



面向 21 世 纪 课 程 教 材
Textbook Series for 21st Century

计算机体系结构

张晨曦 王志英 张春元 戴葵 朱海滨



高等 教育 出 版 社
HIGHER EDUCATION PRESS

面向 21 世 纪 课 程 教 材
Textbook Series for 21st Century

计算机体系结构

张晨曦 王志英 张春元 戴葵 朱海滨

JS85119



高等 教 育 出 版 社
HIGHER EDUCATION PRESS

内容提要

本书是教育部“高等教育面向 21 世纪教学内容和课程体系改革计划”的研究成果，是面向 21 世纪课程教材。本书的主要内容有：计算机体系结构的基本概念、计算机指令集结构设计、流水线技术、指令级并行技术、存储体系、输入/输出系统以及多处理器。本书比较全面和系统地接触了当今计算机体系结构的发展前沿，概念清晰，易于理解，并配有大量的实例分析。

本书可作为计算机专业本科生计算机体系结构课程的教材，也可作为计算机相关专业研究生教材，书中的内容对于从事计算机研究及相关人员亦有很好的参考价值。

图书在版编目(CIP)数据

计算机体系结构 / 张晨曦等编著。—北京：高等教育出版社，2000

教育部面向 21 世纪课程教材

ISBN 7-04-007495-8

I. 计... II. 张... III. 计算机体系结构 - 高等学校
- 教材 IV. TP303

中国版本图书馆 CIP 数据核字(2000)第 17636 号

计算机体系结构

张晨曦 王志英 张春元 戴葵 朱海滨

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>

经 销 新华书店北京发行所

印 刷 中国科学院印刷厂

纸张供应 山东高塘纸业集团总公司

开 本 787×960 1/16

版 次 2000 年 6 月第 1 版

印 张 23.5

印 次 2000 年 6 月第 1 次印刷

字 数 430 000

定 价 19.90 元

凡购买高等教育出版社图书，如有缺页、倒页、脱页等
质量问题，请在所购图书销售部门联系调换。

版权所有 侵权必究

第一作者简介

张晨曦，男，1960年9月生，汉族，福建龙岩人，国防科技大学计算机学院教授，博士生导师。国家级“中青年有突出贡献专家”，中国电子学会理事，全国计算机辅助教育学会理事。主要成就有：先后主持了四项国家自然科学基金项目，曾获“国家杰出青年基金”资助。发表论文80篇，其中在国外发表论文20篇。有18篇被国际著名的八大检索工具收录。撰写专著两部（第二作者），其中《新一代计算机》由荷兰 North-Holland 出版社出版；另一部1992年获“国家教委优秀专著特等奖”，1993年获“全国优秀科技图书一等奖”。获部委级科技进步一等奖两项（排名第二），二等奖一项（排名第一）；获部委级教学成果二、三等奖各一项。负责研制的新型 Internet 动画制作 / 演播系统“网动王”于1999年3月通过了专家鉴定，达到了国际先进水平，并已在远程教育和 CAI 中获得了较广泛的应用。1991年被国家教委授予“做出突出贡献的中国博士”光荣称号，并被评为湖南省科技青年“十佳”之一；1993年被评为“全军优秀教师”，1993年和1995年两次获“霍英东青年教师奖”；1995年获“中国青年科技奖”。从15岁起当中学教师，对教学方法和现代教育技术有深入的研究，提出了面向远程教育和 CAI 的教学方法多媒体图形解析教学法。

通讯地址：湖南长沙 国防科技大学 计算机学院

邮编：410073

网址：<http://www.GotoSchool.net>

E-mail：cchang@public.cs.hn.cn

出版说明

计算机体系结构、计算机组成原理和微型计算机技术是计算机科学与技术专业的核心课程。长期以来，大家普遍感到这三门课程的教材体例和内容陈旧，彼此交叉重复过多，不能适应我国培养面向 21 世纪人才的需要，迫切希望能对它们统一规划，全盘考虑，各有侧重，避免简单重复。为此，在国家教育部高等学校计算机科学与技术教学指导委员会主任孙钟秀院士的领导之下，在 1996 年 9 月上海会议上，经过反复认真讨论最后决定由教学指导委员会副主任陈国良教授负责统一策划，并根据“面向 21 世纪计算机专业内容和课程体系改革”的要求，以“体系结构——组成原理——微机技术”系列教材的形式，组织编写此套书。

按此决议精神，经过半年多的筹备，于 1997 年 3 月在长沙邀请国内著名大学中讲授该课程的一些资深教授，并参照了国际上的同类权威教材，对该系列教材的内容划分和所属重点进行了讨论，确定了统一的编写原则，即计算机体系结构应重点论述计算机系统的各种基本结构、设计技术和性能定量分析方法；计算机组成原理应侧重讨论计算机基本部件的构成和组成方式，基本运算的操作原理和单元的设计思想、操作方式及其实现；而微型计算机技术则应突出应用，详细讲述微处理器芯片、计算机主板、接口技术和应用编程方法。

根据上述确定的原则，经过专家推荐和多方面协商，在 1997 年 10 月的济南会议上，逐一落实了系列教材的作者与审者。其中，计算机体系结构由国防科学技术大学张晨曦教授等主编，复旦大学朱传琪教授主审；计算机组成原理由哈尔滨工业大学唐朔飞教授主编，中国科学技术大学陈国良教授主审；微型计算机技术由上海交通大学孙德文教授主编，西安交通大学鲍家元教授主审。

此后，在 1997 年 11 月的三亚会议上对各书的三级提纲进行了最终统一审定，并约定计算机体系结构、计算机组成原理和微型计算机技术的书稿分别于 1999 年 3 月、8 月和 10 月提交高等教育出版社。最后，在 1998 年 9 月合肥会议上，确定将三本书作为“面向 21 世纪课程教材”向国家教育部提出立项申请，并讨论了要为该系列教材配套 CAI 软件。

此套系列教材的出版，是计算机科学与技术教学指导委员会全体同志和参与编审系列教材的同志们的共同努力、辛勤劳动的结果。我们非常感谢高等教育出版社的支持与鼓励，感谢全国广大读者对此套书的厚望。希望此套教材能为培养我国面向 21 世纪的科技人才发挥应有的作用。

国家教育部高等学校计算机科学与技术教学指导委员会
1999 年 8 月 13 日

前　　言

本书是高等学校计算机专业本科生及研究生计算机体系结构课程的通用教材。为了适应面向 21 世纪计算机类专业教学内容和课程体系改革的需要,国家教育部高等学校计算机科学与技术教学指导委员会统一组织编写了计算机科学与技术专业九五规划教材。其中,《计算机体系结构》、《计算机组成原理》和《微型计算机技术》是重点组织的系列教材。本书是该系列教材之一。

在内容的选择上,本书不打算覆盖计算机体系结构的各个方面,不想成为大而全的参考手册,而是重点论述现代大多数计算机都采用的比较成熟的思想、结构和方法等,并且力求做到深入浅出、通俗易懂。

本书除了着重论述计算机体系结构的基本概念、基本原理、基本结构和基本分析方法以外,还强调采用量化的分析方法。这种方法使我们能更具体、实际地分析和设计计算机体系结构。书中用了大量的例题说明如何进行量化分析。

本书共七章。第一章论述计算机体系结构的概念以及体系结构和并行性概念的发展,并简单地讨论影响计算机系统设计的成本和价格因素。第二章论述计算机指令集结构设计中的一些问题,包括寻址技术、指令集的功能设计、操作数的类型和大小、指令格式等,并且介绍一种指令集结构的实例——DLX。第三章为流水线技术,论述流水线的基本概念、分类以及性能计算方法,并对流水线中的相关问题以及向量计算机进行讨论。第四章为指令级并行,论述利用硬、软件技术开发程序中存在的指令间并行性的技术和方法,包括指令调度、超标量技术、分支处理技术和超长指令字技术。第五章为存储层次,论述 Cache 的基本知识、降低 Cache 失效率的方法、减少 Cache 失效开销的方法以及减少命中时间的方法,并对主存和虚拟存储器进行讨论。第六章为输入输出系统,论述存储设备、总线和通道,并讨论 I/O 与操作系统的关系以及 I/O 系统设计。第七章为多处理机,论述多处理机的存储器体系结构、互连网络以及同步与通信,并对并行化技术和多处理机实例进行讨论。

本书由国防科学技术大学计算机学院张晨曦教授编写第五章,王志英教授编写第七章,张春元博士编写第一章和第四章,戴葵博士编写第二章和第三章,朱海滨博士编写第六章。

本书由复旦大学朱传琪教授主审,提出了宝贵的意见。在本书编写过程中,得到了中国科学技术大学陈国良教授的大力支持,并得到了国防科学技术

大学计算机学院领导和有关师生的多方面帮助。在此一并表示衷心的感谢。

由于作者水平有限，书中难免有错误和不妥之处，敬请读者批评指正。

本书有配套的 CAI 软件和 PowerPoint 讲稿，详见

<http://www.GotoSchool.net>

作　者

1999年3月于长沙

责任编辑 刘 艳

封面设计 张 楠

责任校对 俞声佳

版式设计 马静如

责任印制 宋克学

第一章 计算机体系结构的基本概念

1.1 引论

我们目前使用的数字电子计算机(通常称之为计算机)诞生于 1945 年。半个世纪以来,计算机技术一直处于发展和变革之中。今天,我们用 1 万元人民币购置的个人计算机(PC),其性能已经大大超过了在 20 世纪 60 年代用 1 百万美元购置的计算机。20 世纪 50 年代,人们认为在银行里用计算机来完成现金存取业务的想法是荒唐可笑的,因为当时最便宜的计算机也需要 50 万美元;60 年代时,人们认为用计算机控制汽车行驶是天方夜谭,因为当时计算机的体积就有轿车那么大;70 年代的人们做梦也不会想到计算机系统能随身携带,可带到咖啡屋、汽车或飞机上使用。人们曾经假设,如果汽车制造业能够按计算机产业的速度发展,那么今天人们从美国的东海岸到西海岸,只需花 0.5 美元,用时 5 秒。计算机性能如此高速增长,受益于电路技术和计算机体系结构技术的发展。

在计算机诞生的头 25 年中,计算机性能增长相对缓慢。在这个过程中,电路技术和体系结构同时发挥着作用,其间充满了尝试和创新。目前广泛使用的存储程序计算机的完整概念就是这个时期产生的,我们通常称之为冯·诺依曼(von Neumann)计算机结构。20 世纪 70 年代以后,由于集成电路的出现,计算机性能出现了极大的飞跃,产生了一系列著名的计算机系统,如 IBM 360/370 系列、DEC PDP 系列、CDC 6600/7600 系列等。70 年代计算机性能的增长速度达到每年 25%~30%,这种增长主要归功于以集成电路为代表的电路技术的发展。70 年代末期,出现了微处理器,这是一次计算机设计和制造技术的革命。从 70 年代末到 80 年代中期,采用微处理器的计算机性能增长达到每年 35%;与当时广泛使用的大、中、小型计算机相比,微处理器的性能增长,更多地依赖于集成电路技术的发展。进入 80 年代以后,计算机体系结构产生了一次重大变革,出现了我们现在称之为精简指令集计算机(Reduced Instruction Set Computer,简称 RISC)的处理器设计技术。此后,计算机体系结构不断变革,在计算机体系结构技术发展的促进下,集成电路技术为计算机设计提供的技术空间得到了充分的发挥,计算机系统的性能以每年 50% 以上的速度增长。90 年代中期以来,尤其是 Intel 公司的 Pentium Pro 的出现,基于 RISC 设计技术的微处理器大

批量上市,极大地推动了计算机产业的发展,计算机的系统性能得到全面提升。目前,计算机性能增长达到每年 50% 以上,其中包括器件技术在内的计算机制造技术提供其中约 8%,其余约 42% 的部分主要依靠计算机体系结构发展的支持。

微处理器出现以后,计算机系统设计、计算机市场和计算机应用都出现了较大的变化。首先,计算机用户是最直接的受益者。20世纪 90 年代中期,采用 Digital 的 Alpha 微处理器构成的计算机系统,其性能已经大大超过 80 年代末期的向量巨型计算机,如 Cray Y-MP(可能是 88 年全世界最快的商用计算机)等,而价格仅仅只有巨型计算机的几十分之一。第二,对于市场而言,大批量的微处理器生产促成了计算机产品的批量化、标准化和市场化,这种变化促进了计算机设计、生产和应用的良性发展。微处理器的使用,使绝大多数计算机系统制造商无需再进行中央处理器的设计和制造了,计算机系统设计的复杂性和风险也大大下降。目前,甚至巨型计算机也采用微处理器来实现。第三,大量兼容的微处理器、标准化接口、高度兼容的计算机系统的出现,避免了系统程序和应用程序的重复开发;操作系统和计算机语言的标准化,降低了采用新型体系结构的费用和风险;高级语言,例如 C/C++ 语言,成为计算机系统的必备语言,汇编语言的使用减少使应用开发的难度和风险都大大减少了。

计算机系统的这种变化对现实世界的影响是巨大的。当我们使用 80386 档次微处理器构成系统时,进行字处理大部分采用字符界面,多媒体和所见即所得的概念刚刚诞生,功能十分有限。一台具有基本配置的 80386 计算机系统价格在 2 万元人民币以上,只有极少数人能够购置计算机。32 位的 80386 微处理器,其性能相当于一台百万次计算机。我们目前广泛使用的基于 Pentium II 微处理器计算机系统,其性能相当于一台数千万次的 32 位计算机。多媒体功能已经十分普遍,字符界面几乎淘汰,家庭拥有计算机已经司空见惯,计算机已经可以为我们提供大量非常有用的功能,同时这些功能的使用也相对容易了,这些变化已经使我们的生活出现了更多的所谓数字化的成分。我们可以预计,在计算机系统设计和计算机应用领域中,基于微处理器的系统将在相当长的时间内占支配和统治地位。这种现状对计算机体系结构的研究影响巨大,本书中所研究的很多概念,都是基于这个出发点的。

1.2 计算机体系结构的概念

设计一种新型计算机系统首先必须面临的问题是什么呢?我们会列举出很多因素,其中最主要的有新型计算机的主要特点和性能。它们具体包括:指令集设计、功能组织、逻辑设计、实现技术等。实现技术包括集成电路设计、制造和封

装技术、系统制造、供电、冷却技术等。另外，我们往往要求在限定的造价范围内，使新型计算机具有最高的性能。如何采用先进的计算机体系结构和生产技术，制造出具有高性能价格比(performance cost rate)的计算机系统，是所有通用计算机设计的共同目标。

1.2.1 计算机系统中的层次概念

现代计算机系统是由软件和硬件/固件组成的十分复杂的系统。为了对这个系统进行描述、分析、设计和使用，人们从不同的角度提出了观察计算机的观点和方法。本节从计算机语言的角度，把计算机系统按功能划分成多级层次结构。

随着计算机系统的发展，已设计出一系列语言。从面向机器的语言(如机器语言、汇编语言)，到各种高级程序设计语言(如 Java、C/C++、FORTRAN、Pascal)，到各种面向问题的语言或者叫应用语言(如面向数据库查询的 SQL 语言，面向数字系统设计的 VHDL 语言、面向人工智能的 PROLOG 语言)。计算机语言就是这样由低级向高级发展，高一级语言的语句相对于低级语言功能更强，更便于应用，但又都是以低级语言为基础的。

计算机语言可分成一系列的层次(level)或级，最低层语言的功能最简单、最高层语言的功能最强。对于用某一层语言编写程序的程序员来说，他一般不管其程序在机器中是如何执行的，只要程序正确，他就能得到预期的结果。这样，对这层语言的程序员来说，他似乎有了一种新的机器，这层语言就是这种机器的机器语言，该机器能执行用该层语言编写的全部程序。因此计算机系统就可以按语言的功能划分成多级层次结构，每一层以一种不同的语言为特征。这样，可以把现代计算机系统画成如图 1.1 所示的层次结构。图中第 4 级以上完全由软件实现。我们称由软件实现的机器为虚拟机器(virtual machine)，以区别于由硬件或固件实现的实际机器。

第 1 级是微程序机器级，这级的机器语言是微指令集，程序员用微指令编写的微程序一般是直接由硬件解释实现的。

第 2 级是传统机器级。这级的机器语言是该机的指令集，程序员用机器指令集编写的程序可以由微程序进行解释。这个解释程序运行在第 1 级上。由微程序解释指令集又称作仿真(emulation)。实际上，第 1 级可以有一个或数个能够在它上面运行的解释程序，每一个解释程序都定义了一种指令集。因此，可以通过仿真在一台机器上实现多种指令集。

计算机系统中也可以没有微程序机器级，在这些计算机系统中是用硬件直接实现传统机器的指令集，而不必由任何解释程序进行干预。我们目前使用的 RISC 技术就是采用这样的设计思想，处理器的指令集全部用硬件直接实现以提高指令的执行速度。

第3级是操作系统虚拟机。从操作系统的基本功能来看，一方面它要直接管理传统机器中的软硬件资源，另一方面它又是传统机器的引申。它提供了传统机器所没有的某些基本操作和数据结构，如文件结构与文件管理的基本操作、存储体系和多道程序以及多重处理所用的某些操作、设备管理等。

第4级是汇编语言虚拟机。这级的机器语言是汇编语言，用汇编语言编写的程序，首先翻译成第3级和第2级语言，然后再由相应的机器执行。完成汇编语言翻译的程序就称作汇编程序。

第4级以上出现了一个重要变化。通常的第1、2和3级是用解释(interpretation)方法实现的，而第4级或更高级则经常是用翻译(translation)方法实现。

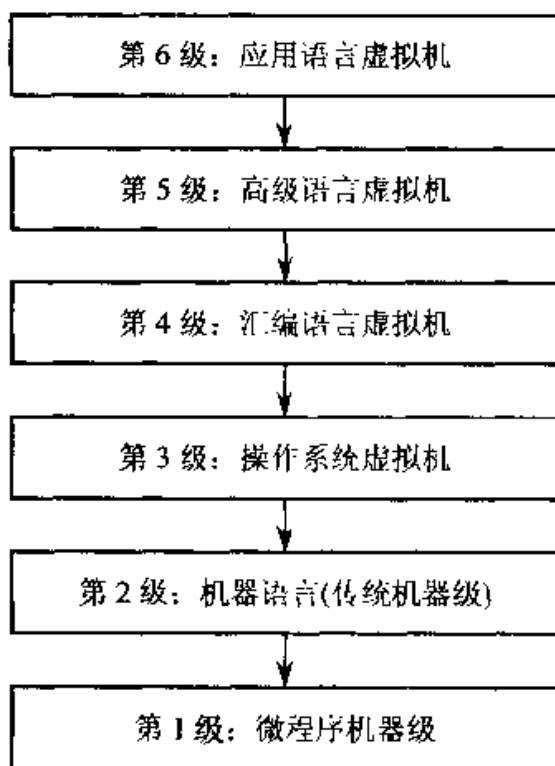


图 1.1 计算机系统的多级层次结构

翻译和解释是语言实现的两种基本技术。它们都是以执行一串 N 级指令来实现 $N+1$ 级指令，但二者仍存在着差别：翻译技术是先把 $N+1$ 级程序全部转换成 N 级程序后，再去执行新产生的 N 级程序，在执行过程中 $N+1$ 级程序不再被访问。而解释技术是每当一条 $N+1$ 级指令被译码后，就直接去执行一串等效的 N 级指令，然后再去取下一条 $N+1$ 级的指令，依此重复进行。在这个过程中不产生翻译出来的程序，因此解释过程是边变换边执行的过程。在实现新的虚拟机器时，这两种技术都被广泛使用。一般来说，解释执行比翻译花的时间多，但存储空间占用较少。

第5级是高级语言虚拟机。这级的机器语言就是各种高级语言，目前高级

语言已达数百种。用这些语言所编写的程序一般是由称为编译程序的翻译程序翻译到第4级或第3级上,如C/C++、Pascal、FORTRAN等,个别的高级语言也用解释的方法实现,如绝大多数BASIC语言系统。

第6级是应用语言虚拟机。这一级是为使计算机满足某种用途而专门设计的,因此这一级语言就是各种面向问题的应用语言。可以设计专门用于人工智能、教育、行政管理和计算机设计等方面的应用程序包,这些虚拟机也是当代计算机应用领域的重要研究课题。应用语言编写的程序一般是由应用程序包翻译到第5级上。

1.2.2 计算机体系结构

计算机体系结构(computer architecture)这个词目前已被广泛使用。Architecture本来用在建筑方面,译为“建筑学”、“建筑术”、“建筑样式”、“构造”、“结构”等。这个词被引入计算机领域后,最初的译法也各有不同,以后趋向译为“体系结构”,但关于它的定义仍未统一。

经典的“计算机体系结构”定义是1964年C. M. Amdahl在介绍IBM 360系统时提出的:计算机体系结构是程序员所看到的计算机的属性,即概念性结构与功能特性。

按照计算机系统的多级层次结构,不同级程序员所看到的计算机具有不同的属性。例如,传统机器程序员所看到的主要属性是该机指令集的功能特性,而高级语言虚拟机程序员所看到的主要属性是该机所配置的高级语言所具有的功能特性。显然,不同的计算机系统,从传统机器级程序员或汇编语言程序员的角度来看,是具有不同属性的。但是,从高级语言(如Pascal)程序员看,它们就几乎没有差别,是具有相同属性的。或者说,这些传统机器级所存在的差别是高级语言程序员所“看不见”的,也是不需要他们知道的。在计算机技术中,对这种本来是存在的事物或属性,但从某种角度看又好像不存在的概念称为透明性(transparency)。通常,在一个计算机系统中,低层机器的属性对高层机器的程序员往往是透明的,如传统机器级的概念性结构和功能特性,对高级语言程序员来说是透明的。由此看出,在层次结构的各个级上都有它的体系结构。Amdahl提出的体系结构是指传统机器级的体系结构,即一般所说的机器语言程序员所看到的传统机器级所具有的属性。

这些属性是机器语言程序设计者(或者编译程序生成系统)为使其所设计(或生成)的程序能在机器上正确运行,所需遵循的计算机属性,包含其概念性结构和功能特性两个方面。目前,对于通用寄存器型机器来说,这些属性主要是指:

- (1) 数据表示(硬件能直接辨认和处理的数据类型);

- (2) 寻址规则(包括最小寻址单元、寻址方式及其表示);
- (3) 寄存器定义(包括各种寄存器的定义、数量和使用方式);
- (4) 指令集(包括机器指令的操作类型和格式、指令间的排序和控制机构等);
- (5) 中断系统(中断的类型和中断响应硬件的功能等);
- (6) 机器工作状态的定义和切换(如管态和目态等);
- (7) 存储系统(主存容量、程序员可用的最大存储容量等);
- (8) 信息保护(包括信息保护方式和硬件对信息保护的支持);
- (9) I/O 结构(包括 I/O 联结方式、处理机/存储器与 I/O 设备间数据传送的方式和格式以及 I/O 操作的状态等)。

这些属性是计算机系统中由硬件或固件完成的功能,程序员在了解这些属性后才能编出在传统机器上正确运行的程序。因此,经典计算机体系结构概念的实质是计算机系统中软硬件界面的确定,界面之上是软件的功能,界面之下是硬件和固件的功能。

这里比较全面地讨论了经典的计算机体系结构概念。随着计算机技术的发展,计算机体系结构所包含的内容是不断变化和发展的。目前经常使用的是广义的计算机体系结构概念,它既包括经典的计算机体系结构的概念范畴,还包括了对计算机组成和计算机实现技术的研究。

1.2.3 计算机组成和计算机实现技术

计算机组成是计算机体系结构的逻辑实现,而计算机实现是计算机组成的物理实现。它们各自包含不同的内容,但又有紧密的关系。

我们以系列机(family machine)为例说明这些概念之间的关系。系列机的出现被认为是计算机发展史上的一个重要里程碑。直到现在,各计算机厂家仍按系列机的思想发展自己的计算机产品。现代计算机不但系统系列化,其构成部件也系列化,如 CPU、硬盘等。至今对计算机领域影响最大也是产量最大的系列计算机莫过于 IBM PC 及其兼容系列机和 Intel 的 80x86 系列微处理器。

所谓系列机,就是指在一个厂家内生产的具有相同的体系结构,但具有不同组成和实现的一系列不同型号的机器。如 IBM 370 系列有 370/115、125、135、145、158、168 等一系列从低速到高速的各种型号。它们各有不同的性能和价格,采用不同的组成和实现技术,但从程序设计者所看到的机器属性却是相同的。在中央处理机中,它们都执行相同的指令集,但在低档机上可以采用指令分

析和指令执行顺序进行的方式,而在高档机上则采用重叠、流水和其他并行处理方式等。

下面从处理器、处理器字宽、I/O 总线、地址空间、寻址方式和计算机结构等方面看一看 IBM PC 系列机。表 1.1 给出了处理器、处理器字宽、I/O 总线、地址空间和寻址方式的比较,图 1.2 为 PC 系列机典型结构的比较。ISA(Industry Standard Architecture)总线是 AT 总线标准化以后的名称。

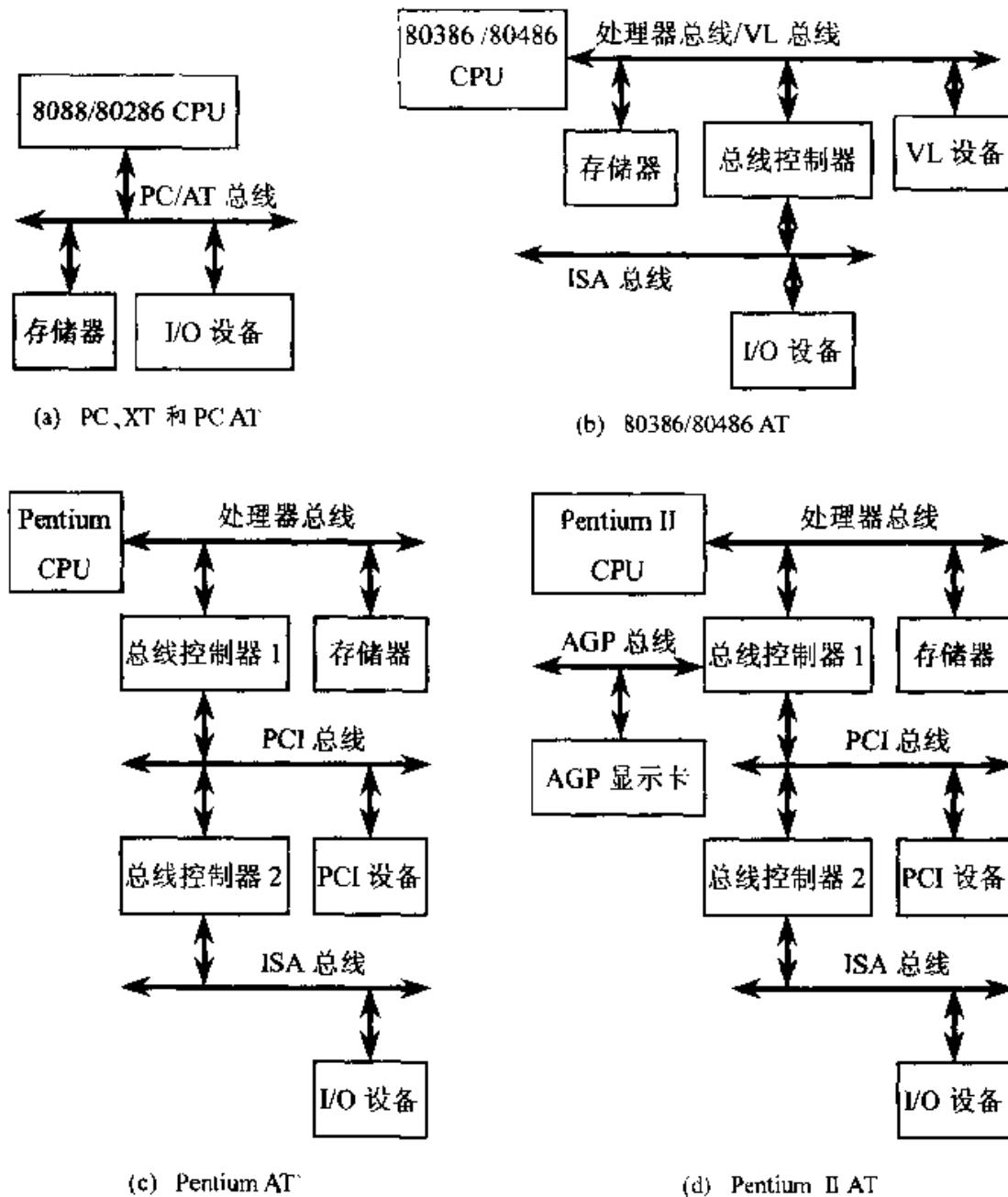


图 1.2 PC 系列机典型结构的发展和比较

表 1.1 PC 系列机特性比较

计算机型号	PC 和 XT	PC AT	80386 AT	80486 AT	Pentium AT	Pentium II AT
处理器型号	8088	80286	80386	80486	Pentium	Pentium II
处理器上市 时间	1979	1982	1985	1989	1993	1997
处理器字宽	16 位	16 位	32 位	32 位	32 位	32 位
I/O 总线	PC 总线	AT(ISA) EISA	ISA/ EISA	ISA + VL	ISA + PCI	ISA + PCI + AGP
地址空间	20 位	20/24 位	20/32 位	20/32 位	20/32 位	20/32(36)位
寻址方式	实地址	实/虚 地址	实/虚 地址	实/虚 地址	实/虚 地址	实/虚地址

从上述系列机的例子可见：一种体系结构可以有多种组成，同样，一种组成可以有多种物理实现。正因为系列机从程序设计者的角度看都具有相同的机器属性，因此按这个属性（体系结构）编制的机器语言程序及编译程序都能通用于各档机器，我们称这种情况下的各档机器是软件兼容的（software compatibility），即同一个软件可以不加修改地运行于体系结构相同的各档机器上，而且它们所获得的结果一样，差别只在于运行时间不同。长期以来，软件工作者希望有一个稳定的环境，使他们编制出来的程序能得到广泛的应用，机器设计者又希望根据硬件技术和器件技术的进展不断地推出新的机器，而系列机的出现较好地解决了软件要求环境稳定和硬件、器件技术迅速发展之间的矛盾，对计算机的发展起到了重要的推动作用。有些计算机厂家为了能利用大计算机厂家的开发成果，也研制一些软件兼容的计算机产品。我们把不同厂家生产的具有相同体系结构的计算机称为兼容机（compatible machine）。兼容机一方面由于采用新的计算机组成和实现技术，因而具有较高的性能价格比；另一方面又可能对原有的体系结构进行某种扩充，从而具有更强的功能（如长城 0520 为 IBM PC 兼容机，但有较强的汉字处理功能）。因此，在市场上有较强的竞争能力。

早在 20 世纪 60 年代，以 Amdahl 公司为代表的接插兼容机 PCM（Plug-Compatible Mainframe）厂家，专门生产在功能上和电气性能上与 IBM 公司相同的主机和设备，它不但可以运行 IBM 公司的软件，而且又可以作为 IBM 产品的替换件插入 IBM 系统。由于采用了新的硬件和器件技术，提高了性能价格比，因而成为 IBM 公司强有力的竞争对手。这种竞争有力地推动了计算机技术和应用的发展。随着标准化技术的发展，目前这种技术已经成为推动计算机系统发展的重要技术之一，有一大批厂家专门生产符合各种标准的设备，这些设备可

以在拥有标准接口的各种计算机上使用,这些厂商称为独立设备生产厂商或者第三方生产厂商。

系列机为了保证软件的兼容,要求体系结构不改变,这无疑又妨碍了计算机体系结构的发展。实际上,系列机的软件兼容还有向上兼容、向下兼容、向前兼容和向后兼容之分。所谓向上(下)兼容指的是按某档机器编制的程序,不加修改就能运行于比它高(低)档的机器。所谓向前(后)兼容指的是按某个时期投入市场的某种型号机器编制的程序,不加修改就能运行于在它之前(后)投入市场的机器。图 1.3 形象地说明了这些概念。

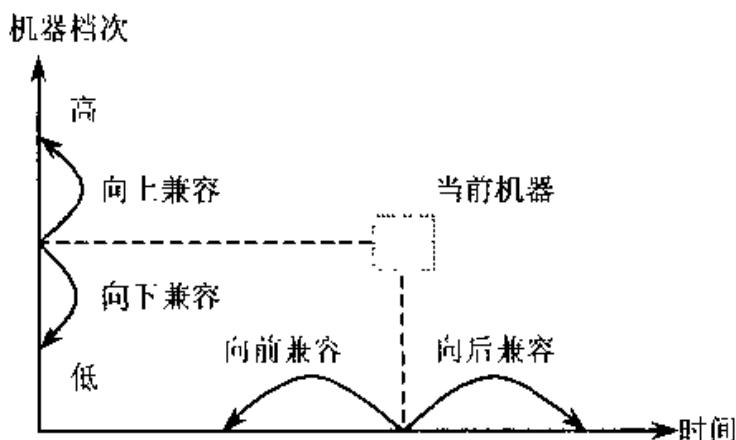


图 1.3 兼容性示意图

为了适应系列机中性能不断提高和应用领域不断扩大的要求,后续各档机器的体系结构也是可以改变的。如增加浮点运算指令以提高速度,或者增加事务处理指令以满足事务处理方面的需要等。但这种改变必须是原有体系结构的扩充,而不是任意地更改或缩小。这样,对系列机的软件向下和向前兼容可以不作要求,向上兼容在某种情况下可能做不到(如在低档机器上增加了面向事务处理的指令),但向后兼容却是肯定要做的。因此,可以说向后兼容是软件兼容的根本特征,也是系列机的根本特征。一个系列机的体系结构设计的好坏,是否有生命力,就看是否能在保证向后兼容的前提下,不断地改进其组成和实现。Intel 公司的 80x86 系列微处理器在向后兼容方面是非常具有代表性的,从 1979 年的 8086 到 1999 年的 Pentium III,增加了保护方式指令集、MMX 指令集和 KNI 指令集,但它保持了极好的二进制代码级的向后兼容性。向后兼容虽然削弱了系列机对体系结构发展的约束,但仍然是体系结构发展的一个沉重包袱,这也是 RISC 微处理器在性能上很快超过传统的 CISC 微处理器的主要原因之一。

1.3 计算机体系结构的发展

计算机体系结构研究计算机系统中软、硬件的界面,即研究哪些功能由软件

完成,哪些功能由硬件完成。实际上,软件和硬件在逻辑功能上是等效的,就是说由软件实现的功能在原理上可以由硬/固件实现。同样,由硬件实现的功能原理上也可以通过软件模拟来实现。但是,软件和硬件在性能上是不等效的。因此,对于计算机系统软硬件功能的分配应保证在满足应用的前提下,充分利用硬件和器件技术的发展,使系统达到较高的性能价格比。

四十多年来,计算机体系结构在存储程序计算机体系结构的范畴内取得了很大进展。近年来,由于技术和应用需求的发展,人们开始探索并实现各种非存储程序的计算机体系结构。

1.3.1 存储程序计算机体系结构及其发展

最早的存储程序计算机是由著名数学家冯·诺依曼等人在 1946 年总结并明确提出来的,因此经常称之为冯·诺依曼结构计算机,其结构如图 1.4 所示。

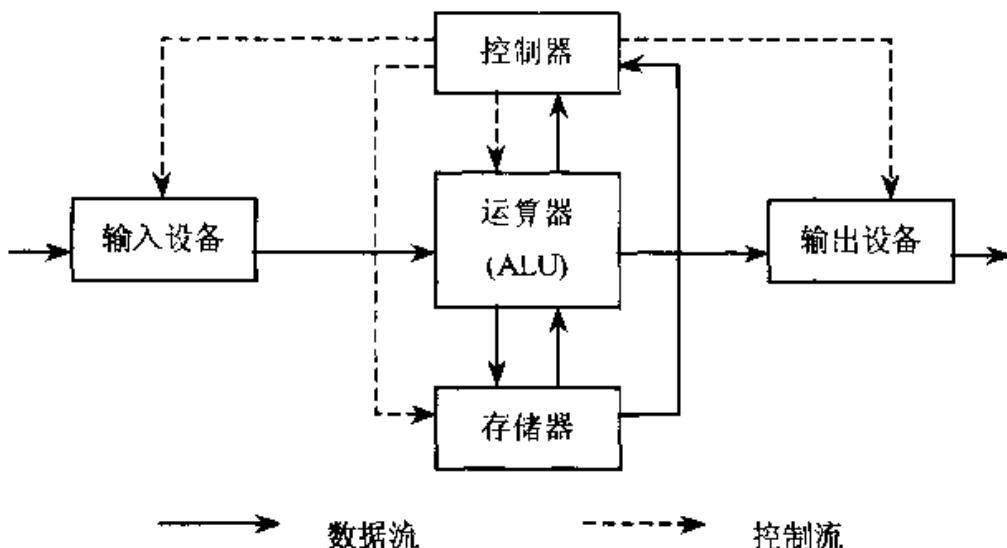


图 1.4 存储程序机器的结构

存储程序计算机在体系结构上的主要特点如下：

- (1) 机器以运算器为中心。输入/输出设备与存储器之间的数据传送都经过运算器;存储器、输入/输出设备的操作以及它们之间的联系都由控制器集中控制。
- (2) 采用存储程序原理。程序(指令)和数据放在同一存储器中,并且没有对两者加以区分。指令和数据一样可以送到运算器进行运算,即由指令组成的程序自身是可以修改的。
- (3) 存储器是按地址访问的、线性编址的空间。每个单元的位数是相同且固定的,称为存储器编址单位。
- (4) 控制流由指令流产生。指令在存储器中按其执行顺序存储,由指令计

