发现关于某共享块的 write-notice 是由申请锁的处理机产生的，即在获得锁之前只有申请锁的处理机修改过该块，那么处理机在获得锁时没有必要置此块无效，即不把此 write-notice 送往申请锁的处理机。

3. 在执行 release 操作时，如果某共享块只在它的 home 结点上被修改过，且其他的处理机都没有该共享块的有效备份，则没必要告诉其他处理机该块已被修改过，即不用发送 write-notice 给相应的锁管理器。

此外，还有一些实现上述协议的软件 DSM(Distributed Shared Memory，分布式共享存储)系统的优化，将在后续章节介绍。

6.4 基于锁的 Cache 一致性协议的正确性

如果一个系统的访存事件次序满足式(5.8)或式(5.7)的要求，则该系统符合域一致性模型的要求。

在上述基本协议中，如果 u 和 v 是同一进程中的两个访存操作且 $u \xrightarrow{PO}$ acq

$\xrightarrow{PO} v$，则 $u^c < v^j$ 成立，其中 $j=1,2,\cdots,N$，c 是发出操作 v 的处理机号。

如果 $\text{acq}(l) \xrightarrow{PO} u \xrightarrow{PO} \text{rel}(l) \xrightarrow{E} \cdots \xrightarrow{E} \text{acq}(l) \xrightarrow{PO} v$，可分为如下两种情况：

1. u 是一个读操作，由于 $u \xrightarrow{PO} \text{rel}(l)$，$u$ 在 rel(l)之前执行完，显然也在取得该 rel(l)释放的锁的 acq(l)操作之前执行完。因此，在该 acq(l)之后发出的所有写操作所写的值对读操作 u 取回的值没有影响。根据定义 3.1 可得 $u^c < v^j$ 成立，其中 $j=1,2,\cdots,N$，c 是发出操作 v 的处理机号。

2. u 是一个写操作。根据基本协议，执行 rel(l)的处理机把与 u 有关的 write-notice 送往锁 l 且把 u 所写的内容送往其 home 结点，而在 rel(l)之后执行 acq(l)获取锁 l 的处理机会根据锁 l 中保存的 write-notice 来使本地相应共享块无效。因此在 rel(l)之后执行 acq(l)获取锁 l 的处理机在访问 u 所写的单元时肯定会发生访存失效，从该单元的 home 结点取回 u 所写的值。也就是说，在执行 acq(l)操作的处理机执行后续指令 v 之前，u 已经相对于该处理机执行完，即 $u^c < v^j$ 成立，其中 $j=1,2,\cdots,N$，c 是发出操作 v 的处理机号。

根据上述分析，基于锁的协议满足式(5.7)对访存事件次序的要求。因此，基于锁的一致性协议是域一致性模型的正确实现。

6.5 小 结

高速缓存技术在共享存储系统中占有重要地位。它不仅可以削弱由处理机和存储器的空间分布引起的大延迟，而且可以减少多个处理机在访问共享存储器时的冲突。然而 Cache 在多处理机中会引起 Cache 一致性问题，即如何使同一单元在不同 Cache 以及主存中的多个备份保持数据一致。高速缓存一致性协议的设计影响着分布式共享存储系统的性能和复杂度。

Cache 一致性协议都是为实现某种存储一致性模型而设计的。存储一致性模型对 Cache 一致性协议提出一致性要求，即 Cache 一致性协议应该实现怎样的“一致性”。Cache 一致性协议通过把某个处理机新写的值传播给其他处理机来具体实现这种一致性。

本章从新值的传播方式(写使无效与写更新)、新值的传播时机(及时传播与延迟传播)、新值的来源(单写协议与多写协议)以及新值传播对象(侦听协议与目录协议)等不同的侧面介绍了 Cache 一致性协议的关键技术。

本章提出的实现域一致性模型的基于锁的新型 Cache 一致性协议消除了传统的目录协议所带来的存储空间的浪费，通过在锁上附带一致性信息来维护一致性。它比目录协议更简单、有效，更有利于系统的扩展。

第7章 共享虚拟存储系统

7.1 引　　言

共享虚拟存储(Shared Virtual Memory,简称 SVM)系统(又称为软件 DSM 系统)在消息传递的硬件上通过软件的方法实现共享存储的编程界面。在过去的十多年中,特别是进入 20 世纪 90 年代以来,共享虚拟存储由于结合了共享存储多处理机系统容易编程和消息传递多计算机系统容易实现的特点而引起了广泛的研究。自从第一个软件 DSM 系统 Ivy[75]诞生以来,人们已经在基于消息传递的 MPP(Massive Parallel Processing,大规模并行计算)系统或工作站网络上实现了若干软件 DSM 系统,如 Midway[18]、Munin[23]、TreadMarks[65]、CVM[66]、JIAJIA[54]和 Brazos[88]等。

Ivy[75]是第一个正式的共享虚拟存储系统。它通过修改操作系统中的存储管理模块,在分布的存储器上建立统一的共享虚拟存储空间,并实现顺序一致的存储一致性模型。但由于优化技术尚未成熟,没有很好地解决软件 DSM 系统中大粒度(通常是存储页)共享导致的假共享(False Sharing)及碎片(Fragmentation)等问题,导致系统性能不佳。与 Ivy 同期的软件 DSM 还有 Linda[12]、Mirage 以及 Emerald 等。

随着 Munin[23]以及 TreadMarks[65]等系统的逐步实现,软件 DSM 的优化措施逐渐成熟,使得软件 DSM 更加实用。其中,具有代表性的两个里程碑式的优化措施是多写协议以及延迟传播技术的提出和应用。在此基础上出现了大量的软件 DSM 系统的优化技术,如动态预取、数据自动迁移、自适应协议技术以及适当的硬件支持等。在这一时期的代表性系统包括 Munin、TreadMarks、Mid-

way、CVM、JIAJIA 等。

此外，随着 SMP(Symmetric Multi Processor，对称多处理机)机群系统的流行，基于 SMP 机群的软件分布式共享存储系统逐渐成为研究的热点。基于 SMP 机群系统的软件 DSM 系统充分利用 SMP 系统的特点，在 SMP 系统内部利用硬件进行共享，而 SMP 系统之间利用软件实现共享。一方面可以利用 SMP 硬件共享的特点提高效率，另一方面给用户提供统一的共享存储编程界面。基于 SMP 机群的系统包括 Shasta[82]、Cashmere-2[92]、SoftFLASH[37]、HL-RL-SMP[80]、Sirocoo[84]、TreadMarks[58]以及 JIAJIA[55]等。

本章第 2 节从实现方式、一致性协议以及编程界面等角度介绍共享虚拟存储系统中存在的问题及关键技术。第 3 节介绍我国自主研制的共享虚拟存储系统 JIAJIA。JIAJIA 实现了上一章提出的基于锁的一致性协议，采用类似于 NUMA(Non-Uniform Memory Access，非一致访存)的存储器组织方式，能够把多个机器的存储器组织起来形成一个更大的存储空间。同时，JIAJIA 还实现了一系列优化策略，如懒惰写识别、写向量、宿主自动迁移、SMP 优化以及数据动态预取等来避免假共享、减少通信、容忍或隐藏通信延迟。第 4 节介绍采用一些广泛使用的测试程序(如 SPLASH2 和 NAS 并行程序集)对 JIAJIA 进行测试的结果，测试内容包括 JIAJIA 与其他软件 DSM 系统(如 CVM)的比较、JIAJIA 与消息传递系统 PVM 的性能比较以及 JIAJIA 的优化措施的效果。第 5 节对软件 DSM 系统和消息传递系统进行了比较分析。第 6 节是本章小结。

7.2 共享虚拟存储系统中的关键技术

7.2.1 实现方式

1. 实现层次

共享虚拟存储系统的实现方式很灵活。一般说来可以在语言、编译器、运行库或操作系统等层次上实现共享虚拟存储系统，有的系统甚至通过一些硬件支持来提高性能。

最早的软件 DSM 系统 Ivy[75]是在操作系统内实现的，它通过修改操作系统中的存储管理模块 MMU(Memory Management Unit)来建立共享虚拟存储器和本地存储器之间的联系并维护数据的一致性。因此，Ivy 的一致性粒度是一个虚存页。当处理机访问一个不在本地内存中的共享页时，系统将产生一个缺页中断。MMU 会根据中断的地址从其他处理机的内存或本地硬盘中取回缺失的页。这种实现方法的主要缺点是需要修改操作系统，而这对大多数人来说是不可能的。在操作系统中实现的其他共享虚拟存储系统还包括 Munin[23]、SoftFLASH[37]及 DSVM6K[21]

等。从商业的角度来看,修改操作系统的方案都是没有前途的。当然,如果能在未来的操作系统核心里加入共享虚拟存储层的功能,那将是非常有用的。正是因为这样,在 ACM(Association for Computing Machinery,计算机协会)每两年召开一次的操作系统设计与实现(Operating System Design and Implementation,简称 OSDI)的会议上,有许多文章讨论这方面的工作。

用户库的软件 DSM 系统的实现方式通过操作系统提供的界面来检测缺页并进行缺页服务。这种方式通常使用 mmap()和 mprotcte()等系统调用来建立共享虚拟存储的映射关系,并修改共享页的访问权限(可读、可写、不可读写)。处理机对共享虚拟存储进行越权访问时,操作系统会产生 SIGSEGV 中断信号,并调用 SIGSEGV 中断处理程序。因此,可以通过修改 SIGSEGV 中断处理程序来进行远程取数和一致性维护。这种方法不用修改操作系统,也不用过多的用户或编译器参与,是目前主流的软件 DSM 实现方法。一些著名系统,如 TreadMarks[65]、CVM[66]、Quarks[68]、Brazos[88]以及 JIAJIA[51]等,都采用了上述方法。

在语言级实现共享虚拟存储系统的典型代表是 20 世纪 80 年代的 Linda[12]和 Orca[16]系统。这种系统需要用户显式地调用访问共享虚拟存储的函数。例如,在 Linda 系统中,共享空间被组织成一个"元组池",用户通过元组的逻辑名来访问池中的共享元组。Linda 系统提供了元组操作的一些专用算子、如插入、删除、读取等。Linda 系统维护一致性的方法也很独特:当处理机想要修改某个共享元组时,必须先把该元组从共享池中删除,修改完后再重新插入。由于语言级实现的系统需要将所有已有的程序改写,难以被用户接受。所以,自 Linda 系统和 Orca 系统之后,近年来已很少有人在这方面进行研究。

也可以通过编译器来实现共享虚拟存储系统,如 Midway[18]系统通过修改可执行代码,在每个访存操作后插入一段代码进行远程访问及一致性维护等操作。用编译器实现共享虚拟存储系统的优点是可以支持细粒度和严格的一致性模型,但由于给每个普通的访存操作都增加了一些额外的开销,使得整个系统的性能有可能下降。另外,这种系统的可移植性也不好。通过编译器实现的系统还包括 Wisconsin 实现的 Blizzard-S 系统。通常,编译器可以作为一种辅助手段对软件 DSM 系统进行优化。例如,通过预编译进行从一种界面到另一种界面的转换,目前已经实现从 OpenMP 的源程序到 TreadMarks 或 JIAJIA 源程序的转换;也可以通过预编译分析并行程序的访存行为,实行预取等优化策略,减少访问失效等。

当然,也有一些系统结合了上述几种实现方法,如 Midway 系统就采用了编译器和用户库结合的方法。NCP2[19]系统通过硬件支持减少系统开销。而 Munin 系统修改了操作系统,提供了用户库调用,也有编译器的支持。

2. 共享粒度

共享虚拟存储系统的实现层次直接决定了系统的共享粒度。共享存储系统的共享粒度是指该系统进行远程访问以及一致性维护的单位。硬件 DSM 系统一般都实现细粒度共享,其共享粒度为 Cache 行。而在软件 DSM 系统中,由于实现方式的限制,共享粒度一般为虚存页。

粗粒度的优点是可以充分利用通信带宽,减少维护一致性所需的目录的容量,并预取后面要用到的数据等。但是,大共享粒度大大增加了软件共享存储系统中假共享情况发生的概率。尽管延迟传播技术与多写技术能有效地缓解这个问题,但假共享问题并没有彻底解决。多写协议以及延迟传播技术只能解决同步区间(Synchronization Interval)内的假共享问题,不能解决跨同步区间的假共享问题。例如,在同步区间内,两个处理机可以同时写同一页的不同部分,但到下一个 barrier 操作时两个处理机中的备份都要被清除掉(由于互相都知道对方已写了),过了 barrier 操作后两个处理机都需重新取页。粗粒度的另外一个问题是内存碎片的问题,即在软件 DSM 系统中,空间分配的最小单位是页,即使用户只需要一个字,系统也会分配一页。

为了进一步缓解粗粒度共享所带来的假共享问题,一些系统通过编译器或硬件支持实现了可变的共享粒度,通常是以页为远程访问的单位,但以较小的块作为一致性维护的单位。例如,Midway 系统通过编译支持,Simple-COMA[81] 系统和 Tempest[79] 系统通过硬件支持实现了细粒度的一致性维护。支持细粒度的系统虽然不会有假共享和碎片问题,但它是以增加通信次数为代价的,而且还需要编译器或硬件支持,使得系统的可移植性降低。

3. 通信机制

通信开销是软件 DSM 中的主要部分之一。研究表明,在典型的通信协议(如 UDP 协议)中,在通信的起始和结束阶段,操作系统和协议的延迟占了通信延迟的绝大部分,而真正由硬件传输引起的延迟相对较小,对于小消息尤其如此。因此,通过用户级消息传递减少操作系统和协议引起的通信开销成为减少软件 DSM 系统通信开销的重要手段。

减少通信开销的另一种方法是利用适当的硬件支持,在这方面目前已有不少工作[19,92,59]。他们的结果表明,用很小的额外硬件来加速通信是值得的,这与 Cox 等人早先的模拟结果相吻合[29]。在软件 DSM 中通过硬件支持减少通信开销的一个重要方式是用硬件支持远程访问,如 SHRIMP 的自动更新策略[59]、Cashemere-2L[92] 中采用的存储通道(memory channel)等。这样不仅可以减少通信时间,而且可以减少中断远程处理机的次数。一些商品化的网络,如 Dolphin 公司的 SCI 网络,也支持远程访问。最新公布的用户级通信标准 VIA(Virtual Interface Architecture,虚拟接口结构)[27] 将此定为标准。

此外,由通信引起的中断开销也不可忽视。在软件 DSM 的典型实现中,一个处理机向另一个处理机发消息时,通常要中断接收消息的处理机。一些研究结果表明,通过由接收消息的处理机主动查询来避免中断开销可以从总体上减少通信开销。中断和查询到底哪种方法更好目前尚无定论。

7.2.2 数据一致性

数据一致性问题是软件 DSM 的核心问题,软件 DSM 中数据一致性的维护直接决定系统的性能及可伸缩性。软件 DSM 中数据一致性的维护包括虚拟存储的组织方式、采用何种存储一致性模型以及 Cache 一致性协议等方面的内容。

1. 存储器组织方式

NUMA(Non-Uniform Memory Access,非一致访存结构)和 COMA(Cache Only Memory Architecture,唯高速缓存结构)是分布式共享存储系统通常采用的两种存储器组织方式。在 NUMA 结构中,共享存储器静态地分布在所有结点上。每个单元有惟一的地址以及由该地址确定的 home 结点。在 COMA 结构中,数据与地址分离,任一单元没有固定的地址和相应的 home,本地存储器类似于一个大容量的 Cache。通常,在 COMA 中每个单元都有一个 owner,而 owner 可以根据访存模式在不同的处理机之间动态地漂移。因此,与 NUMA 相比,COMA 可以减少访问不命中率。COMA 的缺点是在访问不命中时,需要复杂的操作来确定缺失单元的 owner 所在位置;而且在本地存储空间不够而进行替换时,必须确保被替换掉的单元在系统中还有备份。

相应地,软件 DSM 也可以分为两类:采用 NUMA 结构的系统,称为基于宿主(home-based)的系统,而采用 COMA 结构的系统称为无宿主(homeless)的系统。在实现多写技术以及懒惰传播技术的系统中,上述两种结构都通过 twin 和 diff 来检测处理机修改共享页的内容,但它们保存和传播 diff 的方式是不一样的。在基于 home 的系统中,每个共享页都有一个固定的 home 结点。每个处理机都要根据一致性协议的要求(如在一个同步区间结束时)把对 home 不在本地的页面的 diff 传播到 home 结点。在缺页时,缺页的处理机直接从缺失页的 home 结点取数。而在无宿主的协议中,写数的处理机一直保存着它所做的修改,在处理机缺页时,缺页处理机向当前所有持有缺失页可写备份的处理机发出取数请求。写数的处理机在收到取数请求时产生 diff,并把它送到缺页的处理机。

在软件 DSM 中,基于 home 的结构与无 home 的结构各有利弊。基于 home 的系统的优点是:缺页时可以直接向缺失页的 home 结点取数,只需一个消息来回;而在无 home 的系统中缺页时必须向所有当前持有该页可写备份的处理机取数。对 home 在本地的页可以直接访问,没有 twin 和 diff 的开销。由于 diff

及时地送回到 home 结点，没有本地 diff 的积累问题；而在无 home 的系统中每个处理机必须一直保存它所写的值直到下一个全局同步或本地内存放不下时的垃圾清理(garbage collect)。当本地物理内存放不下共享页时，可以把部分 Cache 中的共享页替换回它们的 home 结点，易于实现大内存；而在无 home 的系统中，必须有一种机制保证所替换的页不是该页在系统中的最后一个备份。基于 home 的系统协议简单。基于 home 的系统的缺点是：把 diff 写回到 home 需要额外的消息。在缺页时需要取回整个缺失页，而无 home 的系统只需取回缺失页的 diff。

一些典型的虚拟共享存储系统如 Munin、TreadMarks 和 CVM 等都采用无宿主的结构。而 JIAJIA、Cashmere 等系统则采用基于宿主的结构。

2. 存储器一致性模型

存储器一致性模型从本质上说，是在应用程序与系统之间关于数据一致性的一种约定。一方面，它指明了系统为程序员提供的编程规范；另一方面，它也在很大程度上制约了系统的性能。在软件 DSM 中，通信开销很大，由大共享粒度引起的假共享问题也很严重，实现对访存事件次序要求严格的一致性模型难以提高系统性能。最早的 Ivy 系统就是由于实现了顺序一致性而使得性能不够理想。因此，软件 DSM 系统一般都实现较弱的一致性模型，尤其以释放一致性模型较为普遍。有的系统甚至实现更弱的一致性模型。例如，Munin 和 TreadMarks 系统等实现释放一致性模型，Midway 系统实现单项一致性模型，JIAJIA 系统实现域一致性模型。

3. 高速缓存一致性协议

高速缓存一致性协议是存储一致性模型的具体实现，同时还涉及到存储器组织以及实现方式等内容，直接决定软件 DSM 系统的性能。

由于软件 DSM 中通信与共享的粒度较大、通信开销较大等原因，软件 DSM 的一致性协议与传统的硬件 DSM 的 Cache 一致性协议有很大不同。主要特点是，利用软件 DSM 允许协议比较复杂的特点，以部分 CPU 开销以及编程的复杂性为代价减少消息个数和消息量。

在上一章介绍了 Cache 一致性协议的关键技术，指出一致性协议的实质是将新写的值传播给其他处理机的一种机制。为了实现高效的传播，必须考虑以下几个方面的问题：

(1) 如何传播新值：写使无效与写更新。“写使无效”和“写更新”协议，顾名思义即是对某一共享单元执行写操作后，或者使其他处理机的备份无效，或者以新值替换来维护数据一致性。写使无效协议适用于顺序共享的程序，即在较长时间内只有一个处理机访问一个变量；而写更新协议适用于紧密共享的程序，即多个处理机在一段时间内频繁地访问同一变量。硬件 DSM 的传播方式一般比

较单一,采用写使无效者较多;而软件 DSM 的传播方式比较灵活,不少系统同时实现两种传播方式,并根据应用程序的访存行为自动切换。

(2) 谁产生新值:"多写协议"与"单写协议"。传统的共享存储系统通常采用单写协议,即每次只允许一个处理机写某一共享单位。在软件共享存储系统中,通信开销很大,而且共享粒度通常为一页,假共享会严重地影响系统的性能。多写协议[23]可以有效地解决假共享问题,它允许多个处理机同时读或写分布在同一个共享单位的数据,一个处理机对某一单元的修改对其他处理机来说可以暂时不可见,直到同步操作时才把新值传播给其他处理机,从而避免了不必要的等待。大多数虚拟共享存储系统(如 TreadMarks、Munin、CVM 和 JIAJIA 等)都支持多写协议。

(3) 什么时候传播新值:及时传播与延迟传播。软件 DSM 系统不同于硬件 DSM 系统的一个重要的特征是通信开销大,所以减少通信次数是提高系统性能的重要途径。因此,在 Munin 系统的急切更新释放一致性中,对共享单元的修改在释放锁操作时才传播到各个处理机,这样就可以把多个消息合并成一个消息来传播。TreadMarks 系统的懒惰更新释放一致性则更进一步,它不是在释放锁时由写数的处理机把对共享单元的修改传播给其他处理机,而是把传播新值的操作延迟到获取锁时,而且只传送给需要此数据的处理机。JIAJIA[51] 系统则介于 ERC 和 LRC 之间。在 JIAJIA 系统中,新值在释放锁时被写回它的 home 结点,并在下一次访问时从 home 结点传播到请求结点。

(4) 新值将会传播给谁:侦听协议与目录协议。在侦听协议中,写数的处理机把新写的值广播给所有处理机,其他处理机侦听广播,一旦发现被修改数据处于本地 Cache 中,将立即接收新值。在目录协议中,为每个共享单位维持一目录项用于记录与该共享单位有关的信息,如哪些处理机持有该共享单位的备份、该共享单位在主存及在被访问结点中的状态等。当共享页面被修改后,系统根据目录信息向相应的结点发出"写使无效"或"写更新 "信号。JIAJIA 系统实现的一致性协议最显著的特点是基于锁的机制,它不用目录信息,而只是通过附带在锁上的 write-notice 来维护一致性。

7.2.3 编程接口

软件 DSM 系统的编程界面取决于存储一致性模型与实现方式两个方面。

作为应用程序与系统之间关于数据一致性的一个约定,存储一致性模型从维护数据一致的角度规定了程序员为了维护共享数据的一致性所应遵循的编程规范。一般来说,越严格的存储一致性模型对程序员的要求越低。顺序一致性是程序员最欢迎的,它对程序员没有维护数据一致的额外要求。如果没有编译程序的帮助,单项一致性是程序员所难以接受的。释放一致性与域一致性是比

较好的兼顾程序设计与系统性能的存储一致性模型，因为它们所利用的同步机制是共享存储并行程序中本来就需要的。

软件 DSM 的编程界面取决于系统的实现方式。例如，在语言级实现的系统中，用户需要通过显式的函数调用来访问共享存储器；而在编译器自动并行的实现中，编译器自动插入有关语句，对用户的要求较少。在目前普遍采用的运行库实现方式中，系统给用户提供分配共享空间以及同步的函数调用。一旦共享空间被分配后，用户可以像访问普通变量那样访问共享变量。

从并行模式的角度看，共享存储并行系统的并行模式有单程序流多数据流(Single Program Multiple Data，简称 SPMD)以及分支合并(fork-join)两种。在 SPMD 模式中，多个处理机同时执行同一程序，但操作不同的数据；而在 fork-join 模式中，平时只有主进程执行程序，在需要并行时由主进程通过 fork 操作在其他处理机上分裂出并行进程，并行任务执行完后并行进程再通过 join 操作合并成一个进程。在硬件 DSM 系统(如 SMP)系统中，通常采用 fork-join 的并行模式。软件 DSM 系统一般采用 SPMD 的并行模式，因为软件实现 fork 操作和 join 操作比较困难且开销很大。

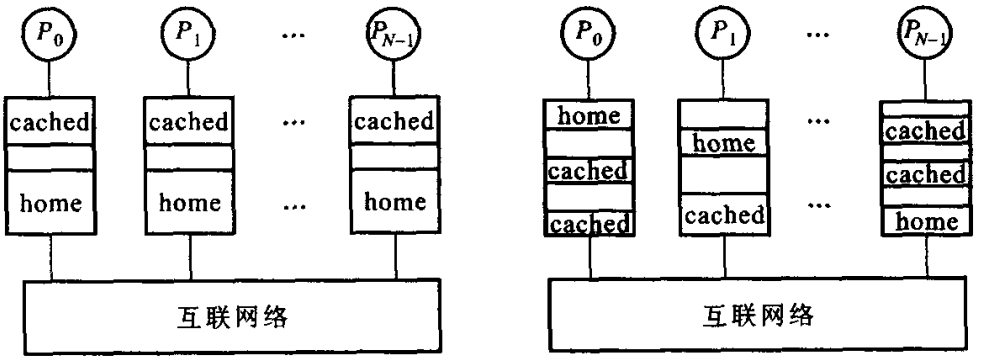
7.3 JIAJIA 共享虚拟存储系统

7.3.1 存储器组织

图 7.1(a)显示了 JIAJIA 系统的共享存储器组织方式。JIAJIA 系统采用类似于 NUMA 的结构。在 JIAJIA 系统中，每个页面都有一个相应的 home 结点。当处理机访问 home 分布在本地的那部分共享内存时，只进行本地访问操作；当处理机访问 home 分布在其他结点上的共享页面时，发生缺页中断，缺页中断服务程序把该页从相应的 home 结点取来放在缺页结点的 Cache 中。此时该页面的虚地址与它在 home 结点中的虚地址是一致的，即不论页面是在 home 结点还是在其他结点，它的地址始终是一致的。所以当发生远程访问时，不需要进行地址转换。图 7.1(b)显示了 JIAJIA 系统的地址映射方式。

JIAJIA 通过系统调用 mmap()分配共享页面，并通过系统调用 mprotect()来设置共享页的访问状态(可读写、只读、不可读写)。当处理机访问一个没有在本地映射的页或对一页进行非法访问时，操作系统会产生 SIGSEGV 中断信号，SIGSEGV 处理程序根据访问失效的种类进行处理。当发生缺页时，从其他结点取来的页面通过 mmap()映射到本地。因为共享页面所需空间有时会远远超过一台机器的内存空间，所以如果不加限制地把共享页面映射到本地结点上，会使内存空间越界，导致系统崩溃。为了避免这种现象的发生，每台处理机都建立一

个 Cache 数据结构用于记录那些非本地的共享页面。当需要映射新的一页到本地时，首先要判断 Cache 是否已满，若是，则需要把一些页面替换出去。



(a) JIAJIA 的存储组织 (b) JIAJIA 的地址映射

图 7.1 JIAJIA 系统的存储器组织与映射

上述存储组织方式使得 JIAJIA 系统的共享空间不受单机内存空间的限制。在其他采用 COMA 结构的软件 DSM 系统(如 Munin、TreadMarks 和 CVM 等)中，由于没有实现替换机制(如前所述，在 COMA 结构中，由于共享页面没有固定的 home，在替换时需要保证被替换页在系统中的最后一个备份不会被清除掉，因此难以实现替换)，每个结点都必须能存储所有共享页面，因此共享空间受单机内存空间的限制。此外，由于采用 COMA 结构，对任一共享页面，每个结点都需要为它保存大量的信息，如在缺页时用于找到该页 owner 的目录、页保护状态、该页的局部与全局地址、twin 以及不断积累的 diff 等。在 JIAJIA 系统中，由于任一页面都有 home，在缺页时可以直接根据页地址到该页的 home 结点取数，对该页修改产生的 diff 也可以及时写回到 home，节省了大量内存开销。在 JIAJIA 系统中，对每个共享页，只需 6 个字节来记录该页的 home 结点，以及在 home 或 Cache 中的位置。此外，对每个 Cache 页，只记录它的地址、保护状态，并在当页状态为可写时，为它保存 twin；对每个 home 页，只记录它的地址以及反映该页是否被本地处理机写过和被其他处理机读过的标志。

7.3.2 基于锁的一致性协议在 JIAJIA 中的实现

JIAJIA 系统实现上一章提出的基于锁的一致性协议。该协议支持域存储一致性模型和写无效的传播策略，并采用多写协议来避免假共享。

在基本协议中，每一页都有固定的 home，不在 home 中的页面可以映射到 Cache 中。在 Cache 中的页面可以处于 3 种状态，即无效(INV)、只读(RO)和可读写(RW)。由于采用了多写一致性协议，同一页面可以同时在不同 Cache 中处于不同的状态。锁实质上是一种特殊的共享变量，因此每个锁也有固定的管理器。

在分配共享空间时,对于每个共享页,根据用户指定的初始分布计算该页的 home 结点,并在该结点中通过 mmap()把该页映射到用户空间中并置为只读状态。其他处理机不用进行存储映射,只需记录该页的 home 结点号以及虚地址等信息。

当处理机发生访问失效时,操作系统产生 SIGSEGV 中断。SIGSEGV 中断处理程序根据发生中断的原因进行处理,有如下 4 种可能的情况:(1) 引起访问失效的页面在本地的 home 中,则肯定是写 RO 页失效,只需记录相应的 write-notice 并把该页置为 RW 状态。(2) 引起访问失效的页面不在本地的 home 中,且处理机发生读不命中,则说明失效页在本地没有有效备份,直接到相应的 home 结点取来该页面放入本地的 Cache 中,并置为 RO 状态。(3) 引起访问失效的页面不在本地的 home 中,处理机发生写不命中且引起访问失效的页面在 Cache 中没有有效备份,则将它从相应 home 结点取来放在 Cache 中置为 RW 状态,产生此页面的 twin 并记录 write-notice。(4) 引起访问失效的页面不在本地的 home 中,处理机发生写不命中且引起访问失效的页面在 Cache 中处于 RO 状态,则直接把状态变为 RW,产生此页面的 twin 并记录 write-notice。在情况(2)或(3) 中,如果 Cache 中已经没有空间存放取回的页面,则需要进行页面替换,如果被替换的页目前处于 RW 状态,则需要为被替换页计算 diff 并写回它的 home。

基于锁的协议通过在锁管理器中的 write-notice 来维护一致性。它要求释放锁的处理机把本临界区中修改页面的 write-notice 送到锁管理器,获取锁的处理机根据锁管理器中的 write-notice 得知哪些页面已被其他处理机修改。为了处理临界区嵌套的情况,JIAJIA 系统使用堆栈结构来记录 write-notice。acquire 操作压栈,release 操作弹栈,弹栈时把栈顶的 write-notice 合并到下一层。为了识别处理机修改了哪些共享页,在一个同步区间(两个相邻同步操作之间的操作组成一个同步区间)开始时,处理机把本机所有的有效页(包括 home 和 Cache 中的有效页)置成 RO 状态(图 6.4 中在 release 或 acquire 操作时页状态从 RW 转为 RO 就是为了软件 DSM 的写检测);在同步区间结束时,把该同步区间中所有被修改页面的 write-notice 置于栈顶。

执行 release 操作时,释放锁的处理机通过比较临界区中已修改的页面和它们的 twin 来产生 diff,合并后送到页面所在的 home 结点;同时还要给相应锁的管理器发送消息,通知该锁已被释放,栈顶用于记录临界区中修改页面信息的 write-notice 也随之发送给锁管理器。

执行 acquire 操作时,请求锁的处理机向相应的锁管理器发出请求后一直处于等待状态,只有获得锁后才能继续进程的执行。锁管理器把所有获取锁的请求按发出的顺序排成队列。当锁被一个处理机释放时,锁管理器发消息给请求

队列中的第一个结点以允许它获得锁，同时把与这个锁相关的 write-notice 也附加在应答消息中，发出请求的处理机在得到响应后，根据应答消息中的 write-notice 把本地 Cache 中的相应页置为无效。对处于 RW 状态的页面，先为它产生 diff 并送回 home 结点，然后才使之无效。

一个 barrier 操作可以看做先释放锁再获得锁。到达 barrier 意味着一个临界区的结束，离开 barrier 意味着新临界区的开始。因此，两个 barrier 操作之间是一个完整的临界区。此外，在执行 barrier 操作时，全部与锁相关的 write-notice 都被清除。

基于锁的协议具有如下优化策略：(1) 在执行 acquire 操作时，如果发现某一 write-notice 在该处理机以往申请该锁时已获得过，即其他处理机写的值已经传播到申请锁的处理机，则此次没必要再把该 write-notice 送往申请锁的处理机；(2) 在执行 acquire 操作时，如果发现关于某一页面的 write-notice 是由申请锁的处理机产生的，即在获得锁之前只有申请锁的处理机修改过某一页面，那么处理机在获得锁时没有必要置此页面无效，即不把此 write-notice 送往申请锁的处理机；(3) 在执行 release 操作时，如果某一页只在它的 home 结点上被修改过，且其他处理机都没有该页的有效备份，则没必要告诉其他处理机该页已被修改过，即不用发送 write-notice 给相应的锁管理器。

为了实现上述第一个优化策略，JIAJIA 系统在锁管理器为每个锁维护一个标识号(incarnation number)，每当该锁的所有权从一个处理机传递到另外一个处理机时，该标识号递增一次。锁管理器在锁中记录一个 write-notice 时，同时记录该锁当前的标识号。处理机获得一个锁时，同时记录该锁当前的标识号。当该处理机下一次再申请同一个锁时，把它上一次获取该锁时记录的标识号传给锁管理器。锁管理器只把标识号大于处理机传过来的标识号的 write-notice 送给申请锁的处理机，从而可以避免把申请锁的处理机中一些有效页无谓地置为无效状态。

为了实现上述第二个优化策略，锁管理器在记录 write-notice 时，同时记录该 write-notice 是由哪个处理机产生的。当处理机获取锁或离开 barrier 操作时，锁管理器只把非申请锁的处理机产生的或由多个处理机同时产生的 write-notice 送给申请锁的处理机。

为了实现上述第三个优化策略，处理机为自己保存的每个 home 页设置一个读标识(read-notice)，记录该页是否在其他处理机中有备份。当某页的 home 处理机收到对该页的取数请求时对该页的 read-notice 置位，在 home 处理机到达 barrier 且自己向 barrier 管理器发送 write-notice 时(说明在 barrier 后所有处理机中的备份都将被置为无效)，对该页的 read-notice 复位。因此，在执行 barrier 或 release 操作时，home 处理机可以根据一页的 read-notice 来判断该页是否在