数器指明每条指令所在单元的地址。一般情况下每执行完一条指令,程序计数器顺序递增。虽然执行顺序可以根据运算结果改变,但是解题算法仍然是也只能是顺序型的。

(5) 指令由操作码和地址码组成。操作码指明本指令的操作类型,地址码指明操作数和操作结果的地址。操作数的类型(定点、浮点或十进制数)由操作码决定,操作数本身不能判定其数据类型。

(6) 数据以二进制编码表示,采用二进制运算。它主要面向数值计算和数据处理。

存储程序计算机体系结构的这些特点奠定了现代计算机发展的基础。但是,在冯·诺依曼等人提出这种结构时,由于硬件价格昂贵,故使硬件完成的功能尽量简单,而把更多的功能交由软件完成。随着计算机应用领域的扩大,高级语言和操作系统的出现,这种功能分配的状况引起了愈来愈多的矛盾,迫使人们不断地对这种体系结构进行改进。下面结合上述六个特点进行简要地分析。

1. 分布的 I/O 处理能力

存储程序计算机以运算器为中心、所有部件的操作都由控制器集中控制,这一特点带来了慢速输入/输出操作占用快速运算器的问题。此时的输入/输出操作和运算操作只能串行,互相等待,大大影响了运算器效率的发挥。尤其是计算机的应用进入商业领域之后,数据处理要求频繁地进行输入/输出操作,上述缺点显得更加突出。

为了克服这一缺点,人们提出各种输入/输出方式,如图 1.5 所示,从而使计算机概念性结构发生了相应的变化。

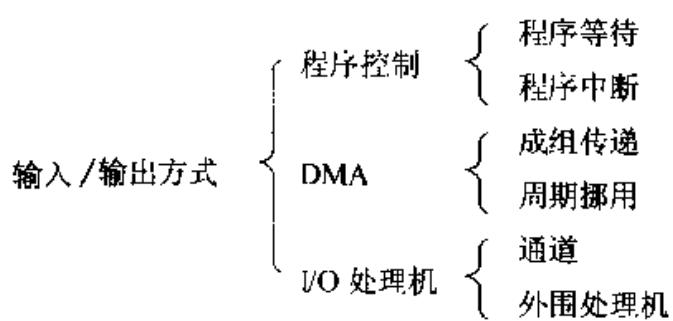


图 1.5 各种输入/输出方式

在程序等待方式之后,很快出现了程序中断的概念,并被应用于输入/输出操作(如 1954 年的 Univac 1103 机)。这时,CPU 执行到一条输入/输出操作指令后,可以不必等待外部设备回答的状态信息而继续执行后续指令,直到外部设备准备好发送/接收数据,向 CPU 发出中断请求,CPU 响应中断请求后才进行输入/输出。这时,计算机结构仍以运算器为中心,但增加了中断接口部件。程序中断概念的引入可以使 CPU 与外部设备在一定程度上并行工作,提高了计算

机的效率，并且可以实现多种外部设备同时工作。中断技术已经成为现代计算机操作系统的技术基础。

随着外部设备种类和数量的增多，中断变得过于频繁，从而浪费了大量 CPU 的时间。于是，对于成块(组)地进行传送的输入/输出信息，出现了所谓 DMA(直接存储器访问)方式。为了实现这种方式，需要在主存和设备之间增加 DMA 控制器(数据通道)，从而形成了以主存为中心的结构。CPU 向 DMA 控制器寄存器置好初始参数后，仍可继续执行其后续指令。当外设准备好发送或接收数据时，便对 CPU 发出 DMA 请求，使输入/输出设备经数据通道直接与主存成组地交换信息。数据通道控制完成这一块数据传送后，向 CPU 发一结束信号，CPU 就可以进行接收或发送数据后的处理工作。

采用 DMA 方式，每传送完一组数据就要中断 CPU 一次。如果该部件能自己控制完成输入/输出的大部分工作，则可使 CPU 进一步摆脱用于管理、控制 I/O 系统的沉重负担，这就出现了 I/O 处理机方式。I/O 处理机几乎把控制输入/输出操作和信息传送的所有功能都从 CPU 那里接管过来。I/O 处理方式有通道方式和外围处理机(I/O 子系统)方式两种。最早采用通道方式的是 IBM 360/370 系统。

2. 保护的存储器空间

虽然传统存储程序计算机的存储程序原理现在仍为大多数计算机所采用，但是否把指令和数据放在同一存储器中，不同计算机却有不同的考虑。将指令和数据存放在同一个存储器中，会带来以下一些好处：指令在执行过程中可以被修改，因而可以编写出灵活的可动态修改的程序；对于存取指令和数据仅需一套读/写和寻址电路，硬件简单；不必预先区分指令和数据，存储管理软件容易实现；程序和数据可以分配于任何可用空间，从而能更有效地利用存储空间等。然而，应看到：自我修改程序是难以编制、调试和使用的；一旦程序出错，进行程序诊断也不容易；程序的修改不利于实现程序的可再入性(reenterability)和程序的递归调用。在开发指令级并行时还可以看到：程序可修改也不利于重叠和流水方式的操作。因此，现在绝大多数计算机都规定：在执行过程中不准修改程序。这是通过存储管理硬件的支持，由操作系统软件实现的，目前几乎所有支持多任务的计算机操作系统都实现了这种管理。

3. 存储器组织结构的发展

按地址访问的存储器具有结构简单、价格便宜、存取速度快等优点。但是在数据处理时，往往要求查找具有某种内容特点的信息。在随机存储器中，虽然可以通过软件，按一定的算法(顺序查找、对分查找或采用 Hash 技术等)完成查找操作，但会由于访问存储器的次数较多而影响计算机系统的性能。按内容访问的相联存储器 CAM(Content Addressed Memory)，则把查找、比较的操作交由存

储器硬件完成。如果让相联存储器除了完成信息检索任务外,还能进行一些算术逻辑运算,则就构成了以相联存储器为核心的相联处理机。

为了减少程序运行过程中访问存储器的次数,人们早在 1956 年的 Pegasus 计算机上就采用了通用寄存器的概念。它不但把变址寄存器和累加器结合起来使用,提供了多累加器结构,而且使中间结果不必访问存储器。这个概念为现代计算机普遍采用,通用寄存器的数量也由几个增加到十几个或几十个,有些 RISC 处理器中增加到几百个。为了进一步减少访问存储器的次数和提高存储系统的速度,人们又提出了在 CPU 和主存之间设置高速缓冲存储器 Cache。这种技术当初是在大型机上采用的,而现在 PC 机也使用了多至 2 MB 的 Cache 存储器。

4. 并行处理技术

传统的存储程序计算机解题算法是顺序型的,即使问题本身可以并行处理,由于程序的执行受程序计数器控制,故只能是串行、顺序地执行。因此,如何挖掘传统机器中的并行性,一直是计算机设计者努力的方向。人们通过改进 CPU 的组成,如采用重叠方式、先行控制、多操作部件甚至流水方式把若干条指令的操作重叠起来;或者在体系结构上使本来可以并行计算的题目能并行计算,如向量计算就可以用一条向量指令并行地对向量的各个元素进行相同的运算。更进一步,如果把一个作业(程序)划分成能并行执行的多个任务(程序段),把每个任务分配给一个处理机执行,则构成了多机并行处理系统。这种多机并行处理系统由于有很高的并行性,因而成为提高计算机系统速度最有潜力的手段之一。随着微处理机的价格迅速下跌,现在已出现了由 2^{16} 个微处理机芯片组成的多处理机系统。

5. 指令集的发展

指令集是传统机器程序员所看到机器的主要属性。指令由操作码和地址码两部分组成,它在两个方面对计算机体系结构设计产生重大影响:一是指令集的功能,二是指令的地址空间和寻址方式。

20 世纪 70 到 80 年代的计算机系统指令的种类愈来愈多,所含指令的数目可达 300~500 条,这种计算机称为复杂指令集计算机 CISC(Complex Instruction Set Computer)。人们原以为庞大的指令集计算机可以提高计算机系统的性能,孰不知日趋庞杂的指令集不但不容易实现,而且还可能降低计算机系统的性能。因此,在 1979 年由 D. A. Patterson 等人提出了精简指令集的设想。他们把指令集设计成只包含那些使用频率高的少量指令,并提供一些必要的指令以支持操作系统和高级语言。按照这个原则而构成的计算机称为精简指令集计算机 RISC。RISC 的技术思想已经成为当代计算机设计的基础技术之一。

早期计算机地址码是直接指出操作数地址的(直接寻址),这是一种简单快

速的寻址方法。随着计算机存储器容量的扩大,指令中地址码的位数已不能满足整个主存空间寻址的要求,使直接寻址方式受到很大限制。现代计算机指令地址码部件一般给出的是形式地址,而由各种寻址方式,按照规定的算法把形式地址转换成访问主存的有效地址,如变址寻址(1949年EDSAC)、间接寻址(1958年IBM 709)、相对寻址(如PDP-11)。由于多道程序的要求,出现了基址寻址(如IBM 360),为了对虚拟存储器进行管理,出现了页式寻址(1959年Atlas)。现代的微、小型计算机由于指令字长较短,一般都具有多种灵活的寻址方式。多种寻址方式在带来灵活性的同时也带来了设计、生产和使用的复杂性,所以采用RISC技术的计算机一般只有常用的几种寻址方式。

1.3.2 计算机的分代和分型

一般认为计算机到目前为止已经发展了五代。这五代计算机分别具有明显的器件、体系结构技术和软件技术特征。表1.2列出了其中的典型特征。

表1.2 五代计算机的特征

第一代 (1945~1954年)	电子管和继电器	存储程序计算机、程序控制I/O	机器语言和汇编语言	普林斯顿ISA、ENIAC、IBM 701
第二代 (1955~1964年)	晶体管、磁芯、印刷电路	浮点数据表示、寻址技术、中断、I/O处理机	高级语言和编译、批处理监控系统	Univac LARC、CDC 1604、IBM 7030
第三代 (1965~1974年)	SSI和MSI、多层印刷电路、微程序	流水线、Cache、先行处理、系列计算机	多道程序和分时操作系统	IBM 360/370、CDC 6600/7600、DEC PDP-8
第四代 (1974~1990年)	LSI和VLSI、半导体存储器	向量处理、分布式存储器	并行与分布处理	Cray-1、IBM 3090、DEC VAX 9000、Convax-1
第五代 (1991年~)	高性能微处理器、高密度电路	超标量、超流水、SMP、MP、MPP	大规模、可扩展并行与分布处理	SGI Cray T3E、IBM SP2、DEC AlphaServer 8400

计算机可以根据价格分为五个档次,包括巨型机(supercomputer)、大型机(mainframe)、中型机、小型机(minicomputer)和微型机(microcomputer)。

概括地分析四十多年来计算机体系结构的进展,可以发现,计算机体系结构的成就不只是表现在巨、大型机上,而且在中、小、微型计算机中也愈来愈多地被采用,同时还包括技术和性能的“下移”。图1.6为计算机系统性能随时间“下

“移”的示意图。

从上述体系结构进展的情况可以看出,新型体系结构的设计一方面是合理地增加计算机系统中硬件的功能比例,使这种体系结构对操作系统、高级语言甚至应用软件提供更多更好的支持;另一方面则是通过多种途径提高计算机体系结构中的并行性等级,使得凡是能并行计算和处理的问题都能并行计算和处理,使这种体系结构和组成对算法提供更多更好的支持。但要记住,无论哪种措施都要在满足应用要求的前提下,使系统具有合理的性能价格比。

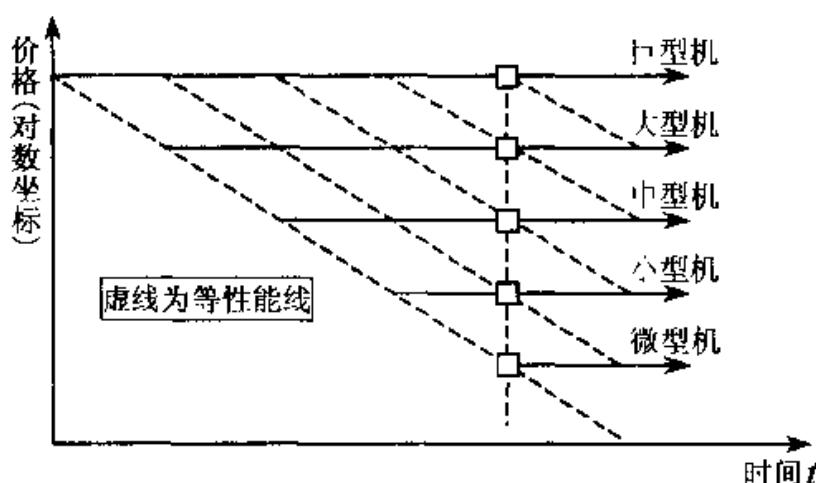


图 1.6 计算机性能下移示意图

1.3.3 应用需求的发展

一种成功的指令集(instruction set, 又称为指令系统), 其结构必须能够适应硬件技术、软件技术及应用特性的变化。设计者尤其要注意计算机使用方法及实现技术的发展趋势, 这样一个新型指令集结构才可能会有数十年的使用寿命, 如 IBM AS/400 计算机的核心技术从 1964 年至今一直在使用。因此为延长采用某种体系结构的计算机使用寿命, 该体系结构必须能够适应技术的变化。

计算机的设计受两方面因素的影响:一方面是计算机现在和未来的使用方法,另一方面是下层的实现技术。软件技术最重要的发展趋势之一,就是程序及数据所使用存储器容量的不断增大。程序所需的存储器容量平均每年递增 1.5%~2%,也就是说计算机的地址位以每年 1/2~1 位的速度递增。如此高的增长速度主要由两方面原因造成:一方面是程序的需要,另一方面是 DRAM 技术的发展。DRAM 技术能够不断降低存储器的位成本。对地址空间增长速度估计不足,是一种指令集结构被淘汰的最主要的原因。

在过去 20 年里,软件技术的另一重要发展趋势是标准的高级语言广泛使用,在应用领域取代了汇编语言,这种变化使编译器更加重要。只有编译器与计

计算机系统紧密联系,才能够生产出更具竞争力的机器。编译器已成为用户与计算机的主要界面。

除了作为用户界面,编译技术正在逐渐增加新的功能,不断提高程序在机器上运行的效率。编译技术的改进既包括传统的优化技术,也包括为完善流水线及存储系统而进行的改进。如何分配编译器与硬件所负担的工作,使处理器能够高效运行,一直是 20 世纪 90 年代以来体系结构领域中最热门的技术话题之一。

1.3.4 计算机实现技术的发展

为了计算机今后的发展,设计者尤其要注意实现技术日新月异的变化,其中有三种实现技术的变化发展极快,而这些技术对于当代计算机的发展非常关键。

(1) 逻辑电路。综合起来看,单芯片上的晶体管数量以每年 60%~80% 的速率增长。1998 年以前,集成电路制造中的金属导线技术没有改进,机器时钟频率的增长完全依赖于导线宽度的减少和工作电压的降低。1998 年 IBM 成功地开发了铜导线技术,使线宽 0.2 μm 以下的半导体器件性能有了很大的变化,这种技术可以将成品微处理器的工作频率提高到 1 GHz 以上。

(2) 半导体 DRAM(动态随机访问存储器)。芯片密度每年增长速率略低于 60%,平均三年增长 4 倍,存储周期时间的减少比较缓慢,大约是每 10 年减少 1/3,DRAM 通过接口的变化改善带宽。目前,DRAM 的设计和制造技术领先于逻辑电路,这主要是因为 DRAM 元件内单元电路中晶体管数量少,并且 DRAM 采用专用技术产生。

(3) 磁盘。最近磁盘密度以每年约 50% 的速度增长,几乎每三年增长 4 倍。1990 年以前磁盘密度每年增长 25%,平均三年增长 2 倍,这表明未来几年内磁盘技术的发展将使磁盘密度继续保持高增长率。磁盘存取时间在过去 10 年内下降 1/3。

这些快速变化发展的技术对微处理器和计算机系统的设计具有很大影响。由于技术和工艺的不断提高,现代微处理器一般只有五年左右的使用寿命,甚至在一种产品的生产周期中(两年设计,两年生产),如 DRAM,其核心技术也在不断变化。设计人员必须经常采用最新技术进行计算机设计,因为最新技术的产品具有最高性价比和性能优势。计算机产品的技术和性能的变化是骤变,而非渐变。比如说,DRAM 技术是三年提高四倍,而不是 18 个月增长 2 倍。这种技术的跃变导致设计技术的跃变,从而使得原先不可能完成的技术变得可以实现。比如说当 MOS 技术在 20 世纪 80 年代初达到单芯片集成 25 000~50 000 个晶体管时,单片 32 位微处理器便实现了,在工艺成熟之后,其性价比飞速提高。这种情况在计算机发展史上十分常见。

1.3.5 体系结构的生命周期

对于一个设计者来说,在进行计算机体系结构设计的时候,必须清楚他现在的位置,或者说这种新的体系结构是否顺应计算机发展的潮流?是超前了,落后了,还是适得其所?这个问题,几乎没有十全十美的答案。从计算机的发展过程中可以看出,任何一种计算机体系结构都是有其生命周期的,从诞生、发展、成熟到消亡。我们关心的是:一种计算机体系结构的生命周期是怎样的?

计算机必须构成系统才能够工作。计算机系统中包括硬件、系统软件和应用软件,计算机的生命周期和系统中各个部分的发展密切相关。一种新的体系结构的诞生,往往以硬件为标志。它的发展和成熟,是以配套的系统软件和应用为标志的。一旦这种体系结构不能够满足应用的需要,这种体系结构就会消亡,新的体系结构也就必然诞生了。一般情况下,从硬件成熟到系统软件(如操作系统)成熟大约需要5~7年的时间,从系统软件成熟到应用软件成熟,大约也需要5~7年时间,再过5~7年的时间,这种体系结构就不会作为主流体系结构而存在了,新的体系结构将替代它。一个计算机体系结构,从产生到消亡,大致需要15~20年的时间,如图1.7所示。其实新旧体系结构在时间上总是有重叠的。

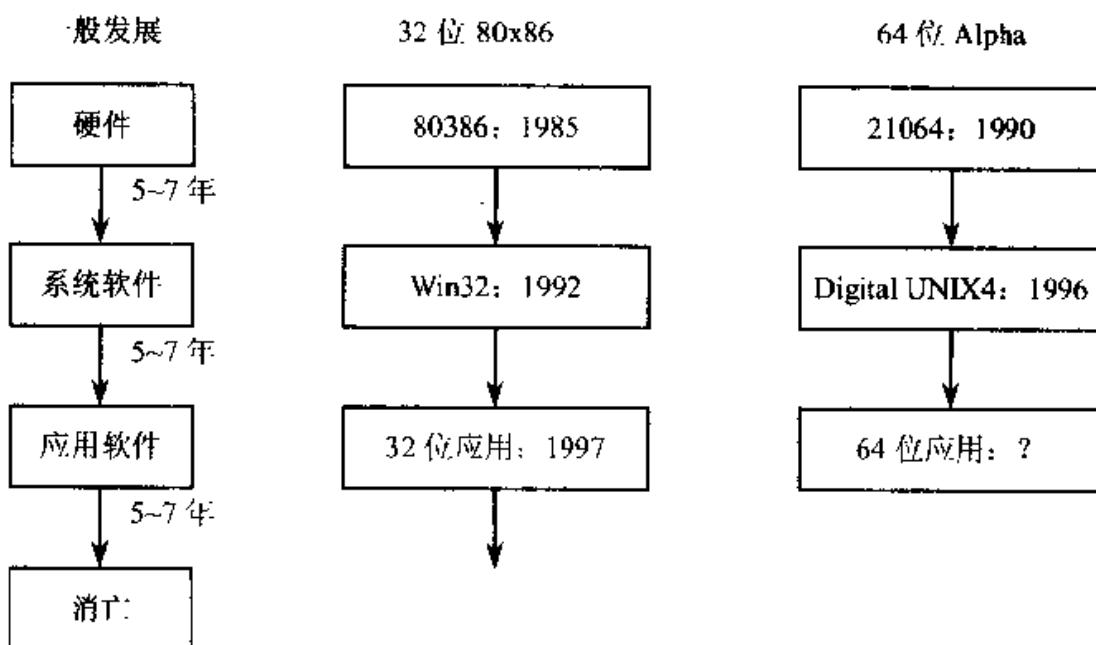


图1.7 体系结构的生命周期

在图1.7中,可以看到两个计算机体系结构发展的实例,一个是Intel的80x86系列微处理器中32位体系结构的发展,另一个是DEC公司(Digital Equipment Company,1997年初被Compaq收购)的64位Alpha体系结构。

对于80x86系列微处理器来说,它的第一个32位微处理器80386投入市场

时，并没有相应的 32 位操作系统，我们仅仅将它当成一个快速的 8086 使用。这种情况一直到 1992 年出现 32 位的 Windows 3.1，就是通常所说的 Win32，32 位的 80x86 处理器才有了一个广泛使用的 32 位体系结构的操作系统和开发环境，而所谓的 32 位应用软件一直到 1997 年才广泛上市。从目前的情况分析，即将进入主流的 64 位系统的生命周期应该比较长，它提供了相对非常大的可寻址空间。

1.4 计算机体系结构中并行性的发展

研究计算机体系结构的目的是提高计算机系统的性能。开发计算机系统的并行性，是计算机体系结构的重要研究内容之一。本节首先叙述体系结构中的并行性概念，然后从单机系统和多机系统两个方面对并行性的发展进行归纳，以期认识计算机体系结构中并行性发展的全貌。

1.4.1 并行性概念

所谓并行性(parallelism)是指在同一时刻或是同一时间间隔内完成两种或两种以上性质相同或不相同的工作。只要时间上互相重叠，就存在并行性。严格来讲，把两个或多个事件在同一时刻发生的并行性叫做同时性(simultaneity)；而把两个或多个事件在同一时间间隔内发生的并行性叫做并发性(concurrency)。以 n 位并行加法为例，由于存在着进位信号的传播延迟时间，全部 n 位加法结果并不是在同一时刻获得的，因此并不存在同时性，而只存在并发性的关系。如果有 m 个存储器模块能同时进行读出信息，则属于同时性。以后，除非特殊说明，本书不严格区分是哪种并行性。

计算机系统中的并行性有不同的等级。从执行程序的角度看，并行性等级从低到高可分为：

- (1) 指令内部并行：指令内部的微操作之间的并行。
- (2) 指令级并行(Instruction Level Parallel, ILP)：并行执行两条或多条指令。
- (3) 任务级或过程级并行：并行执行两个或多个过程或任务(程序段)。
- (4) 作业或程序级并行：在多个作业或程序间的并行。

在单处理机系统中，这种并行性升到某一级别后(如任务级或作业级并行)，则要通过软件(如操作系统中的进程管理、作业管理)来实现。而在多处理机系统中，由于已有了完成各个任务或作业的处理机，其并行性是由硬件实现的。因此，实现并行性也有一个软硬件功能分配问题，往往也需要折衷考虑。

从处理数据的角度，并行性等级从低到高可以分为：

(1) 字串位串:同时只对一个字的一位进行处理。这是最基本的串行处理方式,不存在并行性。

(2) 字串位并:同时对一个字的全部位进行处理,不同字之间是串行的。这里已开始出现并行性。

(3) 字并位串:同时对许多字的同一位(称位片)进行处理。这种方式有较高的并行性。

(4) 全并行:同时对许多字的全部或部分位进行处理。这是最高一级的并行。

在一个计算机系统中,可以采取多种并行性措施。既可以有执行程序方面的并行性,又可以有处理数据方面的并行性。当并行性提高到一定级别时则称之为进入并行处理领域。例如,执行程序的并行性达到任务或过程级,或者处理数据的并行性达到字并位串一级,即可认为进入并行处理领域。所以,并行处理(parallel processing)是信息处理的一种有效形式。它着重挖掘计算过程中的并行事件,使并行性达到较高的级别。并行处理是硬件、体系结构、软件、算法、语言等多方面综合研究的领域。

1.4.2 提高并行性的技术途径

计算机系统中提高并行性的措施多种多样,但就其基本思想而言,都可纳入下列三种途径:

(1) 时间重叠(time-interleaving):在并行性概念中引入时间因素,即多个处理过程在时间上相互错开,轮流重叠地使用同一套硬件设备的各个部分,以加快硬件周转而赢得速度。时间重叠原则上不要求重复的硬件设备。

(2) 资源重复(resource-replication):在并行性概念中引入空间因素,是根据“以数量取胜”的原则,通过重复设置资源,尤其是硬件资源,大幅度提高计算机系统的性能。随着硬件价格的降低,这种方式在单处理机中被广泛采用,而多处理机本身就是资源重复的结果。

(3) 资源共享(resource-sharing):这是一种软件方法,它使多个任务按一定时间顺序轮流使用同一套硬件设备。例如多道程序、分时系统就是遵循资源共享这一途径产生的。资源共享既降低了成本,又提高了计算机设备的利用率。

我们先看一看单机系统中并行性的发展。在发展高性能单处理机过程中,起着主导作用的是时间重叠这个途径。实现时间重叠的基础是部件功能专用化(functional specialization)。就是说把一件工作按功能分割为若干相互联系的部分,把每一部分指定给专门的部件完成;然后按时间重叠原则把各部分执行过程在时间上重叠起来,使所有部件依次分工完成一组同样的工作。例如对于解释指令的五个过程,就分别需要五个专用的部件,即取指令部件(IF)、指令译码部件(ID)、指令执行部件(EX)、访问存储器部件(M)和写结果部件(WB)。把它们

的工作按某种时间重叠关系重叠起来,使得在处理机内部能同时处理多条指令,从而提高处理机的速度,参见图 1.8。这些处理技术开发了计算机系统中的指令级并行。

在单处理机中,资源重复的运用已经普遍起来。不论是在非流水线处理机(如 CDC-6600)中,还是在流水线处理机(如 IBM 360/91)中,多操作部件和多体存储器都是成功应用的结构形式。在多操作部件处理机中,通用部件被分解成专门部件(如加/减法部件、乘法部件、逻辑运算部件等)。一条指令所需的操作部件只要没有被占用,就可以开始执行,这就是指令级并行。进一步,可以重复设置多个相同的处理单元,在同一个控制器指挥下,按照同一条指令的要求对向量的各元素同时进行操作,这就是所谓的并行处理机。从指令和数据处理的角度看,它是一条指令处理多个数据。它在指令内部实现了对数据处理的全并行,从而把单处理机的并行性又提高一步,进入了并行处理领域。并行处理机本身还是单处理机。再提高其并行性,使其达到数据集级并行,多个处理单元同时处理一组数据,这就是阵列处理机。如果进一步提高其并行性,使其达到任务级并行,则每个处理单元都必须有自己的控制器,能独立地解释指令而成为独立处理机,这就进入多处理机范畴,也就是同时有多条指令处理多个数据。这种多处理机系统称为对称型(symmetrical)或同构型多处理机系统(homogeneous multiprocessor system)。它们由多个同类型,至少同等功能的处理机组成,同时处理同一作业中能并行执行的多个任务。

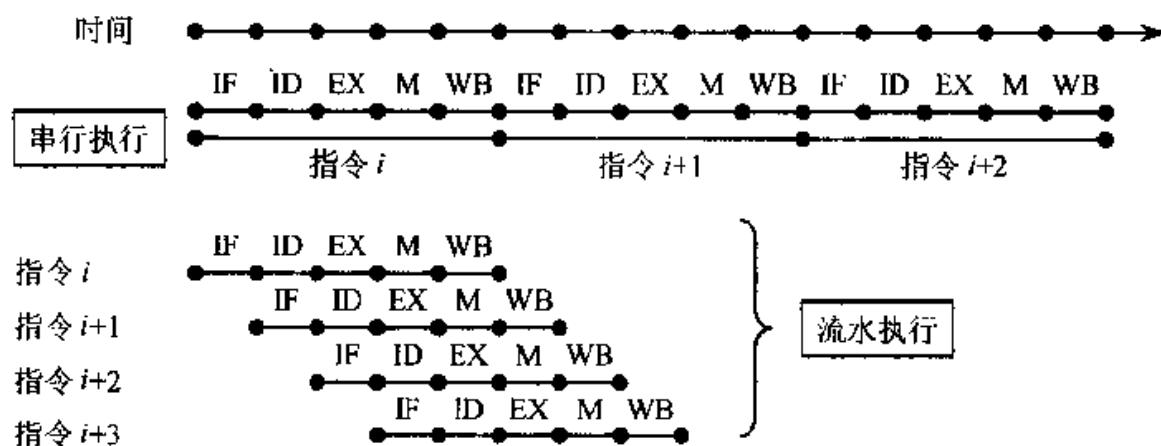


图 1.8 指令的串行执行和流水执行

资源共享的概念,在单处理机系统中实质上是用单处理机模拟多处理机的功能,形成所谓虚拟机的概念。比如分时系统,在多终端情况下,每个终端上的用户感到好像自己有一台处理机一样。远程终端的出现,改变了计算机系统地理上和逻辑上“集中”的局面,开始向“分布”方向发展。当计算机之间互相连接,分工合作时,就进入了多机系统,这种多机系统称为分布处理系统(distributed

processing system)。

下面我们看一看多机系统中并行性的发展。多机系统也遵循着时间重叠、资源重复和资源共享的技术途径,向着三种不同的多处理机方向发展。但在采取的技术措施上与单机系统稍有些差别。

为了反映多机系统各机器之间物理连接的紧密程度和交互作用能力的强弱,我们引进耦合度的概念。多机系统的耦合度,可以分为最低耦合、松散耦合和紧密耦合系统等几类。

(1) 最低耦合(least coupled system)是耦合度最低的系统。除通过某种中间存储介质之外,各计算机之间没有物理连接,也无共享的联机硬件资源。

(2) 松散耦合(loosely coupled system)或间接耦合系统(indirectly coupled system),一般是通过通道或通信线路实现计算机间互连,共享某些外围设备(例如磁盘、磁带等),机间的相互作用是在文件或数据集一级进行。松散耦合系统常表现为两种形式:一种是多台计算机和共享的外围设备连接,不同机器之间实现功能上的分工(功能专用化),机器处理的结果以文件或数据集的形式送到共享的外围设备,供其他机器继续处理。另一种是计算机网,通过通信线路连接,以求得更大范围内的资源共享。

(3) 紧密耦合系统(tightly coupled system)或直接耦合系统(directly coupled system),一般是指机间物理连接的频带较高,它们往往通过总线或高速开关实现互连,可以共享主存。由于具有较高的信息传输率,从而为快速并行处理一个作业或多个任务创造了条件。

在单机系统中,要做到时间重叠必须有多个专用功能部件,即把某些功能分离开由专门部件去完成。而在多处理机中则是将处理功能分散给各专用处理机去完成,即功能专用化。各处理机之间按照时间重叠原理工作。早期是把一些辅助性功能由主机分离出来,交给一些较小的专用计算机去完成。如输入/输出功能的分离,导致由通道向专用外围处理机发展。它们之间往往采取松散耦合方式,形成各种松散耦合系统。这种趋势的发展,使许多主要功能,如数组运算、高级语言编译、数据库管理等,也逐渐分离出来,交由专用处理机完成,机间的耦合程度也逐渐加强,发展成异构型多处理机系统。

最早的多机系统并不是为了提高速度,而是为了在关键性的工作中保证系统的可靠性。通过设置多台相同类型的计算机,使系统工作的可靠性在处理机一级得到提高。各种不同的容错多处理机系统方案对机间互连网络的要求是不同的,但正确性和可靠性是最起码的要求。如果提高对互连网络的要求,使其具有一定的灵活性、可靠性和可重构性,则可将其发展成一种可重构系统(reconfigurable system)。在这种系统中,平时几台计算机都正常工作,像通常的多处理机系统一样。但到故障阶段,就要使系统重新组织,降低档次继续运行,直到

故障排除为止。随着硬件价格的降低,现在人们更多的是通过多处理机的并行处理来提高整个系统的速度。这时,对机间互连网络的性能提出了更高要求。高带宽、低延迟、低开销的机间互连网络,是高效实现程序段或任务一级并行处理的前提条件。为了使并行处理的任务能在处理机之间随机地进行调度,就必须使各个处理机具有同等的功能,成为同构型多处理机。

表 1.3 对上述三种多处理机进行了简单的比较和总结。由表 1.3 可以看出,分布式系统与其他两类多处理机系统在概念上存在着交叉。无论是单机系统还是多机系统,都是按不同的技术途径向三种不同类型的多处理机发展。

表 1.3 三种类型多处理机比较

项目	同构型多处理机	异构型多处理机	分布处理系统
目的	提高系统性能 (可靠性、速度)	提高系统使用效率	兼顾效率与性能
技术途径	资源重复 (机间互连)	时间重叠 (功能专用化)	资源共享 (网络化)
组成	同类型 (同等功能)	不同类型 (不同功能)	不限制
分工方式	任务分布	功能分布	硬件、软件、数据 等各种资源分布
工作方式	一个作业由多机 协同并行地完成	一个作业由多机 协同串行地完成	一个作业由一台处 理机完成,必要时 才请求它机协作
控制形式	常采用浮动控制方式	采用专用控制方式	分布控制方式
耦合度	紧密耦合	紧密、松散耦合	松散、紧密耦合
对互连网络的要求	快速性、灵活性、 可重构性	专用性	快速、灵活、 简单、通用

1.5 定量分析技术基础

我们强调计算机系统的性价比,强调计算机的性能设计,那么性能是如何衡量的呢?本节对其中的一些基本概念和原则进行讨论,在本书的其他章节中将具体讨论各相关部件的性能。

1.5.1 计算机性能的评测

怎样评测一台计算机的性能,与测试者所处的角度有关。计算机用户说机器很快,往往是因为程序运行时间少;而计算中心管理员说机器很快,则往往是因为在一段时间里它能够完成更多的任务。用户关心的是响应时间,即从事件开始到结束之间的时间,也称为执行时间;而管理员关心的是如何提高流量(throughput),即在单位时间内所能完成的工作量。

为了比较不同设计的差别,通常要对两台机器的性能进行比较。假设这两台计算机为 X 和 Y,“X 比 Y 快”的意思是:对于给定任务,X 的响应时间比 Y 少。通常“X 比 Y 快 n 倍”是指

$$\frac{\text{响应时间}_Y}{\text{响应时间}_X} = n$$

由于响应时间与性能成反比,所以上式就变成

$$n = \frac{\text{响应时间}_Y}{\text{响应时间}_X} = \frac{\frac{1}{\text{性能}_Y}}{\frac{1}{\text{性能}_X}} = \frac{\text{性能}_X}{\text{性能}_Y}$$

无论是流量还是响应时间,都是以时间来度量的。它们的相同点是都认为能够以最短时间完成指定任务的计算机就是最快的;不同点是响应时间针对单任务,而流量针对多任务。目前公认的相对可靠的性能评价方法,是使用真实程序的响应时间来衡量。实际上,如果不以响应时间为衡量标准或没有使用真实程序,则容易导致错误的性能评价结论,对计算机设计产生误导甚至引发错误。

响应时间有多种定义。响应时间最直观的定义是计算机完成某一任务所花费的全部时间,包括访问磁盘、访问存储器、输入/输出、操作系统开销等。仔细分析一下就会发现,在多任务系统中,CPU 在一个程序等待 I/O 的同时可以处理另一个程序,从而提高系统的运行效率。在讨论性能时必须把这一点考虑进去。“CPU 时间”的定义就体现了这一点,它表示 CPU 工作的时间,不包含 I/O 等待时间及运行其他程序的时间。很明显,用户看到的响应时间是程序完成任务所花费的全部时间,而不是 CPU 时间。CPU 时间还可细分为用户 CPU 时间和系统 CPU 时间,前者表示用户程序所花费的 CPU 时间,后者表示用户程序运行期间操作系统花费的 CPU 时间。

上面明确了基于响应时间的性能度量方法和基于 CPU 时间的性能度量方法。这里认为“系统性能”对应于响应时间,而“CPU 性能”对应于用户 CPU 时间。下面主要讨论 CPU 性能。

1.5.2 测试程序

既然性能与测试程序的执行时间相关,那么用什么程序作测试呢?如果用户仅仅使用计算机完成某种特定的应用,那么这组应用程序就是评估计算机系统性能的最佳测试程序。用户通过比较在不同系统中这组应用程序的响应时间,就可以知道计算机的性能。然而,这种情况实际上比较少见。大部分人必须依靠其他测试程序以获得机器的性能。目前常用的测试程序可以分为四类,下面按测试可靠性由高至低顺序列出:

(1) 真实程序:这是最可靠的方法。即使用户对计算机性能测试一窍不通,通过运行真实程序,用户也可以清楚地知道计算机的性能。真实程序是随应用而变化的,如运行文本处理软件 Tex(UNIX)、MS Word(Windows)、WPS(DOS)等,运行 CAD 设计的工具 Spice、AutoCAD 等。用户自己的应用系统也是属于这一类程序,如 MIS 系统、工业控制系统等。

(2) 核心程序:它由从真实程序中提取的较短但很关键的代码构成。Livermore Loops 及 LINPACK 是其中使用比较广泛的例子。这些代码的执行时间直接影响到程序总的响应时间。用户不会直接使用核心程序,因为它的功能仅仅是用来测试计算机性能。核心程序可以根据需要评价机器的各种性能,从而解释在运行真实程序时机器性能不同的原因。

(3) 小测试程序:小测试程序代码一般在 100 行以内。用户可以随时编写一些这样的程序来测试系统的各种功能,并产生用户已预知的输出结果,如皇后问题、迷宫问题、快速排序、求素数等,这类流行的测试程序都具有短小、易输入、通用等特点,最适于作一些基本测试。

(4) 合成测试程序:首先对大量的应用程序中的操作进行统计,得到各种操作比例,再按这个比例人为制造出测试程序。Whetstone 与 Dhrystone 是最流行的合成测试程序。在操作类型和操作数类型两个方面,合成测试程序试图保持与大量程序中的比例一致。用户不会自己产生合成测试程序,因为其中没有任何用户能够使用的代码。合成测试程序与实际应用相差更远,核心程序起码是从真实程序中提取出来的,而合成测试程序则完全是人为制造出来的。

要在竞争激烈的计算机市场中生存和发展,就必须努力提高计算机产品的性价比。所以,每家计算机系统设计公司都投入大量的人力、物力资源研究各种通用的测试程序,并针对测试结果,从硬件和软件两个方面对系统设计进行修改和优化,以提高他们的计算机系统的总体测试性能。但是,通用计算机系统一般不针对某一个特定的真实程序进行设计或性能优化,因为这样做不但难度大,而且成本会增加。为了提高测试的公正性,通用测试程序往往由非商业性组织或者第三方厂商提供。通用测试程序在使用时也有明确的要求,如系统配置、数据

精度、编译优化等,以便获得的测试结果具有良好的可比性。

目前有一种日渐普及的测试程序产生方法,就是选择一组各个方面有代表性的测试程序,组成一个通用测试程序集合。这种测试程序集合称为测试程序组件(benchmark suites),它的最大优点是避免了独立测试程序存在的片面性,尽可能全面地测试了一个计算机系统的性能。目前最常见的测试程序组件有SPEC95*(基于UNIX操作系统,包括SPECint*95和SPECfp*95两部分,分别测试系统的整数和浮点性能)和WinBench'98(基于Windows操作系统,从几十种常见的真实程序中截取部分完整的功能,如MS Word中的字符替换、FoxPro中的数据检索、Lotus中的公式计算等)。测试程序组件产生的测试结果比较全面,对计算机系统设计有比较大的指导意义。本书中的一些设计分析就是在SPEC测试结果的基础上进行的。

1.5.3 性能设计和评测的基本原则

我们已知道如何定义、度量和比较计算机系统的性能,下面讨论计算机体系结构设计和分析中最常使用的三条基本原则和方法。

1. 大概率事件优先原则

大概率事件优先原则是计算机体系结构设计中最重要和最常用的原则。这个原则的基本思想是:对于大概率事件(最常见的事件),赋予它优先的处理权和资源使用权,以获得全局的最优结果。

在进行计算机设计时,如果需要权衡,就必须侧重常见事件,使最常发生事件(大概率事件)优先。此原则也适用于资源分配。着重改进大概率事件性能,能够明显提高计算机性能。另外,大概率事件通常比小概率事件简单,而且容易使之更快完成。例如,CPU在进行加法运算时,运算结果无溢出为大概率事件,而溢出为小概率事件。因此应该针对无溢出情况进行CPU优化设计,加快无溢出时加法计算速度。虽然发生溢出时机器速度可能会减慢,但由于溢出事件发生概率很小,所以总体上机器性能还是提高了。

本书中将经常看到该原则的应用。重要的是要能够确定什么是大概率事件,同时要说明针对该事件进行的改进将如何提高机器的性能。

2. Amdahl定律

Amdahl定律既可以用来确定系统中对性能限制最大的部件,也可以用来计算通过改进某些部件所获得的系统性能的提高。Amdahl定律指出:加快某部件执行速度所获得的系统性能加速比,受限于该部件在系统中所占的重要性。

首先,Amdahl定律定义了加速比这个概念。假设对机器进行某种改进,那么机器系统的加速比就是

$$\text{系统加速比} = \frac{\text{系统性能}_{\text{改进后}}}{\text{系统性能}_{\text{改进前}}}$$

或者

$$\text{系统加速比} = \frac{\text{总执行时间}_{\text{改进前}}}{\text{总执行时间}_{\text{改进后}}}$$

系统加速比告诉我们改进后的机器比改进前快多少。Amdahl 定律使我们能够快速得出改进所获得的效益。系统加速比依赖于两个因素：

(1) 可改进部分在原系统计算时间中所占的比例。例如，一个需运行 60 s 的程序中有 20 s 的运算可以加速，那么该比例就是 20/60。这个值用“可改进比例”表示，它总是小于或等于 1 的。

(2) 可改进部分改进以后的性能提高。例如，系统改进后执行程序，其中可改进部分花费的时间为 2 s，而改进前该部分需花费的时间为 5 s，则性能提高为 5/2。用“部件加速比”表示性能提高比，一般情况下它是大于 1 的。

部件改进后，系统的总执行时间等于不可改进部分的执行时间加上可改进部分改进后的执行时间，即

$$\begin{aligned}\text{总执行时间}_{\text{改进后}} &= (1 - \text{可改进比例}) \times \text{总执行时间}_{\text{改进前}} \\ &\quad + \frac{\text{可改进比例} \times \text{总执行时间}_{\text{改进前}}}{\text{部件加速比}} \\ &= [(1 - \text{可改进比例}) + \frac{\text{可改进比例}}{\text{部件加速比}}] \times \text{总执行时间}_{\text{改进前}}\end{aligned}$$

系统加速比为改进前与改进后总执行时间之比，即

$$\text{系统加速比} = \frac{\text{总执行时间}_{\text{改进前}}}{\text{总执行时间}_{\text{改进后}}} = \frac{1}{(1 - \text{可改进比例}) + \frac{\text{可改进比例}}{\text{部件加速比}}}$$

实际上，Amdahl 定律还表达了一种性能增加的递减规则：如果仅仅对计算机中的一部分做性能改进，则改进越多，系统获得的效果越小。Amdahl 定律的一个重要推论是：如果只针对整个任务的一部分进行优化，那么所获得的加速比不大于 $1/(1 - \text{可改进比例})$ 。

从另外一个侧面看，Amdahl 定律告诉我们如何衡量一个“好”的计算机系统：具有高性价比的计算机系统是一个带宽平衡的系统，而不是看它使用的某些部件的性能。

3. 程序的局部性原理

程序的局部性原理是指：程序在执行时所访问地址的分布不是随机的，而是相对地簇聚；这种簇聚包括指令和数据两部分。程序局部性包括程序的时间局部性和程序的空间局部性。程序的时间局部性是指程序即将用到的信息很可能就是目前正在使用的信息。程序的空间局部性是指程序即将用到的信息很可能

与目前正在使用的信息在空间上相邻或者临近。

程序的局部性原理是计算机体系结构设计的基础之一。在很多地方，尤其是在处理那些与存储相关的问题时，经常要使用这个原理。

1.5.4 CPU 的性能

为了衡量 CPU 的性能，可以将程序执行的时间进行分解。首先，将计算机系统中与实现技术和工艺有关的因素提取出来。这个因素就是计算机工作的时钟频率，单位是 MHz；第二，可以测量执行程序使用的总时钟周期数。通过这两个参数就可以知道程序执行的 CPU 时间：

$$\text{CPU 时间} = \frac{\text{总时钟周期数}}{\text{时钟频率}}$$

这两个参数没有反映程序本身的特性。我们还需考虑程序执行过程中所处理的指令数，记为 IC。这样可以获得一个与计算机体系结构有关的参数，即“指令时钟数”CPI(Cycles Per Instruction)：

$$\text{CPI} = \frac{\text{总时钟周期数}}{\text{IC}}$$

程序执行的 CPU 时间就可以写成

$$\text{总 CPU 时间} = \frac{\text{CPI} \times \text{IC}}{\text{时钟频率}}$$

这个公式通常称为 CPU 性能公式，它的三个参数反映了与体系结构相关的三种技术：

- (1) 时钟频率：反映了计算机实现技术、生产工艺和计算机组织。
- (2) CPI：反映了计算机实现技术、计算机指令集的结构和计算机组织。
- (3) IC：反映了计算机指令集的结构和编译技术。

通过改进计算机系统设计，可以相应提高这三个参数的指标，从而提高计算机系统的性能。从目前情况来看，提高某一个参数指标，不会明显影响其他两个指标。这对于综合运用各种技术改进计算机系统的性能是非常有益的。

下面对 CPU 性能公式进行进一步细化。假设计算机系统有 n 种指令，其中第 i 种指令的处理时间为 CPI_i，在程序中第 i 种指令出现的次数为 IC_i，则程序执行时间为

$$\text{CPU 时间} = \frac{\sum_{i=1}^n (\text{CPI}_i \times \text{IC}_i)}{\text{时钟频率}}$$

这个公式同时还反映了计算机系统中每条指令的性能。将上面两个公式合并起来得

$$CPI = \frac{\sum_{i=1}^n (CPI_i \times IC_i)}{IC} = \left(\sum_{i=1}^n CPI_i \times \frac{IC_i}{IC} \right)$$

其中 $\frac{IC_i}{IC}$ 反映了第 i 种指令在程序中所占的比例。上面这些公式均称为 CPU 性能公式。

CPI 的测量比较困难,因为它依赖于处理器组织的细节,如指令流。设计者经常采用指令的平均 CPI 值,该值是通过测量流水线和 Cache 性能,然后计算得出的。

与 Amdahl 定律相比,CPU 性能公式最大的优点是它可以独立涉及到计算机 CPU 性能的各个要素。为使用 CPU 性能评价公式求得 CPU 性能,需要对公式中各独立部分的性能进行测量。开发和使用测量工具,分析测量结果,然后通过权衡各个因素对系统性能的影响,对设计进行修改,是计算机体系结构设计的主要工作。在以后的内容中会看到,这些公式中的各个部分是如何一步一步地测量,然后修改设计,从而使系统性能提高的。

例 1.1 假设我们考虑条件分支指令的两种不同设计方法如下:

- ① CPU_A: 通过比较指令设置条件码,然后测试条件码进行分支;
- ② CPU_B: 在分支指令中包括比较过程。

在两种 CPU 中,条件分支指令都占用 2 个时钟周期而所有其他指令占用 1 个时钟周期,对于 CPU_A,执行的指令中分支指令占 20%;由于每个分支指令之前都需要有比较指令,因此比较指令也占 20%。由于 CPU_A 在分支时不需要比较,因此假设它的时钟周期时间比 CPU_B 快 1.25 倍。哪一个 CPU 更快?如果 CPU_A 的时钟周期时间仅仅比 CPU_B 快 1.1 倍,哪一个 CPU 更快呢?

解 我们不考虑所有系统问题,所以可以用 CPU 性能公式。占用 2 个时钟周期的分支指令占总指令的 20%,剩下的指令占用 1 个时钟周期。所以

$$CPI_A = 0.2 \times 2 + 0.80 \times 1 = 1.2$$

则 CPU 性能为

$$\text{总 CPU 时间}_A = IC \times 1.2 \times \text{时钟周期}_A$$

根据假设有

$$\text{时钟周期}_B = 1.25 \times \text{时钟周期}_A$$

在 CPU_B 中没有独立的比较指令,所以 CPU_B 的程序量为 CPU_A 的 80%,分支指令的比例为

$$20\% / 80\% = 25\%$$

这些分支指令占用 2 个时钟周期,而剩下的 75% 的指令占用 1 个时钟周期,因此:

$$CPI_B = 0.25 \times 2 + 0.75 \times 1 = 1.25$$

因为 CPU_B 不执行比较, 故

$$IC_B = 0.8 \times IC_A$$

因此 CPU_B 性能为

$$\begin{aligned} \text{总 CPU 时间}_B &= IC_B \times CPI_B \times \text{时钟周期}_B \\ &= 0.8 \times IC_A \times 1.25 \times (1.25 \times \text{时钟周期}_A) \\ &= 1.25 \times IC_A \times \text{时钟周期}_A \end{aligned}$$

在这些假设之下, 尽管 CPU_B 执行指令条数较少, CPU_A 因为有着更短的时钟周期, 所以比 CPU_B 快。

如果 CPU_A 的时钟周期时间仅仅比 CPU_B 快 1.1 倍, 则

$$\text{时钟周期}_B = 1.10 \times \text{时钟周期}_A$$

CPU_B 的性能为

$$\begin{aligned} \text{总 CPU 时间}_B &= IC_B \times CPI_B \times \text{时钟周期}_B \\ &= 0.8 \times IC_A \times 1.25 \times (1.10 \times \text{时钟周期}_A) \\ &= 1.10 \times IC_A \times \text{时钟周期}_A \end{aligned}$$

因此 CPU_B 由于执行更多指令条数, 比 CPU_A 运行更快。

1.6 影响计算机体系结构的成本和价格因素

价格是影响一个计算机系统能否成功的最重要因素之一。只有少数计算机, 尤其是巨型计算机, 在设计和制造时不太考虑成本问题。对于大多数计算机系统来说, 成本和价格对于系统的生存和发展是非常重要的。20世纪80年代以来, 如何利用技术优势同时达到高性能和低成本, 已经成为计算机研究的主题之一。计算机专业教科书经常忽略讨论成本和价格问题, 这主要是因为影响成本和价格的因素经常变化, 同时这个问题所涉及到各个方面过于复杂, 难以在计算机专业课程中进行讨论。然而对成本和价格问题的理解对于计算机系统设计人员来说是非常必要的。了解情况的设计者才能对是否采用某种技术(关系到成本问题)做出明智的选择。这里仅仅对成本和价格构成中的部分因素进行简单的介绍, 这部分因素对计算机体系结构的设计会产生一定的影响。其中所举例证使用的数据会随时间而发生变化, 但基本结论和趋势是可信的。

1.6.1 集成电路的成本

集成电路的制造是计算机系统制造过程中一个典型的过程。计算机设计人员可以通过了解集成电路的成本构成, 掌握当前计算机系统的成本概貌。

虽然集成电路的成本相对于时间是按指数下降的, 但是集成电路的基本制

造工艺却没有随时间改变：首先生产圆片(wafer)，在圆片上制造出大量电路单元，圆片经过测试后按照制造的电路单元被切割成基片(die)，如图 1.9 所示。基片在外壳中封装好以后就是集成电路成品。因此一块封装好的集成电路的成本为

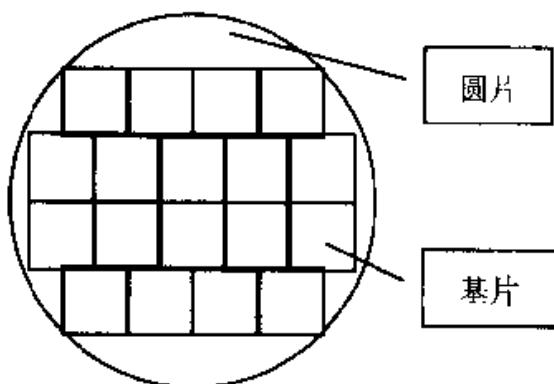


图 1.9 集成电路的圆片和基片

$$\text{集成电路成本} = \frac{\text{基片成本} + \text{基片测试成本} + \text{封装成本}}{\text{最终成品率}}$$

为了能够估计基片成本，我们必须知道圆片成本、每一圆片上的基片数和基片的成品率，这样就能算出基片成本：

$$\text{基片成本} = \frac{\text{圆片成本}}{\text{每块圆片上的基片数} \times \text{基片成品率}}$$

下面分析集成电路的成本与圆片尺寸、基片尺寸的关系。

第一，圆片的基片数一般可由圆片面积除以基片面积求得。精确的公式描述为

$$\text{每块圆片上的基片数} = \frac{\text{圆片面积}}{\text{基片面积}} - \frac{\pi \times \text{圆片直径}}{\sqrt{2} \times \text{基片面积}}$$

式中第二项用于计算在圆片边界上的基片数，这些基片都是不完整的，其数量大约等于圆片圆周除以正方形基片的对角线长。例如，一块直径 20 cm(约 8 英寸)的圆片能够切割出 1 cm 宽的基片个数为

$$3.14 \times 10^2 / 1^2 - (3.14 \times 20 / \sqrt{2} \times 1^2) \approx 269(\text{个})$$

能切割出边长 1.5 cm 的基片个数为

$$\begin{aligned} & \pi \times 10^2 / 1.5^2 - \pi \times 20 / \sqrt{2} \times 1.5^2 \\ & \approx 314 / 2.25 - 62.8 / 2.24 \approx 107(\text{个}) \end{aligned}$$

这仅仅是每一片所能产出完整基片的最大值，另一个问题是圆片上所切割出的这些基片的成品率是多少。假设圆片上的疵点是随机分布的，而产品的成品率与制作工艺的复杂度成反比，那么可得出以下公式：

$$\text{基片成品率} = \text{圆片成品率} \times \left(1 + \frac{\text{疵点密度} \times \text{基片面积}}{\alpha}\right)^{-\alpha}$$

其中圆片成品率考虑的是完全损坏不能使用的圆片,这里简单地假设圆片成品率是100%。疵点密度是一个与生产工艺有关的随机数,它是通过测量获得的,1995年约为0.6~1.2。 α 这个参数主要依赖于集成电路使用的掩膜层数和制造工艺,对于当前广泛使用的多层金属布线CMOS工艺,一般取 $\alpha=3.0$ 。例如,假设疵点密度为0.8个/cm²,圆片成品率是100%,则边长为1cm和1.5cm的基片成品率分别为

$$\text{基片成品率}_{1\text{ cm基片}} = (1 + 0.8 \times 1^2 / 3)^{-3} = 0.49$$

$$\text{基片成品率}_{1.5\text{ cm基片}} = (1 + 0.8 \times 1.5^2 / 3)^{-3} = 0.24$$

而每一圆片上所能生产的成品基片数为

$$\text{成品基片数} = \text{基片成品率} \times \text{每块圆片上的基片数}$$

组合上面的公式,我们可以获得基片成本的公式,即

$$\begin{aligned} \text{基片成本} &= \frac{\text{圆片成本}}{\text{每块圆片上的基片数} \times \text{基片成品率}} \\ &= \frac{\text{圆片成本}}{\left(\frac{\text{圆片面积}}{\text{基片面积}} - \frac{\pi \times \text{圆片直径}}{\sqrt{2} \times \text{基片面积}}\right) \times \text{圆片成品率} \times \left(1 + \frac{\text{疵点密度} \times \text{基片面积}}{\alpha}\right)^{-\alpha}} \end{aligned}$$

公式中的圆片成本、圆片成品率、 α 值及疵点密度等是由生产工艺决定的,唯一可以由设计人员控制的就是基片面积。从公式可以推断,基片成本基本与基片面积的($\alpha+1$)次方成正比,即

$$\text{基片成本} = f(\text{基片面积}^{\alpha+1})$$

对于现代先进工艺来说, α 值一般可以取3,基片成本与基片面积的4次方成正比。对于直径20cm的圆片,可生产132片合格的1cm²基片及25片合格的2.25cm²基片。大部分微处理器尺寸介于上述两者之间。

1995年,直径20cm,3~4层金属布线的生产工艺的成品圆片成本大约为3 000~4 000美元/片。假设生产一片圆片成本为3 500美元,那么生产一片大小为1cm²基片的成本大约为27美元,而生产大小为2.25cm²的基片成本为140美元。

1.6.2 计算机系统的成本和价格

对计算机系统成本产生影响的主要因素有时间、产量、商品化等。

对成本产生最直接影响的是时间。即使实现技术没有变动,计算机系统的制造成本也会不断下降。随着时间的推移,生产工艺会日渐稳定,产品的成品率会不断提高。产品的成本与成品率成反比。规划产品生产周期时,成本随时间变化是非常关键的因素。实际上摩尔定律告诉我们:对于集成电路类产品,如果现在花1

元钱购买的计算能力为 1,那么 18 个月以后花 1 元钱购买的计算能力就是 2。

产量是决定产品成本的第二个关键因素。首先,产量的增加会加速工艺的稳定;第二,产量增加意味着提高了生产效率,降低了成本;第三,产量增加还可降低每台单机必须加入的开发费用,从而使得单机成本下降。统计结论是:无论是集成电路芯片、印刷电路板或系统,如果产量翻一番,那么成本就会减少 10%。

商品化也是影响产品成本的重要因素,但更重要的是它影响产品的价格。所谓商品就是指市场上销售的批量化的产品,例如 DRAM、磁盘、显示器、键盘等。商品化包括建立市场和销售渠道的过程,如广告、代理、维修等。在过去的十多年内,计算机产品的商业化主要集中于 IBM PC 兼容产品的开发与销售。大量的零售商在销售各种产品,竞争非常激烈。这种竞争一方面减少了产品成本与售价之间的差额,另一方面也降低了开发成本和风险。因为商品化的市场同时包括了较大的产量和相对清晰稳定的产品概念,产品部件提供商也展开了竞争,从而导致整个产品成本的下降。

价格与成本是不同的概念。部件的成本会限制设计,但成本并不代表用户必须付出的价格,成本在变成实际价格之前会出现一系列的变化,系统设计者必须清楚设计方案对最终的销售价格的影响。一般情况下,成本每变化 1 000 美元,价格将变化 3 000~4 000 美元。价格上扬时计算机销售势头就不好,产量就会下降,成本就会增大,并导致价格进一步增长。因此小小的成本变化会产生很大的影响。

构成价格的各因素可以通过占成本或价格的百分比来表示。价格与成本的差别也因销售市场的不同而不同。简单地说,商品的标价(价格)由这样一些因素构成:原料成本、直接成本、毛利和折扣。

原料成本是指一件产品中所有部件的采购成本总和。它是价格中最明显的一部分,也是对计算机系统设计影响最明显的部分。

直接成本是指与一件产品生产直接相关的成本,包括劳务成本、采购成本(如运输、包装费用)、零头(剩余的零头)及产品质量成本(如人员培训、生产过程管理等)。直接成本通常是在部件成本上增加 20%~40%。

接下来的一部分叫做毛利。毛利也是公司开支的一部分,但是这一部分开支是无法由一件产品直接支付的,它必须均摊到每一件产品中去。毛利主要包括:公司的研发费用、市场建立费用、销售费用、生产设备维护费用、房租、贷款利息、税后利润和所得税等。原料成本、直接成本和毛利相加,就得到平均销售价格。毛利一般占到平均销售价格的 20%~55%,具体情况取决于产品的独立性。导致低端 PC 产品制造商具有较低的毛利几个主要原因是:首先,由于产品的标准化,他们的研发费用较低。其次,他们采取非直接销售方式(通过邮购、电话订购或零售店),所以其销售成本较低。第三,因为这一类产品缺乏独特性,竞

争激烈导致低价格和低利润,因而毛利也较低。

标价与平均销售价格并不一样,原因之一是公司提供了批量折扣,降低了平均销售价格。另外如果产品通过零售店进行销售,零售商亦需获得标价 40%~50% 的利润。因而平均销售价只能达到标价的 50%~75%。

正如我们所见到的,价格随竞争程度的变化而变化。公司在销售产品时可能无法取得所期望的毛利。在更坏的情况下,价格猛跌而导致无法取得利润。一家公司争夺市场份额的方法就是降低价格,从而提高产品的竞争力,如果产量增加则成本就会减小,就有可能维持利润。它们之间的关系是极其复杂的,这里不可能深入分析。

在美国,大部分公司只将收入的 4% (微机产业)~12% (高端产品产业) 用于研发。这一百分比将不会轻易随时间变化。大型机器一般要投入更大的资金用于研发。一台机器的制造成本增加 10 倍,则其研发成本将会增加更多倍。由于大型机器销售情况通常不如小型机器,大型机器的毛利就会比较高,因此价格就更高。这种投资模式将大型机置于双重危险之中:销售量不大而又需要很高的研发成本比例。这就说明了为什么大型机的价格/成本比总是比小型机高。图 1.10 通过工作站产品成本和价格的分布,对上述概念进行说明。

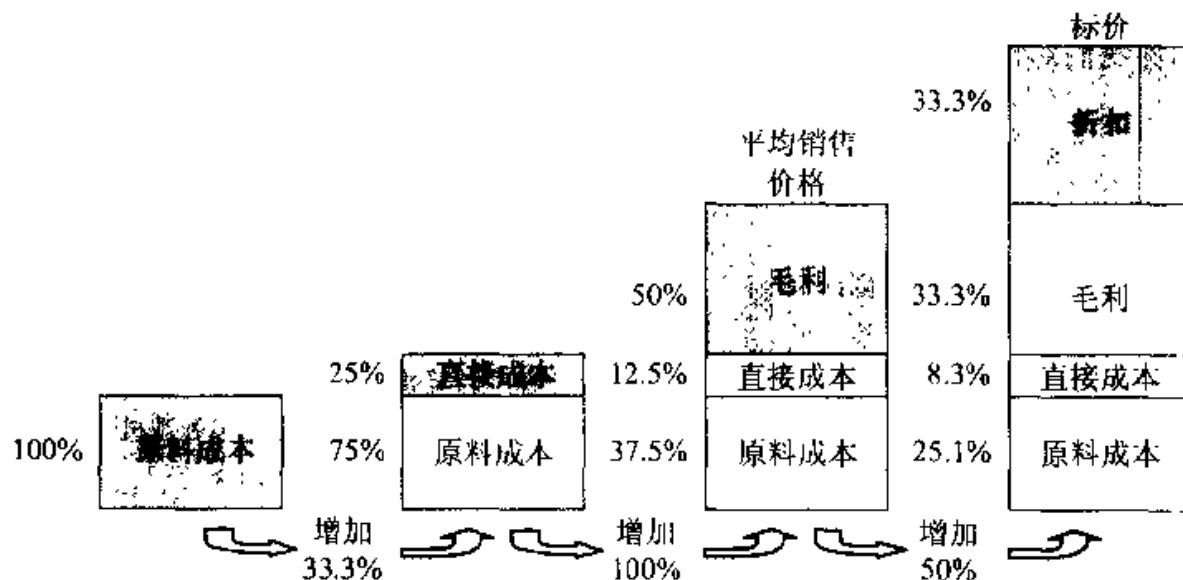


图 1.10 工作站的成本和价格构成

关于成本及成本/性能的问题是很复杂的,计算机系统的设计往往也不是单一目标。一种极端是巨型计算机设计,为达到高性能而不考虑成本;另一种极端是低成本机器,为达到低成本就需要牺牲一些性能,低端的 PC 就属于这一类。位于这两种极端之间的是性能/成本设计,设计者需取得性能与成本之间的平衡。绝大部分工作站、服务器等制造商就属于这一类。在过去的几十年中,计算机尺寸变小,因此低成本设计和成本/性能设计就显得日益重要,即使是巨型计

计算机制造商也发觉成本问题变得日益重要。

1.7 小结

本章主要讨论计算机体系结构的基本概念。

首先,在计算机系统层次结构概念的基础上,讲述了经典计算机体系结构概念,并进一步讨论了计算机组成和计算机实现技术,在此基础上,我们可以更好地理解现代计算机体系结构所研究的范围和内容。

通过存储程序计算机,我们了解了计算机的分代和分型,研究了计算机应用需求和实现技术等方面的发展对计算机体系结构发展的促进作用,总结了计算机体系结构的生命周期。

体系结构研究的主要内容之一就是通过并行性技术提高计算机系统性能。本章介绍了并行性技术的基本概念,这些概念是学习计算机体系结构的基础。

促进现代计算机发展的重要手段之一就是对计算机系统中采用的技术进行定量分析。本章讨论了一些对计算机系统性能进行定量分析的技术、方法、参数和指标等,并给出了贯穿全书的一些指导计算机体系结构设计的基本原则。

本章的最后,简要讨论了影响体系结构设计的成本和价格因素,这些概念会加深我们对计算机体系结构技术的理解。

习 题 一

1.1 解释下列术语:

层次结构	翻译	解释	体系结构
透明性	系列机	软件兼容	兼容机
计算机组成	计算机实现	存储程序计算机	并行性
时间重叠	资源重复	资源共享	同构型多处理机
异构型多处理机	最低耦合	松散耦合	紧密耦合
响应时间	测试程序	测试程序组件	大概率事件优先
系统加速比	Amdahl 定律	程序的局部性原理	CPI
原料成本	直接成本	毛利	折扣
标价			

1.2 假设有一个计算机系统分为四级,每一级指令都比它下面一级指令在功能上强 M 倍,即一条 $r+1$ 级指令能够完成 M 条 r 级指令的工作,且一条 $r+1$ 级指令需要 N 条 r 级指令解释。对于一段在第一级执行时间为 K 的程序,在第二、第三、第四级上的一段等效程序需要执行多少时间?

1.3 传统存储程序计算机的主要特征是什么?存在的主要问题是什么?目前的计算机系统是如何改进的?

1.4 对于一台400 MHz计算机执行标准测试程序,程序中指令类型,执行数量和平均时钟周期数如下:

指令类型	指令执行数量	平均时钟周期数
整数	45 000	1
数据传送	75 000	2
浮点	8 000	4
分支	1 500	2

求该计算机的有效 CPI、MIPS 和程序执行时间。

1.5 计算机系统中有三个部件可以改进,这三个部件的部件加速比如下?

$$\text{部件加速比}_1 = 30$$

$$\text{部件加速比}_2 = 20$$

$$\text{部件加速比}_3 = 10$$

- (1) 如果部件 1 和部件 2 的可改进比例均为 30%,那么当部件 3 的可改进比例为多少时,系统加速比才可以达到 10?
- (2) 如果三个部件的可改进比例分别为 30%、30% 和 20%,三个部件同时改进,那么系统中不可加速部分的执行时间在总执行时间中占的比例是多少?
- (3) 如果相对某个测试程序三个部件的可改进比例分别为 20%、20% 和 70%,要达到最好改进效果,仅对一个部件改进时,要选择哪个部件?如果允许改进两个部件,又如何选择?

第二章 计算机指令集结构设计

计算机指令集结构的设计是计算机体系结构设计的核心问题,是软硬件功能分配最主要的界面,它历来是计算机体系结构设计者、系统软件设计者和硬件设计者所共同关注的问题。

本章首先给出一种指令集结构类型的分类方法,并对各种指令集结构类型的优缺点进行深入分析。然后就指令集结构设计中的诸多问题,如寻址方式、指令集的功能设计、操作数的类型和大小以及指令集格式等进行全面讨论。最后给出一种指令集结构的实例——DLX 指令集结构。本章在论述上述问题的同时,还将给出许多有关指令集结构的测量统计结果,这些结果将对指令集结构设计的综合决策提供极大的帮助和可靠的依据。

2.1 指令集结构的分类

2.1.1 指令集结构的分类

一般来说,可以按如下五个因素考虑对计算机指令集结构进行分类:

1. 在 CPU 中操作数的存储方法;
2. 指令中显式表示的操作数个数;
3. 操作数的寻址方式;
4. 指令集所提供的操作类型;
5. 操作数的类型和大小。

在这五个分类因素中,CPU 中操作数的存储方法,即在 CPU 中用来存储操作数的存储单元的类型,是各种指令集结构之间最主要的区别所在。本节也将据此对指令集结构进行分类。

CPU 中用来存储操作数的存储单元主要有:堆栈、累加器或一组寄存器。另外,指令中的操作数可以显式给出,也可以按照某种约定隐式地给出。比如在“堆栈结构”中,操作数被约定是堆栈的栈顶,而在“累加器结构”中,操作数则是由累加器表示。对于“通用寄存器结构”而言,一般都会明确给出指令中操作数

的存储单元,要么是寄存器,要么是存储器单元。

表 2.1 根据 CPU 内部的存储单元种类,给出了操作数的不同存储方式。从该表中可以看出,对于一条有两个源操作数和一个结果(目的)操作数的指令来说,其显式表示的操作数个数随 CPU 存取操作数的方式不同而不同。因此如果根据 CPU 内部存储单元类型对指令集结构进行分类,一般可以分为堆栈型指令集结构、累加器型指令集结构和通用寄存器型指令集结构。需要指出的是,当前多数指令集结构均可以归结到上述三种结构类型中的某一种,但是有一些指令集结构却是混合型的,比如 Intel 80x86 的指令集结构就是介于通用寄存器型指令集结构和累加器型指令集结构之间的一种类型。

表 2.1 CPU 对操作数的不同存取方式

CPU 提供的 暂存器	每条 ALU 指令显式表 示的操作数个数	运算结果的 目的地	访问显式操作数的 方法
堆 栈	0	堆栈	PUSH/POP
累加器	1	累加器	LOAD/STORE 累加器
一组寄存器	2/3	寄存器或存储器	LOAD/STORE 寄存器或存储器

表 2.2 给出了 $C = A + B$ 表达式在这三种类型指令集结构上的实现方法。假设 A、B、C 均保存在存储器单元中,且在运算过程中一直保持 A 和 B 的值。

表 2.2 $C = A + B$ 表达式在这三种类型指令集结构上的实现方法

堆 栈	累加器	寄存器(寄存器 - 存储器)	寄存器(寄存器 - 寄存器)
PUSH A	LOAD A	LOAD R1,A	LOAD R1,A
PUSH B	ADD B	ADD R1,B	LOAD R2,B
ADD	STORE C	STORE C,R1	ADD R3,R1,R2
POP C			STORE C,R3

如果从指令集结构和编译器之间的关系、指令集结构的实现效率、目标代码的大小三个方面考察表 2.2,可以容易得到如表 2.3 所示的三种类型指令集结构的优缺点。

表 2.3 三种类型指令集结构的优缺点

指令集结构类型	优 点	缺 点
堆栈型	是一种表示计算的简单模型;指令短小	不能随机访问堆栈,从而很难生成有效代码。同时,由于堆栈是瓶颈,所以很难被高效地实现
累加器型	减小了机器的内部状态;指令短小	由于累加器是唯一的暂存器,这种机器的存储器通信开销最大
寄存器型	是代码生成的最一般的模型	所有操作数均需命名,且要显式表示,因而指令比较长

早期的大多数机器都是采用堆栈型或累加器型指令集结构,但是自 1980 年以来的大多数机器均采用的是通用寄存器型指令集结构。这主要有两个方面的原因,一是寄存器和 CPU 内部其他存储单元一样,要比存储器快;其次是对编译器而言,可以更加容易有效地分配和使用寄存器。

由于通用寄存器型指令集结构是现代指令集结构类型的主流,而且可能会在今后延续使用相当长的时间,所以本书后面主要针对这种类型的指令集结构进行讨论。

2.1.2 通用寄存器型指令集结构分类

通用寄存器型指令集结构的一个主要优点就是能够使编译器有效地使用寄存器。这不仅体现在表达式求值方面、更重要的是体现在利用寄存器存放变量所带来的优越性上。

通用寄存器型指令集结构在表达式求值方面,比堆栈型指令集结构和累加器型指令集结构具有更大的灵活性。例如,在一台通用寄存器型指令集结构的机器上求表达式 $(A \times B) - (C \times D) - (E \times F)$ 的值,其中的乘法运算就可以按任意的次序进行。但是在一台堆栈型指令集结构的机器上,该表达式的求值必须按从左到右的顺序进行。

另外,寄存器可以用来存放变量。将变量分配给寄存器,不但能够减少存储器的通信量,加快程序的执行速度(因为寄存器比存储器快),而且和存储器相比,还可以用更少的地址位来寻址寄存器,从而能够有效改进程序的目标代码大小。

编译器设计者通常总是希望 CPU 内部的所有寄存器是公开通用的。而许多老式机器则将其许多寄存器作为专用寄存器,结果导致了通用寄存器数量的减少。如果通用寄存器数量太少,由于表达式求值也需要寄存器,所以即使将变量分配到寄存器中,也不会对程序运行速度的提高带来任何好处。

那么,CPU 到底需要设置多少个寄存器呢? 这主要由编译器使用寄存器的情况来决定。大多数编译器都会为表达式求值保留一些寄存器, 同时还会为传递参数保留一些寄存器, 而用剩下的寄存器来保存变量。只有通过认真研究哪些变量可以分配给寄存器, 以及所采用的寄存器分配算法, 才有可能知道究竟需要给 CPU 设置多少个寄存器。在后面的有关内容中将对此进行详细论述。

深入研究算术逻辑运算指令(ALU 指令)的本质, 可以发现有两种主要的指令特性能够被用来对通用寄存器型指令集结构(GPR: General-Purpose Register)进行进一步细分。一是 ALU 指令到底有两个还是三个操作数。对于有三个操作数的指令, 它包含两个源操作数和一个目的操作数; 对于有两个操作数的指令, 其中一个操作数既作为源操作数, 又作为目的操作数。二是在 ALU 指令中, 有多少个操作数可以用存储器来寻址, 即有多少个存储器操作数。一般来说, ALU 指令有 0~3 个存储器操作数。

基于上述两种 ALU 指令的特性及其所有可能的组合, 可以得到如表 2.4 所示的七种组合类型。

表 2.4 ALU 指令中存储器操作数个数和操作数个数的所有可能组合以及相应的机器实例

ALU 指令中存储器操作数个数	ALU 指令中操作数的最大个数	机器实例
0	2	IBM RT-PC
	3	SPARC, MIPS
1	2	PDP-10, IBM 360, Motorola 68000
	3	IBM 360 的部分指令
2	2	PDP-11, 部分 IBM 360 指令
	3	
3	3	VAX