其他处理机有备份。如果某一页只在它的 home 结点上被修改过,且其他处理机都没有该页的有效备份,则没必要告诉其他处理机该页已被修改过,即不用发送 write-notice 给相应的锁管理器。

7.3.3 JIAJIA 系统的优化

除了上述与协议有关的优化外,JIAJIA 系统还结合实现方法做了如下优化。

1. 懒惰写检测。在软件 DSM 中,为了检测处理机对共享页的修改,通常在一个同步区间开始时把每个处理机持有的有效共享页置为只读状态。当处理机对只读页进行写操作时,发生访问失效并记录 write-notice。这种写检测方法有一定的开销,尤其是对于需要大量共享空间的矩阵类的应用来讲开销很大。懒惰写检测技术通过延迟对 home 页进行写保护,减少没必要的写保护和访问失效开销。其做法是,如果可以确认一个 home 页在其他处理机中没有备份,则在同步区间开始时不对该页进行写保护,直到其他处理机来取数时再对该页进行写保护(如果一个 home 页在其他处理机中没有备份,则没必要知道 home 处理机是否修改过该页)。这样,对于一些只被 home 处理机访问的共享页,就省去了写检测的开销。

2. 宿主自动迁移。基于 home 的软件 DSM 系统的性能对共享页的 home 的分布十分敏感。如果 home 分布得当,大多数访问直接在 home 命中,几乎没有额外开销。如果 home 分布不当,大量的访问需要进行远程取页和写 diff,则开销很大。home 自动迁移技术根据应用程序的访问模式,自动进行 home 的迁移以减少访问开销。JIAJIA 系统的 home 迁移技术的特点是,几乎没有额外的迁移开销。具体做法是,如果在两个 barrier 操作之间,某一共享页只被某个非 home 的处理机修改过,则在执行第二个 barrier 操作时把该共享页的 home 迁移到该处理机。由于迁移的目标处理机刚刚修改过被迁移的共享页,因此持有该页的最新备份,不用进行数据迁移。此外,每个处理机可以根据 barrier 应答消息中附带的 write-notice 知道谁是迁移页的新的 home 处理机,进行 home 迁移时不用额外的消息来通知所有处理机。关于 JIAJIA 系统宿主自动迁移优化的细节详见参考文献[52]。

3. 写向量技术。前面提到,与无 home 的软件 DSM 相比,基于 home 的软件 DSM 的一个缺点是在缺页时需要取回整个缺失页;而无 home 的系统只需取回缺失页的 diff。事实上,这并不是基于 home 的软件 DSM 固有的缺点,JIAJIA 系统的写向量技术可以有效地减少缺页时的消息量。具体做法是,把每个共享页分为若干块,并为每个 home 页维护一个写向量表从而为每个处理机记录该页的哪些块在该处理机上次取页后已经被修改过。这样,在该处理机下一次取

该页时，只需取被修改过的部分。关于JIAJIA系统写向量优化的细节详见参考文献[53]。

4. SMP优化。SMP系统提供了高效的硬件共享。在SMP系统中，多个处理机通过总线共享存储器并通过侦听协议维护处理机之间的一致性。JIAJIA系统针对SMP机群系统的特点进行了优化，即SMP系统内部利用硬件共享，而SMP系统之间利用软件实现共享。一方面可以利用SMP硬件共享的特点提高效率，另一方面给用户提供统一的共享存储编程界面。在JIAJIA系统的SMP优化中，同一结点内的多个处理机通过硬件共享home页而各自保持独立的Cache页，相当于同一结点内的多个处理机把它们各自的home页合并在一起，提高了共享访问在本地命中的概率。结点内的处理机保持各自的Cache，一方面是为了保持协议及实现的简单性，另一方面是减少处理机间的冲突。例如，如果使用硬件共享的Cache页，SMP结点内的一个处理机对某Cache页的修改可能被同一结点内的其他处理机远程访问取回的页覆盖。关于JIAJIA系统SMP优化的细节详见参考文献[55]。

5. 数据动态预取。远程访问开销是软件DSM系统的主要开销。在处理机进行远程取数的过程中，取数的处理机必须等待。JIAJIA系统通过数据自动预取来容忍远程访问延迟。在基于锁的协议中，对一个Cache页的基本操作有两种，一是由于其他处理机的修改引起该Cache页收到无效信号(INV)被置为无效，二是由于本地处理机访问失效引起远程取页(GETP)。JIAJIA系统的预取策略记录每一Cache页的INV和GETP事件并分析其中的规律性。当收到barrier操作或acquire操作的应答信号中关于某Cache页的write-notice(即INV信号)时，如果根据以往事件的规律分析发现GETP将是该Cache页的下一个操作，则发出相应的预取请求。发出预取请求后，处理机不必等待该请求的应答信号。如果有向同一处理机的多个预取请求，则将它们合并成一个消息发出。这样。一方面可以通过预取和其他操作的并行执行来容忍远程访问的延迟，另一方面可以通过把多个取数请求合并来减少消息个数。

7.3.4 编程界面

JIAJIA系统的编程界面与其他软件DSM系统(如TreadMarks)类似。它提供以下基本调用：

1. jia_init(argc,argv)和jia_exit()：初始化和结束系统，这两个函数应该分别在应用程序开始与结束时被调用。

2. jia_alloc(size)：分配共享空间，参数size表示分配的字节数。

3. jia_lock(lockid)和jia_unlock(lockid)：获得和释放编号为lockid的锁。

4. jia_barrier()：barrier操作，所有结点在执行barrier操作时同步。

此外,JIAJIA 系统还提供了一些辅助调用,如用于条件变量同步的 jia _ setcv()、jia _ resetcv()和 jia _ waitcv()调用,用于打印错误信息并结束所有进程的 jia _ error()调用,用于提供运行时间的 jia _ clock()调用,以及用于设置系统优化参数的 jia _ config()调用,等等。

JIAJIA 系统提供了两个变量 jiapid 和 jiahosts,分别表示结点号和结点数。

在运行时,JIAJIA 系统到当前目录下寻找.jiahosts 配置文件。此文件列出了所有用于运行应用程序的结点,每行由结点名称、用户帐号、口令等来描述一个结点,第一行描述的是首先启动进程的结点。

JIAJIA 系统的一个重要特征是允许程序员控制共享空间的初始分布。JIAJIA 系统分配共享空间的基本函数是 jia _ alloc3(size, blocksize, starthost)。该函数从结点号为 starthost 的处理机开始,依次分配 blocksize 字节共享空间,直到分配了 size 字节。如果分配到最后一个处理机还未分配完,则从 0 号结点开始继续分配。

JIAJIA 系统编程界面的另一特点是它还提供了一些类 MPI 的消息传递函数,包括传送函数 jia _ send()、接收函数 jia _ recv()、归约函数 jia _ reduce()和广播函数 jia _ bcast()等。

7.4 性能测试与分析

7.4.1 测试程序

可以用一些被广泛采用的并行程序来评测 JIAJIA 系统。它们包括:SPLASH(或 SPLASH2)中的水分子模拟程序 Water,天体物理多体问题程序 Barnes,海洋模拟程序 Ocean 和 LU 分解程序 LU[87,99];NAS 测试程序集中的蒙特卡罗模拟程序 EP,整数排序程序 IS,多点问题程序 MG 和三维快速傅立叶变换程序 3DFFT[15];TreadMarks 测试程序中的旅行商问题程序 TSP,逐次超松弛法程序 SOR,快速排序程序 QSORT 和遗传链分析程序 ILINK[76];两个实用程序电磁场模拟 EM3D 和大气模型 IAP18,等等。

Water 是一个水分子动力学模拟程序,它逐步地模拟 n 个分子的运动状态。Water 的主要共享数据结构是一个一维数组,其中的每个数组元素记录了一个分子的一些特性参数,包括分子的质心、受力、位移和 6 个方向的导数等。Water 的并行算法把分子数组均匀地分配到各个处理机上。在每个时间步内,每个处理机都需要计算出本机上的每个分子与其他 $n/2$ 个分子之间的作用力。为了减少通信量,每个处理机都维护一个本地备份用以暂时存放作用力。直到所有处理机都完成作用力的计算时才通过用锁保护的临界区对结果进行更新。不同时

间步之间通过执行 barrier 操作同步。

Barnes 程序模拟了天体在万有引力作用下的变迁。它采用基于树结构的 Barnes－Hut 算法。它的主要数据结构是一个 Barnes－Hut 树，在程序中用两个数组来表示。其中，一个数组表示“树叶”，即天体(body)；另一个表示树中间的“节点”(cell)。在并行版本中，只有前者是分配在共享空间的。算法首先初始化天体的位置和速度，然后通过多次迭代来模拟天体的变化。每次迭代包括 4 个步骤，即构造 Barnes－Hut 树、重新分配天体、计算作用力以及更新天体的位置和速度。其中，计算作用力过程花费的时间最多。Barnes 主要采用 barrier 操作实行同步。

Ocean 模拟大范围的海洋运动中涡流等其他因素的作用。它使用一个动态分布的四维共享数组来保存格点数据。barrier 操作是主要的同步方式。

LU 用块分解算法将一个稠密矩阵分解为一个上三角阵和一个下三角阵。在此采用连续块分配的 LU 分解。连续块分配把那些在原来的数组中并不是连续的矩阵块分配在连续的空间内，并分布在对它们进行操作的处理机内存中。块 LU 分解算法是逐步进行的，每步分解一列块。每步首先将对角块进行 LU 分解，然后将对角块下面的块除以分解后的对角块，最后更新矩阵右边余下的块(trailing blocks)。并行主要是在第三阶段进行的。每一步的 3 个阶段之间用 barrier 操作隔开。

EP 的主要目的是产生一组高斯分布的数对。该程序特别适合于并行，程序中惟一的通信是在程序的最后进行一次累积操作。

IS 是一个用“桶排序”算法进行整数排序的程序。它把 key 分配在所有处理机上，每个处理机都有私有的“桶”，且所有处理机共享一个公用的“桶”。首先，每个处理机计算私有“桶”中 key 的个数。然后，在用锁保护的临界区中把这些值累加到公用“桶”中。最后，形成一个顺序的数组。

MG 是一个计算三维势场的多点问题程序，采用 barrier 操作进行同步。

3DFFT 是一个三维 FFT 的计算程序，主要的输入、输出数据都分配在共享空间，采用 barrier 操作进行同步。

SOR 采用红黑格的逐次超松弛法解偏微分方程。SOR 的并行程序将红黑两个数组分成大小基本相同的长方块，由每个处理机负责计算一块数据。由于采用 5 点差分格式，只有涉及到带状区域的边缘行的计算时才会发生通信。每完成一次迭代后，各处理机都通过 barrier 操作进行同步，以便使每个处理机都能看到其他处理机修改过的最新数据。

TSP 用分支限界算法解决旅行商问题。它的主要数据结构包括一个用于存放路径的存储池，一个包含指向路径的指针的优先队列，一个用来存放指向存储池中未用路径的指针堆栈，以及目前最短的路径。在寻找最短路径的过程中，

多个处理机通过用锁保护的临界区不断地访问这些共享数据。

QSORT 用并行的快速排序算法将 2 的幂次个互不相同的整数排成由小到大的序列。算法的基本操作是将一个无序的序列分成左右两个子序列，使左侧序列中的所有元素都小于右侧序列中的任一元素。算法用一个子序列堆栈记录尚未完成的工作，栈中的每一项都记录了一个子序列的最左和最右元素的数组下标。当子序列的长度小于一定阈值时，改用冒泡排序算法。在 JIAJIA 系统的程序中，工作栈是共享的，且通过锁予以保护。算法一开始，先由 0 号处理机将序列分成若干个子序列(数目约是处理机数的 4 至 5 倍)，形成任务栈，再由所有处理机一起从任务栈中取出任务，进行子序列的排序。

ILINK 的主要任务是在染色体上定位某个特定的疾病基因，其主要数据结构是由 genarray 组成的缓冲池，而 genarray[j]则包含了个体拥有基因型 j 及其相关的表现型的概率。在并行算法中，对每个个体的 genarray 的更新是并行进行的。在 JIAJIA 系统的并行程序中，整个缓冲池由所有处理机共享，通过栅障实现必要的同步。

EM3D 是中国科学院电子所高功率微波与电磁辐射开放实验室的一个生产性程序，它主要采用时域有限差分算法(Finite Difference Time Domain，简称 FDTD)来计算波导加载谐振腔里的谐振频率。它将计算空间分成 N 个区域。一个处理器对应一个子空间，这个处理器负责自己子空间内场量的运算和存储。按照 FDTD 算法的特点，因为相邻子空间的公共面上场的迭代涉及相邻的两个子空间，所以相邻的处理器需要通信，以交换场量信息、保证整个场量的迭代同步进行。在 JIAJIA 并行程序中，3 个电场分量、3 个磁场分量以及 8 个系数分量都定义为共享变量。在每一迭代结束后用 barrier 操作实现同步。

IAP18 是从 SGI Origin 2000 的 doacross 制导的并行程序中移植到 JIAJIA 系统上的。该程序计算 18 层的气候模型。通过 barrier 操作实现进程间同步。

7.4.2 JIAJIA 与 CVM 系统的比较

比较测试在曙光 1000A 机群系统上进行，该机群包括由 100 Mb/s 交换以太网连接的 8 个结点，每个结点有 1 个 PowerPC 604 处理机以及 256 MB 内存。测试程序包括 Water、Barnes、LU、EP、IS、SOR 以及 TSP。为了和其他软件 DSM 系统进行比较，所有这些应用程序也在美国马里兰大学研制的 CVM 上进行了相应的测试。

表 7.1 给出了每个应用程序的特性、串行程序运行时间以及并行程序在 JIAJIA 和 CVM 上的八机运行时间和加速比。由于内存限制，8 192×8 192 的 LU 和 SOR 的串行时间是根据问题规模的估计时间，由于不能支持大内存，上述规模的问题在 CVM 上无法运行。

表 7.1 JIAJIA 与 CVM 的比较

问题	规模	共享内存	Barrier 数	Lock 数	串行时间	八机时间		八机加速比	
						JIA	CVM	JIA	CVM
Water	1 728 分子	484 KB	35	520	178.00	26.47	39.79	6.72	4.47
Barnes	16 384 体	1.6 MB	28	64	413.24	64.75	66.02	6.38	6.26
LU	2 048×2 048	32 MB	128	0	84.86	25.04	35.21	3.39	2.41
LU	8 192×8 192	512 MB	512	0	5 464.80*	815.26	—	6.62	—
EP	2^{24}	4 KB	1	8	49.69	6.25	6.25	7.95	7.95
IS	2^{24}	4 KB	30	80	30.10	4.84	4.59	6.22	6.56
SOR	2 048×2 048	16 MB	200	0	68.44	11.45	15.25	5.98	4.49
SOR	8 192×8 192	256 MB	200	0	1 235.76*	166.20	—	7.44	—
TSP	20 个城市	788 KB	0	1 121	175.36	33.25	76.20	5.27	2.30

*:由于内存限制,8 192×8 192 的 LU 和 SOR 在单机上无法运行,表中时间为根据问题规模的估计时间,8 192×8 192LU 的串行时间等于 4 096×4 096LU 串行时间(683.10 s)的 8 倍;8 192×8 192 SOR 的串行时间等于 4 096×4 096 SOR 串行时间(31.18 s)的 4 倍。

表 7.2 给出了八机运行的部分统计信息,包括消息个数、消息量、缺页数(即 SIGSEGV 中断的次数)以及远程访问数等。

表 7.2 JIAJIA 与 CVM 的八机运行统计信息

应用程序	消息个数		消息量(KB)		缺页中断数		远程访问数	
	JIA	CVM	JIA	CVM	JIA	CVM	JIA	CVM
Water	10 828	160 134	16 850	72 408	4 892	30 899	2 847	19 416
Barnes	37 018	114 284	80 569	64 960	34 144	37 370	17 844	18 193
LU2048	25 992	49 283	99 950	203 604	32 663	23 998	12 072	11 874
LU8192	386 664	—	1 569 946	—	1 108 875	—	189 720	—
EP	77	105	60	65	22	23	14	14
IS	1 050	984	896	2 710	230	240	140	150
SOR2048	8 412	11 288	11 763	12 662	2 800	9 650	2 800	2 786
SOR8192	8 413	—	46 135	—	2 800	—	2 800	—
TSP	19 773	15 773	24 265	4 948	8 312	8 773	5 580	6 069

从表 7.1 可以看出,对于大多数程序,JIAJIA 系统获得了较好的性能和加速比。除了 IS 程序,JIAJIA 系统的性能均优于 CVM。

在 EP 程序中,JIAJIA 系统和 CVM 都获得了线性加速比,这是因为 EP 基本上没有什么同步,只在程序结束时有少量的通信。

在程序 LU 和 SOR 中,加速比随着问题规模的增大而增大,这是因为 LU 和 SOR 中的计算通信比都随问题规模的增加而线性增加。LU 和 SOR 的每个迭代步之间都用 barrier 操作进行同步,且在 LU 中只有修改右下角矩阵部分是并行完成的,因此 2 048×2 048 规模的 LU 和 SOR 的加速比都不是很高。在 LU 和 SOR 中,JIAJIA 系统性能优于 CVM 的主要原因是基于 home 的结构使得 JIAJIA 不用为在 home 命中的写操作计算 diff 而且 JIAJIA 系统在访存不命中时的一致性开销较小。从表 7.2 中可以看出,在程序 LU 和 SOR 中,JIAJIA 系统和 CVM 的远程访问次数相当,但 CVM 的消息个数和消息量都比 JIAJIA 系统多。

Water 和 Barnes 都是解决多体问题的应用程序,表现出紧密共享的访存行为。与 CVM 的懒惰更新释放协议相比,JIAJIA 系统的锁协议普通访存失效的开销较小,但同步操作的开销较大。程序 Water 和 Barnes 所需的共享页不多,但这些共享页被所有处理机频繁地访问。因此,在 Water 和 Barnes 程序中,JIAJIA 系统同步操作开销大的缺陷表现得不明显(JIAJIA 中同步操作的开销与每个处理机当前拥有的共享页数有关),而 JIAJIA 对普通访存失效开销小的优势却十分突出。从表 7.2 中可以看出,在 Water 程序中,JIAJIA 系统的远程访问数比 CVM 少,因此消息数也比 CVM 小;而在 Barnes 程序中,JIAJIA 系统和 CVM 的远程访问数差不多,但 JIAJIA 系统的消息数要小得多。表 7.2 还反映出,对于相同的消息个数,CVM 的消息量通常比 JIAJIA 系统少。在 Water 程序中,CVM 的消息个数是 JIAJIA 系统的十几倍,但消息量只有 JIAJIA 系统的 4 倍左右;在 Barnes 程序中,CVM 的消息数是 JIAJIA 系统的 3 倍,但消息量反而比 JIAJIA 系统略少。这是因为在由缺页引起的远程访问中,JIAJIA 系统要取一整页,而 CVM 只取已被修改的 diff 而已。

IS 的主要计算是每个处理机根据自己拥有的 key 值把它归类到不同的"桶"中,而把不同处理机中每个"桶"的值累加在一起时才需要通信。因此在 IS 程序中,key 的数目决定了计算量,"桶"的数目决定了通信量。在此固定每个处理机 key 的数目为 2^{24},"桶"的数目为 1 024,通信量相对较少,因此 JIAJIA 系统和 CVM 的八机加速比都大于 6。在程序 IS 中,CVM 的性能略优于 JIAJIA 系统。这主要是因为程序 IS 的主要同步和通信是在程序的最后各个处理机通过临界区把局部计算的值累加到共享区中,JIAJIA 系统在执行 release 操作时必须把产生的 diff 送回其 home,每个处理机都要花费大量的时间顺序地进入和退出临界区,而 CVM 只需在本地保留 diff,传递消息的数量少于 JIAJIA 系统。

在程序 TSP 中,所有的同步都通过临界区进行。每个处理机频繁地通过临

界区访问路径和优先级队列，表现出紧密共享的访存行为。从表 7.1 和表 7.2 中可以看出，在程序 TSP 中，JIAJIA 系统的消息数和消息量都多于 CVM，但性能却明显优于 CVM。进一步的分析表明，CVM 性能差的主要原因是单机运行的时间长，需要 478.07 s。如果与自己的单机时间相比，CVM 的加速比优于 JIAJIA 系统，达到 6.29。从表 7.2 还可以看出，在程序 TSP 中，JIAJIA 系统和 CVM 的远程访问数和消息数差别不大，但 CVM 的消息量却比 JIAJIA 系统少得多。同样，这是由于在远程访问时，JIAJIA 系统要取一整页，而 CVM 只取已被修改的 diff。

关于 JIAJIA 系统与 CVM 比较的详细分析见参考文献[4]。

7.4.3 JIAJIA 与 PVM 系统的比较

比较测试在上述 8 个结点的曙光 1000A 机群系统上进行。测试程序包括 SOR、Water、TSP、IS、Barnes、ILINK 和 QSORT。这些测试程序的 PVM 版本都是由美国莱斯大学的 TreadMarks 小组提供的。

1. 执行时间的比较

由于时间有限，大部分应用程序都只计算了一种问题规模。对程序 SOR，计算了 4 096×4 096 和 8 192×8 192 两种规模，后者已经超过单机内存的容量，因此没有单机的结果。表 7.3 给出了这些应用程序在两种环境下的性能，从中可以看出，JIAJIA 系统的性能与 PVM 完全相当。

表 7.3 JIAJIA 与 PVM 环境下的计算时间(单位:s)

处理机数	1		2		4		8	
应用程序	PVM	JIAJIA	PVM	JIAJIA	PVM	JIAJIA	PVM	JIAJIA
SOR 4096	25.74	26.48	13.13	13.53	7.25	7.21	4.17	3.96
SOR 8192	–	–	64.72	65.42	32.76	33.07	16.50	16.71
Water	197.7	178.83	91.52	92.14	49.80	48.35	25.46	26.32
TSP	258.3	258.52	135.6	138.21	71.21	74.56	36.70	38.27
IS	31.05	30.97	16.84	15.68	8.32	8.30	4.94	4.86
Barnes	145.4	130.46	72.85	68.71	38.04	37.23	20.24	22.52
ILINK	403.2	404.93	221.5	245.95	220.4	175.27	144.9	167.00
QSORT	179.1	138.31	102.6	78.99	103.6	57.41	102.4	729.18

从表中可以看出，除程序 QSORT 外，其余程序的 PVM 版本与 JIAJIA 版本的性能符合得很好，几乎相差无几。且随着处理机数的增加，大部分程序体现了

较好的性能改善。由于人们对 ILINK 了解很少,只是做了简单的移植,所以目前尚无法解释 8 个结点并行计算时性能提高不多的原因。至于 QSORT 在 PVM 环境下的性能远远不如在 JIAJIA 环境下的性能,主要的原因在于二者在具体程序设计上有所不同。JIAJIA 程序中并行任务先由 0 号处理机静态产生(这部分时间当然也计入了性能统计),而在 PVM 程序中,并行任务是各个从进程动态产生并交由主进程管理的。

2. 消息量的比较

分布式共享存储系统为了支持远程内存访问和维护整个共享存储空间的一致性,需要传送大量消息。例如,在 JIAJIA 系统中,就有将远程页面取入本地 Cache 的 GETP 消息、将临界区内修改过的内容(diff)送往主结点的消息、将临界区内哪些页面被修改过的信息(write-notice)送往锁存储区的消息等。相比之下,精心设计的消息传递程序却可以将消息的数量降到最小程度。从这个意义上说,消息传递程序发送消息的数量是 DSM 系统发送消息数量的下界,也是 DSM 系统应当努力追求的一个目标。考虑到通信量包括两个方面,即发送消息的个数和实际传送的字节数[10],为此在 8 机并行计算的条件下,分别测量了前述 7 个应用程序在 PVM 和 JIAJIA 环境下发送消息的个数和实际传送的总字节数,具体结果列在表 7.4 中。

表 7.4 JIAJIA 与 PVM 的消息量

应用	消息个数		消息量(KB)	
程序	PVM	JIAJIA	PVM	JIAJIA
SOR 4096	294	853	2 294	2 316
SOR 8192	294	1 411	4 588	4 613
Water	620	22 009	9 122	40 333
TSP	1 114	9 421	25	11 489
IS	140	1 050	573	894
Barnes	168	37 010	15 191	80 523
ILINK	6 615	207 881	47 583	391 062
QSORT	24 565	10 128	128 821	34 490

表 7.4 中列出的消息量的差异可以分为下列 4 种情况:

(1) 对 SOR 这样比较规整的矩阵计算问题,由于数据划分非常均匀,计算时需要的非本地数据不多,JIAJIA 系统发送的消息虽然多一些,但传送的总字节数已与 PVM 非常接近。

(2) 对使用较少共享存储空间的不规则问题,如 Water、TSP、Barnes 等,JIAJIA 系统发送的消息量和传送的字节数均比 PVM 多数倍乃至数十倍。但是,由于 JIAJIA 系统采用了多种通信与计算重叠的优化技术,而且这些问题本身计算量较大,因此增加的通信量对整体性能影响不大。

(3) 对 ILINK 这个结构尚不清楚的问题,由于共享数据的分配未达到最优,在通信量上出现了与 PVM 的较大差距,且总体性能也不稳定(参见表 7.3)。

(4) 由于具体编程方案上的差异,程序 QSORT 的 PVM 版本通信量极大,导致它的总体性能远低于 JIAJIA 版本的性能。

关于 JIAJIA 系统与 PVM 比较的详细分析见参考文献[5]。

7.4.4 部分优化措施的效果

可以用曙光 2000 - I 的 8 个结点测试前述懒惰写检测、宿主自动迁移以及写向量技术的优化效果。曙光 2000 - I 的每个结点包含 1 个 300 MHz 的 PowerPC 604 处理器以及 256 MB 内存。结点间通过 100 Mb/s 的交换式以太网相连。测试程序包括 Water、Barnes、LU、MG、3DFFT、SOR、TSP、ILINK、IAP18 以及 EM3D。表 7.5 给出了上述应用程序的一些特征以及在曙光 2000 - I 上串行程序的运行时间。

表 7.5 部分优化措施的测试程序及其串行时间

程序	规模	共享内存	Barr 数	Lock 数	串行时间(s)
Water	1 728 mole.,10iters	0.5 MB	70	1 040	255.28
Barnes	16 384 bodies	1.6 MB	28	64	279.76
LU	4 096×4 096	128 MB	256	0	605.24
SOR	4 096×4 096,100iters	64 MB	200	0	267.86
ILINK	LGMD - 1 - 2 - 3	12 MB	740	0	487.50
TSP	- f20 - r15	0.8 MB	0	1 167	129.61
MG	256×256×256,8iters	443 MB	592	0	295.12*
3DFFT	128×128×128,4iters	96 MB	12	0	70.89
EM3D	120×60×416,100iters	160 MB	200	0	664.00
IAP18	144×91×18,1day	20 MB	5 400	0	2 508.00

* 由于内存限制,256×256×256 的 MG 在单机上无法运行,表中串行时间为根据问题规模的估计时间,等于 128×128×128 的 MG 串行时间(36.89 s)的 8 倍。

为了测试上述优化措施的效果,每个测试程序都在如下 4 种设置的 JIAJIA 系统中运行:没有上述 3 种优化的普通 JIAJIA(用 JIA 表示),经懒惰写检测优化的 JIAJIA(用 JIA_l 表示),经懒惰写检测以及写向量优化的 JIAJIA(用 JIA_{lw} 表示),以及经上述所有 3 种优化的 JIAJIA(用 JIA_{lwm} 表示)。除了 LU_p 和 SOR_p,其他程序的共享数据都根据最佳方案进行初始分布。而 LU_p 和 SOR_p

是 LU 和 SOR 的初始数据按页在各处理机间均匀分布的情况。

图 7.2 以直方图的方式给出了各测试程序在 JIA、JIA_l、JIA_{lw} 以及 JIA_{lwm} 中的八机运行时间。其中，各测试程序在 JIA 中的运行时间在相应直方图的顶上(单位：s)，而在 JIA_l、JIA_{lw} 以及 JIA_{lwm} 中的运行时间可以根据直方图的比例得出。在图 7.2 中，并行执行时间分成缺页时间(即 SIGSEGV 服务时间)、同步时间、服务时间(即 SIGIO 服务时间)以及计算时间 4 个部分。其中，前 3 部分时间是在运行时统计的系统开销，计算时间是实际运行时间减去系统开销所得到的。表 7.6 给出了并行程序运行的一些统计数据，包括消息量、远程取页数、远程写 diff 数，以及写本地只读 home 页引起的访问失效数。

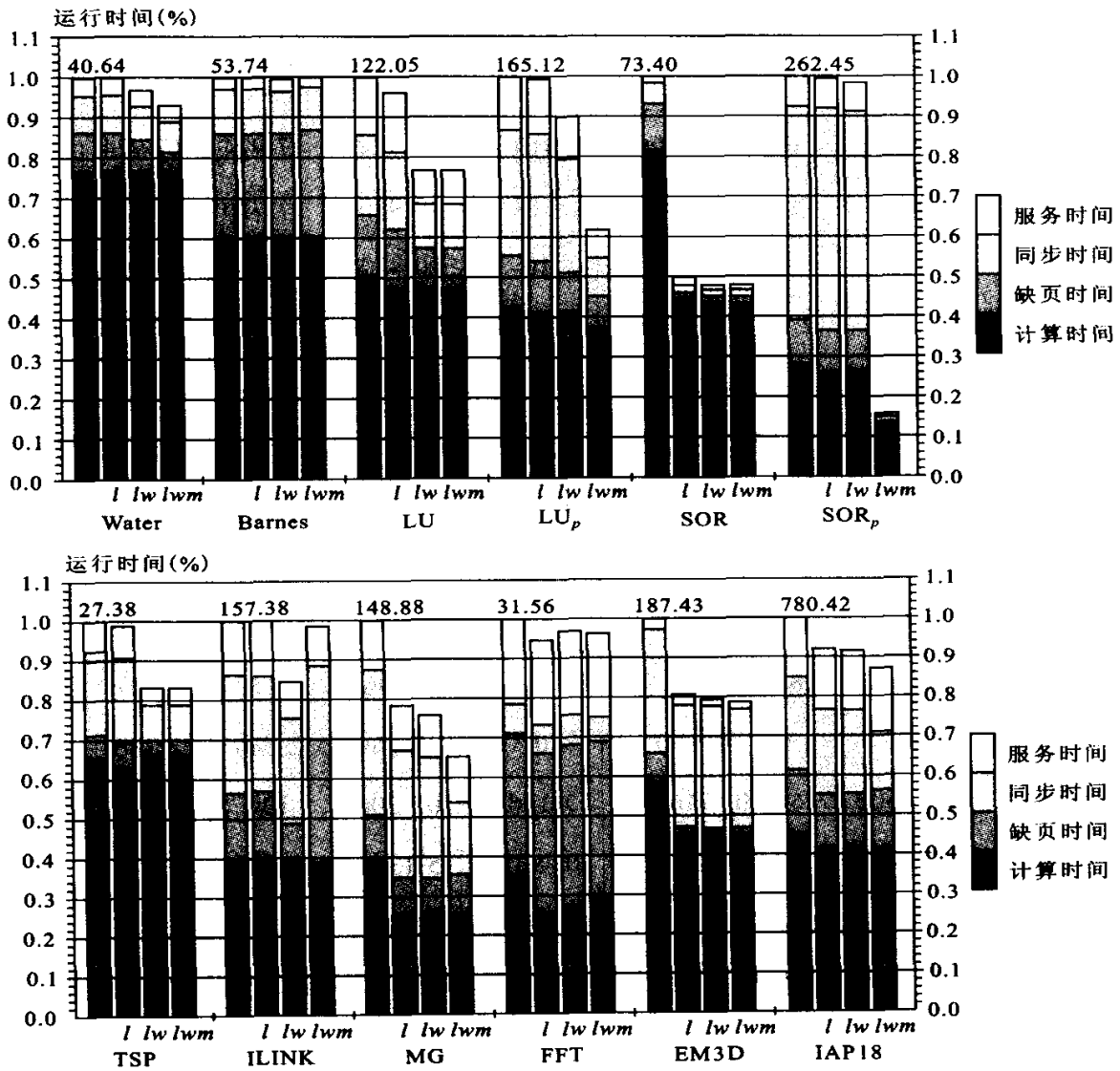


图 7.2 部分优化措施的并行计算时间比较

表 7.6 部分优化措施的并行执行统计数据

应用程序	消息量(MB)				远程取页数			
	JIA	JIA_l	JIA_{lw}	JIA_{lwm}	JIA	JIA_l	JIA_{lw}	JIA_{lwm}
Water	40	40	30	29	3 678	3 678	3 680	3 774
Barnes	75	75	72	72	9 692	9 692	9 692	9 692
LU	396	396	101	101	47 228	47 228	47 228	47 228
LU_p	470	469	261	118	43 451	43 331	43 331	49 303
SOR	23	23	2	2	2 800	2 800	2 800	2 800
SOR_p	533	533	51	3	2 400	2 400	2 400	2 789
ILINK	479	479	179	217	54 577	54 777	54 577	60 354
TSP	43	43	7	7	5 201	5 203	5 090	5 090
MG	653	653	609	410	17 924	17 924	17 924	23 444
3DFFT	149	149	149	149	17 976	17 976	17 976	17 976
EM3D	89	89	9	10	10 472	10 469	10 469	13 949
IAP18	2 890	2 890	2 740	2 644	269 583	269 589	269 589	300 312
应用程序	Diff 数				本地访问失效数			
	JIA	JIA_l	JIA_{lw}	JIA_{lwm}	JIA	JIA_l	JIA_{lw}	JIA_{lwm}
Water	2 370	2 370	2 370	1 883	340	339	339	819
Barnes	7 575	7 575	7 575	7 572	848	848	848	849
LU	0	0	0	0	135 963	16 709	16 709	16 709
LU_p	118 967	118 967	118 967	1 799	16 996	2 130	2 101	15 135
SOR	0	0	0	0	818 800	2 786	2 786	2 786
SOR_p	716 600	716 600	716 600	7 166	102 200	199	2 773	2 786
ILINK	27 467	27 467	27 467	10 801	5 136	3 352	3 352	9 931
TSP	3 414	3 439	3 350	3 350	294	278	364	364
MG	27 960	27 960	27 960	5 862	368 760	21 265	21 266	27 365
3DFFT	56	56	56	56	45 064	14 344	14 344	14 344
EM3D	9 900	9 900	9 900	3 729	905 100	10 256	10 256	13 721
IAP18	137 280	137 280	137 280	94 970	2 248 940	249 831	249 831	280 553