仔细研究表 2.4 不难看出, 实际上可以将当前大多数通用寄存器型指令集结构进一步细分为三种类型, 即寄存器-寄存器型(R-R: Register-Register)、寄存器-存储器型(R-M: Register-Memory)和存储器-存储器型(M-M: Memory-Memory)。

对某种通用寄存器型指令集结构而言, 如果其所有 ALU 指令都不包含存

储器操作数,那么称这种指令集结构为寄存器-寄存器型指令集结构,或者是Load/Store型指令集结构;如果ALU指令中包含有存储器操作数,那么根据拥有一个存储器操作数或一个以上存储器操作数,可以分别称其为寄存器-存储器型指令集结构或存储器-存储器型指令集结构。这三种通用寄存器型指令集结构的优缺点如表2.5所示。

表2.5 常见的三种通用寄存器型指令集结构的优缺点

指令集结构类型	优 点	缺 点
寄存器-寄存器型 (0,3)	简单,指令字长固定,是一种简单的代码生成模型,各种指令的执行时钟周期数相近	和指令中含有对存储器操作数访问的指令集结构相比,指令条数多,因而其目标代码较大
寄存器-存储器型 (1,2)	可以直接对存储器操作数进行访问,容易对指令进行编码,且其目标代码较小	指令中的操作数类型不同。在一条指令中同时对一个寄存器操作数和存储器操作数进行编码,将限制指令所能够表示的寄存器个数。由于指令的操作数可以存储在不同类型的存储器单元,因此每条指令的执行时钟周期数也不尽相同
存储器-存储器型 (3,3)	是一种最紧密的编码方式,无需“浪费”寄存器保存变量	指令字长多种多样。每条指令的执行时钟周期数也大不一样,对存储器的频繁访问将导致存储器访问瓶颈问题

注:表中(m, n)的含义是,指令的 n 个操作数中有 m 个存储器操作数

一般来说,指令格式和指令字长越单一,编译器的工作就越简单。如果指令集结构的指令格式和指令字长具有多样性,则可以有效地降低程序的目标代码长度。但是这种多样性也可能会增加编译器和CPU实现的难度。另外,CPU中寄存器的个数也会影响指令的字长。当然,表2.5中所表示的优缺点也并非是绝对的。

2.2 寻址技术

在通用寄存器型指令集结构中,一般利用寻址方式指明指令中的操作数是一个常数、一个寄存器操作数或者是一个存储器操作数。表2.6列出了当前指令集结构中所使用的一些操作数寻址方式。

对于立即值寻址方式而言,虽然它所指明的操作数就在指令中,但是这里也认为它是一种操作数的寻址方式。我们把由程序计数器决定的寻址方式,叫做“PC 相对寻址”,PC 相对寻址主要用在控制转移指令中指定目标指令代码的地址。另外,在表 2.6 的自增/自减寻址方式及缩放寻址方式中,变量 d 用来指明被指令存取的数据项有多少,这也说明,只有当所存取的所有数据元素在存储器中彼此相邻,这几种寻址方式才有用。

表 2.6 当前指令集结构中所使用的一些操作数寻址方式

寻址方式	指令实例	含 义
寄存器寻址	ADD R4 , R3	$Regs[R4] \leftarrow Regs[R4] + Regs[R3]$
立即值寻址	ADD R4 , #3	$Regs[R4] \leftarrow Regs[R4] + 3$
偏移寻址	ADD R4 , 100(R1)	$Regs[R4] \leftarrow Regs[R4] + Mem[100 + Regs[R1]]$
寄存器间接寻址	ADD R4 , (R1)	$Regs[R4] \leftarrow Regs[R4] + Mem[Regs[R1]]$
索引寻址	ADD R3 , (R1 + R2)	$Regs[R3] \leftarrow Regs[R3] + Mem[Regs[R1] + Regs[R2]]$
直接寻址或 绝对寻址	ADD R1 , (1001)	$Regs[R1] \leftarrow Regs[R1] + Mem[1001]$
存储器间接寻址	ADD R1 , @(R3)	$Regs[R1] \leftarrow Regs[R1] + Mem[Mem[Regs[R3]]]$
自增寻址	ADD R1 , (R2) +	$Regs[R1] \leftarrow Regs[R1] + Mem[Regs[R2]]$ $Regs[R2] \leftarrow Regs[R2] + d$
自减寻址	ADD R1 , -(R2)	$Regs[R2] \leftarrow Regs[R2] - d$ $Regs[R1] \leftarrow Regs[R1] + Mem[Regs[R2]]$
缩放寻址	ADD R1 , 100(R2)[R3]	$Regs[R1] \leftarrow Regs[R1] + Mem[100 + Regs[R2] * d]$

注:全书把 C 语言标注方法进行扩展,并用来作为硬件描述的符号系统。该表用了两个非 C 特性的符号:左箭头“ \leftarrow ”表示赋值操作;“Mem”表示存储器的名字。 $Mem[R1]$ 指的是由寄存器 R1 中内容指定地址的存储器单元中的内容。

在指令集结构中采用多种寻址方式可以显著地减少程序的指令条数。但这同时也可能增加实现的复杂度和使用这些寻址方式的指令的执行时钟周期数(CPI)。所以,有必要对各种不同寻址方式的使用情况进行统计分析,用以指导

选择指令集结构所应支持的寻址方式。

在 VAX 指令集结构的机器上运行 gcc、Spice 和 Tex 基准程序，并对各种寻址方式的使用情况进行统计，可以得到如图 2.1 所示的统计结果。这里只给出了使用频率超过 1% 的那些寻址方式。

从图 2.1 可以看出，立即值寻址方式和偏移寻址方式的使用频率十分高。所以，下面着重论述这两种常用寻址方式的一些性质。

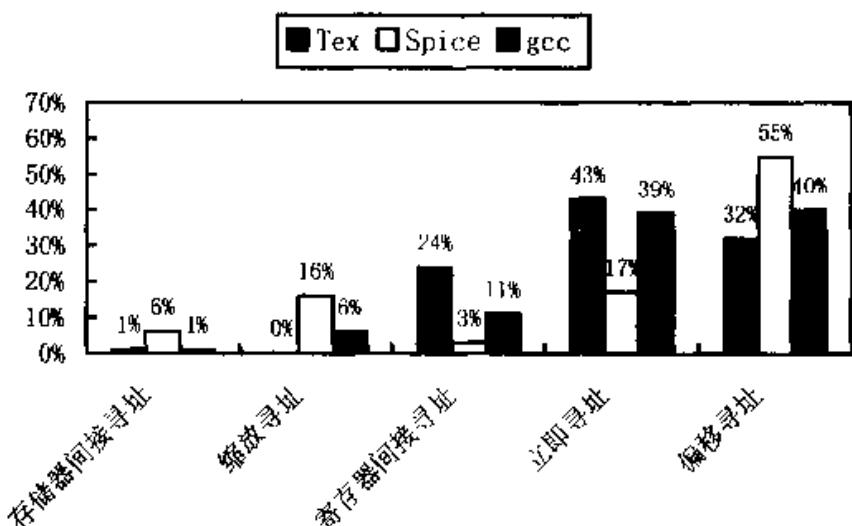


图 2.1 寻址方式使用情况统计结果

如果要在一种指令集结构设计中设置偏移寻址方式，那么首先必须知道各种偏移量大小的使用情况，这样才能比较容易确定指令集到底应该支持多大的偏移量，以及在指令中需要设置多少位的偏移量字段来表示偏移量。这一选择十分重要，因为它直接影响到指令的字长。

在一台寄存器—寄存器型指令集结构的机器上分别运行 SPECint92 基准程序集 (compress、espresso、eqntott、gcc、li) 和 SPECfp92 基准程序集 (dudoc、ear、hydro2d、mdljdp2、su2cor)，并对采用偏移寻址方式的数据访问指令所使用偏移量的大小进行测量统计，可以得到如图 2.2 所示的各种偏移量字段大小的使用情况。

图 2.2 中 x 轴标记的是对偏移量大小进行 $\log_2(\cdot)$ 运算所得到的结果，也就是偏移量字段的位数。从图 2.2 可以看出，程序所使用的偏移量大小分布十分广泛，这主要是因为在存储器中所保存的数据并不是十分集中，需要使用不同的偏移量大小才能对其进行存取。另外，较小的偏移量和较大的偏移量均占有相当大的比例。这一统计结果为选择偏移量字段的大小提供了定量的依据。比如，如果将偏移量字段的大小设置为 12~16 位。图 2.2 表明这种长度可以支持上述 75%~99% 基于偏移寻址方式的数据访问中偏移量大小的表示。

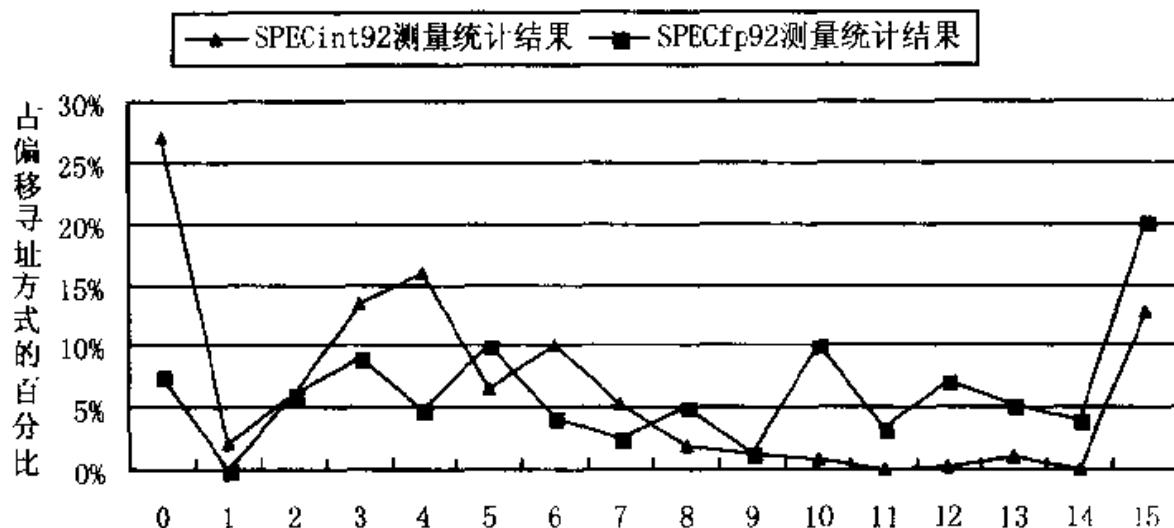


图 2.2 各种偏移量字段大小的使用情况

算术运算、分支比较等指令通常都要用到立即值寻址方式。对指令集结构设计而言,首先要确定所有的指令都要具有立即值寻址方式,还是只有部分指令具有立即值寻址方式。图 2.3 表示在一种 Load/Store 型指令集结构中,一些指令使用立即值寻址方式的频率。从图 2.3 中可以看出,比较指令和 ALU 指令使用立即值寻址方式十分频繁。

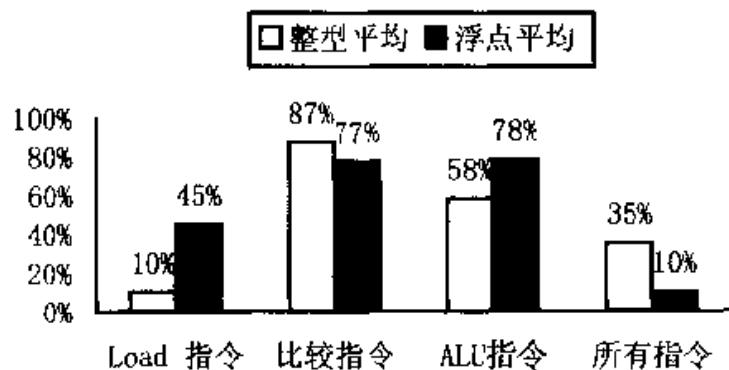


图 2.3 指令使用立即值寻址方式的频率

另外,和偏移寻址模式一样,指令所能够支持的立即值大小也会影响指令的字长。所以,我们还需确定指令所使用的立即值大小的范围。

立即值大小的使用分布情况如图 2.4 所示,x 轴标记的是用于表示立即值的位数。从图中可以看出最常用到的是较小的立即值,然而有时也会用到较大的立即值(主要是用于地址计算)。在指令集结构设计中,至少要将立即值的大小设置为 8~16 位,因为图 2.4 表明这种立即值字段的大小可以覆盖所有使用立即值寻址方式指令的 50%~80%。

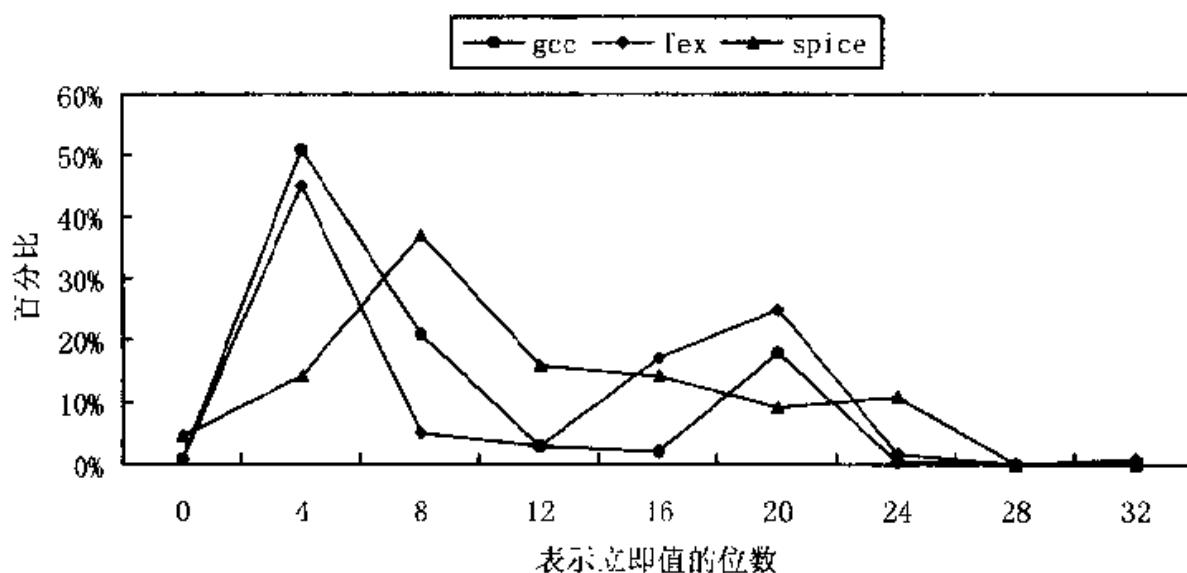


图 2.4 不同立即值大小的使用分布图

2.3 指令集结构的功能设计

大多数指令集结构所支持的操作可以被归纳为表 2.7 所示的一些类型。所有的指令集结构一般都会对前三种类型的操作提供相应的指令。在各种指令集中对系统功能的支持程度会随着结构的不同而有较大的差异，但是所有的指令集结构都必须对基本的系统功能提供一些指令支持。另外，大多数具有浮点计算功能的指令集结构均提供了支持浮点操作的浮点指令。对后面三种类型的操作而言，有些指令集结构可能根本不会对其提供任何指令支持，而有的指令集结构则可能提供了和这些操作相对应的指令。

表 2.7 指令集结构中操作的分类

操作类型	实 例
算术和逻辑运算	整数的算术和逻辑操作：加、减、与、或等
数据传输	Load/Store
控制	分支、跳转、过程调用和返回、自陷等
系统	操作系统调用、虚拟存储器管理等
浮点	浮点操作：加、乘等
十进制	十进制加、十进制乘、十进制到字符的转换等
字符串	字符串移动、字符串比较、字符串搜索等
图形	像素操作、压缩/解压操作等

那么，一种指令集结构中的指令到底要支持哪些类型的操作呢？这就是所

谓的指令集结构功能设计问题。当前在这一问题的处理上有两种截然不同的方向。一个方向是强化指令功能,实现软件功能向硬件功能转移,基于这种指令集结构而设计实现的计算机系统称为复杂指令集计算机(CISC)。另一个方向是20世纪80年代发展起来的精简指令集计算机(RISC),其目的是尽可能地降低指令集结构的复杂性,以达到简化实现,提高性能的目的,这也是当今指令集结构功能设计的一个主要趋势。

下面将分别从这两个方向讨论指令集结构功能设计的一些问题。另外,由于分支和跳转等控制操作的行为在上述操作类型中较为特殊,所以在本节的最后将深入讨论支持控制操作的控制流指令。

2.3.1 CISC计算机指令集结构的功能设计

CISC结构和RISC结构的重要区别之一就是在于其指令的功能强弱上。一般来说,CISC结构追求的目标是强化指令功能,减少程序的指令条数,以达到提高性能的目的。通过对诸如IBM 370、VAX-11/780、Intel公司的APX等CISC计算机进行分析,可以发现增强这些机器的指令功能主要是从如下几个方面着手。

1. 面向目标程序增强指令功能

对大量目标程序及其执行情况进行统计分析,可以发现有些指令或者指令串的使用频率较高。如果增强这些指令的功能,并加快其执行,或者将常用的指令串用一条新的指令来替代,不但会减少目标程序存取指令的次数,加快目标程序的执行,而且也会有效地缩小程序目标代码的长度。这种面向目标程序增强指令功能主要利用如下一些方法。

(1) 提高运算型指令功能

在一些高性能计算机中,为了提高运算速度,减少程序调用的额外开销,将那些常用的计算函数及子程序用一条指令来实现,如 \sin 、 \cos 、 \tan 、 e^x 等。至于一台机器将哪些子程序指令化,这要在增加硬设备与提高性能之间进行合理的折衷。

(2) 提高传送指令功能

在一些数据处理计算机和通用计算机中,一般都设有成组传送指令。其主要功能是将主存中的一组数据传送到CPU。由于对向量、数组等处理的需要,也在寄存器之间或寄存器与处理部件之间设置了一些一次就能传送多个数据的指令,如向量计算机中的向量传送指令等。

(3) 增加程序控制指令功能

为了缩小程序空间,提高软件运行效率,尤其是提高系统软件的运行效率,在CISC计算机中一般均设置了多种程序控制指令。

在 VAX-11/780 机器指令集结构中有转移指令和跳转指令。转移指令提供一个偏移量，并把它与现行程序计数器相加而获得新地址；跳转指令则是利用正常的寻址方式指令把新地址放入程序计数器。因为大多数程序转移都是转到现行指令相近的单元，所以转移指令比跳转指令更有效。VAX-11/780 机器有 29 种转移指令，在此就不一一赘述了。

实际上，如果没有这些功能较强的控制指令，目标程序也能正确运行，只不过这些功能需要几条指令或一段程序才能完成。但是实现这些具有较强功能的指令，硬件开销较大，所以只有频繁使用的子程序或指令串，用较强功能的指令替换才合算。

2. 面向高级语言和编译程序改进指令系统

目前，高级语言大都是由编译程序编译的。但是应该看到，由于高级语言和机器语言有较大的语义差距，使得经编译程序形成的目标程序往往比人们直接用机器语言编写的程序要长，需要更多的运行时间；在用户程序编译和运行的过程中，编译程序运行时间本身在总运行时间中占较大的比例，况且，即使编译过程通过了，目标程序也可能通不过，这时再由目标程序往回查找源程序的错误也较为困难。为了解决这些问题，在 CISC 计算机的指令集结构功能设计中，可以考虑面向高级语言和编译程序的优化来增强相应指令功能。

(1) 增加对高级语言和编译系统支持的指令功能

一方面，可以对源程序中各种高级语言语句进行使用频度的统计与分析，对于使用频度高的语句，可以设置专门的指令或采取措施增加相应指令的功能，以提高其编译速度和执行速度。另一方面，可以从面向编译程序，尤其是从优化代码生成的角度进行考虑，增加指令集结构的规整性来改进指令系统。所谓规整性，是指没有或尽可能减少例外的情况和特殊的应用，以及所有运算都能对称、均匀地在存储器单元或寄存器单元之间进行。如 IBM 370 的指令集结构就多少有些不规整，有些指令只能使用指定的寄存器。这样，在编译时，对这些指令就需先把它们要用到的寄存器腾出来，并把要用到的数据移入这些指定的寄存器。这些都增加了不是运算所必须的数据传送操作，并使得对通用寄存器的优化管理更加复杂化。

(2) 高级语言计算机指令系统

采用了上述各种对高级语言和编译程序提供支持的措施后，机器语言和高级语言的语义差距比传统的冯·诺依曼型机器缩小了许多。这种机器统称为面向高级语言(HL)的机器。可以看出，在语言编译实现中，对高级语言解释的分量明显增大。

如果进一步增大解释的比重，直至几乎没有语义差距，则可达到使高级语言成为机器的汇编语言，即高级语言和机器语言是一一对应的，这种机器称为间接

执行型高级语言机器。它用汇编的方法(可以用软件实现,也可以用硬件实现)把高级语言源程序翻译成机器语言程序。

当然,高级语言机器本身也可以没有机器语言,或者说高级语言就作为机器语言。它可以直接由硬件或固件对高级语言源程序的语句逐条进行解释并执行。这样既没有编译,也不用汇编。由于是逐条解释,当发现有程序设计的错误时,错误现场容易保存,因而也容易排除错误。另外,解释方法对实现交互式语言有利。这种机器称为直接执行型高级语言机器。

3. 面向操作系统的优化实现改进指令系统

计算机体系结构与操作系统是紧密联系的,操作系统的实现在很大程度上取决于体系结构的支持。这主要表现在对中断处理、进程管理、存储管理和保护、系统工作状态的建立与切换等的支持。就指令系统而言,虽然它不能全而反映体系结构对操作系统的支持,但对其主要方面均有反映。我们可以通过设置支持系统工作状态和访问方式转移的指令、支持进程转移的指令、支持进程同步和互斥的指令等措施,来达到优化实现操作系统的目的。

2.3.2 RISC 计算机指令集结构的功能设计

在 20 世纪 70 年代后期,人们已经感到日趋庞杂的指令系统不仅不易实现,而且还有可能降低系统的效率。1979 年,以 Patterson 为首的一批科学家对指令集结构的合理性进行了深入研究,研究结果表明,CISC 结构存在着如下缺点:

1. 在 CISC 结构的指令系统中,各种指令的使用频率相差悬殊。据统计,有 20% 的指令使用频率最大,占运行时间的 80%。也就是说,有 80% 的指令只在 20% 的运行时间内才会用到。
2. CISC 结构指令系统的复杂性带来了计算机体系结构的复杂性,这不仅增加了研制时间和成本,而且还容易造成设计错误。
3. CISC 结构指令系统的复杂性给 VLSI 设计带来了很大负担,不利于单片集成。
4. CISC 结构的指令系统中,许多复杂指令需要很复杂的操作,因而运行速度慢。
5. 在 CISC 结构的指令系统中,由于各条指令的功能不均衡性,不利于采用先进的计算机体系结构技术(如流水技术)来提高系统的性能。

针对上述缺点,Patterson 等人提出了 RISC 结构的设想。RISC 计算机的指令系统只包含那些使用频率很高的指令和一些必要指令。

实际上,在许多机器中,最经常执行的操作都是其指令集结构中一些简单的操作。比如,对在 Intel 80x86 上执行的标量程序进行统计,可以发现如表 2.8 所示的 10 种简单指令几乎占据了所有执行指令的 96%。根据“快速实现常见

事件”(make the common case fast)的设计原理,应该使得这些指令的实现尽可能地快。

表 2.8 Intel 80x86 最常用的 10 条指令

执行频率排序	80x86 指令	指令执行频率(%执行指令总数)
1	Load	22%
2	条件分支	20%
3	比较	16%
4	Store	12%
5	加	8%
6	与	6%
7	减	5%
8	寄存器—寄存器间数据移动	4%
9	调用	1%
10	返回	1%
合 计		96%

根据上述测试统计结果可以看到,进行 RISC 计算机指令集结构的功能设计时,不能简单地着眼于精简指令系统上,更重要的是克服 CISC 结构的缺点,使计算机体系结构更加简单、合理和高效,机器速度更快,程序运行时间缩短,从而提高计算机系统的性能。

因此,进行 RISC 计算机指令集结构的功能设计时,必须遵循如下原则:

- 选取使用频率最高的指令,并补充一些最有用的指令;
- 每条指令的功能应尽可能简单,并在一个机器周期内完成;
- 所有指令长度均相同;
- 只有 Load 和 Store 操作指令才访问存储器,其他指令操作均在寄存器之间进行;
- 以简单有效的方式支持高级语言。

当然,要保证在指令集结构功能设计中遵循上述原则,离不开对高级语言各类语句的使用频率、指令的执行频率以及它们对程序运行速度的影响进行测试统计分析,只有这样,才能在大量定量测试统计分析结果的基础上,设计出最简洁、高效的指令集结构。

2.3.3 控制指令

长期以来,对改变控制流的指令一直都没有统一的术语。1950 年以前,一

般称控制指令为“转移”，从 1960 年 IBM 370 出现以后，开始使用“分支”(branch)这一术语来描述控制指令。后来的一些机器，也引入了其他一些描述控制指令的术语。为统一起见，本书对控制指令的描述统一约定为：当控制指令为无条件改变控制流时，称之为“跳转”，而当控制指令是有条件改变控制流时，称之为“分支”。另外，还可以按照如下四种操作来区分控制流程的各种改变情况，即条件分支(conditional branch)、跳转(jump)、过程调用(call)和过程返回(return)。

上述四种改变控制流程的操作各不相同，因此可能要针对每种操作设计不同的指令。我们首先希望知道这些操作在程序中出现的相对频率。为此，在一台 Load/Store 型指令集结构的机器上执行 SPECint 92 和 SPECfp 92 基准程序，可以得到如图 2.5 所示的测试统计结果。

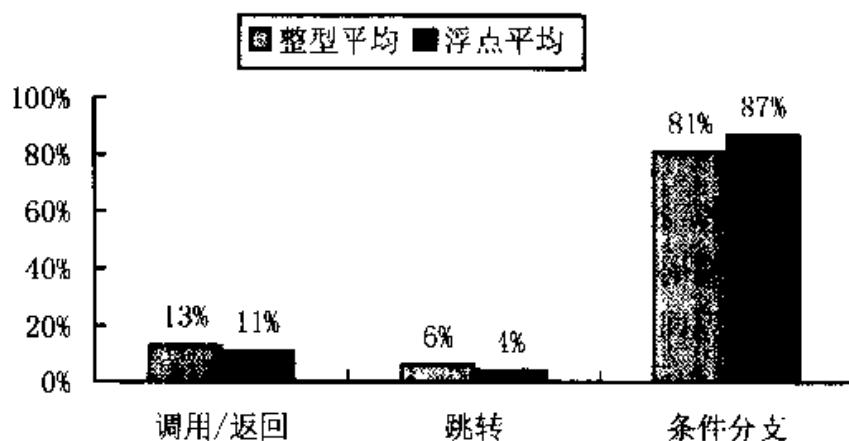


图 2.5 控制指令的使用频率

从图 2.5 可以看出，改变控制流的大部分指令是条件分支指令，因此，在进行条件分支指令设计时，确定如何表示分支条件就显得非常的重要了。表 2.9 列出了当前常用的三种表示分支条件的技术及其优缺点。

表 2.9 表示分支条件的主要技术及其优缺点

表示分支条件的技术	测试分支条件的方法	优点	缺点
条件码(CC)	在程序的控制下，由 ALU 操作设置特殊的位	可以自由设置分支条件	因为必须从一条指令将分支条件信息传送到分支指令，所以 CC 是额外状态，条件码限制了指令顺序
条件寄存器	根据比较结果测试条件寄存器	简单	占用了一个寄存器
比较且分支	比较操作是分支指令的一部分，通常这种比较是受一定限制的	一条指令完成了两条指令的功能	分支指令的操作增多

对于改变控制流的指令来说,除了要指出控制流改变的条件之外,还必须明确指出控制流改变的目标地址。在绝大多数情况下,指令中都会给出目标地址,但是过程返回指令却不能在指令中直接给出返回目标地址,因为对返回操作而言,在编译的时候,其返回目标地址是未知的。

指定目标地址最一般的方法就是在指令中提供一个和程序计数器(PC)的值相加的偏移量。控制指令所采用的这种寻址方式叫做“PC 相对寻址”(PC-relative)。在控制指令中使用 PC 相对寻址方式会带来许多优点,因为控制转移的目标通常离当前指令很近,用相对于当前 PC 值的偏移量来指明目标地址时,可以有效地缩短指令中表示目标地址的字段的长度。另外,使用 PC 相对寻址方式,可以使代码在执行时与它被载入的位置无关,这一特性也叫做“位置无关”(position independence),它能够简化程序的链接,特别是对在执行时才被链接的程序来说,这一特性非常有用。

由于控制指令一般采用 PC 相对寻址方式来确定其转移目标地址,所以关键问题是:转移目标离当前控制指令的偏移量有多大?如果能够知道这一偏移量大小的分布,则对确定控制指令中偏移量字段的长度非常有帮助,同时这也会影响到指令的字长和编码。

对过程的调用和返回而言,除了要改变控制流之外,可能还包括机器状态的保存(至少必须保存过程调用的返回地址)。某些指令集结构提供了保存指令,而某些指令集结构则需编译器来产生机器状态保存指令。

一般有两种策略来保存寄存器的内容:一种是“调用者保存”(caller saving)策略,一种是“被调用者保存”(callee saving)策略。如果采用调用者保存策略,那么在一个调用者调用别的过程时,必须保存调用者所要保存的寄存器,以备调用结束返回后,能够再次访问调用者。如果采用被调用者保存策略,那么被调用的过程必须保存它要用的寄存器,保证不会破坏过程调用者的程序执行环境,并在过程调用结束返回时,恢复这些寄存器的内容。

但是,在某些情况下却必须采用调用者保存策略,这是因为在多个不同的过程中,有可能都要用到公共的全局变量。比如,假设过程 P1,调用过程 P2,两者都使用全局变量 X。现 P1 已经把 X 放置在一个寄存器中了,它就得在调用 P2 之前,确保已经将 X 存放在 P2 知道的位置。但由于存在独立编译过程的可能性,加之编译器的能力所限,即使可以保证 P2 不会改变保存变量 X 的寄存器,但是如果 P2 调用一个将改变该寄存器的另外一个过程 P3,那么当过程调用返回后,由于 P3 已经改变了保存变量 X 的寄存器中的内容,从而改变了 P1 的执行环境。因此当前大多数编译器都保守地采用调用者保存策略,以保存“任何可能”在调用中被访问的寄存器的内容。

在两种策略均可采用的情况下,有些用被调用者保存比较好,有些则用调用

者保存比较好,有的编译器也混合使用这两种方法,但是这将增加实现编译器的复杂性。

2.4 操作数的类型、表示和大小

操作数类型和操作数表示也是软硬件的主要界面之一。操作数类型是面向应用、面向软件系统所处理的各种数据结构;而操作数表示是硬件结构能够识别、指令系统可以直接调用的那些结构。因此,操作数表示所表征的那些操作数类型,是应用软件和系统软件所处理的操作数类型的子集。如何确定这个子集,是体系结构设计者所面临的困难问题之一。

之所以困难,是因为从原理上说,计算机即使只具有最简单的操作数表示,如只有整数(定点)表示法,也可以通过软件方法处理各种复杂的操作数类型,但是这样会大大降低系统的效率。如果各种复杂的操作数类型均包含在操作数表示之中,无疑会大大提高系统的效率,但是所花费的硬件代价也很高。因此,确定操作数表示实际上也是软硬件取舍折衷的问题。

一般来说,操作数的类型主要有:整数(定点)、浮点、十进制、字符串、向量、堆栈等。对这些操作数类型的表示主要有如下两种方法:

1. 操作数的类型可以由操作码的编码指定,这也是最常见的一种方法;
2. 数据可以附上由硬件解释的标记,由这些标记指定操作数的类型,从而选择适当的运算。然而有标记数据的机器却非常少见。

对于某种操作数的类型,一般都会给定其大小。一般的操作数类型大小选择主要有:字节、半字(16位)、单字(32位)、和双字(64位)。

字符可以用 ASCII 码表示,为一个字节大小;整数则几乎都是用二进制补码表示,其大小可以是字节、半字或单字。浮点操作数可以分为单精度浮点(单字大小)和双精度浮点(双字大小),直到 20 世纪 80 年代初期,大多数计算机厂家都一直采用各自的浮点操作数表示方法,然而在过去几年内,IEEE 754 浮点操作数表示标准已经成为大多数新型计算机系统的共同选择。

某些计算机结构还提供了有限的字符串运算功能,比如字符串的比较和移动操作等。字符串操作数类型的数据表示通常是将字符串中的每个字符当作一个字节来看待。对商业应用而言,某些结构也支持十进制操作数类型,其表示方法通常采用“压缩十进制”或“二进制编码十进制”。压缩十进制数据表示法用 4 位二进制数编码数字 0~9,然后将两个十进制数字压缩在一个字节中存储。如果将十进制数字直接用字符串来表示,就叫做“非压缩十进制”表示法。压缩十进制表示法和非压缩十进制表示法之间相互转换的操作分别称为“压缩”与“解压”操作。

在指令集结构设计中,为了确定操作数字段的长度应该为多少位,应该支持哪些数据类型的存取操作等诸多问题,经测试统计 SPECint92 基准程序和 SPECfp92 基准程序对字节、半字、单字和双字四种大小操作数的访问情况,得到如图 2.6 所示的结果。

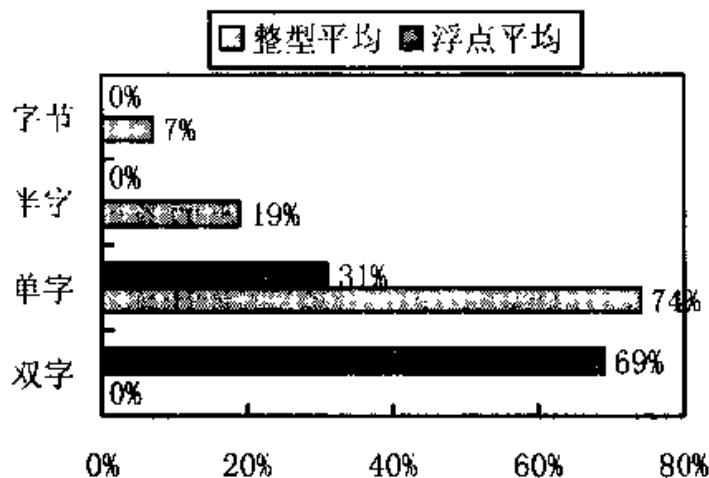


图 2.6 访问不同操作数大小的频率

从图 2.6 中可以看出,基准程序对单字和双字的数据访问具有较高的频率,所以如果选择操作数字段的长度为 32 位,那么它可以有效支持 8、16、32 位整型操作数,以及 32 位浮点操作数的表示。当然,如果定义操作数字段长度为 64 位,则更具有一般性。

2.5 指令集格式的设计

指令集中的每条指令由操作码和地址码组成,指令集格式的设计就是要确定操作码字段和地址码字段的大小及其组合形式,以及各种寻址方式的编码方法。在指令集格式的设计中,体系结构设计者必须在以下三个方面进行折衷:

- 尽可能增加寄存器数目和寻址方式类型;
- 充分考虑寄存器字段和寻址方式字段对指令平均字长的影响,以及它们对目标代码大小的影响;
- 在具体实现中能够容易地处理所设计出的指令集格式。

本节将围绕上述三个方面,说明在指令集格式设计中经常采用的一些技术和方法,至于如何进行具体的折衷,则由体系结构设计者根据具体情况系统地进行综合分析和设计。

2.5.1 寻址方式的表示方法

CPU 在执行指令的时候,首先对指令进行译码,然后根据译码结果确定指令所表示的操作,并根据指令所指定的寻址方式迅速定位指令执行所需要的操作数。寻址方式的表示在指令集格式设计中有着极其重要的地位。

通常,在指令中有两种表示寻址方式的方法。一种是将寻址方式编码于操作码中,由操作码在描述指令操作的同时,也描述了相应操作的寻址方式;一种是为每个操作数设置一个地址描述符,由该地址描述符表示相应操作数的寻址方式(如图 2.7 所示)。

操作码	地址描述符 1	地址码字段 1	...	地址描述符 n	地址码字段 n
-----	---------	---------	-----	-----------	-----------

图 2.7 利用地址描述符表示寻址方式的方法

至于在指令集格式设计中到底选择哪种表示寻址方式的方法,则主要由指令集结构所采用的寻址方式种类及其适用范围,以及操作码与寻址方式之间的独立程度来决定。如果某些指令集结构的指令有 1~5 个操作数,每个操作数有 10 种寻址方式,对于这种大规模的操作数和寻址方式组合,如果将寻址方式表示在操作码中显然不合适,因为它不仅增加了指令条数,导致指令的多样性,而且增加了 CPU 对指令译码的难度。所以在这种情况下通常采用增设地址描述符的方法来描述寻址方式。

然而,对诸如 Load/Store 类型指令集结构的指令而言,由于只有 1~3 个操作数,而且只有有限的几种寻址方式,所以在这些指令集结构中,将寻址方式编码于操作码中是十分合适的。

2.5.2 指令集格式的选择

图 2.8 表明了三种指令集编码格式,其中:第一种是变长编码格式,当指令集结构包含多种寻址方式和操作类型时,这种编码方式可以有效减少指令集结构的平均指令长度,降低目标代码的长度。但是,这种编码格式也会使各条指令的字长和执行时间大不一样。多数 CISC 计算机的指令集结构均采用了这种编码格式。

第二种是固定长度编码格式,它将操作类型和寻址方式组合编码在操作码中,所有指令的长度是固定唯一的。当寻址方式和操作类型非常少时,这种编码格式非常好,它可以有效降低译码的复杂度,提高译码的性能。一般 RISC 计算机的指令集结构,如 DLX、MIPS、Power PC 和 SPARC 等微处理器的指令集结

构,均采用这种类型的编码格式。

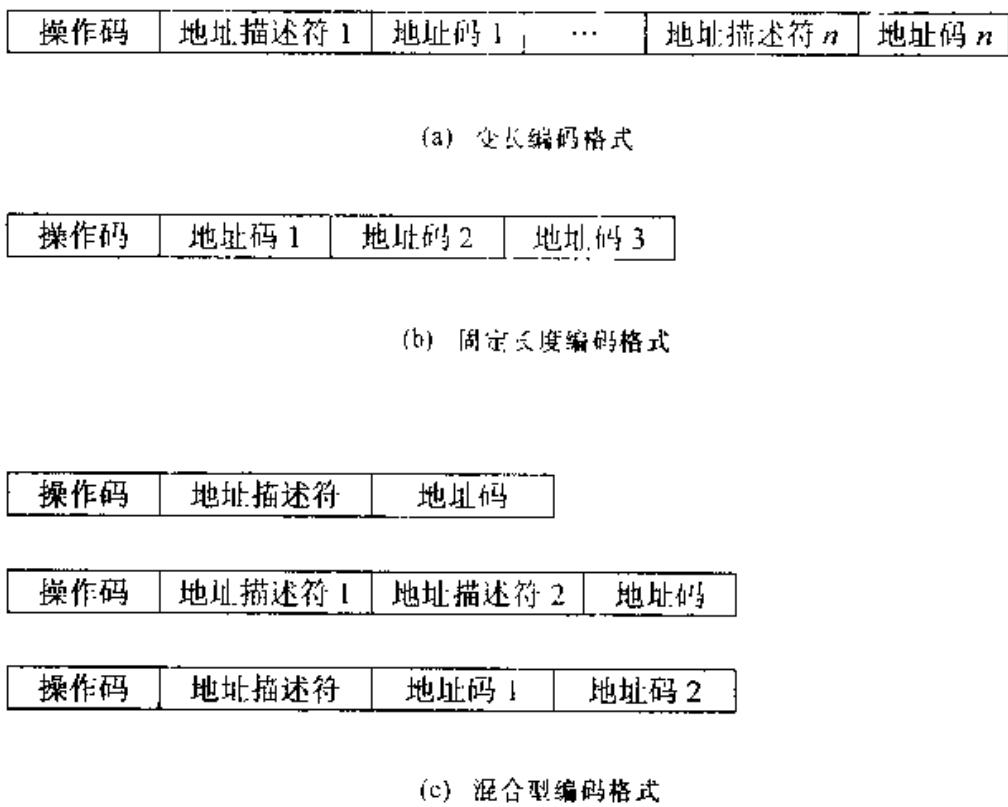


图 2.8 三种基本的指令集编码方式

第三种编码格式为混合型编码格式,其目的是通过提供一定类型的指令字长,期望能够兼顾降低目标代码长度和降低译码复杂度两个目标。它也是指令集结构通常采用的一种编码格式,如 IBM 360/370 和 Intel 80x86 的指令集结构均采用的这种编码格式。

一般来说,如果体系结构设计者感兴趣的是程序的目标代码大小,而不是性能,就可以采用变长编码格式。但如果感兴趣的是性能,而不是程序的目标代码大小,则可以选择固定长度编码格式。当前许多体系结构设计者都选用了固定长度编码格式,虽然在追求短小目标代码方面做出了牺牲,但获得了许多在实现和性能上的好处。

至此,已经对计算机指令集结构设计的基本知识进行了全面的讨论,但是在介绍指令集结构实例——DLX 指令集结构之前,让我们首先看看当前的编译技术和计算机体系结构设计之间的关系。

2.6 编译技术与计算机体系结构设计

一般来说,当前大多数程序都是用高级语言编写的,而在机器上运行的是编

译器生成的程序目标代码。由此可以认为,指令集结构就是编译器的目标,计算机系统的性能是否能充分发挥,在很大程度上受编译器的影响,所以对当今编译技术的深入了解是设计并有效实现指令集结构的关键之一。

本节把计算机体系结构和编译技术紧密结合,并从现代编译技术的观点来讨论指令集结构设计的关键目标,即哪些指令集结构特性可以为生成高质量的程序目标代码提供保证?如何才能非常容易地为指令集结构编写出相应的高效编译器?

2.6.1 现代编译器的结构和相关技术

现代编译器结构大致如图 2.9 所示,它主要由 2~4 遍(pass)组成,遍可以简单地看成是编译的一个阶段,在这个阶段中,编译器读入整个程序,并完成对整个程序的相应转换。编译器的优化程度越高,其遍数也就越多。

编译器设计者的首要目标就是要保证编译器的正确性,要使所有有效的程序能够被正确地编译。第二个目标是使编译出的代码在目标机器上的执行速度要尽可能快。另外,还可能有其他一些编译器设计者所追求的目标,比如编译的速度、对程序调试的支持、以及各种语言之间互操作的可能性等。

在正常情况下,编译器的一遍扫描会逐渐将程序从高级而抽象的表示转换成低级的表示形式,最终达到机器指令表示形式。这种层次转换结构有助于管理中间代码之间的复杂转换过程,并且容易写出正确无误的编译器。

实际上,编译器对程序进行优化编译工作量的多少和编译器设计的复杂度等问题之间也存在一个折衷关系。编译器对程序进行优化的遍数越多,编译器设计就越复杂,程序的编译时间就会越长。虽然上述层次转换结构有益于降低编译器设计的复杂度,但是这也表示编译器必须按照事先安排的顺序执行各种转换。

在图 2.9 所示的优化编译器结构中,有些高级优化在并不知道结果代码会是什么的情况下就已经完成了,并且一旦完成了这样的优化转换,编译器便无法再重复以前的优化转换。从全局优化的角度来说,这种限制并不合适,但是如果取消这种限制,那么无论是在编译时间上,还是在编译器设计的复杂度上都是不允许的。编译器设计者将这一限制称为“按序转换问题”(phase-ordering problem)。

更具体一些,可以用“全局公共子表达式消去法”的例子来说明这个问题。“全局公共子表达式消去法”的优化方法是设法找出在同一表达式中出现的公共子表达式,并把第一次公共子表达式计算出的值存放在一个临时变量中,然后利用该临时变量的值消去表达式中的其他相同公共子表达式的计算,显然,这种优化可以有效地降低一个表达式的计算量。但是,为了使这种优化真正起到提高程序执行速度的作用,该临时变量一定要分配给一个寄存器。如果将该临时变

量存放在存储器中,那么以后从存储器中重新读入该临时变量的值所需要的访存开销将会把这种优化所带来的好处抹杀掉。

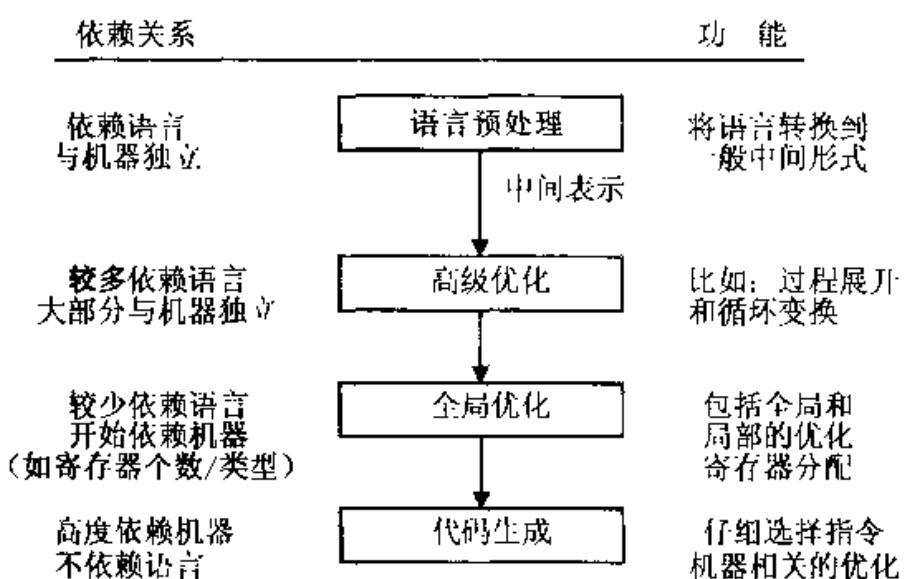


图 2.9 现代编译器的简要结构

按序转换的限制将会使这个问题变得更加复杂,因为寄存器分配通常是在全局优化结束和代码生成之前做的,所以完成这种“全局公共子表达式消去”优化的优化器就必须假设后面的寄存器分配器将会把该临时变量分配给某一寄存器。

实际上,无论是在提高机器代码速度方面,还是在实现其他优化技术方面,寄存器分配都是一个核心问题。当前在编译器中所采用的寄存器分配算法均是基于“图着色法”发展而来的,图着色法的基本思想是根据将要分配给寄存器的各个可能候选变量和它们的使用范围构造相干图,然后再用该图来分配寄存器。

如图 2.10 所示,分配给寄存器的每个可能候选变量(A、B、C、D)均对应于图中的一个结点(图 2.10(b)),结点之间的连线则表示各个变量的使用范围,这样由候选变量结点和表示各个变量使用范围的连线就构成了各个变量之间的相干图。在相干图的基础上,遵循“相邻结点不能着以相同颜色,而非邻的结点可以着以相同颜色”的原则对各个结点进行着色,可以得到如图 2.10(c)所示的着色图,可以看出图中的结点着有两种颜色,一种是白色,一种是灰色。我们可以将寄存器 R1 分配给着白色结点的候选变量,而将寄存器 R2 分配给着灰色结点的候选变量。由此可以得到如图 2.10(d)所示的寄存器分配结果。从寄存器分配结果可以看出,不会将重叠使用的变量分配给相同的寄存器,从而可以保证合理地使用寄存器。

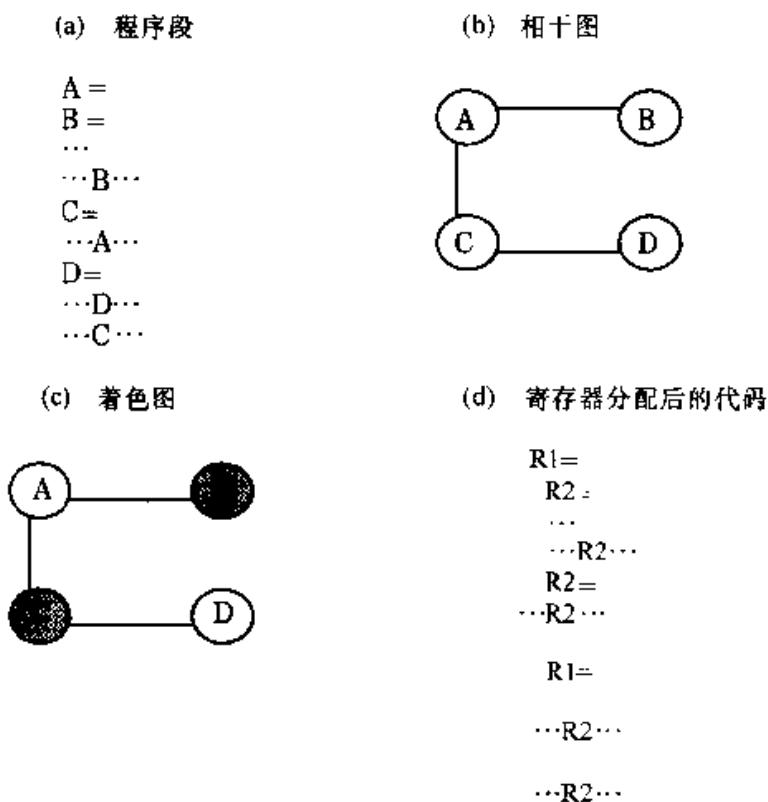


图 2.10 图着色法

图着色问题是一个 NP 完全问题,但是人们已经提出了一些有效的启发式图着色算法。实际应用表明,在利用图着色法进行寄存器分配的时候,如果至少有 16 个通用寄存器(当然,多多益善)用来分配变量,则可以得到效果非常好的寄存器分配结果。然而,当寄存器个数很少时,图着色法就可能无法顺利完成分配寄存器的任务。

另外,根据各种转换的特点,可以将当前编译器所实现的优化分为如下几种类型:

1. 高级优化(high-level optimizations):通常是对原始程序进行优化,并将优化结果送到后续的优化扫描中。
2. 局部优化(local optimizations):仅仅是一系列在线性的程序段(也称为基本块)中进行的优化。
3. 全局优化(global optimizations):在局部优化的基础上,考虑到各种分支情况,对循环和分支进行一系列的优化转换。
4. 寄存器分配(register allocation)。
5. 基于机器的优化(machine-dependent optimizations):目的是充分利用机器结构的某些特点进行代码优化。

表 2.10 列出了在上述各种优化中所采用的一些实际优化方法。表 2.11 列出了采用一些优化方法后程序运行速度和未采取这些优化方法前程序运行速度

相比在性能上提高的百分比。

表 2.10 各种优化过程所采用的一些实际优化方法

优化类型和方法	说 明
高级优化	处于或接近源代码级别,与机器独立无关
过程集成	用过程体代替过程调用语句
局部优化	线性代码序列内的优化
消去公共子表达式	设法找出在同一表达式中出现的公共子表达式,并把第一次公共子表达式计算出的值存放在一个临时变量中,然后它利用该临时变量值消去表达式中的其他相同公共子表达式的计算
常数传递	将所有被赋值为常数的变量用该常数的值代替
降低堆栈的高度	对表达式进行重新组织,尽量减少表达式求值时所需要的资源
全局优化	非线性代码序列的优化
消去公共子表达式	和局部优化中的消去公共子表达式优化类似
拷贝传递	如果变量 A 已经赋值为 X,则用 X 代替所有地方的 A
代码移动	如果在循环中的某段代码在每次循环中均是计算相同的值,则将这段代码移到循环的外部
消去索引变量	简化/消去在循环中的数组地址计算
基于机器的优化	依赖于机器的特点
降低计算量	比如,可以用加操作和移位操作来代替一个常数的乘操作
流水线调度	重新组织指令序列以提高流水线性能
分支偏移的优化	选择能够达到分支目标的最短分支偏移量

表 2.11 优化对程序执行性能的影响

进行的优化	性能提高的百分比
只进行过程集成	10%
只进行局部优化	5%
局部优化 + 寄存器分配	26%
局部优化 + 全局优化	14%
局部优化 + 全局优化 + 寄存器分配	63%
局部优化 + 全局优化 + 寄存器分配 + 过程集成	81%

2.6.2 现代编译技术对计算机体系结构设计的影响

编译器和高级语言之间的相互作用对程序如何使用指令集结构有着显著的影响,要对这些影响有一个明确的认识,必须要明确以下三个重要问题:

1. 如何分配和寻址变量?用多少个寄存器分配变量比较合适?
2. 优化技术对指令使用频度有何影响?
3. 程序中有哪些控制指令?其使用频度如何?

为了回答第一个问题,首先让我们考察一下当前高级语言用来分配数据的三个不同区域,这三个区域分别是:

1. 堆栈(stack):堆栈主要用来分配局部变量。堆栈内容的变化分别体现了过程的调用和返回情况。堆栈中的对象是相对于栈顶指针来寻址的,并且多数是简单变量。堆栈主要用来记录程序运行环境,而不是用来进行表达式求值。
2. 全局数据区(global data area):全局数据区用来分配被静态说明的对象,如全局变量和常量,在这些对象中,数组和其他联合类型数据结构占有相当大的比例。
3. 堆(heap):堆主要用来动态分配一些不适合放在堆栈中的对象,这些对象一般都不是简单变量。要用指针才能访问堆中的对象。

一般来说,可以将寄存器分配给堆栈和全局数据区中的对象。而对于那些要分配给堆的对象,由于要通过指针才能访问到这些对象,所以不能将寄存器分配给它们。

就当前编译器技术而言,许多堆变量都是有别名的。例如,考虑如下代码序列,其中“&”表示返回一个变量的地址,“*”则是取该地址处保存的值:

p = &a	取变量 a 的地址,并将其保存在 p 中;
a = ...	直接对变量 a 进行赋值;
* p = ...	利用指针 p 对 a 进行赋值;
... a ...	访问 a。

在上述代码序列中,就不能够将变量 a 分配给寄存器,否则会生成错误的代码。别名会导致很多问题,这主要是因为编译器要确定某一指针到底指向哪个对象是一件十分困难的事情,有时甚至是不可能的事情。所以,当前的多数编译器一般都采用保守的寄存器分配策略。比如对某个过程来说,只要可能有指针指向该过程中的一一个局部变量,编译器就不会将寄存器分配给该过程的任何一个局部变量。

另外,对某些全局变量和一些堆栈变量而言,它们也可能有“别名”,即有多种方法指明访问这些变量的地址,所以也不能将寄存器分配给它们,否则将导致非法分配。

由于编译器不能将程序中所有的变量都分配给寄存器,所以在程序执行过程中,还存在如下几种类型的存储器访问:

1. 未分配的访问:虽然可以将某些变量分配给寄存器,但是由于机器中寄存器数目限制等诸多原因,并没有将其分配给寄存器,由此而带来的对这些变量的存储器访问;
2. 对全局变量的访问:由于未将寄存器分配给某些全局变量,由此而带来的对这些全局变量的存储器访问;
3. Load/Store 访问:主要是将存储器中的内容读入寄存器,或将寄存器的内容写回存储器,这里我们称之为 Load/Store 访问;
4. 必要的堆栈变量访问:对于一些具有别名的堆栈变量,由于不能将寄存器分配给它们,所以必须通过对存储器的访问完成对这些堆栈变量的访问;
5. 计算型的访问:包括所有对堆变量的存储器访问,以及由指针和数组索引实现的存储器访问等。

如果在一台在 Load/Store 机器上运行 gcc 和 Tex 基准程序,在为机器配置不同寄存器个数的情况下,可以测得如图 2.11 所示的上述各种类型的存储器访问次数占所有存储器访问次数的百分比。

从图 2.11 中可以看出,上述第 2 到第 5 种类型的存储器访问次数并不会随着机器中配置的寄存器个数的变化而变化,因为它们所要访问的变量都无法分配给寄存器。但是未分配类型的存储器访问次数,却会随着寄存器个数的增多而减少。由此可见,在进行计算机体系结构设计时,为机器设置多个寄存器对减少访问存储器的开销、提高程序的执行速度是非常有益的。

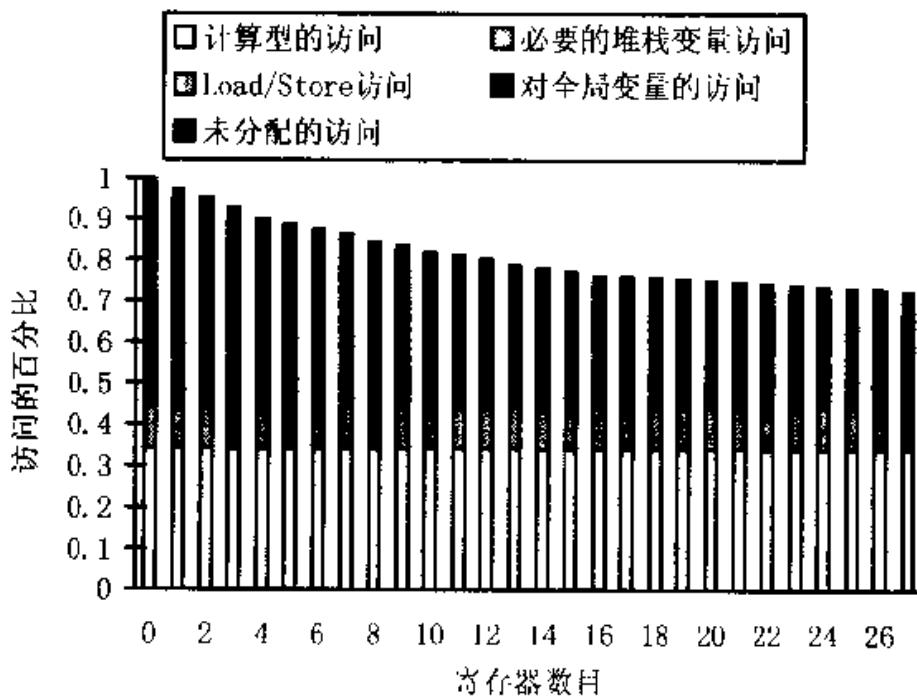


图 2.11 各类存储器访问百分比随寄存器数目变化情况

另外,从图中还可以看到,计算型的访问占有相当大的比例,因此从计算机体系结构的角度考虑如何有效支持计算型的存储器访问(诸如数组访问等),也是值得仔细研究的问题。

那么,编译器的优化技术对指令使用频度又有何影响呢?程序中有哪些控制指令,其使用频率如何?在MIPS机器的编译器上采用不同的优化级别分别对基准程序集SPEC92中的hydro2d和li两个基准程序进行编译,可以得到如图2.12所示的各类指令数目随优化级别变化的情况。

图2.12中的优化级别0表示不进行优化;优化级别1表示编译器进行局部优化、代码调度和局部寄存器分配;优化级别2表示编译器进行全局优化、循环转换和全局寄存器分配;优化级别3表示除了包含优化级别2的优化工作之外,还包含过程集成的优化工作。

从图2.12可以看出,优化可以明显地降低程序中指令的条数,但是却增加了分支/调用等控制指令在程序中所占的比例,即编译器优化技术可以有效减少整型ALU指令的条数,但是却难以减少分支指令的条数。因此在计算机体系结构设计中应该注重支持对控制指令的加速。

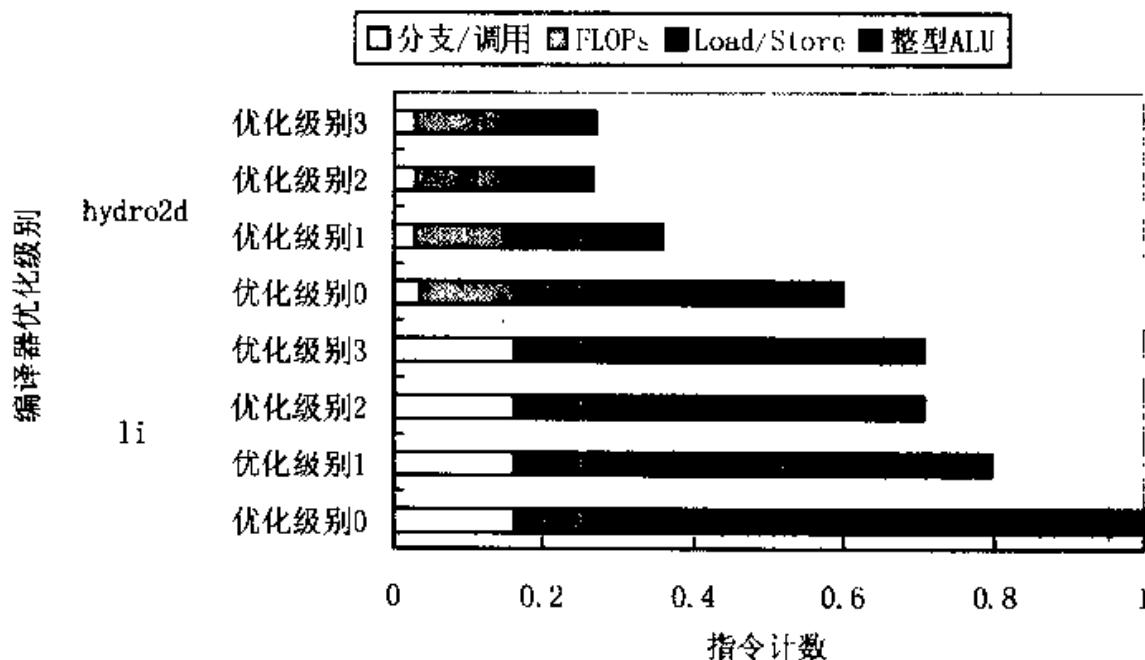


图 2.12 指令计数随优化级别的变化情况

2.6.3 计算机体系结构对当前编译技术的影响

当前编译器的复杂之处并不在于翻译像 $A = B + C$ 这样简单的表达式。大多数程序的局部模块都是十分简单的,编译器只需对这些局部模块进行一些简单的翻译就够了。但是由于当前程序规模越来越大,而且程序各个局部模块之

间的相互关系十分复杂,加之当前编译器多数采用按序转换结构,所以编译器的真正复杂之处在于决定哪一种代码顺序最为理想,即当前编译技术的难点在于优化。

编译器设计者通常在编译器设计上都遵循一个基本原则,那就是“对程序中经常出现的部分要尽量加速,而对较少出现的部分要力求正确”。也就是说,如果知道哪些部分在程序中经常出现,哪些部分在程序中很少出现,并且为它们生成代码一样简单的话,则对很少出现部分的代码质量就可以不作过高的要求,不过一定要保证其正确性。

指令集结构的如下一些设计原则非常有助于编译器的设计。但是也不能将它们看作是绝对不能侵犯的原则,只是如果遵循这些原则,设计编译器就会更加容易,编译器也更易于生成正确高效的代码。

1. “规整性”:如果希望指令集结构具有规则性,那么就要求指令集的三个主要元素(操作、数据类型和寻址方式)必须正交(orthogonal)。所谓正交,就是上述三种元素中的任何两个元素在指令集结构中彼此独立无关。比如,对指令集结构中的操作和寻址方式两种元素而言,如果每一种操作都可采用任何一种寻址方式,并且所有寻址方式均是可用的,则称操作和寻址方式是正交的。指令集结构的这一性质有利于简化代码生成。如果指令集结构不具备这一性质,就可能会带来许多问题。比如,指令集结构限制某一类指令可以使用的寄存器,那么可能会使编译器在进行代码生成的时候,发现有许多“空闲”的寄存器,却唯独没有该指令可以使用的寄存器。

2. “提供基本指令,而非解决方案(solution)”:试图用指令集结构去“匹配”某种高级语言的做法是十分不可取的。如果试图用指令集结构对某种高级语言提供极大的支持,可能会导致这种指令集结构只支持该高级语言,或者是指令集结构中的指令对该高级语言的正确有效实现支持太多,而对其他高级语言的正确有效实现支持不够。

3. “简化方案的折衷取舍标准”:编译器设计者所面临的最棘手的问题就是要计算出每段代码采用哪一种指令顺序才是最高效的。以前主要是用指令条数和目标代码长短作为取舍标准,但是正如我们在上一章中所看到的,这两个标准并不能真实反映代码执行效率,因为它们并没有包含每种指令的执行时间。如果机器具有 Cache 和流水线,那么取舍标准就会变得更加复杂,因为即使是同一指令在同一机器上执行,其两次执行时间也可能大不一样。无论如何,计算机体系结构设计者所做的任何一件事只要能够帮助编译器设计者了解各种代码序列的执行效率和代价,就非常有助于编译器的优化。

4. “对于在编译时就已经知道的量,提供能够将其变为常数的指令”:编译器设计者并不希望机器在运行程序时还要去解释一个在编译时就已经知道大小

的变量。

2.7 DLX 指令集结构

本节将讨论一种称为 DLX 的 Load/Store 型指令集结构。DLX 是一种多元未饱和型指令集结构。称之为多元未饱和型结构,是因为它不仅体现了当今多种机器 (AMD29K、DEC station 3100、HP850、IBM 801、Intel i860、MIPS M/120A、MIPS M/1000、Motorola 88k、RISC I、SGI4D/60、SPARC station 1、Sun 4/110、Sun 4/260 等) 指令集结构的共同特点,而且它还将会体现未来一些机器的指令集结构的特点。这些机器的指令集结构设计思想都和 DLX 指令集结构的设计思想十分相似,它们都强调:具有一个简单的 Load/Store 指令集;注重指令流水效率;简化指令的译码;高效支持编译器。

2.7.1 DLX 指令集结构

1. DLX 中的寄存器

DLX 中有 32 个通用寄存器(GPRs),分别将其命名为 R0,R1,⋯,R31。每个通用寄存器的长度为 32 位。寄存器 R0 的值总是为 0,稍后我们将会看到如何利用该寄存器实现一些有用的操作。

另外,DLX 还有 32 个浮点寄存器(FPRs),分别将其命名为 F0,F1,⋯,F31。每个浮点寄存器的长度为 32 位。这些浮点寄存器可以用来保存 32 位的单精度浮点数,或者通过相邻两个浮点寄存器奇偶对 $F_i F_{i+1}$ ($i = 0, 2, 4, \dots, 30$) 来保存双精度浮点数,这种组合而成的 64 位双精度浮点寄存器在 DLX 中分别被命名为 F0,F2,⋯,F28,F30。

另外,还有一些特殊的寄存器(比如用来保存浮点操作结果信息的浮点状态寄存器)可以和通用寄存器相互进行数据传送。DLX 也提供了在 FPR 和 GPR 之间移动数据的指令。

2. DLX 的数据类型

DLX 提供了多种长度的整型数据和浮点数据。对整型数据而言,有 8 位、16 位和 32 位多种长度;对浮点数据而言,有 32 位单精度浮点数和 64 位双精度浮点数。浮点数据表示采用的是 IEEE 754 标准。DLX 操作都是对 32 位整型数据以及 32 位或 64 位浮点数据进行的。

当 8 位和 16 位整型数据载入到寄存器中时,用 0 或数据的符号位来填充 32 位通用寄存器中的剩余位。一旦载入,就将其作为 32 位整型数据进行处理。

3. DLX 的寻址方式和数据传送

DLX 提供了寄存器寻址、立即值寻址、偏移寻址和寄存器间接寻址四种寻址

方式。寄存器寻址字段的大小为 5 位,用来标识 32 个通用寄存器或浮点寄存器。

DLX 的存储器地址采用的是高端字节表示顺序,存储器按字节寻址,其地址宽度为 32 位。由于 DLX 是一种 Load/Store 结构,所以它通过寄存器(通用寄存器和浮点寄存器)和存储器之间的数据传送操作完成对存储器的访问。

由于 DLX 支持上述数据类型,所以对通用寄存器而言,相应的存储器访问数据大小有 8 位、16 位和 32 位;而对浮点寄存器而言,相应的存储器访问数据大小有 32 位(单精度浮点数)和 64 位(双精度浮点数)。值得注意的是,DLX 的所有存储器访问均需对齐。

4. DLX 的指令格式

因为 DLX 只有四种寻址方式,所以将其寻址方式编码在操作码中。为了简化指令译码,并充分发挥流水线的效率,所有 DLX 指令的字长均是 32 位,其中用 6 位表示操作码。DLX 中各种类型指令的格式如图 2.13 所示。

I 型指令

6	5	5	16
操作码 op	rs1	rd	立即值

字节、半字、字的载入和储存;

$rd \leftarrow rs1 \text{ op 立即值}$ 。

R 型指令

6	5	5	5	11
操作码 op	rs1	rs2	rd	Func

寄存器—寄存器 ALU 操作: $rd \leftarrow rs1 \text{ Func } rs2$;

函数对数据的操作进行编码: 加、减 …;

对特殊寄存器的读/写和移动。

J 型指令

6	26
操作码 op	与 PC 相加的偏移量

跳转, 跳转并链接, 从异常 (exception) 处自陷和返回。

图 2.13 DLX 的指令格式布局

5. DLX 中的操作

DLX 除了支持上面提到的一些简单操作之外,还支持其他一些操作。DLX 指令中的操作可以分为四种类型,即 Load 和 Store 操作、ALU 操作、分支和跳转

操作、浮点操作。

在后面对上述各种操作类型进行论述的过程中,将采用 C 语言的扩充形式来表示指令的含义,为此,首先对指令含义的一些扩充表示方法做如下约定,并且在以后各章节的相关说明中也都将遵循这些约定:

① 符号“ \leftarrow ”表示数据传送操作,其后附带一个下标 n ,也即“ \leftarrow_n ”表示传送一个 n 位数据。

② 符号“##”用来表示两个域的串联操作,可以出现在数据传送操作的任何一边。

③ 域的下标用来表明从该域中选择某一位。域中的位从最高位开始标记,并且起始标记为 0。下标可以是一个单独的数字,如 $\text{Regs}[R4]_0$ 表示选择寄存器 R4 中内容的符号位;下标也可以是一个范围,如 $\text{Regs}[R3]_{24..31}$ 表示选择寄存器 R3 中内容的最低一个字节。

④ 上标表示复制一个域,如 0^{24} 可以得到一个 24 位全为 0 的域。

⑤ 变量 Mem 用来表示存储器中的一个数组,存储器按照字节寻址,它可以传送任何数目的字节。

为了进一步说明上述约定表示方法的用途,现设 R8 和 R10 均为 32 位寄存器,那么, $\text{Regs}[R10]_{16..31} \leftarrow_{16} (\text{Mem}[\text{Regs}[R8]]_0)^8 ## \text{Mem}[\text{Regs}[R8]]$ 的含义是:以 R8 中的内容作为存储器地址寻址存储器相应单元的字节,将该字节的符号扩展形成 8 位的域,并和该字节的值串联形成一个 16 位的值,然后将该值传送到寄存器 R10,保存在寄存器 R10 的低 16 位,而寄存器 R10 的高 16 位保持不变。

下面对 DLX 中的四种操作类型分别进行论述:

(1) Load 和 Store 操作

可以对 DLX 的所有通用寄存器和浮点寄存器进行 Load(载入)和 Store(储存)操作,但是对通用寄存器 R0 的 Load 操作没有任何效果。表 2.12 给出了载入和储存指令的一些实例。

(2) ALU 操作

在 DLX 中,所有的 ALU 指令都是寄存器 - 寄存器型指令,其运算包含了简单的算术和逻辑运算,如加、减、AND、OR、XOR 和移位。另外,DLX 还允许所有这些指令对立即值进行操作,立即值以 16 位符号扩展形式出现。LHI(Load 高位立即值)操作将立即值载入到一个寄存器的高半部分,而该寄存器的低半部分则设置为 0。这样就可以通过两条 Load 指令构造一个 32 位的常数。

表 2.12 DLX 中 Load 和 Store 指令实例

指令实例	指令名称	含义
LW R1,30 (R2)	载入整型字	$Regs[R1] \leftarrow_{32} Mem[30 + Regs[R2]]$
LW R1,1000 (R0)	载入整型字	$Regs[R1] \leftarrow_{32} Mem[1000 + 0]$
LB R1,40 (R3)	载入字节	$Regs[R1] \leftarrow_{32} (Mem[40 + Regs[R3]]_0)^{24} \# \# Mem[40 + Regs[R3]]$
LBU R1,40 (R3)	载入无符号字节	$Regs[R1] \leftarrow_{32} 0^{24} \# \# Mem[40 + Regs[R3]]$
LH R1,40 (R3)	载入整型半字	$Regs[R1] \leftarrow_{32} (Mem[40 + Regs[R3]]_0)^{16} \# \# Mem[40 + Regs[R3]] \# \# Mem[41 + Regs[R3]]$
LF F0,50 (R3)	载入单精度浮点	$Regs[F0] \leftarrow_{32} Mem[50 + Regs[R3]]$
LD F0,50 (R2)	载入双精度浮点	$Regs[F0] \# \# Regs[F1] \leftarrow_{64} Mem[50 + Regs[R2]]$
SW 500 (R4),R3	储存整型字	$Mem[500 + Regs[R4]] \leftarrow_{32} Regs[R3]$
SF 40(R3),F0	储存单精度浮点	$Mem[40 + Regs[R3]] \leftarrow_{32} Regs[F0]$
SD 40 (R3),F0	储存双精度浮点	$Mem[40 + Regs[R3]] \leftarrow_{32} Regs[F0]$ $Mem[44 + Regs[R3]] \leftarrow_{32} Regs[F1]$
SH 502 (R2),R31	储存整型半字	$Mem[502 + Regs[R2]] \leftarrow_{16} Regs[R31]_{16..31}$
SB 41(R3),R2	储存整型字节	$Mem[41 + Regs[R3]] \leftarrow_8 Regs[R2]_{24..31}$

正如上面所提到的, R0 主要用来合成一些有用的操作。比如, Load 一个常数就可以看作是一次简单的立即值加操作, 其中一个源操作数是 R0; 寄存器—寄存器间的数据移动也可以看作是一次简单的加, 其中一个源操作数是 R0。这两个操作可以分别用 LI 和 MOV 表示。

在 DLX 指令集中, 还有一些寄存器比较指令(=、≠、<、>、≤、≥), 如果比较结果为真, 这些指令就在目标寄存器中填入 1(表示真), 否则填入 0(表示假)。因为这些比较操作指令要对目标寄存器进行“设置”, 所以也称它们为“设置相等”、“设置不等”、“设置小于”等指令。DLX 同样也提供了这些比较指令的立即值形式, 表 2.13 给出了 ALU 操作指令的一些实例。