1. 懒惰写检测的效果。从图 7.2 中可以看出,懒惰写检测可以显著提高 LU、SOR、MG、EM3D 和 IAP18 的性能。从表 7.6 中可以看出,JIA_l 的本地缺页数比 JIA 要少得多。这是因为在懒惰写检测中,不对没有远程备份的 home 页进行写检测,减少了本地访问失效次数。这样,不仅减少了同步时间(写保护在同步操作中进行),而且减少了访问失效时间,这一点从图 7.2 中可以看出来。此外,图 7.2 还反映出在程序 SOR 中,JIA_l 性能的提高主要得益于计算时间的减少,说明频繁的访问失效对处理机的流水线、Cache 命中率以及 TLB 命中率等有较大影响。

2. 写向量的效果。写向量技术通过把一页分成若干块并在远程取页时只取被修改过的块来减少消息量。从表 7.6 和图 7.2 中可以看出,在大多数程序中,JIA_{lw} 都能有效地减少消息量,并明显地提高了 LU、LU_p、TSP 和 ILINK 的性能。由于远程访问是在 SIGSEGV 服务程序中进行,并由 SIGIO 服务程序提供服务,消息量的减少使得缺页时间和服务时间也相应减少,这一点在 Water、LU、LU_p、SOR、TSP、ILINK 和 EM3D 中比较明显。此外,在临界区密集的 TSP 程序中,缺页时间的减少能够缩短临界区,从而减少获取锁时的同步等待时间。

3. Home 自动迁移的效果。在 JIAJIA 系统的 home 自动迁移优化过程中,共享页的 home 根据程序的访问特征自动迁移到单写的处理机,以减少写共享页的 twin 和 diff 等开销。从图 7.2 中可以看出,在 Water、LU_p、SOR_p、MG、EM3D 以及 IAP18 中,JIA_{lwm} 获得了比 JIA_{lw} 更好的性能;而在 ILINK 程序中,JIA_{lwm} 的性能不如 JIA_{lw}。根据表 7.6 中的统计信息,在 SOR_p、LU_p、ILINK、MG、EM3D、Water 以及 IAP18 中,JIA_{lwm} 产生的 diff 远远少于 JIA_{lw},说明在这些程序中,单写是比较重要的共享行为。由于 diff 是在同步操作时产生的,并在 SIGIO 中断服务程序中写回到它们的 home,因此在多数程序中,JIA_{lwm} 访问失效开销以及服务开销都比 JIA_{lw} 小。分析表明,在程序 ILINK 中,虽然 JIA_{lwm} 也明显地减少了 diff 个数,但由于 JIA_{lwm} 把一些频繁访问的页迁移到 0 号处理机,使该处理机成为访问的瓶颈,增加了其他处理机缺页时的等待时间。

关于写向量及 home 自动迁移优化的详细分析见参考文献[52]及[53]。

7.4.5 SMP 优化的效果

关于 SMP 优化的测试是在由 Myrinet 相连的 4 台双 CPU 的 Ultra-2 工作站机群上进行的。每台工作站有 256 MB 内存。测试程序包括 Water、Barnes、Ocean、LU、MG、3DFFT、SOR、TSP、EM3D 以及 IAP18。表 7.7 给出了这些应用程序的特征以及串行时间。

表 7.7 SMP 优化的测试程序及其串行时间

程序	规模	共享内存	Barr 数	Lock 数	串行时间(s)
Water	1 728 mole.,10iters	0.5 MB	70	1 040	491.99
Barnes	16 384 bodies	1.6 MB	28	64	321.83
Ocean	514×514	60 MB	858	1 568	54.88
LU	2 048×2 048	32 MB	128	0	133.45
SOR	2 048×2 048,100iters	16 MB	200	0	89.44
TSP	-f 20 -r15	0.8 MB	0	1 167	216.70
MG	256×256×256,8iters	443 MB	592	0	398.56*
3DFFT	128×128×128,4iters	96 MB	12	0	74.05
EM3D	120×60×416,10iters	160 MB	20	0	101.46
IAP18	144×91×18,1 day	20 MB	5 400	0	1 994.95

*:由于内存限制,256×256×256 的 MG 的单机上无法运行,表中串行时间为根据问题规模估计的时间,等于 128×128×128 的 MG 串行时间(49.82 s)的 8 倍。

为了测试 SMP 优化的效果,JIAJIA 把 4 台双 CPU 的工作站分别当做 8 台单机(记为 JIA)以及 4 台 SMP(记为 JIA_{smp})来运行上述程序。图 7.3 以直方图的形式给出了各测试程序在 JIA 以及 JIA_{smp} 中的运行时间。其中,各测试程序在 JIA 中的运行时间在相应直方图的顶上(单位:s),而在 JIA_{smp} 中的运行时间可以根据直方图的比例得出。在图 7.3 中,并行执行时间分成缺页时间(即

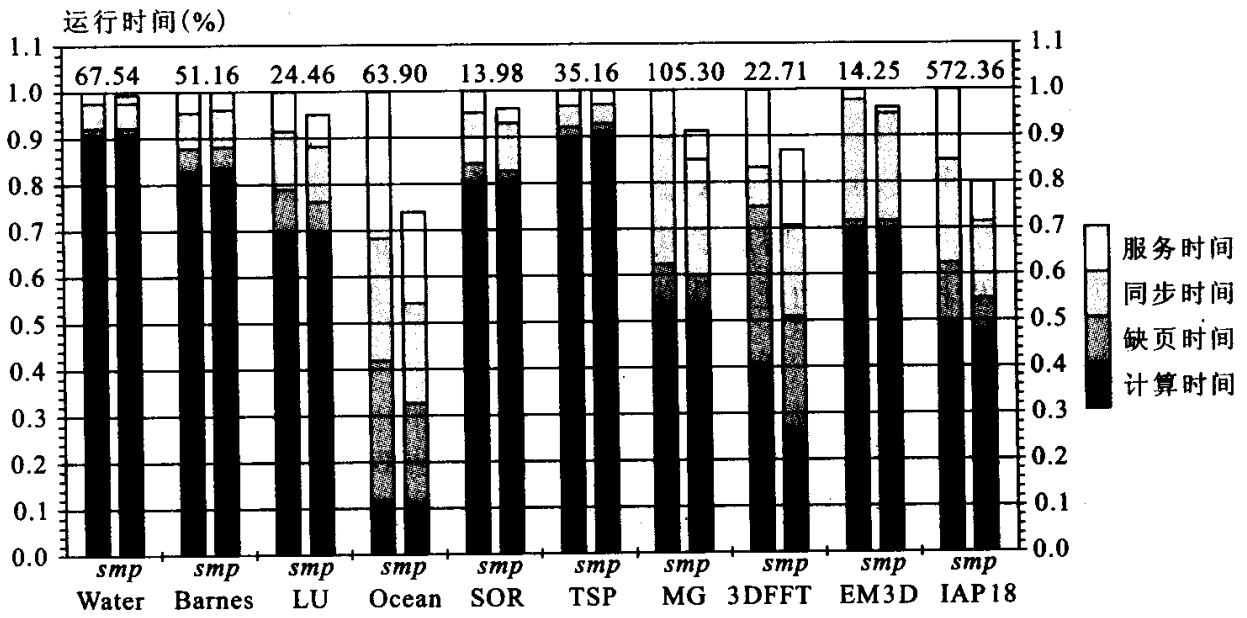


图 7.3 SMP 优化的并行计算时间比较

SIGSEGV 服务时间)、同步时间、服务时间(即 SIGIO 服务时间)以及计算时间 4 个部分。表 7.8 给出了并行程序运行的一些统计数据,包括消息量、远程取页数以及远程写 diff 数。

表 7.8 SMP 优化的并行执行统计数据

应用程序	消息量(MB)		远程取页数		Diff 数	
	JIA	JIA_{smp}	JIA	JIA_{smp}	JIA	JIA_{smp}
Water	36	29	3 420	2 870	1 897	1 373
Barnes	81	65	9 076	7 408	510	353
Ocean	891	563	107 073	67 733	1 294	1 132
LU	25	18	12 072	8 292	0	0
SOR	12	5.3	2 823	1 216	0	0
TSP	7.4	6.3	4 965	4 237	3 236	2 780
MG	553	260	32 508	18 025	43 192	17 016
3DFFT	149	129	17 976	15 408	56	48
EM3D	2.4	1.1	1 200	528	990	420
IAP18	2 943	1 355	276 643	136 474	131 136	49 920

从表 7.8 以及图 7.3 中可以看出,在所有 10 个测试程序中,JIA_{smp} 的远程取页数、diff 数以及消息量都比 JIA 少,并且在 7 个程序中取得了明显的性能提高。一般地说,JIA_{smp} 对所需共享内存大且加速比不好的应用程序提高性能的效果明显。由于远程访问是在 SIGSEGV 服务程序中进行的,写 diff 是在同步操作中进行的,且这两类消息都必须在 SIGIO 服务程序中得到服务,因此远程访问数以及 diff 数的减少将导致上述 3 类开销的减少,这一点可以从图 7.3 中看出,在程序 3DFFT 中,SMP 优化还导致了计算时间的减少,说明在程序 3DFFT 中,缺页中断对处理机的流水线、Cache 以及 TLB 的命中率有较大影响。

关于 SMP 优化的详细分析见参考文献[55]。

7.4.6 预取优化的效果

预取优化的测试在曙光 2000 - I 的 16 个结点上进行。测试程序包括 Water、Barnes、Ocean、MG、3DFFT、SOR、ILINK 以及 TSP。表 7.9 给出了上述应用程序的一些特征以及在曙光 2000 - I 上串行程序的运行时间。

表 7.9 预取优化的测试程序及其串行时间

程序	规模	共享空间	Barr 数	Lock 数	串行时间(s)
Water	1 728 mole.10iters	0.5 MB	70	1 040	251.96
Barnes	16 384 bodies	1.6 MB	28	64	279.76
Ocean	514×514	75 MB	858	1 568	40.59
SOR	8 192×8 192,100iters	256 MB	200	0	1 330.40
ILINK	LGMD-1-2-3	12 MB	740	0	487.50
TSP	-f 20 -r 15	0.8 MB	0	1 167	133.98
MG	256×256×256,16iters.	443 MB	1 184	0	590.24*
3DFFT	128×128×128,16iters.	96 MB	36	0	239.51

*:由于内存限制,256×256×256 的 MG 在单机上无法运行,表中串行时间为根据问题规模的估计时间,等于 128×128×128 的 MG 串行时间(73.78 s)的 8 倍。

JIAJIA 系统的预取优化分析每一 Cache 页的 INV 和 GETP 事件的周期性。如果发现其中某一种访问模式重复了一定的次数,就根据该模式在执行 barrier 操作或 acquire 操作时对该页发出预取,并合并向同一处理机的多个预取请求。为了测试预取的效果,在此分别测试了无预取的 JIAJIA(记为 JIA)、同一访存模式出现 2 次时预取的 JIAJIA(记为 JIA_{p2})以及同一访存模式出现 3 次时预取的 JIAJIA(记为 JIA_{p3})。

图 7.4 以直方图的方式给出了各测试程序在 JIA、JIA_{p2} 以及 JIA_{p3} 中的运行时间。其中,各测试程序在 JIA 中的运行时间在相应直方图的顶上(单位:s),而在 JIA_{p2} 和 JIA_{p3} 中的运行时间可以根据直方图的比例得出。在图 7.4 中,并行执行时间分成缺页时间(即 SIGSEGV 服务时间)、同步时间、服务时间(即 SIGIO 服务时间)以及计算时间 4 个部分。表 7.10 给出了并行程序运行的一些统计数据,包括消息数、消息量以及远程取页数。

从表 7.10 以及图 7.4 中可以看出,JIAJIA 系统的预取可以大大减少缺页的次数以及消息个数(由于合并预取),由于错误的预取引起的额外的消息量不多(在 JIA_{p3} 中平均只有 7%)。预取减少了缺页开销。由于预取在同步操作中进行,某些程序的同步开销略有增加。此外,多个处理机同时预取多页加剧了冲突,使得某些程序的远程服务开销略有增加。总的来说,JIA_{p2} 和 JIA_{p3} 在多数程序中能明显提高性能,包括 Ocean、ILINK、TSP、MG、3DFFT 以及 SOR。在程序 Water 中,JIA_{p2} 降低了性能而 JIA_{p3} 提高了性能。在程序 Barnes 中,JIA_{p2} 和 JIA_{p3} 都降低了性能。

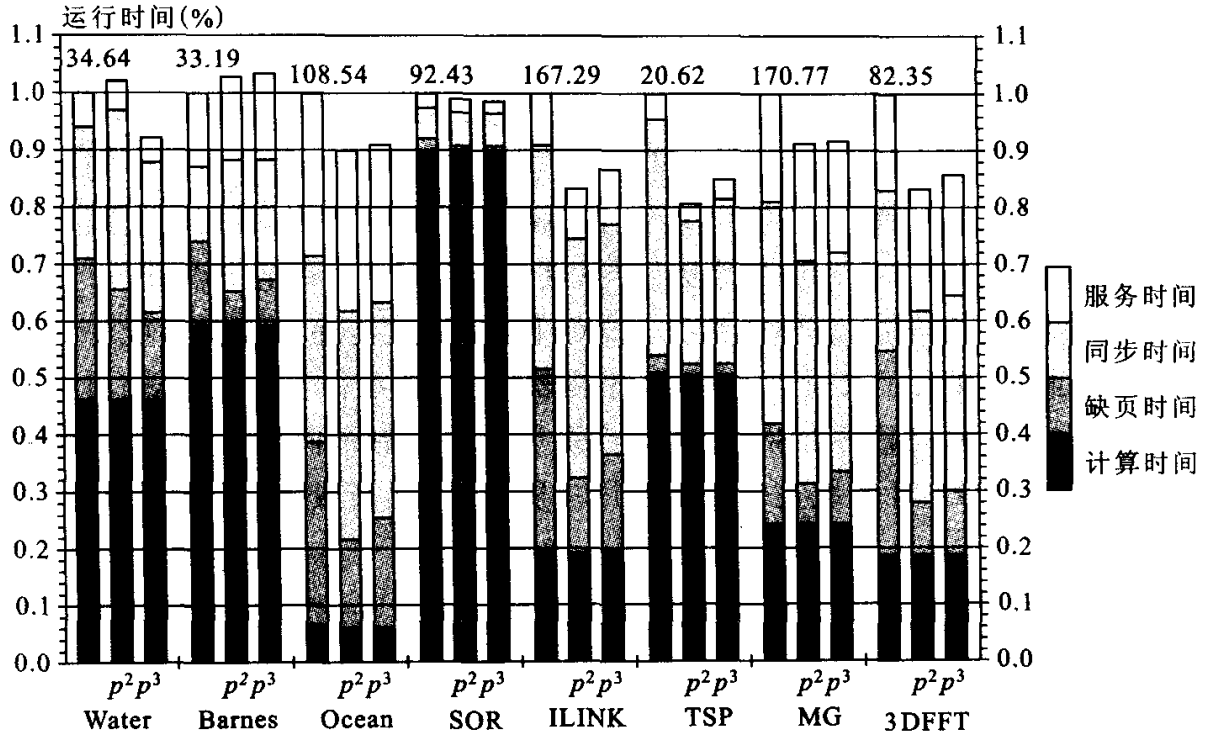


图 7.4 预取优化的并行时间比较

表 7.10 预取优化的并行执行统计数据

问题	消息个数			消息量(MB)			远程取页数		
	JIA	JIA_{p2}	JIA_{p3}	JIA	JIA_{p2}	JIA_{p3}	JIA	JIA_{p2}	JIA_{p3}
Water	34 507	29 913	29 276	61	81	62	8 511	1 516	2 283
Barnes	44 266	24 815	29 933	145	191	191	19 494	6 339	9 577
Ocean	405 678	245 524	284 203	1 097	1 246	1 179	181 477	51 762	81 825
SOR	30 060	18 360	18 478	42	42	42	12 000	240	360
ILINK	313 341	236 527	254 273	384	418	415	136 805	38 541	73 405
TSP	19 266	11 040	12 593	13	13	13	6 851	1 608	2 399
MG	240 557	150 895	170 439	1 565	1 760	1 681	88 982	15 888	31 774
3DFFT	133 110	42 000	48 510	551	549	549	65 760	11 580	15 450

关于 JIAJIA 系统预取优化的详细分析见参考文献[57]。

7.5 软件共享存储与消息传递的编程环境

自从第一个软件 DSM 系统 Ivy 诞生以来，人们致力于软件 DSM 系统的研

究已经有十几年的历史。但迄今为止,软件 DSM 系统还未成为主流的并行编程环境。像软件 DSM 这样实用性较强的研究方向,经过了十几年的发展,没有形成实用化的主流系统,但人们对它持续了十几年的研究热情不仅没有消退,而且近年来有越来越热的趋势,这在迅速发展的计算机领域是罕见的。与此相对应的是,消息传递的并行编程环境(如 PVM 或 MPI)在推出不久很快就成为主流的并行计算编程环境。究其原因,一是在并行计算(尤其是机群并行计算)发展之初,软件 DSM 还不成熟的情况下,人们除了消息传递的编程环境外别无选择,PVM 或 MPI 占了天时;二是近几年在软件 DSM 取得了长足发展,具备实用性条件之后,人们对软件 DSM 存在的许多误解并没有消除。最近,越来越多的迹象表明,软件 DSM 系统正被更多的用户所接受,逐渐成为主流的并行计算环境。