(3) 分支和跳转操作

在 DLX 中, 对程序流程的控制是通过一些跳转和分支指令来实现的。根据描述目标地址的方法和是否链接可以将跳转操作指令分为四种类型。其中两种类型的跳转指令用带符号位的 26 位偏移量加上程序计数器的值来确定跳转的目标地址。另外两种类型的跳转指令则指定一个寄存器, 由寄存器中的内容决定跳转的目标地址。跳转有两种类型, 一种是简单跳转, 另一种是跳转并链接(用于过程调用)。后者将返回一个地址, 即将下一条顺序指令地址(返回地址)

保存在寄存器 R31 中。

表 2.13 ALU 指令实例

指令实例	指令名称	含义
ADD R1,R2,R3	加	$Regs[R1] \leftarrow Regs[R2] + Regs[R3]$
ADDI R1,R2,#3	和立即值相加	$Regs[R1] \leftarrow Regs[R2] + 3$
LHI R1,#42	载入高位立即值	$Regs[R1] \leftarrow 42 \# 0^{16}$
SLLI R1,R2,#5	逻辑左移的立即值形式	$Regs[R1] \leftarrow Regs[R2] \ll 5$
SLT R1,R2,R3	设置小于	$if (Regs[R2] < Regs[R3]) Regs[R1] \leftarrow 1 \text{ else } Regs[R1] \leftarrow 0$

DLX 中的所有分支指令都是条件分支指令, 其源操作数寄存器中包含了一个数值或某个比较结果。分支指令测试该源操作数寄存器中的值是 0 还是非 0, 从而决定分支是否成功。分支目标地址由一个带符号的 26 位偏移量加上程序计数器的值来确定, 分支目的地址指向下一条要执行的指令。表 2.14 给出了一些典型的分支和跳转指令。

表 2.14 典型的分支和跳转指令

指令实例	指令名称	含义
J name	跳转	$PC \leftarrow name;$ $((PC + 4) - 2^{25}) \leqslant name \leqslant ((PC + 4) + 2^{25})$
JAL name	跳转并链接	$Regs[R31] \leftarrow PC + 4; PC \leftarrow name;$ $((PC + 4) - 2^{25}) \leqslant name \leqslant ((PC + 4) + 2^{25})$
JALR R2	寄存器型跳转并链接	$Regs[R31] \leftarrow PC + 4; PC \leftarrow Regs[R2];$
JR R3	寄存器型跳转	$PC \leftarrow Regs[R3];$
BEQZ R4,name	“等于 0”分支	$if (Regs[R4] == 0) PC \leftarrow name;$ $((PC + 4) - 2^{15}) \leqslant name \leqslant ((PC + 4) + 2^{15})$
BNEZ R4,name	“不等于 0”分支	$if (Regs[R4] != 0) PC \leftarrow name;$ $((PC + 4) - 2^{15}) \leqslant name \leqslant ((PC + 4) + 2^{15})$

(4) 浮点操作

在 DLX 中, 浮点指令的操作数来源于浮点寄存器, 同时该浮点指令还指明了相应的操作是单精度浮点操作还是双精度浮点操作。

DLX 的浮点操作有: 加、减、乘、除。后缀 D 代表双精度浮点操作, 而后缀 F

代表单精度浮点操作(如: ADDD、ADDF、SUBD、SUBF、MULTD、MULTF、DIVD、DIVF)。值得提出的是,DLX 的浮点比较操作将设置浮点状态寄存器中的位,如果比较结果为真,则将该位设置为 1;如果比较结果为假,则将该位设置为 0。浮点分支指令 BFPT 和 BFTF 则测试该寄存器的值以决定分支是否成功。

另外,操作 MOVF 将一个单精度浮点寄存器的内容拷贝至另一个单精度浮点寄存器; MOVD 则将一个双精度浮点寄存器的内容拷贝至另一双精度寄存器; MOVFP2I 和 MOVI2FP 操作则是在一个浮点寄存器和通用寄存器之间移动数据。如果要将一个双精度浮点数移入两个通用寄存器则需要两条指令。另外,DLX 还提供了在 32 位浮点寄存器中进行整数乘除操作的指令。

表 2.15 列出了 DLX 所有指令及其含义。为了明确哪些指令是最常用的,分别用 SPECint92 和 SPECfp92 基准程序集对 DLX 进行统计,可以分别得到表 2.16 和表 2.17 的指令使用频率测试统计结果。对于测试统计结果中使用频率大于 1% 的指令,以直方图的形式分别表示在图 2.14 和图 2.15 中。

表 2.15 DLX 中的所有指令及其含义

指令类型	操作码	含义
数据传送	LB,LBU,SB	载入字节,载入无符号字节,储存字节
	LH,LHU,SH	载入半字,载入无符号半字,储存半字
	LW,SW	载入字,储存字
	LF,LD,SF,SD	载入单精度浮点数,载入双精度浮点数,储存单精度浮点数,储存双精度浮点数
	MOVI2S, MOVS2I	将通用寄存器中的内容移入特殊寄存器,将特殊寄存器中的内容移入通用寄存器
	MOVF, MOVD	将一个单精度/双精度浮点寄存器的内容拷贝到另一个单精度/双精度浮点寄存器
	MOVFP2I, MOVI2FP	将 32 位浮点寄存器中的内容移入整型寄存器,将 32 位整型寄存器中的内容移入浮点寄存器
算术/逻辑	ADD, ADDI, ADDU, ADDUI	带符号加,带符号立即值加,无符号加,无符号立即值加
	SUB, SUBI, SUBU, SUBUI	带符号减,带符号立即值减,无符号减,无符号立即值减

续表

指令类型	操作码	含义
算术/逻辑	MULU, MULTU, DIV, DIVU	带符号乘, 无符号乘, 带符号除, 无符号除
	AND, ANDI	与, 和立即值与
	OR, ORI, XOR, XORI	或, 和立即值或, 异或, 和立即值异或
	LHI	载入高位立即值
	SLL, SRL, SRA, SLLI, SRLI, SRAI	包含了立即值(S_I)和变量(S_)形式的移位操作, 移位有: 逻辑左移, 逻辑右移和算术右移
控制	S_, S_I	设置条件, “_”可以是 LT, GT, LE, GE, EQ, NE
	BEQZ, BNEZ	根据指定通用寄存器的内容等于/不等于 0 分支
	BFPT, BFPPF	测试浮点状态寄存器中的比较位为真/假进行分支
	J, JR	跳转, 基于寄存器的跳转
	JAL, JALR	跳转并链接, 基于寄存器的跳转并链接
	TRAP	转换到操作系统
	RFE	从异常恢复用户模式
浮点	ADD, ADDF	双精度浮点加, 单精度浮点加
	SUBD, SUBF	双精度浮点减, 单精度浮点减
	MULTD, MULTF	双精度浮点乘, 单精度浮点乘
	DIVD, DIVF	双精度浮点除, 单精度浮点除
	CVTF2D, CVTF2I, CVTD2F, CTD2I, CVTI2F, CVTI2D	转换指令, CVTx2y 表示从类型 x 转换到类型 y, 其中 x 和 y 可以是 I(整型)、D(双精度浮点)、F(单精度浮点)
	- D, - F	双精度浮点和单精度浮点比较, “_”可以是 LT, GT, LE, GE, EQ, NE, 根据比较结果设置浮点状态寄存器中的位

表 2.16 基于 SPECint92 基准程序集的指令使用频率测量统计结果

指令	compress	eqntori	Espresso	gcc(cc1)	li	整型平均
载入	19.8%	30.6%	20.9%	22.8%	31.3%	26%
存储	5.6%	0.6%	5.1%	14.3%	16.7%	9%
加	14.4%	8.5%	23.8%	14.6%	11.1%	14%
减	1.8%	0.3%		0.5%		0%
乘				0.1%		0%

续表

指令	compress	eqntott	Espresso	gcc(cc1)	li	整型平均
除						0%
比较	15.4%	26.5%	8.3%	12.4%	5.4%	13%
载入立即值	8.1%	1.5%	1.3%	6.8%	2.4%	3%
条件分支	17.4%	24.0%	15.0%	11.5%	14.6%	16%
无条件分支	1.5%	0.9%	0.5%	1.3%	1.8%	1%
调用	0.1%	0.5%	0.4%	1.1%	3.1%	1%
返回,跳转	0.1%	0.5%	0.5%	1.5%	3.5%	1%
移位	6.5%	0.3%	7.0%	6.2%	0.7%	4%
与	2.1%	0.1%	9.4%	1.6%	2.1%	3%
或	6.0%	5.5%	4.8%	4.2%	6.2%	5%
其他(异或,非)	1.0%		2.0%	0.5%	0.1%	1%
载入浮点数						0%
存储浮点数						0%
浮点加						0%
浮点减						0%
浮点乘						0%
浮点除						0%
浮点比较						0%
浮点寄存器移动						0%
其他浮点操作						0%

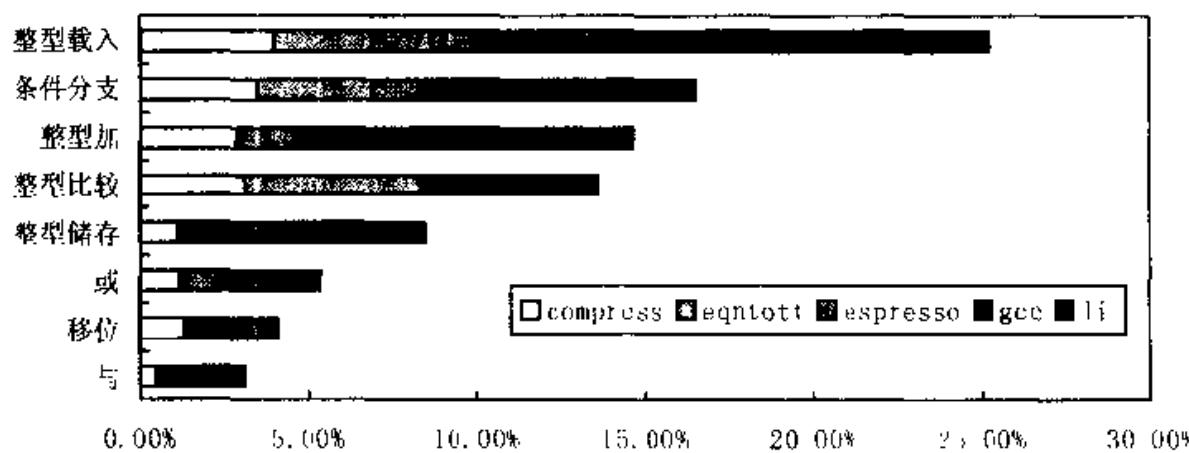


图 2.14 指令使用频率的整型平均

表 2.17 基于 SPECfp92 基准程序集的指令使用频率测量统计结果

指令	doduc	ear	hydro2d	mdljdp2	su2cor	浮点平均
载入	1.4%	0.2%	0.1%	1.1%	3.6%	1%
储存	1.3%	0.1%		0.1%	1.3%	1%
加	13.6%	13.6%	10.9%	4.7%	9.7%	11%
减	0.3%		0.2%		0.7%	0%
乘						0%
除						0%
比较	3.2%	3.1%	1.2%	0.3%	1.3%	2%
载入立即值	2.2%		0.2%	2.2%	0.9%	1%
条件分支	8.0%	10.1%	11.7%	9.3%	2.6%	8%
无条件分支	0.9%	0.4%		0.4%	0.1%	0%
调用	0.5%	1.9%			0.3%	1%
返回,跳转	0.6%	1.9%			0.3%	1%
移位	2.0%	0.2%	2.4%	1.3%	2.3%	2%
与	0.4%	0.1%			0.3%	0%
或		0.2%	0.1%	0.1%	0.1%	0%
其他(异或,非)						0%
载入浮点数	23.3%	19.8%	24.1%	25.9%	21.6%	23%
储存浮点数	5.7%	11.4%	9.9%	10.0%	9.8%	9%
浮点加	8.8%	7.3%	3.6%	8.5%	12.4%	8%
浮点减	3.8%	3.2%	7.9%	10.4%	5.9%	6%
浮点乘	12.0%	9.6%	9.4%	13.9%	21.6%	13%
浮点除	2.3%		1.6%	0.9%	0.7%	1%
浮点比较	4.2%	6.4%	10.4%	9.3%	0.8%	6%
浮点寄存器移动	2.1%	1.8%	5.2%	0.9%	1.9%	2%
其他浮点操作	2.4%	8.4%	0.2%	0.2%	1.2%	2%

2.7.2 DLX 指令集结构效能分析

综上所述,DLX 指令集结构的指令格式、寻址方式和操作都非常简单。也许有人担心,这些特性会使目标代码中指令条数增多,导致程序运行时间加长,从而使这种指令集结构的机器性能不会太高,但情况并非如此。实际上,回顾一下前一章所给出的 CPU 性能公式,可以发现程序在 CPU 上执行的时间是与指令计数和每条指令的平均时钟周期数,以及时钟周期时间三个因素密切相关的。

因此不能仅仅从指令计数来考虑指令集结构的性能。

为了考察 DLX 结构的效能, 我们选择出现于 20 世纪 70 年代中期的 VAX 指令集结构作为参考结构。之所以选择 VAX, 是因为 VAX 的设计思想和 DLX 的设计思想截然不同,VAX 中提供了多种指令格式、大量的寻址方式,所有的寻址方式均可以适用于各种类型的指令操作,并且指令的操作也比较复杂。

当然 VAX 这种设计思想是有其时代背景的, 从第一章我们已经了解到 DRAM 的容量以三年四倍的速度提高, 所以 20 世纪 70 年代中期的 DRAM 芯片的容量不会超过当今 DRAM 芯片容量的 1/1000, 当时机器的存储空间十分紧张, 自然对目标代码的长短有很高的要求。VAX 采用这种设计思想的重要目标之一就是希望尽可能缩短目标代码, 减少存储器空间的浪费。

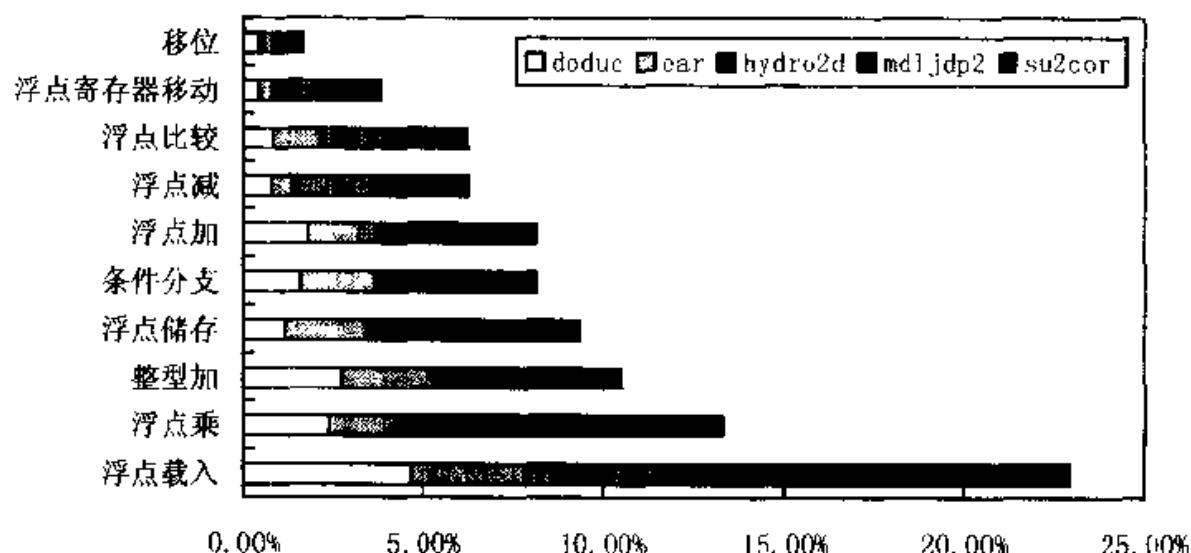


图 2.15 指令使用频率的浮点平均

这里选择 VAX 8700 作为 VAX 指令集结构的机器代表, 而选择 MIPS M2000 作为 DLX 指令集结构的机器代表。图 2.16 是这两种机器运行 SPEC89 基准程序测量结果的比较, 它表明了两种机器执行的指令条数的比值、CPI 的比值和以时钟周期为单位的性能比值。

因为 VAX 8700 和 MIPS M2000 的组织相似, 所以假设它们的时钟周期时间一样。MIPS M2000 执行的指令数大约为 VAX 8700 的两倍, 而 VAX 8700 的 CPI 将近是 MIPS M2000 的六倍, 所以 MIPS M2000 在性能上将近是 VAX 8700 的三倍。另外, 由于 MIPS 结构十分简单, 所以实现 MIPS 的 CPU 比实现 VAX 的 CPU 所需要的硬件要少得多。

也正是由于这种性能价格比上的差异使得过去研制 VAX 机器的公司现在已抛弃了 VAX 指令集结构, 而采用了和 DLX 类似的指令集结构。

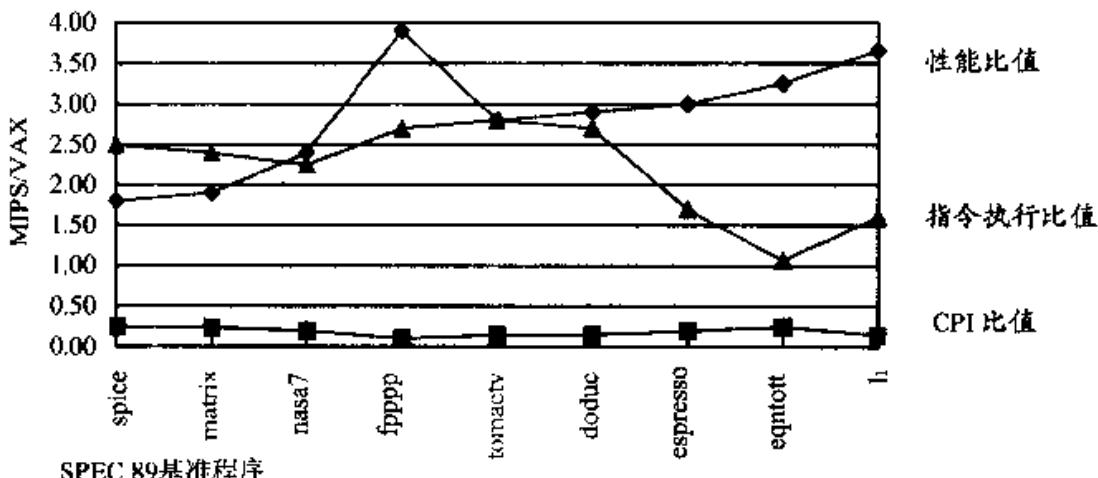


图 2.16 VAX 8700 和 MIPS M2000 的比较结果

2.8 小结

在计算机发展早期,由于受到计算机硬件技术的限制,当时的指令集十分简单,所以计算机体系结构也受到了极大限制。但是随着计算机及硬件技术的发展,一旦条件允许,计算机体系结构设计者总是想方设法支持高级语言。对如何更好地支持程序有效运行的研究大致经历了三个阶段:在 20 世纪 60 年代,堆栈结构十分普及,大家认为这种结构可以很好地和高级语言相结合,这也许是基于当时的编译技术而做出的一种选择。在 70 年代,体系结构设计者的主要精力都集中在如何降低软件开销上;所以人们致力于用硬件代替软件的工作,甚至致力于设计出能够简化软件设计者编程的高级结构,从而导致了高级语言计算机体系结构的发展,设计出了诸如 VAX 这样的寻址和数据类型十分丰富高度正交的复杂指令集结构。到了 80 年代以后,由于有成熟的编译技术支持,所以人们的眼光又重新回到了如何提高机器性能上,这直接导致了 RISC 精简指令集结构的出现。

在 1970 年到 1985 年之间,许多人都认为计算机体系结构设计者的工作就是设计计算机指令集,所以当时的一些教科书都十分强调指令集结构设计,同时这些教科书还希望读者对一些典型机器的长处和弱点有较深刻的认识,似乎每个人都有机会设计指令集。

当前,计算机体系结构的定义已经扩展到整个计算机系统设计和评价的全过程,而并不仅仅是定义指令集,所以有许多问题等待着体系结构设计者们去研究。因此本书论述的是计算机设计,而不仅仅是指令集设计。

习 题 二

2.1 解释下列术语

堆栈型机器	累加器型机器	通用寄存器型机器	有效地址
按序转换问题	CISC	RISC	图着色法
高级优化	局部优化	全局优化	指令集结构的正交特性

2.2 堆栈型机器、累加器型机器和通用寄存器型机器各自有什么优缺点?

2.3 常见的三种类型的通用寄存器型机器的优缺点各有哪些?

2.4 指令集结构设计所涉及的内容有哪些?

2.5 简述 CISC 计算机结构指令集功能设计的主要目标。从当前的计算机技术观点来看, CISC 结构有什么缺点?

2.6 简述 RISC 结构的设计原则。

2.7 简述操作数的类型及其相应的表示方法。

2.8 表示寻址方式的主要方法有哪些? 简述这些方法的优缺点。

2.9 通常有哪几种指令格式,请简述其适用范围。

2.10 为了对编译器设计提供支持,在进行指令集结构设计时,应考虑哪些问题?

2.11 请根据 CPU 性能公式简述 RISC 机器和 CISC 机器的性能特点。

2.12 现有如下 C 语言源代码:

```
for (i=0; i<=100, i++)
    A[i] = B[i] + C;
```

其中,A 和 B 是两个 32 位整数的数组,C 和 i 均是 32 位整数。假设所有数据的值及其地址均保存在存储器中,A 和 B 的起始地址分别是 0 和 5000,C 和 i 的地址分别是 1500 和 2000。在循环的两次迭代之间不将任何数保存在寄存器中。

- (1) 请写出该 C 语言源程序的 DLX 实现代码。
- (2) 该程序段共执行了多少条指令?
- (3) 程序对存储器中的数据访问了多少次?
- (4) DLX 代码的大小是多少?

2.13 参考习题2.12,现假设将 i 的值和数组变量的地址在程序运行过程中,只要有可能就一直保存在寄存器中。

- (1) 请写出该 C 语言源程序的 DLX 实现代码。
- (2) 该程序段共执行了多少条指令?
- (3) 程序对存储器中的数据访问了多少次?
- (4) DLX 代码的大小是多少?

2.14 读写存储器的频率、访问指令和数据的频率是设计存储器系统的重要依据之一。参考表 2.16 中的整型平均指标,求:

- (1) 所有对数据的存储器访问所占的百分比;
- (2) 所有数据访问中读操作所占的百分比;
- (3) 所有存储器访问中读操作所占的百分比,

2.15 对表2.16中的所有类型的指令,现通过测量其 CPI,得到如下结果:

指 令	时钟周期
所有的 ALU 指令	1
Load/Store 指令	1.4
成功的条件分支指令	2.0
失败的条件分支指令	1.5
跳转指令	1.2

假设 60% 的条件分支指令分支转移成功,同时将表 2.16 中其他一些类别的指令(没有被包含在上述类别中的指令)看作是 ALU 指令,请根据 gcc 和 cspresso 基准程序计算上述各种类型指令出现的平均频率,以及这两个基准程序的有效(等效)CPI。

第三章 流水线技术

在计算机体系结构设计中,为了提高执行部件的处理速度,经常在部件中采用流水线技术,这是一种性能价格比较高的方法。本章首先论述流水线的基本概念、流水线的分类和流水线的性能计算方法,然后基于 DLX 指令集结构的流水实现,对流水线中的相关等问题进行深入讨论,并给出一种流水线处理器实例。最后,将讨论流水线技术在向量处理机中的应用。

3.1 流水线的基本概念

3.1.1 流水线的基本概念

每当谈起“流水线”,人们就可能会将其和生产车间的产品生产流水线联系起来。的确,计算机技术中的“流水线”概念也正是由此得来的。为了对计算机技术中的流水线概念有明确的认识,让我们首先看看产品生产流水线的作用。

假设:某产品的生产需要 4 道工序。该产品生产车间以前只有 1 个工人,1 套生产该产品的机器。该工人工作 8 小时,可以生产 120 件(即每 4 分钟生产 1 件)。现车间主任希望将该产品的日产量提高到 480 件,那么他如何能够实现其目标呢?一种办法是再聘请 3 名工人,同时再购买 3 套生产该产品的机器。让 4 名工人同时工作 8 小时,可以达到期望的日产量目标。

但是这种方法需要购买 3 套机器,其费用已经超出了车间主任现有的经济承受能力。这时车间工程师提出了一套技术改造方案:产品生产采用流水线生产方式,将原来的机器按照 4 道工序重新进行改造组合,将 4 道生产工序分离开来,使得每道工序的生产时间一样,均为 1 分钟;同时车间再聘请 3 名工人,让每个工人负责该产品生产的一道工序,每完成一道工序,就将半成品传给下一道工序的工人,由他去完成别的产品生产工序,直至生产出完整的产品。如此连续作业,工作 8 小时(如图 3.1 所示)。

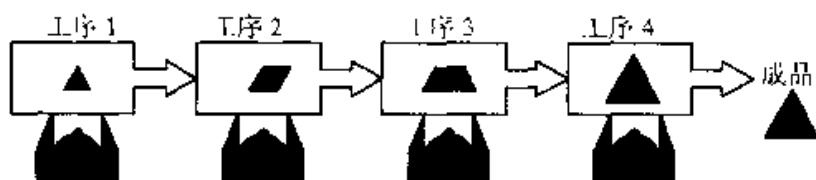


图 3.1 车间产品生产流水线示意图

主任采纳了车间工程师的意见，并照此办理，在没有花钱购置机器的情况下，就达到了所期望的日产量目标。那么这种节省开销的方案是如何满足车间主任要求的呢？

实际上，可以用图 3.2 来表示 4 个工人的生产过程，其中每个箭头表示工人们生产出一件成品。从图中可以看出，每个工人连续工作 8 小时，实际上是重复了 480 次（每分钟 1 次）他所负责的工序，而第 1 件成品生产出来到第 477 件成品生产出来之间，所有工人均是在并行地完成自己所负责的工作，从而使得在第 1 件成品生产出来之后，每隔 1 分钟就生产出来 1 件成品。按 8 小时计算，整个车间的日产量是 480 件。

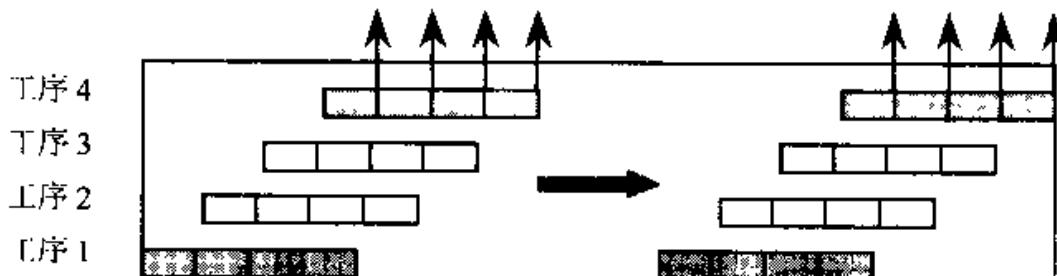


图 3.2 流水线生产过程的抽象描述

这种流水工作方式的主要特点是：每件产品还是要经过 4 道工序处理，单件产品的加工时间并没有改变，但是它将各个工人的操作时间重叠在一起，使得每件产品的产出时间从表面上看是从原来的 4 分钟缩减到 1 分钟，提高了产品的产出率。另外，它和第一种方案相比，明显节省了硬件投入，的确是一种性价比非常好的方案。

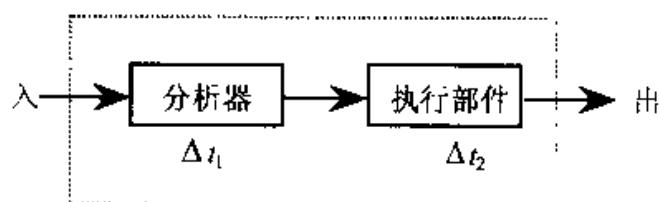


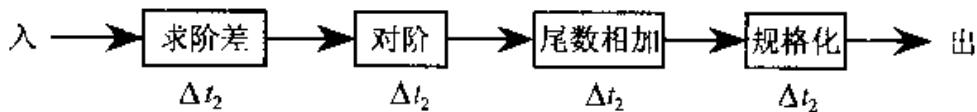
图 3.3 指令执行的简化结构

可以将上述思想引入到计算机技术中来。比如，可以将一条指令的解释过程分解成“分析”与“执行”两个子过程。每个子过程分别在指令分析器和执行部件这两个独立的部件上实现（如图 3.3 所示）。所以，不必等待上一条指令的“分析”和“执行”子过程完成后，才送入下一条指令。指令分析器在完成上一条指令的“分析”子过程，并将结果送入执行部件去实现“执行”子过程的同时，就可开始接收下一条指令，并进行“分析”子过程。若“分析”时间和“执行”时间相等，如为 Δt_1 ，则从执行一条指令的全过程来看，每条指令的执行需要 $T = 2\Delta t_1$ 才能完

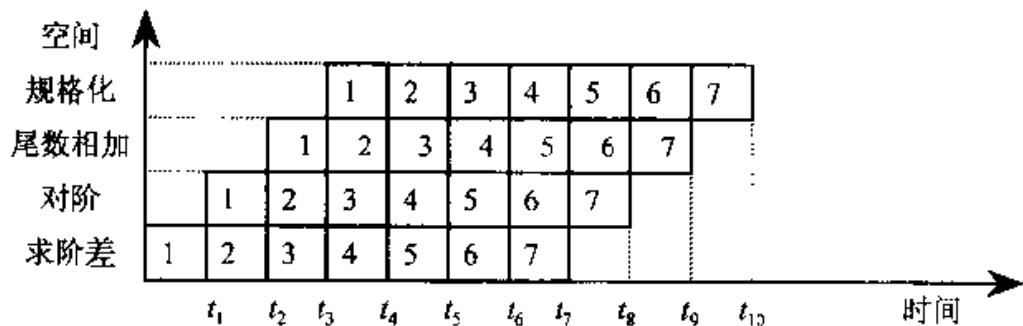
成；然而，从机器的输出端来看，都是每隔 Δt_1 就能给出一条指令的执行结果。处理机的速度提高了一倍。

将这种思想用于提高其他执行部件的速度上，同样可以收到明显的加速效果。例如，对于浮点加法器而言，可以把浮点加法的全过程分解成求阶差、对阶、尾数相加和规格化四个子过程，让每个子过程都在各自独立的部件上完成。设各独立部件完成相应工作（以后称为段）所需时间都为 Δt_2 ，如图 3.4(a) 所示。若在输入端连续作几次加法，那么在第一个 Δt_2 内，第一次加法在求阶差段；第二个 Δt_2 内，第一次加法在对阶段，第二次加法则进入求阶差段；第三个 Δt_2 内，第一次加法在尾数相加段，第二次加法在对阶段，第三次加法则在求阶差段……由此可见，虽然每次加法操作所需时间都是 $T = 4\Delta t_2$ ，但从加法器的输出端来看，却是每隔一个 Δt_2 给出一个加法结果。这样，速度提高了三倍。

由于这种工作方式与上述车间生产流水线概念相类似，因此，将上述浮点加法器称为浮点加法流水线。



(a) 浮点加法流水线



(b) 描述流水线工作的时空图

图 3.4 流水技术原理

所谓流水线技术，是指将一个重复的时序过程，分解成为若干个子过程，而每一个子过程都可有效地在其专用功能段上与其他子过程同时执行。

描述流水线的工作，常采用时(间)-空(间)图的方法。图 3.4(b) 是浮点加法流水线的时空图，横坐标表示时间，纵坐标代表流水线的各段，图中的数字代表各次加法在流水线中流动的过程。由此可看出，流水技术具有如下特点：

1. 流水过程由多个相联系的子过程组成，每个子过程称为流水线的“级”或“段”。流水线的段数也称为流水线的“深度”或“流水深度”。

2. 每个子过程由专用的功能段实现；
3. 各个功能段所需时间应尽量相等，否则，时间长的功能段将成为流水线的瓶颈，会造成流水线的“堵塞”和“断流”，这个时间一般为一个时钟周期(拍)或一个机器周期；
4. 流水线需要有“通过时间”(第一个任务流出结果所需的时间)，在此之后流水过程才进入稳定工作状态，每一个时钟周期(拍)流出一个结果；
5. 流水技术适合于大量重复的时序过程，只有输入端能连续地提供任务，流水线的效率才能充分发挥。

3.1.2 流水线的分类

流水线可按不同的观点进行分类，一般来说流水线可以分为如下几种类型。

1. 单功能流水线 (unifunction pipelines) 和多功能流水线 (multifunction pipelines)

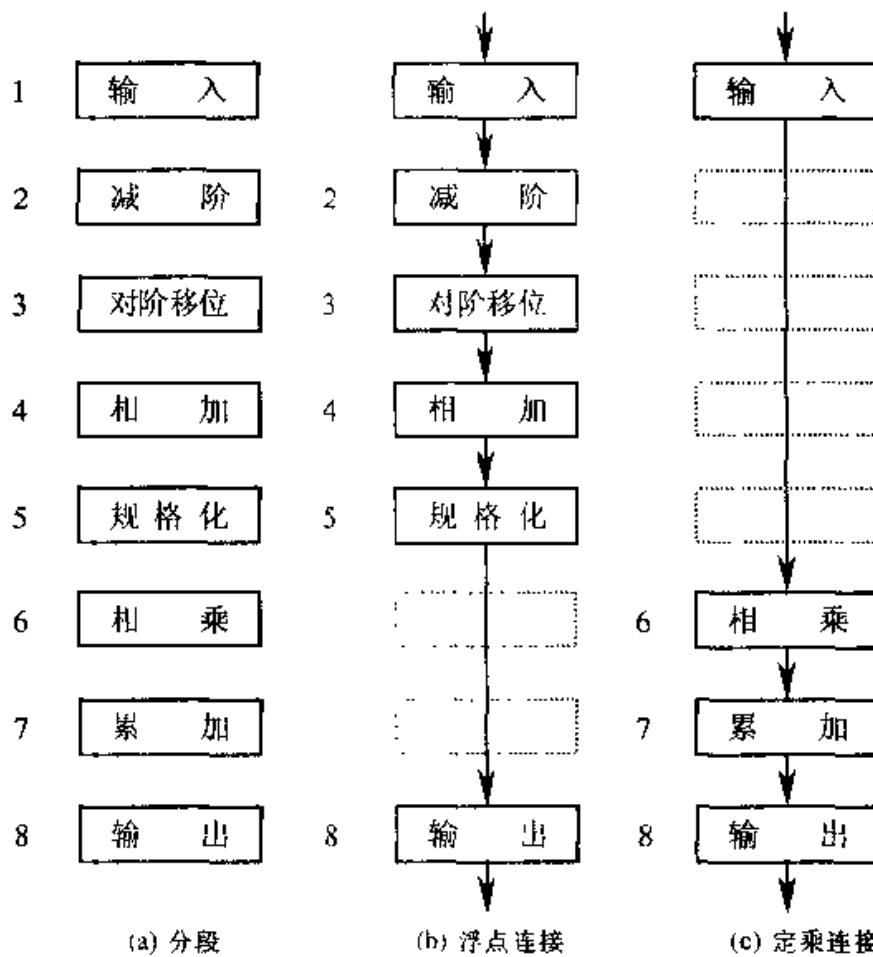


图 3.5 TI ASC 的多功能流水线

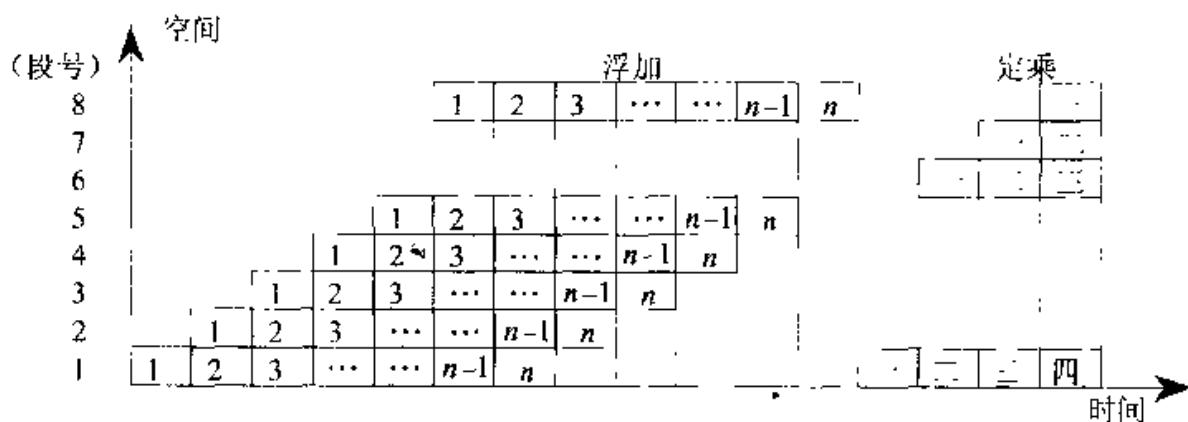
这是按照流水线所完成的功能来分类的。所谓单功能流水线，是指只能完成一种固定功能的流水线，如前面介绍的浮点加法流水线，要完成多种功能，可

采用多个单功能流水线。如 Cray-1 有 12 个单功能流水线, YH-1 有 18 个单功能流水线。

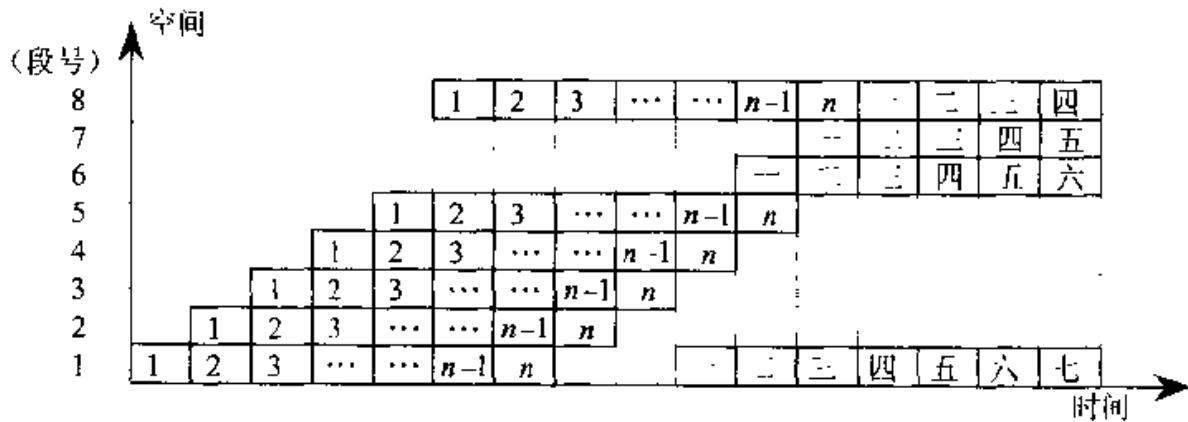
而所谓多功能流水线,是指流水线的各段可以进行不同的连接,从而使流水线在不同的时间,或者在同一时间完成不同的功能。例如 TI ASC 的运算器就是多功能流水线,如图 3.5(a)所示。它由 8 段组成,当要进行浮点加、减法运算时,各段的连接如图 3.5(b)所示;当要进行定点乘法运算时,各段的连接如图 3.5(c)所示。

2. 静态流水线(static pipelines)和动态流水线(dynamic pipelines)

这是按照同一时间内各段之间的连接方式来分类的。所谓静态流水线,是指在同一时间内,流水线的各段只能按同一种功能的连接方式工作。例如,上述 ASC 的 8 段只能或者是都按浮点加、减运算连接方式工作,或者是都按定点乘运算连接方式工作,不能在同一时间有的段在进行浮点加、减运算,而有的段又在进行定点乘运算。因此,在静态流水线中,只有当输入的是一串相同的运算操作时,流水的效率才能得以发挥。如果流水线输入的是一串不同运算相间的操作,例如是浮加、定乘、浮加、定乘……的一串操作,则这种静态流水线的效率会降到和顺序处理方式的一样。



(a) 静态流水线



(b) 动态流水线

图 3.6 静、动态流水线时空图

所谓动态流水线,是指在同一时间内,当某些段正在实现某种运算(如定乘)时,另一些段却在实现另一种运算(如浮加)。这样,就不是非得相同运算的一串操作才能流水处理。显然,这对提高流水线的效率很有好处,然而,这却会使流水线的控制变得很复杂。目前,绝大多数的流水线是静态流水线。从图 3.6 给出的静态和动态流水线的时空图,可以很清楚地看到它们工作方式的不同。

3. 部件级、处理机级及处理机间流水线

这是按照流水的级别来进行分类的。所谓部件级流水线,又叫运算操作流水线(arithmetic pipelines)。它是把处理机的算术逻辑部件分段,以便为各种数据类型进行流水操作,我们已经在前面论述了其相应的实例。

所谓处理机级流水线,又叫指令流水线(instruction pipelines),它是把解释指令的过程按照流水方式处理。因为处理机要处理的主要时序过程就是解释指令的过程,这个过程当然也可分解为若干个子过程。把它们按照流水(时间重叠)方式组织起来,就能使处理机重叠地解释多条指令。从这个意义上说,可以把前面将指令解释过程分解为“分析”与“执行”两个子过程的最简单的指令流水线,它同时只能解释两条指令。下一节将详细介绍指令流水线,这也正是本章的重点之一。

所谓处理机间流水线,又叫宏流水线(macro pipelines)。它是由两个以上的处理机串行地对同一数据流进行处理,每个处理机完成一项任务。如图 3.7 所示,第一个处理机对输入的数据流完成任务 1 的处理,其结果存入存储器中。它又被第二个处理机取出进行任务 2 的处理……依此类推。这一般属于异构型多处理机系统,它对提高各处理机的效率有很大的作用。

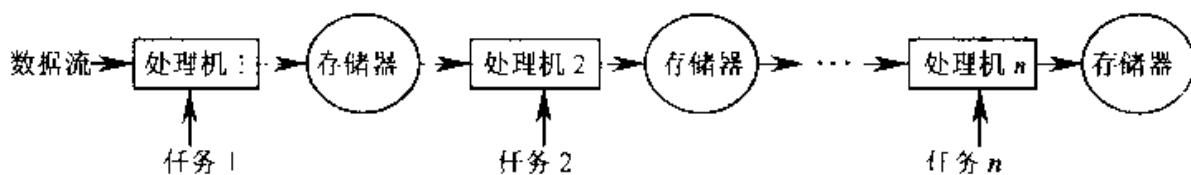


图 3.7 宏流水线

4. 标量流水处理机和向量流水处理机

这是按照数据表示来进行分类的。所谓标量流水处理机(scalar pipelining processor),是指处理机不具有向量数据表示,仅对标量数据进行流水处理,如 IBM 360/91, Amdahl 470V/6 等。而向量流水处理机(vector pipelining processor),是指处理机具有向量数据表示,并通过向量指令对向量的各元素进行处理。所以,向量处理机是向量数据表示和流水技术的结合,如 TI ASC、STAR-100、CYBER-205、CRAY-1、YH-1 等。关于它们的结构和特点,也是本章论述的重点之一。

5. 线性流水线和非线性流水线

这是按照流水线中是否有反馈回路来进行分类的。所谓线性流水线(linear pipelines),是指流水线的各段串行连接、没有反馈回路。而非线性流水线(non-linear pipelines),是指流水线中除有串行连接的通路处,还有反馈回路。图3.8就是一个非线性流水线,虽然它由4段S1~S4组成,但由于有反馈回路,从输入到输出可能要依次流过S1、S2、S3、S4、S2、S3、S4、S3各段(图中 \otimes 代表多路开关)。在一次流水过程中,有的段要被多次使用。非线性流水线常用于递归(recurrence),或组成多功能流水线。在非线性流水线中,一个重要的问题是确定什么时候向流水线引进新的输入,从而使新输入的数据和先前操作的反馈数据在流水线中不产生冲突,此即所谓的流水线调度问题。

此外,还可按照输出端任务流出顺序与输入端任务流入顺序是否相同,将流水线分为顺序流动流水线和异步流动流水线(或称为无序流水线、错序流水线、乱序流水线),这里就不再赘述。下面将以DLX指令流水线为例来论述流水线技术。

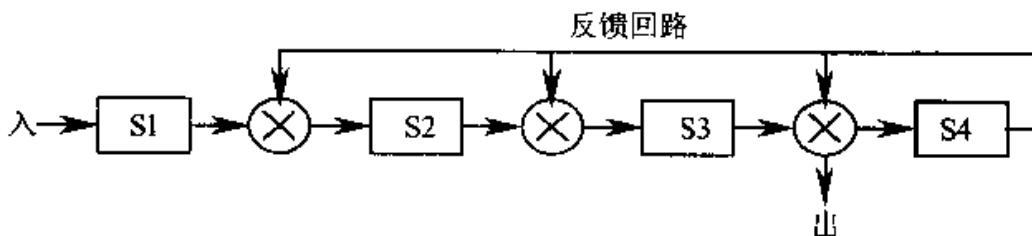


图 3.8 非线性流水线

3.2 DLX 的基本流水线

3.2.1 DLX 的一种简单实现

上一章论述了DLX这种Load/Store型指令集结构,为了说明如何流水实现DLX指令集结构,我们首先论述在不流水的情况下,是如何实现DLX的。图3.9给出了实现DLX指令的一种简单数据通路。可以看出,它能够在以下5个时钟周期内实现一条DLX指令:

1. 取指令周期(IF)

$$IR \leftarrow \text{Mem}[PC]$$

$$NPC \leftarrow PC + 4$$

其操作为:根据PC值从存储器中取出指令,并将指令送入指令寄存器IR;PC值增加4,指向顺序的下一条指令,并将下一条指令的地址放入临时寄存器

NPC 中。

2. 指令译码/读寄存器周期(ID)

$$A \leftarrow \text{Regs}[IR_{6..10}]$$

$$B \leftarrow \text{Regs}[IR_{11..15}]$$

$$\text{Imm} \leftarrow ((IR_{16})^{16} \# IR_{16..31})$$

其操作为：进行指令译码，读 IR 寄存器（指令寄存器），并将读出结果放入两个临时寄存器 A 和 B 中。同时对 IR 寄存器中内容的低 16 位进行符号扩展，然后将符号扩展之后的 32 位立即值保存在临时寄存器 Imm 中。

指令的译码操作和读寄存器操作是并行进行的，因为 DLX 指令格式中操作码在固定位置，从而为这种并行提供了可能，这种技术也称为“固定字段译码”(fixed-field decoding)技术。值得注意的是，在上述过程中，可能读出了一些在后面周期中并不会使用到的寄存器内容，但是这并不会影响指令执行的正确性。相反，却可以有效地降低问题的复杂性。

另外，由于立即值在 DLX 指令格式中处于固定位置，因此这里也对其进行符号扩展，以便在下一个周期能使用它。当然由于指令的不同，也许在后面的周期中并不会用到这个立即值，但无论如何，提前形成立即值总是有益无害的。

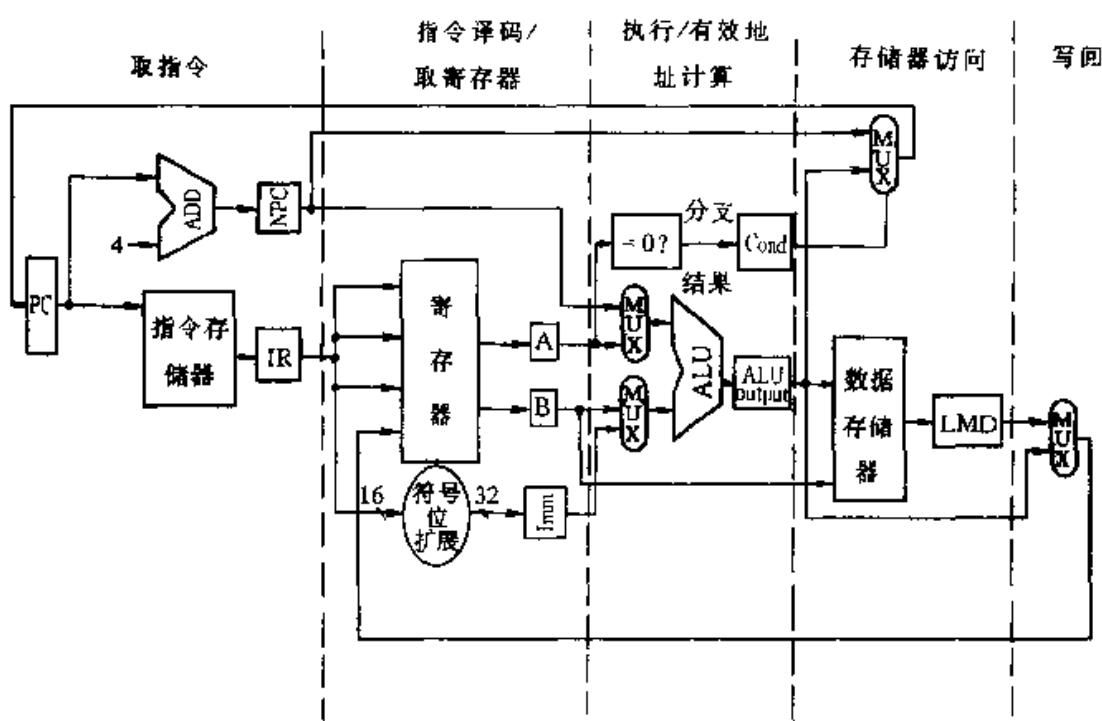


图 3.9 实现 DLX 指令的一种简单数据通路

3. 执行/有效地址计算周期(EX)

在前一个周期已经准备好了指令要处理的操作数之后，就开始执行/有效地址计算周期。根据指令的不同，可以将该周期的操作分为以下几种类型：

(1) 存储器访问

$$\text{ALUoutput} \leftarrow A + \text{Imm}$$

当指令为存储器访问指令时,该周期的操作为:ALU 将操作数相加形成有效地址,并将结果放入临时寄存器 ALUoutput 中。

(2) 寄存器 - 寄存器 ALU 操作

$$\text{ALUoutput} \leftarrow A \text{ op } B$$

当指令为寄存器 - 寄存器 ALU 操作指令时,该周期的操作为:ALU 根据操作码指出的功能对临时寄存器 A 和 B 中的值进行处理,并将结果送入临时寄存器 ALUoutput 中。

(3) 寄存器 - 立即值 ALU 操作

$$\text{ALUoutput} \leftarrow A \text{ op Imm}$$

当指令为寄存器 - 立即值 ALU 指令时,该周期的操作为:ALU 根据操作码指出的功能对临时寄存器 A 和 Imm 中的值进行处理,并将结果送入临时寄存器 ALUoutput 中。

(4) 分支操作

$$\text{ALUoutput} \leftarrow \text{NPC} + \text{Imm};$$

$$\text{Cond} \leftarrow (A \text{ op } 0)$$

当指令为分支指令时,该周期的操作为:ALU 将临时寄存器 NPC 和 Imm 中的值相加,得到分支的目标地址。同时,对在前一个周期读入到寄存器 A 的值进行检查,决定分支是否成功。op 由分支操作码决定。比如,op 对 BEQZ 指令来说就是“==”。

上面将有效地址计算周期和执行周期合并为一个时钟周期,这是由 DLX 指令集结构本身的特点所允许的,因为在 DLX 指令集结构中,没有任何指令需要同时计算数据的存储器地址、计算分支指令的目标地址和对数据进行处理。另外,上面四种操作类型中没有包含各种形式的跳转操作,它们和分支操作十分相似,这里就不再赘述。

4. 存储器访问/分支完成周期(MEM)

在该周期处理的 DLX 指令仅仅有 Load、Store 和分支三种指令,下面分别讨论它们在该周期内所完成的操作。

(1) 存储器访问操作

$$\text{LMD} \leftarrow \text{Mem}[\text{ALUoutput}] \text{ 或者 } \text{Mem}[\text{ALUoutput}] \leftarrow B$$

存储器访问操作包含了 Load 和 Store 两种类型的操作。如果指令是 Load 指令,就将临时寄存器 ALUoutput 中的值作为访存地址,从存储器中读出相应的数据,并放入临时寄存器 LMD 中;如果指令是 Store 指令,就将临时寄存器 B 中的值按照临时寄存器 ALUoutput 所指明的地址写入存储器。

(2) 分支操作

```
if (cond) PC ← ALUoutput else PC ← NPC
```

如果分支条件寄存器中的内容为真,表明分支转移成功,选择临时寄存器 ALUoutput 中的值作为分支目标地址,并将其放入 PC 中。否则,它将临时寄存器 NPC 中的值送入 PC 中,作为下一条指令地址。

5. 写回周期(WB)

不同的指令在写回周期完成的工作也不一样,这里按如下指令类型对写回周期所要完成的工作进行说明。

(1) 寄存器 - 寄存器型 ALU 指令

```
Regs[IR16..20] ← ALUoutput
```

(2) 寄存器 - 立即值型 ALU 指令

```
Regs[IR11..15] ← ALUoutput
```

(3) Load 指令

```
Regs[IR11..15] ← LMD
```

上述指令均是将结果写入寄存器文件。无论结果是来自于存储器系统(临时寄存器 LMD 中的内容),还是来自于 ALU 的计算结果(临时寄存器 ALUoutput 中的内容),都由操作码决定将其送入目标寄存器相应的域中。

从图 3.9 中可以看出,分支指令需要 4 个时钟周期完成,其他指令需要 5 个时钟周期完成。假设分支指令数占指令总数(也称混合指令数)的 12%,则其 CPI 是 4.88 个时钟周期。由此可见,上述实现无论是从性能方面,还是从硬件开销方面来看,都称不上是一种优化实现。

由于 ALU 指令在 MEM 周期不做任何工作,所以可以在 MEM 周期就完成 ALU 指令,这并不影响执行部件的时钟速度,但是可以降低 CPI。假设 ALU 指令数占混合指令条数的 44%,则 CPI 为 4.44,这种改进所带来的加速比为 $4.88/4.44 = 1.1$ 。

如果还要降低 CPI,就有可能会延长时钟周期时间,因为在每个时钟周期中可能需要完成更多的工作,因此,在时钟周期时间和 CPI 之间也存在着折衷关系,这种折衷取舍需建立在对系统仔细分析的基础之上。

为此,可以考虑一种降低 CPI 的极限情况,那就是用单周期实现来代替上述多周期实现,即每条指令在一个较长的时钟周期内完成。这时机器的 CPI 是 1,但是其时钟周期时间却是上述多周期实现的时钟周期时间的 5 倍,所以从单条指令的执行时间上看,并没有多少改善。虽然在单周期实现中,由于每条指令必须通过所有功能单元,可以在实现中省去一些临时寄存器,但是在实际中一般并不采用单周期实现方法,其主要原因有:

1. 对多数机器而言,单周期实现并不是非常有效的。不同的指令所需完成

的操作大不一样,因而不同指令实现所需要的时钟周期时间也大不一样。

2. 从第一节中的产品生产流水线实例可以看出,基于单周期实现提高程序的执行速度需要重复设置指令执行功能部件,而基于多周期实现提高程序的执行速度则可以采用流水技术共享指令执行功能部件中的功能单元。

另外,在上述多周期实现中,除了可以优化 CPI 外,还可以省去一些冗余的硬件。比如,上述实现中有两个 ALU,一个进行 PC 值的增加,一个进行有效地址计算和 ALU 运算,由于两个 ALU 的操作并不在同一个时钟周期内进行,所以可以将其合并在一起,并通过增加多路器的方法由多种操作共享。同样,指令和数据也可以保存在同一个存储器中。

3.2.2 基本的 DLX 流水线

在上述 DLX 多周期实现的基础上,可以将每一个时钟周期看作是流水线的一个时钟周期,使图 3.9 中的数据通路成为一条指令流水线。硬件每个时钟周期启动一条新的指令,并执行 5 条不同指令中的某一部分。我们可以用时空图的另外一种形式将该流水线描述为如图 3.10 所示的流水过程。从图 3.10 可以看出,每条指令仍然需要 5 个时钟周期完成,但是指令执行的吞吐量却有很大提高。

指令编号	时钟周期								
	1	2	3	4	5	6	7	8	9
指令 i	IF	ID	EX	MEM	WB				
指令 $i+1$		IF	ID	EX	MEM	WB			
指令 $i+2$			IF	ID	EX	MEM	WB		
指令 $i+3$				IF	ID	EX	MEM	WB	
指令 $i+4$					IF	ID	EX	MEM	WB

图 3.10 一种简单的 DLX 流水线

从图 3.10 可以看出,上述 DLX 流水线十分简单,但是要使 DLX 指令的各种组合能够在上述流水线中真正流水起来,充分发挥流水线的效率,并不是一件很容易的事情,还有许多问题有待解决。

为了使多条指令能够在流水线中重叠执行,首先必须保证在指令重叠时,不存在任何流水线资源冲突问题,也就是确保流水线的各段不会在同一个时钟周期内使用相同的数据通路资源。图 3.11 从使用流水线资源的角度描述了上述流水线的流水过程。从图中可以看到,在同一个时钟周期内每条指令所使用的功能单元都不同,所以多条指令的重叠执行“基本上”没有资源冲突。

图 3.11 将指令存储器(IM)和数据存储器(DM)分隔开,避免了取指令操作和访问数据操作之间存在访问存储器冲突。值得注意的是,如果流水线的时钟

周期和非流水实现的一样,那么流水线的存储器带宽必须是非流水实现的 5 倍,这是为获取较高性能所必须付出的开销之一。

另外,从图 3.11 还可以看到,流水线中的 ID 段和 WB 段都要使用寄存器文件:在 ID 段对寄存器文件进行读操作,在 WB 段对寄存器文件进行写操作。那么,如果读操作和写操作都是对同一寄存器进行,又将如何?下节将对这个问题进行深入讨论。

其次,在图 3.11 中没有考虑 PC 的问题。流水线为了能够在每个时钟周期启动一条新的指令,就必须在每个时钟完成 PC 值的增值操作,并保存增值后的 PC 值。对上述 DLX 流水线来说,这些操作必须在 IF 段完成,以便为取下一条指令做好准备。

当流水线执行分支指令时,会出现新的问题。分支指令可能会改变 PC 的值,但是它只有在 MEM 段结束时才能完成改变 PC 值的操作。所以针对这个问题,我们需要重新组织上述流水线的数据通路,争取在 IF 段中完成改变 PC 值的操作,这里引出了一个如何处理分支指令的问题,我们将在后面详细论述、分析和讨论这一问题。

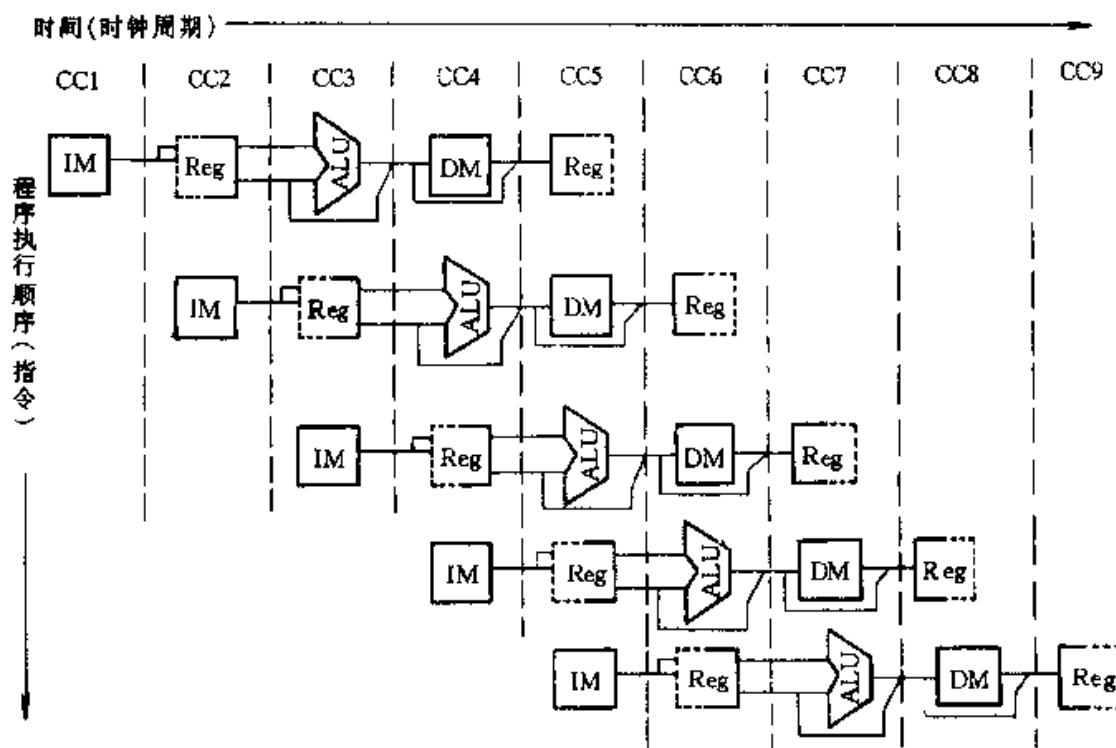


图 3.11 随时间移动的数据通路序列

通过上面的讨论可以看到,在 DLX 基本流水线中,每个时钟周期都要用到其所有的流水段,一个流水段中的所有操作必须在一个时钟周期内完成。特别是要使数据通路完全流水,就必须保证从一个流水段传输到下个流水段的数据

都被保存在寄存器文件中。为此,我们对图 3.9 所示的多周期实现数据通路进行了一定的改进,改进后的流水线数据通路如图 3.12 所示。

图 3.9 中的 PC 值多路选择器在图 3.12 中已经被移到了 IF 段,这样就可以保证对 PC 值的写操作只会在一个流水段内出现,否则当分支转移成功的时候,流水线中两条指令都会试图在不同的流水段写 PC 值,从而发生写冲突。

值得注意的是,如果在流水过程中仅仅使用先前非流水数据通路中的一些寄存器,那么当这些寄存器中保存的临时值还在为流水线中某条指令所用时,就可能会被流水线中其他的指令所重写,从而导致错误的执行结果。所以,我们在图 3.12 中的每个流水段之间设置了一些流水线寄存器文件。流水线寄存器文件保存着从一个流水段传送到下一个流水段的所有数据和控制信息。随着流水过程的进行,这些数据和控制信息从一个流水线寄存器文件拷贝到下一个流水线寄存器文件,直到不再需要为止。

寄存器文件由它们相连的流水段的名称来标记。当一条指令流水执行时,在各个时钟周期之间用来保存临时值的所有寄存器都包含在相应的寄存器文件中,每个寄存器被看作是相应寄存器文件的一个域。这些寄存器也称为流水线寄存器。比如,IF/ID 寄存器文件就是连接流水线 IF 段和 ID 段的寄存器文件,指令寄存器 IR 是 IF/ID 寄存器文件的一个域,被记为 IF/ID.IR。

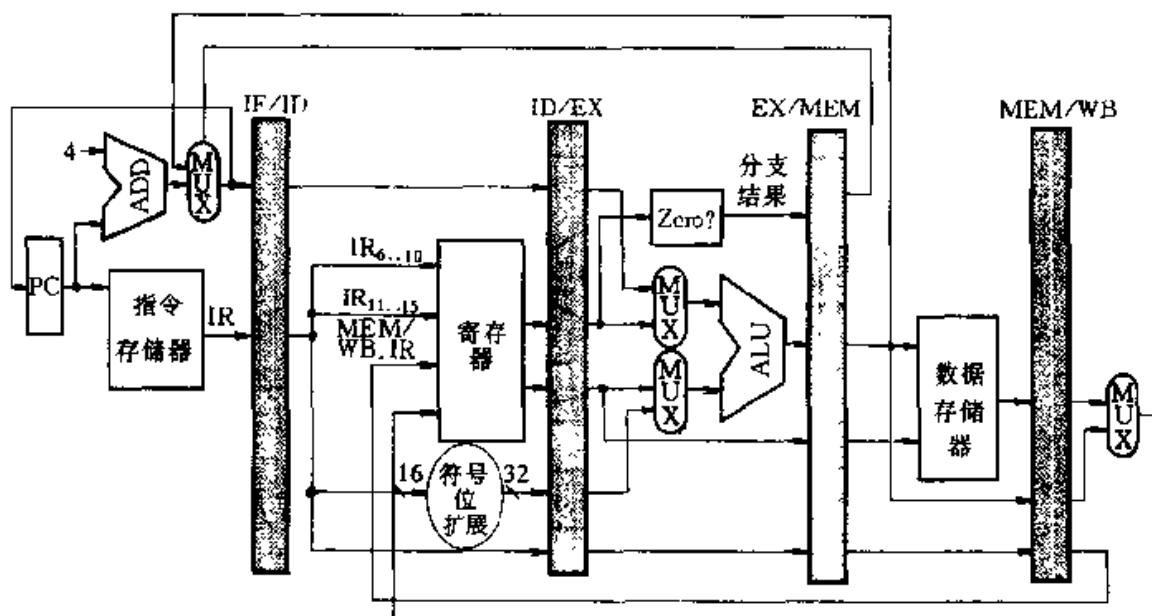


图 3.12 DLX 流水线的数据通路

在 DLX 流水线中,每个流水段所完成的操作如表 3.1 所示。

表 3.1 DLX 流水线的每个流水段的操作

流水段	任何指令类型		
	ALU 指令	Load/Store 指令	分支指令
IF	IF/ID. IR \leftarrow Mem[PC]; IF/ID. NPC, PC \leftarrow (if EX/MEM. cond {EX/MEM. NPC} else {PC + 4});		
ID	ID/EX. A \leftarrow Regs[IF/ID. IR _{6..10}]; ID/EX. B \leftarrow Regs[IF/ID. IR _{11..15}]; ID/EX. NPC \leftarrow IF/ID. NPC; ID/EX. IR \leftarrow IF/ID. IR; ID/EX. Imm \leftarrow (IR ₁₆) ¹⁶ # IR _{16..31} ;		
EX	EX/MEM. IR \leftarrow ID/EX. IR; EX/MEM. ALUoutput \leftarrow ID/EX. A op ID/EX. B 或 EX/MEM. ALUoutput \leftarrow ID/EX. A op ID/EX. Imm; EX/MEM. cond \leftarrow 0;	EX/MEM. IR \leftarrow ID/EX. IR; EX/MEM. ALUoutput \leftarrow ID/EX. A + ID/EX. Imm;	EX/MEM. ALUoutput \leftarrow ID/EX. NPC + ID/EX. Imm; EX/MEM. cond \leftarrow 0 (ID/EX. A op 0);
MEM	MEM/WB. IR \leftarrow EX/MEM. IR; MEM/WB. ALUoutput \leftarrow EX/MEM. ALUoutput;	MEM/WB. IR \leftarrow EX/MEM. IR; MEM/WB. LMD \leftarrow Mem[EX/MEM. ALUoutput]; 或 Mem[EX/MEM. ALUoutput] \leftarrow EX/MEM. B;	
WB	Regs[MEM/WB. IR _{16..20}] \leftarrow MEM/WB. ALUoutput; 或 Regs[MEM/WB. IR _{11..15}] \leftarrow MEM/WB. ALUoutput;	Regs[MEM/WB. IR _{11..15}] \leftarrow MEM/WB. LMD;	

为了控制该基本的 DLX 流水线,还需要确定如何控制图 3.12 中的四个多路器。ALU 输入端的两个多路器根据 ID/EX 寄存器的 IR 域所指出的指令类型来控制。其中,上面一个 ALU 多路器的输入由当前指令是否为分支指令确定,下面一个多路器的输入由当前指令是寄存器 - 寄存器型 ALU 指令还是其他类型的指令来控制。IF 段的多路器选择增长以后的 PC 值或者 EX/MEM. NPC 的值(分支的目标地址)作为下一条指令的地址,该多路器由 EX/MEM.

cond 域来控制。第四个多路器由在 WB 段的指令是 Load 指令还是 ALU 指令来控制。

3.2.3 流水线性能分析

1. 吞吐率(throughput rate)

吞吐率是衡量流水线速度的重要指标。它是指在单位时间内流水线所完成的任务数或输出结果的数量。

(1) 最大吞吐率 TP_{\max}

这是指流水线在连续流动达到稳定状态后所得到的吞吐率。如果流水线各段时间 Δt 相等, 即 $\Delta t_i = \Delta t_0$, 如图 3.13 所示, 则有:

$$TP_{\max} = \frac{1}{\Delta t_0} \quad (3.1)$$

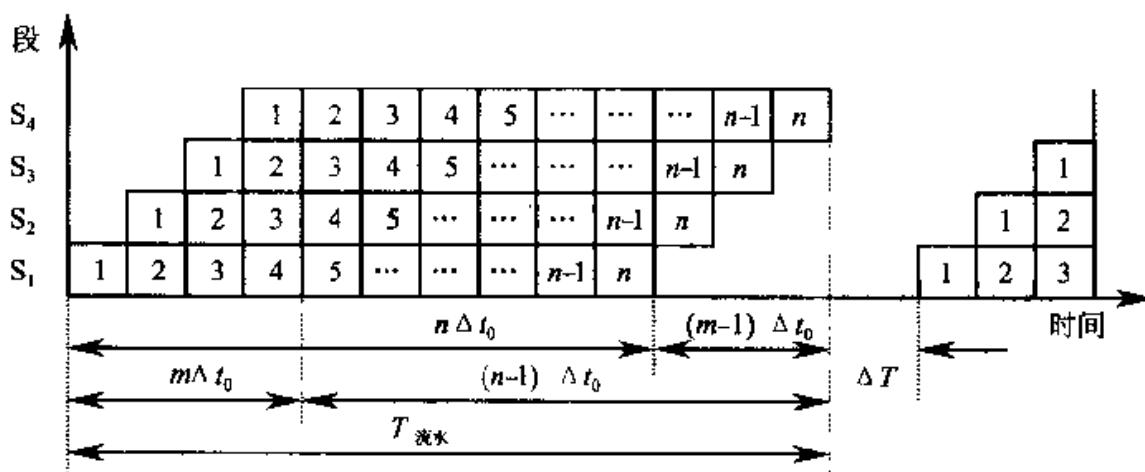


图 3.13 各段时间相等的流水线的时空图

若流水线各段时间不等, 如图 3.14(a)所示的四段流水线中, $\Delta t_2 = 3\Delta t_1 = 3\Delta t_3 = 3\Delta t_4 = 3\Delta t_0$, 其时空图如图 3.14(b)所示, 则有:

$$TP_{\max} = \frac{1}{\max |\Delta t_i|} = \frac{1}{3\Delta t_0} \quad (3.2)$$

即最大吞吐率取决于流水线中是慢的一段所需的时间, 这段就成了流水线的瓶颈。为了解决瓶颈, 可以将瓶颈段再细分。对于图 3.14(a)的流水线, 其瓶颈在 S_2 段, 故将 S_2 分成 $S_{2-1}, S_{2-2}, S_{2-3}$ 三段, 每段时间都为 Δt_0 , 如图 3.15(a)所示。然而, 当瓶颈段不能再细分时, 可采用图 3.15(b)的办法。重复设置瓶颈段, 使其并行工作, 其时空图如图 3.15(c)所示。这时, 最大吞吐率仍然能达到 $1/\Delta t_0$ 。但是, 在并行段之间的任务分配和同步都比较复杂。

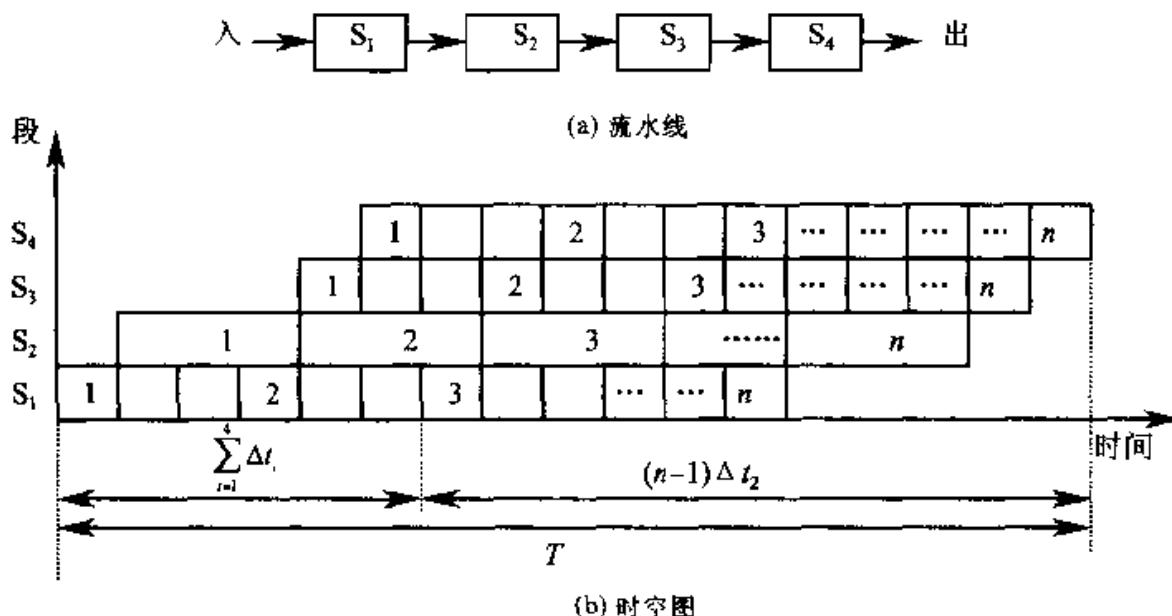
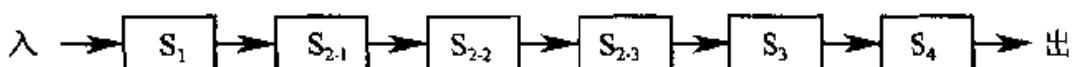
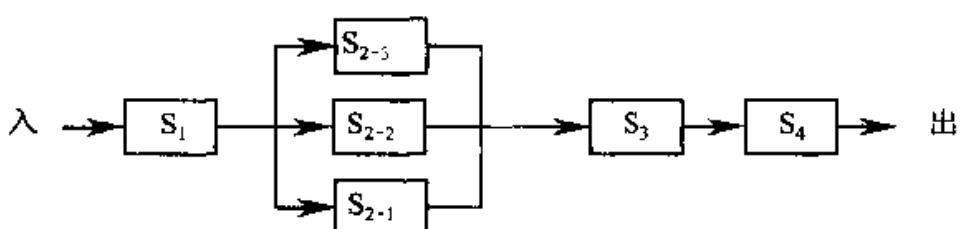