7.5.1 软件 DSM 与消息传递环境的可编程性

在消息传递的并行编程环境中,程序设计者必须对任务和数据进行划分,并安排在并行计算时处理机间的所有通信。在软件 DSM 系统中,由于系统提供了共享存储空间,用户只要进行任务的划分,不必进行数据的划分,也不用确切地知道并行计算过程中处理机之间的通信。

从通信的角度看,消息传递的并行程序比共享存储的并行程序进程间关联复杂,体现在空间关联和时间关联两个方面。在空间关联方面,发送数据的进程需要关心自己产生的数据被谁用到,而接收数据的进程需要关心它用到了谁产生的数据;在时间关联方面,发送数据的进程通常必须在数据被接收后才能继续,而接受数据的进程必须等待收到数据后才能继续。在共享存储的并行程序中,处理机间的通信通过访问共享存储器完成,用户只需考虑进程间同步,不用考虑进程间通信。尤其是对于比较复杂的数据结构的通信,如 struct{int * pa; int * pb;int * pc},消息传递的并行程序比共享存储的并行程序复杂得多。

从数据划分的角度看,消息传递的并行程序必须考虑诸如数组名称以及下标变换等因素,在串行程序并行化时,需要修改大量程序代码。而在共享存储的环境中进行串行程序并行化时,不用进行数组名称以及下标变换,对代码的修改量很少。

此外,对于一些在编程时难以确切知道进程间通信的程序,用消息传递的方法很难进行并行化。图 7.5 是多项式图像纠正的串行程序和基于 JIAJIA 的并行程序片段。纠正的方法是通过多项式建立源图像和目标图像间的对应关系,即对于目标图像的每个点坐标,通过多项式(程序中 xy()是计算多项式的函数)计算相应点在源图像中的位置并把目标图像中该点的像素值置为源图像中相应点的像素值。由于图像所需的空间大于单机内存空间,因此在串行程序中没有

把图像取进内存。在 JIAJIA 系统中,由于可以把多机内存合并成大共享空间,所以可以把输入输出图像(即程序中的 in 和 out 共享数组)取进内存。不难看出,在这个程序中,处理机间的通信是动态的,在编程时难以确定。当这个程序在消息传递的编程环境中并行时,较可行的方法是让每个处理机都持有源图像的一个完整备份。但这样做一方面增加了通信量(初始化时 0 号处理机需要把 in 数组传播给所有处理机),另一方面在源图像太大、单机内存放不下时难以实行。

```
for (i=0,i<lin2;i++)
for (j=0;j<col2;j++){
  x=xy (P,j,i,aa)+0.5;
  y=xy(P,j,i,bb)+0.5;
  if(x<0||x>=col1||y<0||y>=lin1){
    putc(0,fp2);
  } else{
    fseek(fp1,y*col1+x,0);
  }putc(getc (fpin),fpout);
}
```

(a) 串行程序

```
start=lin2/jiahosts*jiapid;
end=start+lin2/jiahosts;
for (i=start;i<end;i++)
for (j=0;j<col2;j++){
  x=xy(P,i,j,aa)+0.5;
  y=xy(P,j,i,bb)+0.5;
  if(x<0||x>=col1||y<0||y>=lin1){
    out[i*col2+j]=0;
  }else{
    out[i*col2+j]=in[y*col1+x];
  }
}
jia_barrier;
```

(b) 并行程序

图 7.5 图像纠正程序片段

当然,共享存储的并行程序虽然不用考虑数据划分,但为了得到较好的系统性能,通常需要考虑共享数据在处理机间的分布。但数据分布与数据划分是很不同的。消息传递程序中的数据划分通常需要改变对被划分对象的引用,包括数组的名字以及数组下标等;而数据分布只修改分配部分,不需修改引用部分。例如,在图 5.2 的矩阵乘法程序中,利用 JIAJIA 系统的共享空间分配函数 jia_alloc()来分配 a、b、c 数组。由于在矩阵乘法时任务是按块来划分的,为了增加访问局部性,减少远程访问的次数,应该分配共享空间如下:

```
a=(double*)jia_alloc3(N*N*8,N*N*8/jiahosts,0);
a=(double*)jia_alloc3(N*N*8,N*N*8/jiahosts,0);
a=(double*)jia_alloc3(N*N*8,N*N*8/jiahosts,0);
```

而对 a、b、c 数组的访问不变。显然,共享存储并行程序中的数据分布与消息传递并行程序中的数据划分是本质不同的。

软件共享存储在可编程性方面的另一个优势是,减少了编译器在自动并行编译及程序转换时的负担。由于不用考虑数据划分以及处理机间的通信,面向

共享存储编程界面的自动编译或程序转换工具比面向消息传递界面的自动编译或程序转换工具要容易得多。例如,在面向消息传递界面进行自动并行编译时,编译器常常难以准确分析处理机间的通信,只好采用比较保守的策略,把所涉及的所有数组都进行传播;而面向共享存储界面进行自动编译时,编译器不用考虑通信及数据划分,运行系统会根据访问的需要自动进行处理机间的通信。

7.5.2 软件 DSM 与消息传递环境的性能

软件 DSM 系统未能成为主流并行计算环境的一个重要原因是其性能不够理想。一般来说,除了像图 7.5 中的在编程时难以确定通信的程序外,手工精雕细刻的消息传递并行程序的消息量是最少的,是共享存储并行程序通信量的下限。但近几年软件 DSM 技术的发展,尤其是多写协议以及延迟传播技术的提出,已经大大减少了软件 DSM 系统的开销。近年来网络带宽的迅速发展也减少了并行程序的性能对通信量的敏感性。此外,与消息传递系统中集中的通信方式相比,软件 DSM 的分散的以页为单位的通信有利于减少网络冲突。这些因素使得近期的软件共享存储系统能获得与消息传递系统可比的性能。除了前面对 JIAJIA 系统和 PVM 的比较之外,参考文献[76]也对基于 PVM 系统和 TreadMarks 系统的并行程序进行了性能比较,得出了软件 DSM 并行程序的性能在消息传递并行程序性能的 80% 之上的结论。因此,软件 DSM 系统和消息传递系统在程序的可并行性上没有本质上的区别。一般来说,在消息传递系统中可并行化的程序也可用软件 DSM 进行并行化而得到可比的性能。

在实现自动任务迁移以及负载自动平衡方面,软件 DSM 系统比消息传递系统具有更强的优势。一般来说,消息传递系统的任务划分一旦确定,很难在运行时进行动态的修改,因为数据的划分已经固定。而在软件 DSM 系统中,在运行时进行任务自动划分就容易得多。目前,在 JIAJIA 系统上已经实现了简单的循环级负载自动平衡[86]。

软件共享存储系统在性能方面的另一优势是可以更好地利用硬件支持。例如,在 SMP 机群系统中,软件 DSM 系统可以实现在 SMP 结点内部利用硬件共享而在 SMP 结点之间利用软件实现共享。因此,在 SMP 结点内部通信时,软件 DSM 通过访存指令在存储器总线上直接交换数据(SMP 的侦听协议可以通过总线在处理机 Cache 间交换信息),显然比通过函数调用进行通信的消息传递快得多。Ocean 程序在一台四 CPU 的 Sparc 20 工作站上运行时,如果把 4 个 CPU 当做 4 台独立的机器,在通信最优的情况下,JIAJIA 程序四机运行时间为 106 s(单机运行时间仅为 36 s),而如果把 4 个 CPU 当做一个 SMP 结点中的 4 个处理机,利用硬件共享,则 JIAJIA 程序四机运行时间仅为 18 s。可见,SMP 内硬件共享通信比消息传递通信快得多。此外,在结点之间,软件 DSM 可以更好地

利用支持远程访问(或远程 DMA)的网络来减少通信开销。VIA(Virtual Interface Architecture,虚拟接口结构)标准[27]就支持远程 DMA 的功能。表 7.11 给出了 Dolphin 公司的 SCI 网络中对于不同长度的消息共享存储通信和消息传递通信的延迟。从表中可以看出,在支持远程访问的网络中,共享存储的通信开销比消息传递的通信开销要小。

表 7.11　Dolphin SCI 网络通信延迟(μs)

消息长度	消息传递通信	共享存储通信
16	14.3	0.22
256	17.5	4.5
4096	83.3	71.4

可见,在难以完全由硬件实现共享存储的情况下,利用软件实现共享存储,并由硬件提供必要的支持,可以既改善机群系统可编程性,又有效提高系统的性能。

7.6 小　结

共享虚拟存储系统由于结合了共享存储多处理机系统容易编程和消息传递多计算机系统容易实现的特点而引起了广泛的研究。然而,软件 DSM 系统中较大的共享和通信粒度(通常是存储页)会导致假共享及额外的通信量等问题。此外,在软件 DSM 系统中,尤其是在基于工作站网络的系统中,通信开销很大。本章先从实现方式、一致性协议以及编程界面等角度介绍了共享虚拟存储系统中存在的问题和关键技术。然后,介绍了我国自主研制的共享虚似存储系统 JIAJIA。JIAJIA 实现了基于锁的新型一致性协议,采用类似于 NUMA 的存储器组织方式,能够把多个机器的存储器组织起来形成一个更大的存储空间。同时,JIAJIA 系统还实现了若干优化策略,如懒惰写识别、写向量、宿主自动迁移、SMP 优化以及数据动态预取等,来避免假共享、减少通信次数和通信量并容忍或隐藏通信延迟。

采用一些被广泛使用的测试程序,如 SPLASH2 和 NAS 并行程序集,对 JIAJIA 系统进行测试。测试结果表明,同近期实现的共享虚拟存储系统(如 CVM)比较,JIAJIA 系统不仅具有更高的性能,而且可以解决更大规模的问题;基于 JIAJIA 的并行程序性能与基于消息传递环境 PVM 的并行程序性能相当;JIAJIA 系统的优化措施可以有效地降低系统开销,提高系统性能。

此外,本章还就可编程性和性能两方面对软件 DSM 和消息传递并行环境进行了比较,指出软件 DSM 经过十几年的发展,已经到了推向实用的阶段。在

难以完全实现硬件共享的机群系统中，利用软件实现共享存储，并由硬件提供必要的支持，是既能改善系统可编程性，又能有效提高性能的方法。

JIAJIA是一个免费软件，目前已被20多个国家和地区的100多个科研单位使用。关于JIAJIA的更多信息可以从http://www.ict.ac.cn/chpc/index.html处获得。

第8章 总 结

8.1 本书内容总结

随着单处理机的性能越来越接近物理极限，当前的高性能计算机系统都采用并行处理结构。从存储管理的角度看，并行处理系统可分为共享存储多处理机系统和消息传递多计算机系统两类。分布式共享存储多处理机系统以其方便的编程环境和良好的可伸缩性而成为计算机体系结构发展的重要方向[17]。与集中式共享存储系统(如并行向量机系统和 SMP 系统)相比，分布式共享存储系统具有更好的可伸缩性。与消息传递的多计算机系统相比，共享存储系统支持传统的单地址编程空间，因此，更具有通用性。

然而，在共享存储系统中，多个处理机对同一地址空间的共享大大增加了系统复杂性，对系统正确性和系统性能产生了重要影响。在单处理机系统中，只有一个处理机访问存储器，处理机的访存行为比较简单。取数操作总是取回“最近”一个对同一单元的存数操作所写的值，而存数操作惟一地确定“此后”对同一单元的取数操作所取回的值。而在分布式共享存储系统中，多个处理机同时访问共享存储器。访存时间的不一致以及同一单元的多个备份破坏了存储访问的不可分割性，使得同一单元内容的变化在不同的时刻被不同的处理机所接受，在单机系统中的所谓“最近”、“此后”的概念不复存在。为了保证正确性，需要对访存操作的发生次序进行严格的限制，许多在单处理机中行之有效的提高性能的技术，如流水、多发射、预取、缓存等，不能在共享存储系统中盲目使用，因为这不利于提高性能。同时，维持 Cache 一致性需要复杂的硬件，从而影响了共享存储系统的可伸缩性。

可见,共享存储多处理机中的存储系统有着不同于其他计算机存储系统的特征,带来了一些新问题,集中体现在访存事件及其发生次序上。针对这一问题,本书从访存事件次序的角度系统地研究了共享存储系统中维护数据一致性、提高性能和增加系统的可伸缩性等方面的问题。

第二章利用集合论的方法建立了一个共享存储系统的执行正确性模型。该章从一个简单的程序模型开始,首先讨论了串行执行的正确性。然后,在研究冲突访问对执行结果的重要影响的基础上,定义了共享存储系统中执行的概念。并根据一个正确的并行执行的结果应等于同一程序的一个串行执行的结果这一顺序一致性的要求,讨论并证明了使一个并行执行正确的充要条件。具体地说,判断一个并行执行正确与否的标准是其结果是否等于同一程序在单机多进程环境下的某一执行的结果。决定一个执行结果的关键因素是此执行中冲突访问的执行次序。程序 PRG 的一个并行执行 $E(PRG)$ 从本质上说是程序 PRG 的冲突访问对集 $C(PRG)$ 的一个无圈定序。$E(PRG)$ 正确的充要条件是 $E(PRG) \cup PO(PRG)$ 无圈。在 $E(PRG) \cup PO(PRG)$ 中无弦的圈称为 $E(PRG) \cup PO(PRG)$ 的关键圈。在 $E(PRG) \cup PO(PRG)$ 中的一个关键圈中不可能有连续的 $\xrightarrow{PO}$ 边;由连续的 $\xrightarrow{E}$ 边构成的路径只能以 $w \xrightarrow{E} w$、$w \xrightarrow{E} r$、$r \xrightarrow{E} w$ 或 $r \xrightarrow{E} w \xrightarrow{E} r$ 的方式出现,其中 r 是读操作,w 是写操作。在 $E(PRG) \cup PO(PRG)$ 中的一个关键圈中两个相邻的存数操作之间只有 5 种可能路径。

第三章讨论正确执行的访存事件次序条件。它首先根据在分布式共享存储系统中访存操作可分割的特点,通过把任一访存操作分成若干子操作,建立了一个共享存储访问模型。并在该模型以及执行正确性模型的基础上,研究保证执行正确的访存事件次序条件。首先,说明了写一致条件是共享存储系统正确执行的前提。然后,给出一个较通用的满足顺序一致性的访存事件次序条件。根据该条件,证明了顺序一致性的一个典型实现(GPPO 条件)的正确性,该条件要求系统中所有处理机根据指令在程序中出现的次序执行指令。并在此基础上推出了一种乱序执行的方案,证明了只要满足一定条件,访存操作就可以越过它前面的指令执行而不会破坏顺序一致性。

第四章把前两章对执行正确性模型与访存事件发生次序的研究同一个具体的基于目录的 Cache 一致性协议结合起来,讨论在基于目录的 Cache 一致性协议中访存事件次序条件的实现方法,并提出通过猜测执行实现乱序执行的方案。此章还建立了一个地址流驱动的模拟模型来评价不同的访存事件次序条件对性能的影响。模拟结果表明:本书提出的乱序执行方案能有效地提高顺序一致共享存储系统的性能,延迟无效的猜测执行技术由于比检测纠正的猜测执行技术能更精确地判断猜测执行的正确性,因而其性能也优于后者。此外,在本书所做

的模拟中，访存冲突对系统性能有重要影响。由于在乱序执行的情况下，一个处理机可以并行执行多条指令，加剧了访存冲突，因此，乱序执行的作用只有在访存冲突不严重的情况下才能充分发挥出来。

第五章建立了存储一致性模型的一个数学模型。该模型可以看做是把前几章建立的执行正确性模型及访存正确性条件由顺序一致性推广到其他一致性模型的情况。与传统的从访存事件次序的角度来刻画存储一致性模型的方法不同，该模型从存储一致性所体现出的行为的角度来描述存储一致性模型。基于并行执行的行为由该执行中冲突访问的执行次序确定的这种认识，本章把存储一致性模型看做是一种处理机间同步机制，该同步机制确定处理机间冲突访问的执行次序。并行执行的结果由同步序与程序序共同确定。存储一致性模型 M 中的一个执行本质上是对程序中同步操作的一个定序。存储一致性模型 M 中正确程序的条件是，对该程序在 M 中的任一执行，程序中所有冲突访问都被程序序或同步序定序。一个系统满足存储一致性模型 M 的条件是，该系统对于 M 中正确程序的任一执行，该执行的同步序与程序序一致。存储一致性模型的实现本质是该模型对访存事件次序的一组限制。本章基于这种认识，以顺序一致性、释放一致性以及域一致性为例给出了证明存储一致性模型正确实现的方法。

第六章讨论高速缓存一致性协议中的一些关键问题。高速缓存技术在共享存储系统中占有重要地位。它不仅可以削弱由处理机和存储器的空间分布引起的延迟，而且可以减少多个处理机在访问共享存储器时的冲突。然而 Cache 在多处理机中会引起 Cache 一致性问题，即如何使同一单元在不同 Cache 以及主存中的多个备份保持数据一致的问题。Cache 一致性协议通过把某个处理机新写的值传播给其他处理机来具体实现这种一致性。本章从新值的传播方式(写使无效与写更新)、新值的传播时机(及时传播与延时传播)、新值的来源(单写协议与多写协议)以及新值传播给谁(侦听协议与目录协议)等不同的侧面介绍了 Cache 一致性的关键技术。在此基础上，本章提出了实现域一致性模型的基于锁的新型 Cache 一致性协议。该协议消除了传统的目录协议所带来的存储空间的浪费，通过在锁上附带一致性信息来维护一致性。它比目录协议更简单、有效，更利于系统的扩展。

第七章讨论共享虚拟存储系统中的关键问题。共享虚拟存储系统由于结合了共享存储多处理机系统容易编程和消息传递多计算机系统容易实现的特点而引起了广泛的研究。然而。软件 DSM 系统中较大的共享和通信粒度(通常是存储页)会导致假共享及碎片等问题。此外，在软件 DSM 系统中，尤其是在基于工作站网络的系统中，通信开销很大。本章先从实现方式、一致性协议以及编程界面等角度介绍了共享虚拟存储系统中存在的问题和关键技术。然后，介绍

了我国自主研制的共享虚拟存储系统 JIAJIA。JIAJIA 实现了上一章提出的基于锁的一致性协议，采用类似于 NUMA 的存储器组织方式，能够把多个机器的存储器组织起来形成一个更大的存储空间。同时，JIAJIA 系统还实现了若干优化策略，如懒惰写识别、写向量、宿主自动迁移、SMP 优化以及数据动态预取等来避免假共享、减少通信次数和通信量，并容忍或隐藏通信延迟。另外，采用一些被广泛使用的测试程序，如 SPLASH2 和 NAS 并行程序集，对 JIAJIA 系统进行了测试。测试结果表明，同近期实现的共享虚拟存储系统如 CVM 比较，JIAJIA 系统不仅具有更高的性能，而且可以解决更大规模的问题；基于 JIAJIA 的并行程序性能与基于消息传递环境 PVM 并行程序的性能相当；JIAJIA 系统的优化措施可以有效地降低系统开销，提高系统性能。此外，在本章中，还就可编程性和性能两方面对软件 DSM 和消息传递并行环境进行了比较，指出软件 DSM 经过十几年的发展，已经到了推向实用的阶段。在难以完全实现硬件共享的机群系统中，利用软件实现共享存储，并由硬件提供必要的支持，是既能改善系统可编程性，又能有效提高性能的方法。

8.2 共享存储系统发展趋势

共享存储系统的发展经历了以下几个阶段。

从 20 世纪 70 年代到 20 世纪 80 年代中期，并行处理系统主要是 SMP 系统以及(并行)向量机系统。这两类系统虽然在结构上不一样，但都支持共享存储的编程界面，并行编程容易；而且处理机之间耦合紧密，并行效果明显。因此，这两类系统很快成为并行计算机系统发展的主流。目前，这两类系统在技术上已经成熟，被用户普遍接受，是市场上主要的并行处理产品，被广泛应用在科学计算、事务处理、服务器等各个领域。SMP 系统还常被当做基本结点构成更大的并行处理系统。但这两类系统由于其系统结构本身的特点而难以扩展到很大的规模。

随着人们对高性能计算的需求不断扩大，SMP 系统和并行向量机系统已经不能满足大规模并行计算的要求。由于难以立即找到可伸缩性好的共享存储系统结构，消息传递的大规模并行处理系统在 20 世纪 80 年代后期及 20 世纪 90 年代中前期得到迅速发展。这一时期 MPP 系统的互联网络成为并行系统结构的研究热点。不少公司推出了消息传递的 MPP 产品，如 Thinking Machine 公司的 CM5，Intel 公司的 Paragon，IBM 公司的 SP2，以及 Cray 公司的 T3D，等等。由于可编程性差等原因，这些系统主要被用于科学计算，很少被用在事务处理等其他领域。同时，为了找到可伸缩性好的共享存储系统结构，人们对分布式共享存储系统展开了深入研究，其中有代表性的系统包括美国斯坦福大学的 DASH、

美国麻省理工学院的 Alewife 以及 Kendall Square Research 机构的 KSR1 等。

从 20 世纪 90 年代后期开始,分布式共享存储技术逐渐成熟。以 SGI 公司的 Origin 2000 为标志,共享存储系统再次受到计算机生产厂家以及并行处理用户的青睐,成为主流的并行处理系统。这一阶段的主要系统还包括 Sun 公司的 Starfire 系统、Compaq 公司的 Wildfire 系统以及 IBM/Sequent 公司的 NUMA-Q 系统等。Starfire 实现了 64 个处理机的超级 SMP 系统,该系统采用交叉开关互连,并采用多条总线利用侦听协议进行一致性维护。Wildfire 系统使用交叉开关互连并通过目录机制维护一致性。NUMA-Q 系统采用 SCI 互联网络及一致性维护机制实现共享存储。这些共享存储系统一般包含几十个结点,主要用于服务器及事务处理等。而 SGI 公司的 Origin 2000 共享存储系统则主要作为计算服务器,它采用 CC-NUMA 的结构,使用目录机制维护一致性并通过专门的网络设计使得访问远程共享存储器的延迟仅为访问本地存储器的 2 到 3 倍,最多可到几百甚至上千个结点。Cray T3E 系统也支持共享存储,但没有一致性的维护,该系统可伸缩到几千个结点。此外,共享虚拟存储技术的发展使得在消息传递系统或工作站机群上通过软件实现的共享存储系统具备了实用的条件。在这一阶段,随着一些专门生产并行机的公司的倒闭或兼并,基于消息传递的 MPP(massive Parallel Processing 大规模并行计算)系统逐渐从主流的并行处理市场退出,应用领域主要局限于大规模科学计算。同时,由于消息传递系统比较容易实现,成为实现超大规模并行处理的重要手段,其研制也逐渐成为政府行为,如美国的 ASCI 计划。

根据共享存储系统的发展历程,可以看出如下发展趋势:

1. 大规模和超大规模的、以科学计算为目的的 MPP 系统由于其实现复杂性,仍以消息传递为主。如 Intel 公司的 ASCI Red、IBM 公司的 ASCI Blue Pacific 以及 SGI 公司的 ASCI Blue Moutain 等系统都采用消息传递的结构。一方面,这几个系统处理机数都在 5 000 个以上,峰值速度在 3TFLOPS 以上,系统设计主要解决如何把成千上万个处理机高效地连接在一起以及系统稳定性等问题,难以容忍在这么庞大的系统中进行共享存储组织以及维护数据一致性所带来的复杂度。另一方面,这类系统应用面较窄,主要用于科学计算,如上述 3 个系统分别安装在美国的 3 个核武器研究实验室用于核模拟。

2. 中小规模的并行处理系统,尤其是服务器系统,将主要采用共享存储结构。虽然共享存储系统从集中式(SMP 系统和并行向量机系统均为集中式共享存储系统)到分布式的发展过程中存在一个不易衔接的时期,在这个时期内消息传递系统取得了很大发展。但由于其固有的缺陷,消息传递系统的应用主要局限于科学计算以及其他一些比较容易并行执行的领域,难以取代共享存储系统在传统并行处理市场(如以数据库为核心的事务处理、服务器以及一些科学计算

等)中的地位。可以预见,共享存储系统会逐渐成为中小规模(从几个到几百个结点)的并行处理及服务器的主流系统,并且随着硬件技术的进一步发展,共享存储系统的规模会进一步扩大。

3. 在机群系统中实现软件共享存储系统或通过适当硬件支持实现软硬件结合的共享存储系统也是并行处理系统的重要发展方向。事实上,随着网络技术的发展,机群系统和 MPP 系统的界限越来越模糊。一方面,网络硬件技术的发展使得网络的通信带宽有了长足的提高,如千兆位以太网、Myrinet 及其他高性能网络的发展使得网络带宽已经接近或超过 PCI 总线所能提供的带宽。另一方面,用户级通信技术的发展使网络的延迟得以大大降低,如在 Myrinet 上用户级通信的延迟仅为 6~8 μs,而在 Dolphin 的 SCI 网络上用户级通信的延迟在 1 μs 之内。与消息传递系统[如 MPI(Message Passing Interface,消息传递接口]相比,软件 DSM 系统除了在编程界面上的优势外,在支持远程访问的网络相连的机群系统以及 SMP 机群系统中还具有性能上的优势。因此,在机群系统中共享虚拟存储系统会成为一种主流的并行计算环境,但不可能完全取代 MPI 等消息传递系统。

附录：中英文术语对照

acquire，获取操作
barrier，栅障操作
Cache，高速缓冲存储器
Cache Coherence Protocol，高速缓存一致性协议
Cache-Only Memory Architecture (COMA)，唯高速缓存结构
diff，页差、块差
directory，目录
Distributed Shared Memory (DSM)，分布式共享存储
Eager Release Consistency (ERC)，急切更新释放一致性
Entry Consistency (EC)，单项一致性
Grid Computing，格网计算
home，宿主
home-based，基于宿主
homeless，无宿主
Lazy Release Consistency (LRC)，懒惰更新释放一致性
lock，锁
Massive Parallel Processing，大规模并行计算
Memory Consistency Model，存储一致性模型
Message Passing，消息传递
Metacomputing，元计算
multicomputer，多计算机
Multiple Writer，多写
multiprocessor，多处理机
Non-Uniform Memory Access (NUMA)，非一致访存结构
owner，属主

Probowner，可能属主
Processor Consistency (PC)，处理机一致性
Relaxed Consistency Model，弱一致性模型
release，释放操作
Release Consistency (RC)，释放一致性
Scope Consistency (ScC)，域一致性
Sequential Consistency (SC)，顺序一致性
sequential sharing，顺序共享
Shared Memory，共享存储
Shared Virtual Memory，共享虚拟存储
snoopy，侦听
Symmetric MultiProcessor (SMP)，对称多处理机
synchronization interval，同步区间
tight sharing，紧密共享
twin，页备份、块备份
Weak Consistency (WC)，弱一致性
Write Atomic，写不可分割
Write Invalidate，写使无效
Write Nonatomic，写可分割
write-notice，写标志
write update，写更新

后记：博士生创新能力的培养点滴

创新能力的培养是博士生教育的核心，博士生的学位论文必须有所创新。下面是笔者在攻读博士学位期间和毕业后工作中，对创新能力培养的一些体会。

1．导师起着关键作用

博士导师对博士生创新能力的培养起着十分重要的作用。导师要有很强的责任心。一方面，导师要对国家负责，招收学生的根本目的是为国家培养人才；另一方面，导师要对学生负责，除了让博士生学到知识，更重要的是教会学生做研究的方法，不能把学生当做廉价劳动力。如果让学生参与一些项目的研究，要尽量让博士生参加研究性的项目，避免让他们参与开发性的项目。有的导师招收博士生后，让博士生做大量的开发工作，直到毕业前一年甚至半年才让学生开始看文章，这样是培养不出好学生的。

导师既要给学生创造自由的研究环境，又要给学生必要的压力。在我攻读学位期间，我的导师夏培肃院士经常告诫我，一篇博士论文一定要有明显的创新，要有自己的思想和理论。否则，即使有很大的工作量，只可以算做若干篇硕士论文，也不能称其为一篇博士论文。我的博士学习生涯中，有近两年的时间就是在怕毕不了业的压力下度过的。直到阅读了大量的文献，对国际上相关领域的研究有了深刻的了解，才自然地有了一些新想法。

对于博士生的研究方向，导师不能不管，但也不必管得太细，以免限制学生本身创造性的发挥。导师应该向学生介绍本学科目前国际上的研究热点和发展方向，让学生自己去选择题目。在研究工作中，导师要注意对博士生研究方法和能力的培养。我在 1995 年 6 月完成博士论文的初稿，一直到 1996 年 2 月 29 日才答辩，其间根据夏老师的要求修改了二十多稿，历时 8 个月。夏老师对我的论文从学术内容，到章节的安排、单词的用法甚至标点符号，都进行了仔细的推敲和修改。一个在我看来是理所当然的结论，夏老师却要求我给出严格的证明。到答辩时，我的论文与其初稿相比已经是脱胎换骨、面目全非了。在这 8 个月的论文修改过程中，我受益匪浅。可以说，是导师言传身教地教会了我如何做学

问。

如果有可能，导师可以给学生创造条件，让学生接触国内外本领域的一些权威人士和学术带头人。这对学生毕业后的发展是很有好处的。我的导师夏培肃院士是中国计算机事业的创始人之一，是国内计算机系统结构领域的泰斗，我从她那里也获益颇多。

2．大量阅读文章，把握国际研究前沿

创新是对人类现有知识的拓展。博士论文的创新必须基于对所研究领域国际同行工作的全面了解。这就需要阅读大量论文，了解本领域现有的工作基础和需要解决的关键问题，甚至了解国际同行工作的特点。应该达到这样的程度：在谈到所研究方向的情况时，能列举出在该方向国际上较有名的研究组以及他们的工作成果及最新进展。一般来说，这个过程需要一年到两年的时间。我本人以及我的师兄唐志敏和马余泰，都花费了一年半以上的时间来阅读论文。根据我自己的体会，我很难相信，那些被导师的、与论文无关的项目占用了大量的时间，只在毕业的前一年（有的只有半年）才开始做论文的学生能做出合格的博士毕业论文。有些博士论文，连所研究领域的综述都写不好，更谈不上创新。

阅读文章是一个认识不断加深的过程，需要经过几个阶段。刚开始是了解阶段，了解自己所要研究的领域中国际同行的一些工作，只有一些点滴的了解，还没有全局的观念。随着阅读文章的深入，慢慢地会产生一些问题和迷惑，一方面，对国际同行的工作有了较全面的认识，觉得要解决的问题很多；另一方面，觉得自己能想到的别人都想到了，无从下手。这时候再看别人的文章时，不仅要了解，而且要注意领会，对一些核心的文章，可能要反复地看，同时结合自己的迷惑和问题进行深入的思考。这样过了一段时间，终于产生豁然开朗的感觉，觉得自己接触到了一些问题的本质。这时再回顾别人的工作，就能够发现其中一些不足的地方或值得改进的地方。

3．理论与实践结合，进行创新性研究工作

对工学学生来说，接下来可以进行一些实验和模拟。由于已经对别人的方法有了透彻的了解，在实验中就可能提出自己的方法或对别人方法的改进。即使自己的方法不如别人也不要灰心，而是要分析原因。在实验过程中，对于碰到的问题再去有针对性地阅读文章，这种有针对性的阅读会对别人工作的细微之处有更深的了解。如此反复，不断经历从实践到认识，再用认识指导实践的过程[1]，就能不断加深认识，发现问题，创新的火花也会从中迸发。从而对本研究方向的发展做出自己的贡献。

最后，当自己对所研究的领域有了系统的了解和创新后，再看论文时，就会觉得虽然文章多如牛毛，但很难看到让人爱不释手的好文章了。同时，也觉得自己以前发表的文章大部分是浅显的。在这种认识的基础上，才能避免为了写文

章而写文章的情况。我产生这种认识和体会的时候，是在博士毕业之后的若干年。

4．要有献身精神，注意学术道德的培养

在读书期间，应逐渐确立自己的人生方向，如果觉得自己不适合做学问或对金钱十分感兴趣，迟早要离开做学问的地方。

做科学研究是要有献身精神的。在生活上，不仅要准备过比较清贫的生活，而且要容忍各种社会压力和误解。在工作中，要全身心地投入，没有上班和下班的时间区别，即使在走路或吃饭的时候，头脑中还常常萦绕着一些问题，半夜也会为一个新想法的产生而兴奋得不能入睡。

做学问是一种踏踏实实的工作，需要长期的积累，不能有浮躁的心理。创新思想的产生，常常是在长期平淡的工作之后，因此，需要非常的耐心和持之以恒的决心。做学问最讲究实事求是，不能有半点的虚假。有就是有，没有就是没有。写文章的正确态度是，在做了大量的理论和实践研究之后，觉得自己的工作已经可以总结一下，不要为了文章而写文章，更不要为了文章的数量而涉足抄袭以及一稿多投这些禁区。论文的署名更要实事求是，须知做论文的作者既是一种荣誉，更是一种责任。近年经常看到一些知名学者由于学生的虚假论文而受连累的报道，我们要引以为戒。现在在学术界有一种认识：在学生完成的论文中，都要把导师作为作者之一。这即使在国外的许多地方也是天经地义、无可厚非的。但我的老师夏培肃院士却不允许这样做，除非论文中确实包括她的实质性的工作。在我入学之初，我的师兄就告诉过我，如果不经夏老师的同意而把她列为论文的作者之一，会受到严厉的批评。后来，夏老师说，她在英国爱丁堡大学的博士导师也是这样要求的。在这方面，作为夏老师的弟子，都是十分小心的，并且我们也这样要求我们自己的学生。

5．我国博士生的教育体制需要改革

一般说来，要在3年时间内完成优秀的博士学位论文是非常困难的。博士生从学习到创新心态的转换需要一个过程，创新能力的形成也需要一个过程，掌握国际研究前沿以及做出创新性的工作需要积累和实践。此外，博士生不可避免地需要做一些与研究方向关系不大的事情。因此，3年的时间是十分紧迫的。而且，不同的研究领域所需的时间也不同，博士生的个人素质也有所差异。因此，一概以3年而论是不科学的，应该对具体情况进行具体分析，在教育部门所做的规定的基础上（如基础课的有关规定），由导师确定毕业时间。同时，逐渐形成对导师的监督机制。

此外，应该克服博士生中“只要念够年头，就一定能取得学位”的心理，改革博士生的答辩体制。表面上，答辩能不能通过取决于导师，其实导师也有苦衷。普遍的问题是，博士生3年时间一到，教育部门就开始给导师施加压力，如果不

能及时毕业，那么学生以后的宿舍、助学金以及其他的一系列问题都会成为导师难以处理的问题。有的研究生教育部门甚至以限制招生为威胁。因此，导师必须在3年内让博士生毕业，否则就是导师的过错。这是引起导师把关不严的重要原因。十年树木，百年树人。培养一个人才不是轻而易举的事情。最近开始实行的硕博连读制度很受导师的欢迎，但也存在管理太死的问题。因此，我国博士生教育体制仍需要改革。

参考文献

1 毛泽东．“实践论”，《毛泽东选集》第一卷．北京：人民出版社，1991

2 胡伟武，夏培肃．“顺序一致共享存储系统中的乱序执行技术：基本理论”，《计算机学报》20(6)．北京，1997

3 胡伟武，夏培肃．“顺序一致共享存储系统中的乱序执行技术：模拟实现”，《计算机学报》20(6)．北京，1997

4 胡伟武，施巍松，唐志敏．“基于新型 Cache 一致性协议的共享虚拟存储系统”，《计算机学报》22(5)．北京，1999

5 唐志敏，施巍松，胡伟武．“曙光 1000A 上消息传递和共享存储的比较”，《计算机学报》23(2)．北京，2000

6 Adve S, Hill M. “Weak Ordering: A New Definition”, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990

7 Adve S, Hill M, Vernon M. “Comparison of Hardware and Software Cache Coherence Schemes”, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991

8 Adve S, Gharachorloo K. “Shared Memory Consistency Models: A Tutorial”, *IEEE Computer*, 1996

9 Agarwal A, Simoni R, Hennessy J, Horowitz M. “An Evaluation of Directory Schemes for Cache Coherence”, *Proceedings of the 15th International Symposium on Computer Architecture*, 1988

10 Agarwal A, Lim B, Kranz D, Kubiatowicz J. “APRIL: A Processor Architecture for Multiprocessing”, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990

11 Agarwal A, Chaiken D, Johnson K, Kranz D, Kubiatowicz J, Kurihara K, Lim B, Maa G, Nussbaum D. “The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor”, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991

12 Ahuja S, Carriero N, Gelernter D. “Linda and Friends”, *IEEE Computer*, 1986

13 Archibald J, Baer J. “Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model”, *ACM Transactions on Computer Systems*, 1986

14 Archibald J. "A Cache Coherence Approach for large Multiprocessor Systems", *Proceedings of the 1988 International Conference on Supercomputing*, 1988

15 Bailey D, Barton J, Lasinski T, Simon H. "The NAS Parallel Benchmarks", *Technical Report 103863*. NASA, 1993

16 Bal H, Bhoedjang R, Hofman R, Jacobs C, Langendoen K, Ruhl T. "Performance Evaluation of the Orca Shared-object System", *ACM Trans. on Computer Systems* 16(1), 1998

17 Bell G. "Scalable, Parallel Computers: Alternatives, Issues, and Challenges", *International Journal of Parallel Programming*, Vol. 22, No. 1, 1994

18 Bershad B, Zekauskas M, Sawdon W. "The Midway Distributed Shared Memory System", *Proceedings of the 38th IEEE International Computer Conference*, 1993

19 Bianchini R, Kontothanassis L, Pinto R, Maria M, Abud M, Amorim C. "Hiding Communication Latency and Coherence Overhead in Software DSMs", *the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996

20 Bitar P, Despain A. "Multiprocessor Cache Synchronization: Issues, Innovations, Evolution", *Proceedings of the 11th International Symposium on Computer Architecture*. New York: IEEE, 1986

21 Blount M, Butrico M. "DVSM6K: Distributed Virtual Shared Memory on the RISC System/6000", *Proceedings of the 1993 IEEE Spring COMPCON*, 1993

22 Censier L, Feautrier P. "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, Vol. C-27, No. 12, 1978

23 Carter J, Bennet J, Zwaenepoel W. "Implementation and Performance of Munin", *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991

24 Chaiken D, Field C, Kurihara K, Agarwal A. "Directory-Based Cache Coherence in Large-Scale Multiprocessors", *Computer*, Vol. 23, No. 6, 1990

25 Chaiken D, Kubiatowitz J, Agarwal A. "Limitless Directories: A Scalable Cache Coherence Scheme", *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991

26 Coller W. "Reasoning About Parallel Architectures". Englewood Cliffs, NJ: Prentice-Hall, 1992

27 Compaq, Intel, Microsoft. "Virtual Interface Architecture Specification Version 1.0", Technique Report, http://www.viarch.org, 1997

28 Cox A, Fowler R. "Adaptive Cache Coherency for Detecting Migratory Shared Data", *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993

29 Cox A, Dwarkadas S, Keleher P, Lu H, Rajamony R, Zwaenepoel W. "Software versus Hardware Shared Memory Implementation: A Case Study", *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994

30 Dubios M, Briggs F. "Effects of Cache Coherence in Multiprocessors", *IEEE Transactions on Computers*, Vol. C-31, No. 11, 1982

31 Dubios M, Scheurich C, Briggs F. "Memory Access Buffering In Multiprocessors", *Proceedings of the 13th International Symposium on Computer Architecture*, 1986

32 Dubios M, Scheurich C, Briggs F. "Synchronization, Coherence, and Event Ordering in Multiprocessors", *Computer*, Vol. 21, No. 2, 1988

33 Dwarkadas S, Hardavellas N, Kontothanassis L, Nikhil R, Stets R. "Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory", *Proceedings of the 13th International Parallel Processing Symposium*, 1999

34 Eggers S, Katz R. "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988

35 Eggers S, Katz R. "Evaluation of the Performance of Four Snooping Cache Coherence Protocols", *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989

36 Enderton H B. *Elements of Set Theory*. London: Academic Press Inc Ltd, 1977

37 Erlichson A, Nuckolls N, Chesson G, Hennessy J. "SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory", *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996

38 Gharachorloo K, Lenoski D, Laudon J, Gibbons P, Gupta A, Hennessy J. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990

39 Gharachorloo K, Gupta A, Hennessy J. "Two Techniques to Enhance the Performance of Memory Consistency Models", *Proceedings of the 1991 International Conference on Parallel Processing*, 1991

40 Gharachorloo K, Gupta A, Hennessy J. "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors", *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991

41 Gharachorloo K, Gupta A, Hennessy J. "Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors", *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992

42 Gjessing S, Gustavson D, Goodman J, James D, Kristiansen E. "The SCI Cache Coherence Protocol", *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991

43 Goodman J. "Using Cache Memory to Reduce Processor-Memory Traffic", *Proceedings of the 10th International Symposium on Computer Architecture*. New York: IEEE, 1983

44 Goodman J. "Cache Consistency and Sequential Consistency", *Technical Report No. 61*. SCI committee, 1989

45 Gupta A, Weber W, Mowry T. "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes", *Proceedings of 1990 International Conference on Parallel Processing*, Vol. 1, 1990

46 Hagersten E, Landin A, Haridi S. "Multiprocessor Consistency and Synchronization Through Transient Cache States", *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991

47 Hu W. "Correct Event Ordering in Shared-Memory Systems" *Ph. D. Thesis*. Chinese Academy of Sciences: Institute of Computing Technology, 1996. Available at www.itc.ac.cn/chpc/hww.

48 Hu W, Shi W, Tang Z, Li M. "A Lock-Based Cache Coherence Protocol for Scope Consistency", *Journal of Computer Science and Technology*, Vol. 13, No. 2, 1998

49 Hu W, Shi W, Tang Z. "A Framework of Memory Consistency Models", *Journal of Computer Science and Technology*, Vol. 13, No. 2, 1998

50 Hu W, Xia P. "Out-of-Order Execution in Sequentially Consistent Shared Memory Systems: Theory and Experiments", *Journal of Computer Science and Technology*, Vol. 13, No. 2, 1998

51 Hu W, Shi W, Tang Z. "Reducing System Overhead in Home-Based Software DSMs", *Proceedings of the 13th International Parallel Processing Symposium*, 1999

52 Hu W, Shi W, Tang Z. "Home Migration in Home-Based Software DSMs", *Proceedings of the 1st Workshop on Software Distributed Shared Memory*, 1999

53 Hu W. "Reducing Message Overheads in Home-Based Software DSMs", *Proceedings of the 1st Workshop on Software Distributed Shared Memory*, 1999

54 Hu W, Shi W, Tang Z. "JIAJIA: A Software DSM System Based on a New Cache Coherence Protocol", *Proceedings of the 1999 InternationalConference on High Performance Computing and Networking Europe*, LNCS 1593. Amsterdam, 1999

55 Hu W, Zhang F, Liu H. "A New Home-based Software DSM Protocol for SMP Clusters", *Proceedings of the Euro-Par 2000*, 2000

56 Hu W, Re L, Zhang F, Shi W, Tang Z. "Running Real Applications on Software DSM", *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region*, 2000

57 Hu W, Zhang F, Liu H. "Dynamic data prefetching in Home-based Software DSMs", *Journal of Computer Science and Technology*, Vol. 16, No. 3, 2001

58 Hu Y, Lu H, Cox A, Zwaenepoel W. "OpenMP for Networks of SMPs", *Proceedings of the 13th International Parallel Processing Symposium*, 1999

59 Iftode L, Singh J, Li K. "Scope Consistency: A Bridge between Release Consistency and Entry Consistency", *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996

60 Iftode L. "Home-based Shared Virtual Memory", *Ph. D. Thesis*. Princeton University, 1998

61 Karlin A, Manasse M, Rudolph L, Sleator D. "Competitive Snoopy Caching", *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986

62 Karlsson M, Stenstrom P. "Effectiveness of Dynamic Prefetching in Multiple Writer Distributed Virtual Shared Memory Systems", *Journal of Parallel and Distributed Computing*, Vol. 43, No. 7, 1997

63 Katz R. "Implementing a Cache Coherence Protocol", *Proceedings of the 12th International Symposium on Computer Architecture*. New York: IEEE, 1985

64 Keleher P, Cox A, Zwaenepoel W. "Lazy Release Consistency for Software Distributed Shared Memory", *Proceedings of the 19th Annual Symposium on Computer Architecture*, 1992

65 Keleher P, Dwarkadas S, Cox A, Zwaenepoel W. "TreadMarks Distributed Shared Memory on Standard Workstations and Operating Systems", *Proceedings of the 1994 Winter USENIX Conference*, 1994

66 Keleher P. "The Relative Importance of Concurrent Writers and Weak Consistency Models", *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996

67 KSR1 Technical Summary. Kendall Square Research. 1992

68 Khandekar D. "Quarks: Distributed Shared Memory as a Building Block for Complex Parallel and Distributed Systems", *Master's thesis*. Department of Computer Science, The University of Utah, 1996

69 O'Krakka B, Newton A. "An Empirical Evaluation of Two Memory-Efficient Directory Methods", *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990

70 Lamport L. "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, 1978

71 Lamport L. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs", *IEEE Transactions on Computers*, Vol. C-28, No. 9, 1979

72 Lee R, Yew P, Lawrie D. "Multiprocessor Cache Design Considerations", *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987

73 Lenoski D, Laudon J, Gharachorloo K, Gibbons P, Gupta A, Hennessy J. "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessors", *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990

74 Lenoski D, Laudon J, Joe T, Nakahira D, Stevens L, Gupta A, Hennessy J. "The DASH Prototype: Implementation and Performance", *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992

75 Li K. "IVY: A Shared Virtual Memory System for Parallel Computing", *Proceedings of the 1988 International Conference on Parallel Processing*, Vol. 2, 1988

76 Lu H, Dwarkadas S, Cox A, Zwaenepoel W. "Quantifying the Performance Differences Between PVM and TreadMarks", *Journal of Parallel and Distributed Computing*, Vol. 43, No. 2, 1997

77 Mowry T, Gupta A. "Tolerating Latency Through Software-controlled Prefetching in Shared-memory Multiprocessors", *Journal of Parallel and Distributed Computing* 12(2), 1991

78 Papamarcos M, Patel J. "A Low-Overhead Coherence Solution for Multiprocessors", *Proceedings of the 11th International Symposium on Computer Architecture*. New York: IEEE, 1984

79 Reinhardt S, Larus J, Wood D. "Tempest and Typhoon: User-level Shared Memory", *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994

80 Samanta R, Bilas A, Iftode L, Singh J. "Home-based SVM Protocols for SMP Clusters: Design and Performance", *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, 1998

81 Saulsbury A, Wilkinson T, Carter J, Landin A. "An Argument for Simple COMA", *Proceedings of the 1st IEEE Symposium on Parallel and Distributed Processing Techniques and Applications*, Vol. I, 1998

82 Scales D, Gharachorloo K, Aggarwal A. "Fine-grain Software Distributed Shared Memory on SMP Clusters", *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, 1998

83 Scheurich C, Dubois M. "Correct Memory Operation of Cached-Based Multiprocessors", *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987

84 Schoinas I, Falsafi B, Hill M, Larus J, Wood D. "Sirocco: Cost-effective Fine-grain Distributed Shared Memory", *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1998

85 Shasha D, Snir M. "Efficient and Correct Execution of Parallel Programs That Share Memory", *ACM Transactions on Programming Languages and System* 10(2), 1988

86 Shi W, Hu W, Tang Z, Eskicioglu R. "Dynamic Task Migration in Home-based Software DSM System", *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*. CA: Redondo Beach, 1999

87 Singh J, Weber W, Gupta A. "SPLASH: Stanford Parallel Applications for Shared Memory", *Computer Architecture News* 20(1), 1992

88 Speight W, Bennett J. "Brazos: A third generation DSM system", *Proceedings of the 1997 USENIX Windows/NT Workshop*, 1997

89 Stenstrom P. "A Survey of Cache Coherence Schemes for Multiprocessors", *Computer*, Vol. 23, No. 6, 1990

90 Stenstrom P, Joe T, Gupta A. "Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures", *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992

91 Stenstrom P, Brorsson M, Sandberg L. "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing", *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993

92 Stets R. "Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network", *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, 1997

93 Tang C. "Cache System Design in the Tightly Coupled Multiprocessor System", *Proceed-*

ings National Computer Conference ,1976

94 Thacker C,Stewart L,Satterthwaite E. "Firefly:A Multiprocessor Workstation", *IEEE Transaction on Computers*,Vol.37,No.8,1988

95 Thapar M,Delagi B,Flynn M. "Linked List Cache Coherence for Scalable Shared Memory Multiprocessors", *Proceedings of the 7th International Parallel Processing Symposium* ,1993

96 Tomasevic M,Milutinovic V. "A Simulation Study of Snoopy Cache Coherence Protocols", *Proceedings of the 25th Hawaii International Conference on System Sciences*,1992

97 Tomasevic M, Milutinovic V. "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors,Part 1", *IEEE Micro*,Vol.14,No.5,1994

98 Tomasevic M, Milutinovic V. "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors,Part 2", *IEEE Micro*,Vol.14,No.6,1994

99 Woo S,Ohara M,Torrie E,Singh J,Gupta A. "The SPLASH-2 Programs:Characterization and Methodological Considerations", *Proceedings of the 22th Annual Symposium on Computer Architecture*,1995

100 Zucker R,Baer J. "A Performance Study of Memory Consistency Models", *Proceedings of the 19th Annual International Symposium on Computer Architecture*,1992

{
  "filename": "5YWx5Lqr5a2Y5YKo57O757uf57uT5p6E77ya5YWo5Zu95LyY56eA5Y2a5aOr5a2m5L2N6K665paHXzExMDY2MzUyLnppcA==",
  "filename_decoded": "\u5171\u4eab\u5b58\u50a8\u7cfb\u7edf\u7ed3\u6784\uff1a\u5168\u56fd\u4f18\u79c0\u535a\u58eb\u5b66\u4f4d\u8bba\u6587_11066352.zip",
  "filesize": 12548365,
  "md5": "476977ac071b323591ea69c1e2788ab5",
  "header_md5": "e8197d73965d9c2f36b0b40e1ceac8bf",
  "sha1": "4b4d49e435091f1dff732daffc679f4aab3857ac",
  "sha256": "740b0e8f7b7eb5f7ef0a2ce15482855877e1d91c7dbf1d4d20376609107ac31e",
  "crc32": 3726107130,
  "zip_password": "",
  "uncompressed_size": 13058986,
  "pdg_dir_name": "\u2563\u2593\u2567\u03c6\u2524\u00b5\u2524\u00f3\u2567\u2561\u2550\u2502\u255c\u00df\u2563\u2563\u00fa\u2551\u255a\u00bd\u2563\u00b7\u2559\u253c\u2568\u03c0\u2593\u2310\u2569\u2510\u2564\u00ba\u256c\u2557\u252c\u2588\u256c\u2500_11066352",
  "pdg_main_pages_found": 137,
  "pdg_main_pages_max": 137,
  "total_pages": 147,
  "total_pixels": 737384928,
  "pdf_generation_missing_pages": false
}