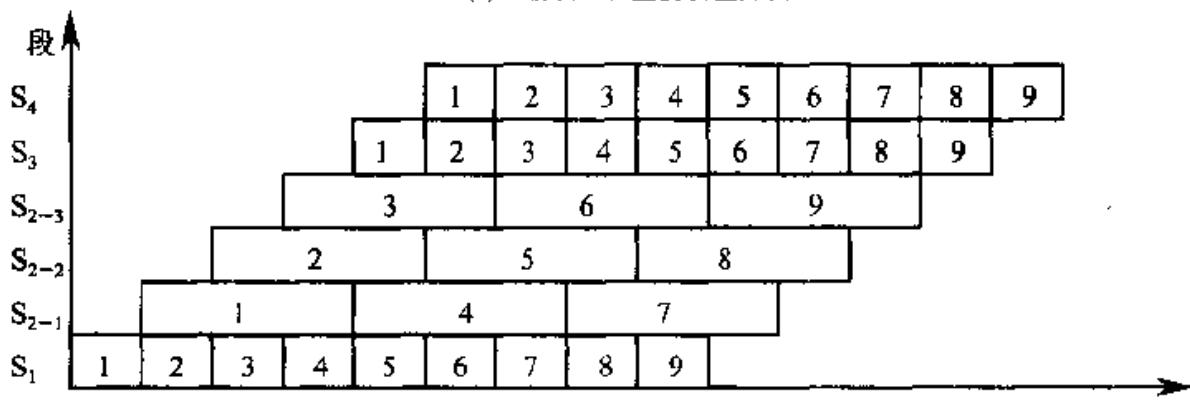
图 3.14 各段时间不等的流水线及其时空图



(a) “瓶颈”段细分方法



(b) “瓶颈”段重复设置方法



(c) 与(b)对应的流水线时空图

图 3.15 提高流水线吞吐率的方法

基于上述流水线性能分析可知,对于指令流水线而言,流水线增加了指令的吞吐率,但是它并不会真正减少一条指令总的执行时间。实际上,由于流水线控制等而带来的额外开销,反而会使每条指令的执行时间都会稍微有所增加。指令吞吐率的提高意味着即使单条指令执行并没有变快,但是程序运行会更快些,程序总执行时间也将会有有效减少。

最大吞吐率一般是机器说明书中给出的值。但实际上由于流水线有通过时间,输入的任务可能跟不上流水的需要,再加上程序中分支和相关等问题的影响,所以,流水线的实际吞吐率小于最大吞吐率。

(2) 实际吞吐率

设流水线由 m 段组成,完成 n 个任务的实际吞吐率可计算如下:

如果各段时间相等,则可参照图 3.13,完成 n 个任务所需的时间为

$$T_{\text{流水}} = m \cdot \Delta t_0 + (n - 1) \Delta t_0 \quad (3.3)$$

所以,实际吞吐率为

$$TP = \frac{n}{T_{\text{流水}}} = \frac{n}{m \Delta t_0 + (n - 1) \Delta t_0} = \frac{1}{\left(1 + \frac{m-1}{n}\right) \Delta t_0} = \frac{TP_{\max}}{1 + \frac{m-1}{n}} \quad (3.4)$$

由此看出:实际吞吐率小于最大吞吐率,它除了与 Δt_0 有关外,还与段数 m 、任务数 n 有关,只有当 $n \gg m$ 时,其 $TP \approx TP_{\max}$ 。

如果各段时间不等,则可参照图 3.14,完成 n 个任务的实际吞吐率为

$$TP = \frac{n}{\sum_{i=1}^m \Delta t_i + (n - 1) \Delta t_j} \quad (3.5)$$

式中 Δt_j 为最慢一段所需时间。

2. 加速比(speedup ratio)

流水线的加速比是指 m 段流水线的速度与等功能的非流水线的速度之比。对于连续完成 n 个任务来讲,如果流水线各段时间相等,则所需时间如式(3.3)所示。在等效的非流水线上所需时间为 $T_0 = n \cdot m \cdot \Delta t_0$,故加速比 S 为

$$S = \frac{T_0}{T_{\text{流水}}} = \frac{n \cdot m \cdot \Delta t_0}{m \Delta t_0 + (n - 1) \Delta t_0} = \frac{n \cdot m}{m + n - 1} = \frac{m}{1 + \frac{m-1}{n}} \quad (3.6)$$

可以看出:当 $n \gg m$ 时, $S \rightarrow m$ 。即线性流水线各段时间相等时,其最大加速比等于流水线的段数。从这个意义上讲,流水线的段数愈多愈好,但这会对流水线的设计带来许多问题,后而将对此进行详细论述。

如果各段时间不等,则有

$$S = \frac{n \cdot \sum_{i=1}^m \Delta t_i}{\sum_{i=1}^m \Delta t_i + (n - 1) \Delta t_j} \quad (3.7)$$

3. 效率(efficiency)

效率是指流水线的设备利用率。由于流水线有通过时间(第一个任务输入后到其完成的时间)和排空时间(最后一个任务输入后到完成的时间),所以,在连续完成 n 个任务的时间内,每段都不是在满负荷地工作。

如果各段时间相等,则从图 3.13 可看出,各段的效率 e_i 是相等的,都等于 e_0 。即

$$e_0 = e_1 = e_2 = \cdots = e_m = \frac{n \cdot \Delta t_0}{T_{\text{流水}}} \quad (3.8)$$

所以整个流水线的效率为

$$E = \frac{e_1 + e_2 + \cdots + e_m}{m} = \frac{m \cdot e_0}{m} = \frac{m \cdot n \cdot \Delta t_0}{m \cdot T_{\text{流水}}} \quad (3.9)$$

上式的分母 $m \cdot T_{\text{流水}}$,是时空图中 m 个段和 T 流水时间所围成的总面积;而分子 $m \cdot n \cdot \Delta t_0$ 是时空图中 n 个任务实际占用的面积。所以,从时空图上看,所谓效率,就是 n 个任务占用的时空区和 m 个段总的时空区之比。即效率含有时间和空间两个方面的因素。因为:

$$E = \frac{n \cdot \Delta t_0}{T_{\text{流水}}} = \frac{n}{m + n - 1} = \frac{1}{1 + \frac{m-1}{n}} \quad (3.10)$$

所以,当 $n \gg m$ 时, $E \approx 1$ 。这个结论和上面分析吞吐率及加速比的结论是一致的。比较式(3.6)和式(3.10)可以得出

$$E = \frac{S}{m} \quad \text{或} \quad S = m \cdot E \quad (3.11)$$

可以看出,效率是实际加速比(S)和最大加速比(m)之比。只有当 $E=1$ 时,才有 $S=m$,实际加速比才能达到最大。比较式(3.4)和式(3.10)也可以得出

$$E = TP \cdot \Delta t_0 \quad \text{或} \quad TP = \frac{E}{\Delta t_0} \quad (3.12)$$

即当 Δt_0 不变时,流水线的效率和吞吐率成正比。这就是说,为提高效率所采取的措施,对提高吞吐率也有好处。

如果流水线各段时间不等,此时各段的效率不等。参照图 3.14 可以得出

$$E = \frac{n \text{ 个任务占用的时空区}}{m \text{ 个段总的时空区}} = \frac{n \cdot \sum_{i=1}^m \Delta t_i}{m \left[\sum_{i=1}^m \Delta t_i + (n-1)\Delta t_j \right]} \quad (3.13)$$

比较式(3.5)和式(3.13),可以得出

$$E = TP \cdot \frac{\sum_{i=1}^m \Delta t_i}{m} \quad \text{或} \quad TP = E \cdot \frac{m}{\sum_{i=1}^m \Delta t_i} \quad (3.14)$$

由于除了最慢的段以外,其他段都出现较多的空白区,整个流水线的效率 E 是较低的。而这时, $m / \sum_{i=1}^m \Delta t_i$ 值也是小于 1 的,所以对吞吐率的影响比较严重。因此,像图 3.15 所采用的提高吞吐率的两种办法,都是由于能减少时空图中的空白区而提高效率。

对于非线性流水线和多功能流水线,也可仿照上述对线性流水线的性能分析方法,在正确画出时空图的基础上,分析其吞吐率和效率等。上述性能指标为从各个角度分析流水线的性能提供了依据。

4. 流水线性能分析举例

例 3.1 设在图 3.6(a)所示的静态流水线上计算 $\sum_{i=1}^4 A_i B_i$, 流水线的输出可以直接返回输入端或暂存于相应的流水线寄存器中,试计算其吞吐率、加速比和效率。

解 首先,应选择适合于流水线工作的算法。对于本题,应先计算 $A_1 B_1$ 、 $A_2 B_2$ 、 $A_3 B_3$ 和 $A_4 B_4$;再计算 $A_1 B_1 + A_2 B_2$ 及 $A_3 B_3 + A_4 B_4$;然后求总的累加结果。

其次,画出完成该计算的时空图,如图 3.16 所示,在该图的下面,给出输入、输出的变化,图中阴影部分表示该段在工作。

由图可见,它在 20 个 Δt 时间内,给出 7 个结果。所以吞吐率为

$$TP = 7 / (20\Delta t)$$

如果不用流水线,由于一次求积需 $4\Delta t$,一次求和需 $6\Delta t$,则产生上述 7 个结果共需 $(4 \times 4 + 3 \times 6)\Delta t = 34\Delta t$ 。所以加速比为

$$S = (34\Delta t) / (20\Delta t) = 1.7$$

该流水线的效率可由阴影区和 8 个段总时空区的比值求得

$$E = (4 \times 4 + 3 \times 6) / (8 \times 20) \approx 0.21$$

由此看出:在求解此问题时,该流水线的效率不高。主要原因是,静态多功能流水线在对某种功能进行流水处理时,总有某些段处在空闲状态;在进行功能切换时,增加了前一种功能的排空时间和后一种功能的通过时间;此外,还需要把输出回传到输入(相关)。因此,不是每拍都有数据输入。由此看出,流水线最适合于解输入与输出之间无任何联系与相关的一串相同运算。

例 3.2 假设前面 DLX 非流水线实现的时钟周期时间为 10 ns, ALU 和分支操作需要 4 个时钟周期,访问存储器操作需 5 个时钟周期,上述操作在程序中出现的相对频率分别是:40%、20% 和 40%。在基本的 DLX 流水线中,假设由于时钟扭曲和寄存器建立延迟等原因,流水线要在其时钟周期时间上附加 1 ns 的额外开销。现忽略任何其他延迟因素的影响,请问:相对于非流水实现而言,基本的 DLX 流水线执行指令的加速比是多少?

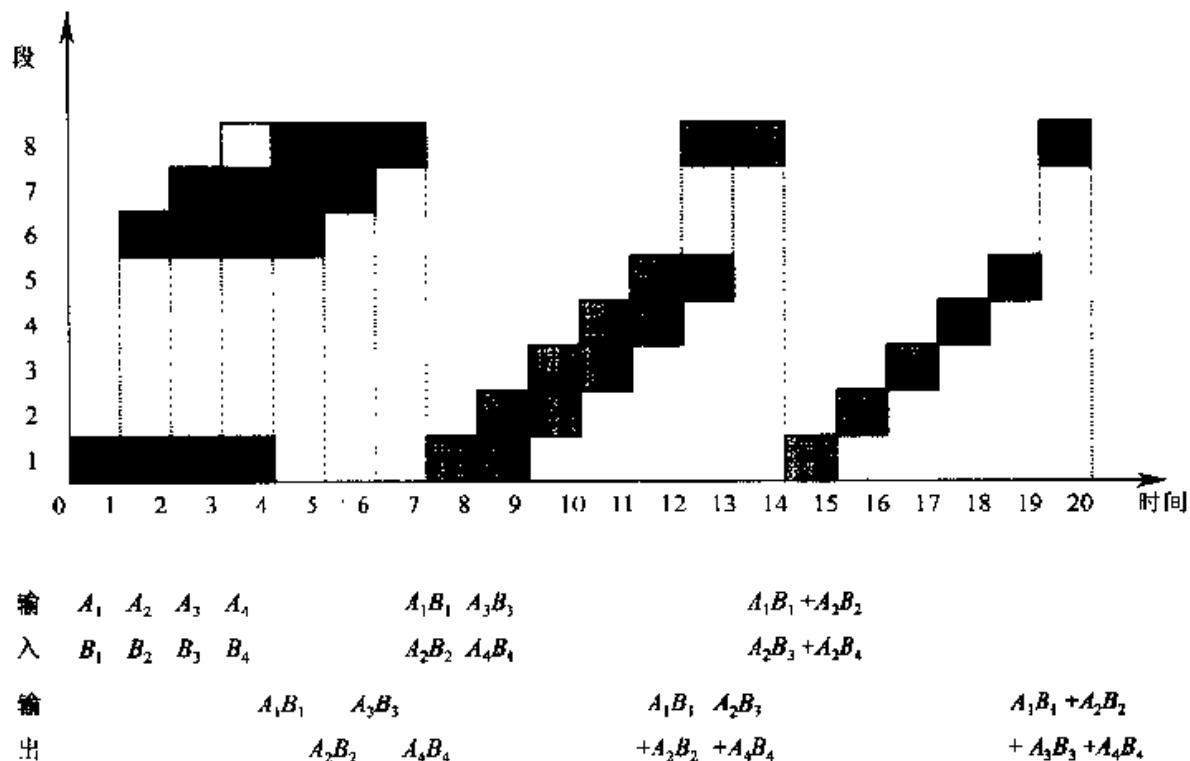


图 3.16 完成乘加运算的多功能静态流水线

解 当非流水执行指令时, 指令的平均执行时间为

$$\begin{aligned}
 TPI_{\text{非流水}} &= 10 \text{ ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5) \\
 &= 10 \text{ ns} \times 4.4 \\
 &= 44 \text{ ns}
 \end{aligned}$$

在流水实现中, 指令执行的平均时间是最慢一段的执行时间加上额外开销, 即

$$TPI_{\text{流水}} = 10 \text{ ns} + 1 \text{ ns} = 11 \text{ ns}$$

所以基本的 DLX 流水线执行指令的加速比为

$$\begin{aligned}
 S &= \frac{TPI_{\text{非流水}}}{TPI_{\text{流水}}} \\
 &= \frac{44 \text{ ns}}{11 \text{ ns}} = 4
 \end{aligned}$$

例 3.3 在 DLX 的非流水实现和基本流水线中, 5 个功能单元所需要的执行时间分别是: 10 ns、8 ns、10 ns、10 ns 和 7 ns。现假设流水线机器的时钟周期时间要附加 1 ns 的额外开销, 求相对于非流水实现指令而言, 基本的 DLX 流水线的加速比是多少?

解 因为非流水机器以单时钟周期方式执行所有指令, 每条指令执行的平

均时间就是机器的时钟周期时间,而时钟周期时间等于指令在执行过程中每一步的执行时间之和,即

$$\begin{aligned} TPI_{\text{非流水}} &= 10 \text{ ns} + 8 \text{ ns} + 10 \text{ ns} + 10 \text{ ns} + 7 \text{ ns} \\ &= 45 \text{ ns} \end{aligned}$$

在流水实现中,指令的平均执行时间是最慢一段的时间加上额外开销,即

$$TPI_{\text{流水}} = 10 \text{ ns} + 1 \text{ ns} = 11 \text{ ns}$$

这是平均指令执行时间。所以流水线的加速比为

$$\begin{aligned} S &= \frac{TPI_{\text{非流水}}}{TPI_{\text{流水}}} \\ &= \frac{45 \text{ ns}}{11 \text{ ns}} = 4.1 \end{aligned}$$

从上面例 3.2 和例 3.3 可以看出,流水线的性能受限于最慢一级流水段的操作时间,流水段的操作时间不平衡限制了流水线的性能。

另外,流水线的额外开销对其性能也有较大影响,这些额外开销主要来自于流水线寄存器的延迟和时钟扭曲。流水线寄存器或锁存器具有一定的建立时间和传输延迟,这些延迟加长了流水线的时钟周期时间。

前面曾谈到增加流水线的段数可以提高流水线的性能,但是流水线段数的增加受限于这些额外开销,因为增加流水线的段数意味着每段的时钟周期时间减小,一旦流水线的时钟周期时间降低到和额外开销一样小的时候,再细分流水线就没有任何作用了,这时在流水线的一个时钟周期内根本没有时间来完成流水段所规定的操作。

正是由于这些额外开销对流水线的性能有较大的影响,所以设计者必须选择高性能的寄存器作为流水线寄存器。Earle 锁存器(1965 年由 J. G. Earle 发明)的以下特点使其成为流水线寄存器的一种较好的选择:

1. 它对时钟扭曲不敏感;锁存器的延迟是一个常数,一般是两级门的延迟,从而避免了数据通过锁存器时可能产生的时钟扭曲;
2. 在锁存器中可以执行两级逻辑运算,而不会增加锁存器的延迟时间。从而可以隐藏锁存器产生的额外开销。

实际上在例 3.1 中,我们已经看到流水线完成任务之间如果存在依赖关系,将对流水线的性能产生较大的影响。对指令流水线来说,也是如此。如果流水线中的每条指令都相互独立,则可以充分发挥上述 DLX 基本流水线的性能。但在实际中,流水线中流动的指令极有可能会相互依赖,即它们之间存在着相关关系,那么在这种情况下又如何提高流水线的性能呢? 这正是下面要着重讨论的问题。

3.3 流水线中的相关

一般来说,流水线中的相关主要分为以下三种类型:

1. 结构相关:当指令在重叠执行过程中,硬件资源满足不了指令重叠执行的要求,发生资源冲突时将产生“结构相关”。
2. 数据相关:当一条指令需要用到前面指令的执行结果,而这些指令均在流水线中重叠执行时,就可能引起“数据相关”。
3. 控制相关:当流水线遇到分支指令和其他会改变 PC 值的指令时就会发生“控制相关”。

一旦流水线中出现相关,必然会给指令在流水线中的顺利执行带来许多问题,如果不能很好地解决相关问题,轻则影响流水线的性能,重则导致错误的执行结果。消除相关的基本方法是让流水线暂停执行某些指令,而继续执行其他一些指令。

在本章中解决相关问题的一些方法中,我们约定:当一条指令被暂停时,在该暂停指令之后发射的所有指令都要被暂停,而在该暂停指令之前发射的指令则可继续进行,在暂停期间,流水线不会取新的指令。本节我们针对上述三种类型的相关进行讨论。

3.3.1 流水线的结构相关

如果某些指令组合在流水线中重叠执行时,产生资源冲突,则称该流水线有结构相关。为了能够在流水线中顺利执行指令的所有可能组合,而不发生结构相关,通常需要采用流水化功能单元的方法或资源重复的方法。

许多流水线机器都是将数据和指令保存在同一存储器中。如果在某个时钟周期内,流水线既要完成某条指令对数据的存储器访问操作,又要完成取指令的操作,那么将会发生存储器访问冲突问题(如图 3.17 所示),产生结构相关。为了解决这个问题,可以让流水线完成前一条指令对数据的存储器访问时,暂停取后一条指令的操作(如图 3.18 所示)。该周期称为流水线的一个暂停周期。暂停周期一般也被称为“流水线气泡”,或简称为“气泡”。从图 3.18 可以看出,在流水线中插入暂停周期可以消除这种结构相关。

也可以用如图 3.19 所示的时空图来表示上述暂停情况。在图 3.19 中,将暂停周期标记为 stall,并将指令 $i+3$ 的取指令操作右移一个时钟周期。所以,流水线要到第 9 个时钟周期才完成指令 $i+3$,在时钟周期 8 时,流水线中没有任何指令流出。

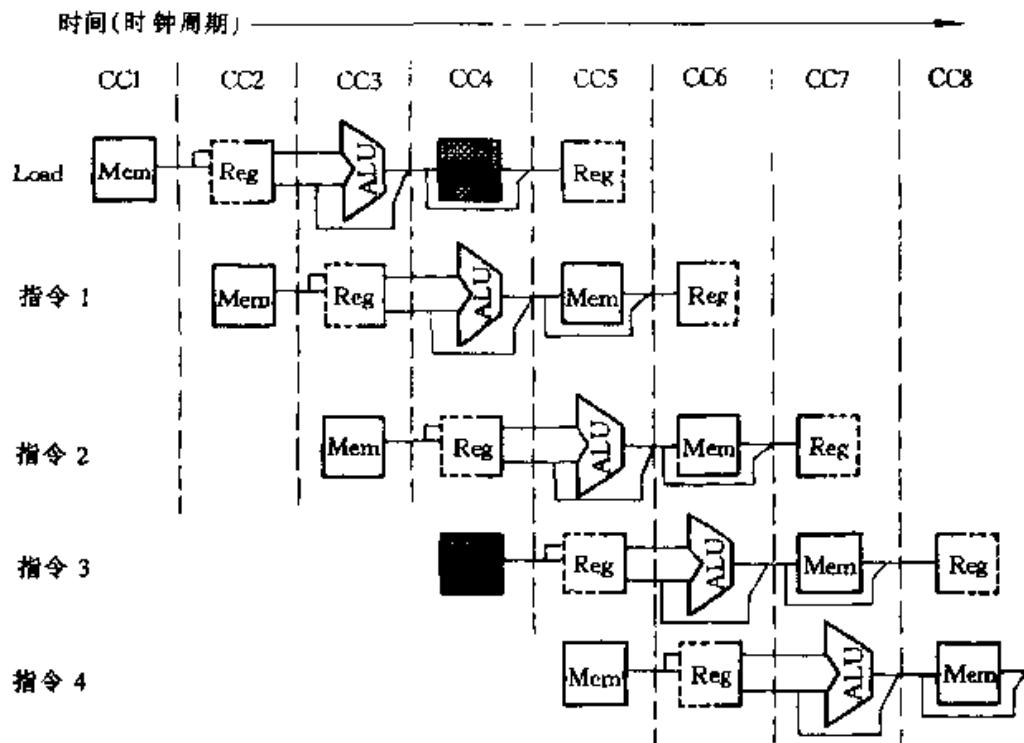


图 3.17 由于存储器访问冲突而带来的流水线结构相关

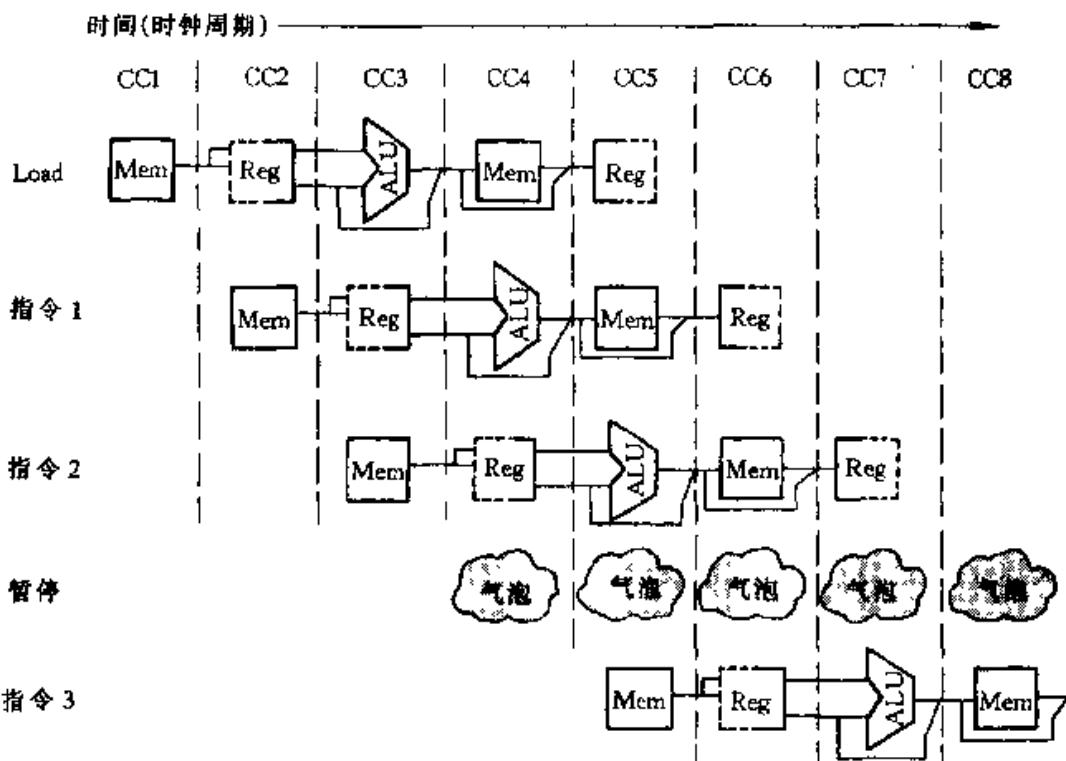


图 3.18 为消除结构相关而插入的流水线气泡

由上可知, 为消除结构相关而引入的暂停将影响流水线的性能。为了避免结构相关, 可以考虑采用资源重复的方法。比如, 在流水线机器中设置相互独立

指令编号	时钟周期									
	1	2	3	4	5	6	7	8	9	10
指令 i	IF	ID	EX	MEM	WB					
指令 $i+1$		IF	ID	EX	MEM	WB				
指令 $i+2$		IF	ID	EX	MEM	WB				
指令 $i+3$			Stall	IF	ID	EX	MEM	WB		
指令 $i+4$				IF	ID	EX	MEM	WB		
指令 $i+5$					IF	ID	EX	MEM		
指令 $i+6$						IF	ID	EX		

图 3.19 由于消除结构相关引入暂停的流水线时空图

的指令存储器和数据存储器;也可以将 Cache 分割成指令 Cache 和数据 Cache。

假设不考虑流水线其他因素对流水线性能的影响,显然如果流水线机器没有结构相关,那么其 CPI 也较小。然而,为什么有时流水线设计者却允许结构相关的存在呢?这主要有两个原因:一是为了减少硬件代价,二是为了减少功能单元的延迟。

如果为了避免结构相关而将流水线中的所有功能单元完全流水化,或者设置足够的硬件资源,那么所带来的硬件代价必定很大。例如,对流水线机器而言,如果要在每个时钟周期内,能够支持取指令操作和对数据的存储器访问操作同时进行,而又不发生结构相关,那么存储总线的带宽必须要加倍。同样,一个完全流水的浮点乘法器需要许多逻辑门。假如在流水线中结构相关并不是经常发生,那么就不值得为了避免结构相关而增加大量硬件代价。

另外,完全可以设计出比完全流水化功能单元具有更短延迟时间的非流水化和不完全流水化的功能单元,如 CDC 7600 和 MIPS R2010 的浮点功能单元就选择了具有较短延迟、而又不是完全流水化的设计方法。

例 3.4 当前许多机器都没有将浮点功能单元完全流水,比如在 DLX 实现中,浮点乘需要 5 个时钟周期,但是对该指令不流水。请分析由此而引起的结构相关对 mdlijdp2 基准程序在 DLX 上运行的性能有何影响?为简单起见,假设浮点乘法服从均匀分布。

解 从第二章的一些测量结果可知,在 mdlijdp2 基准程序中,浮点乘法出现的频率是 14%。而本章所给出的 DLX 流水线处理乘法的频率最高能够达到 20%,即每 5 个时钟周期进行一次浮点乘操作。当浮点乘法不是成群地聚集在一起,而是服从均匀分布时,就表明浮点乘法指令完全流水化所能够获得的性能好处可能很少。最好的情况是,浮点乘法操作和其他操作重叠没有一点性能损失;最坏的情况是,所有的浮点乘法指令聚集在一起,并且这些指令(占总指令数的 14%)需要 5 个时钟周期。因而,如果流水线基本的 CPI 是 1,那么在这种情况下由于流水线暂停所带来的 CPI 增量是 0.56。

3.3.2 流水线的数据相关

1. 数据相关简介

当指令在流水线中重叠执行时,流水线有可能改变指令读/写操作数的顺序,使得读/写操作顺序不同于它们非流水实现时的顺序,这将导致数据相关。首先考虑下列指令在流水线中的执行情况:

ADD	R1,	R2,	R3
SUB	R4,	R1,	R5
AND	R6,	R1,	R7
OR	R8,	R1,	R9
XOR	R10,	R1,	R11

ADD 指令后的所有指令都要用到 ADD 指令的计算结果,如图 3.20 所示,ADD 指令在 WB 段才将计算结果写入寄存器 R1 中,但是 SUB 指令在其 ID 段就要从寄存器 R1 中读取该计算结果,这种情况就叫做“数据相关”。除非有措施防止这一情况出现,否则 SUB 指令读到的是错误的值。所以,为了保证上述指令序列的正确执行,流水线只好暂停 ADD 指令之后的所有指令,直到 ADD 指令将计算结果写入寄存器 R1 之后,再启动 ADD 指令之后的指令继续执行。

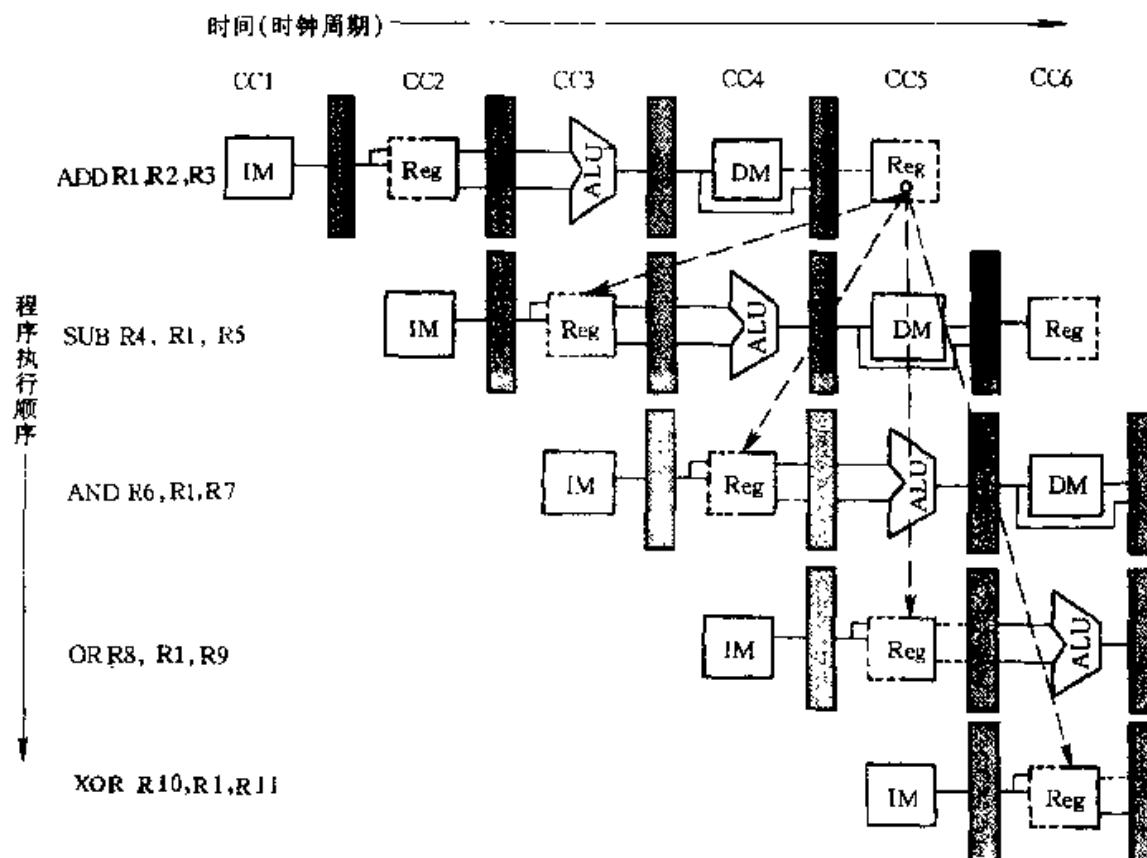


图 3.20 流水线的数据相关

从图 3.20 还可以看到, AND 指令同样也将受到这种相关关系的影响。ADD 指令只有到第五个时钟周期末尾才能结束对寄存器 R1 的写操作, 所以 AND 指令在第四个时钟周期从寄存器 R1 中读出的值也是错误的。而 XOR 指令则可以正常操作, 因为它是在第六个时钟周期才读寄存器 R1 的内容。

另外, 利用 DLX 流水线的一种简单技术, 可以使流水线顺利执行 OR 指令。这种技术就是: 在 DLX 流水线中, 约定在时钟周期的后半部分进行寄存器文件的读操作, 而在时钟周期的前半部分进行寄存器文件的写操作。在本章的图中, 我们将寄存器文件的边框适当地画成虚线来表示这种技术。

2. 通过定向技术减少数据相关带来的暂停

图 3.20 中的数据相关问题可以采用一种称为定向(也称为旁路或短路)的简单技术来解决。定向技术的主要思想是: 在某条指令(如图 3.20 中的 ADD 指令)产生一个计算结果之前, 其他指令(如图 3.20 中的 SUB 和 AND 指令)并不真正需要该计算结果, 如果能够将该计算结果从其产生的地方(寄存器文件 EX/MEM)直接送到其他指令需要它的的地方(ALU 的输入寄存器), 那么就可以避免暂停。基于这种考虑, 定向技术的要点可以归纳为:

- (1) 寄存器文件 EX/MEM 中的 ALU 的运算结果总是回送到 ALU 的输入寄存器。
- (2) 当定向硬件检测到前一个 ALU 运算结果的写入寄存器就是当前 ALU 操作的源寄存器时, 那么控制逻辑将前一个 ALU 运算结果定向到 ALU 的输入端, 后一个 ALU 操作就不必从源寄存器中读取操作数。

图 3.20 还表明, 流水线中的指令所需要的定向结果可能并不仅仅是前一条指令的计算结果, 而且还有可能是前面与其不相邻指令的计算结果。图 3.21 是采用了定向技术后上述例子的执行情况, 其中寄存器文件和功能单元之间的虚线表示定向路径。上述指令序列可以在图 3.21 中顺利执行而无需暂停。

上述定向技术可以推广到更一般的情况, 可以将一个结果直接传送到所有需要它的功能单元。也就是说, 一个结果不仅可以从某一功能单元的输出定向到其自身的输入, 而且还可以定向到其他功能单元的输入。考虑以下指令序列:

ADD	R1,	R2,	R3
LW	R4,	0(R1)	
SW	12(R1),		R4

不难看出, 这三条指令之间都存在数据相关关系。为了消除由于这些数据相关而带来的暂停, 可以采用如图 3.22 所示的定向技术, 将寄存器 R1 的值定向到 ALU, 将寄存器 R4 的值定向到数据存储器的输入。

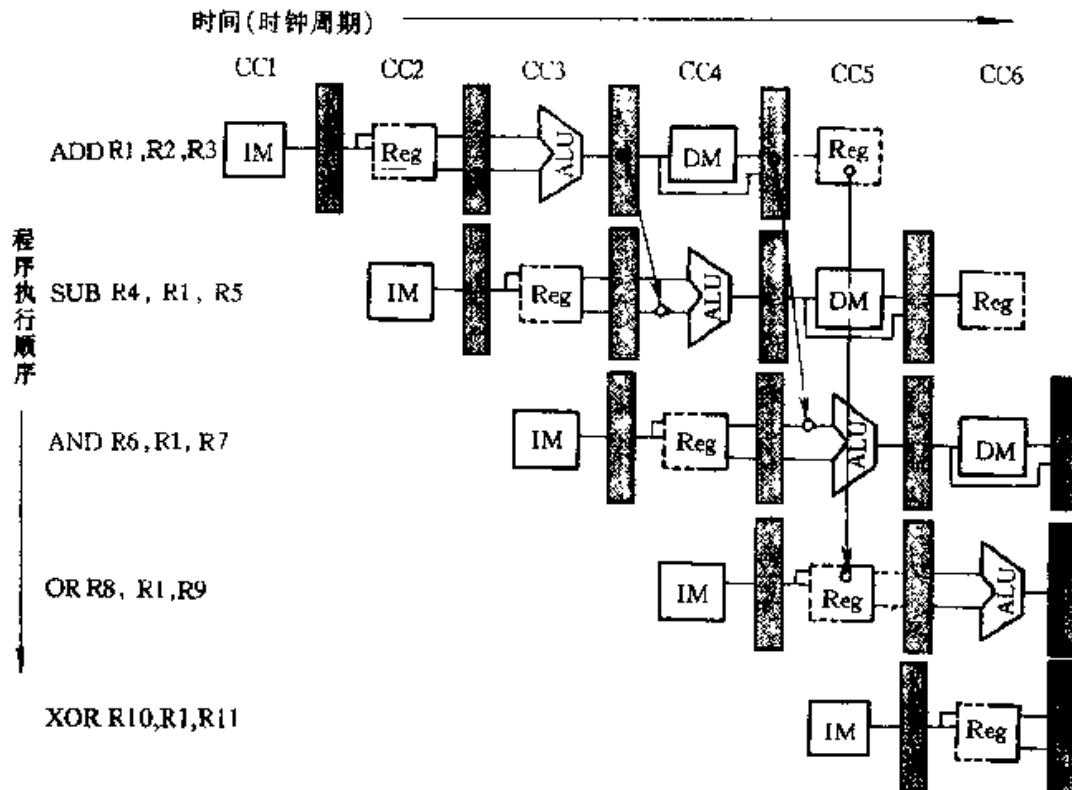


图 3.21 采用定向技术消除数据相关

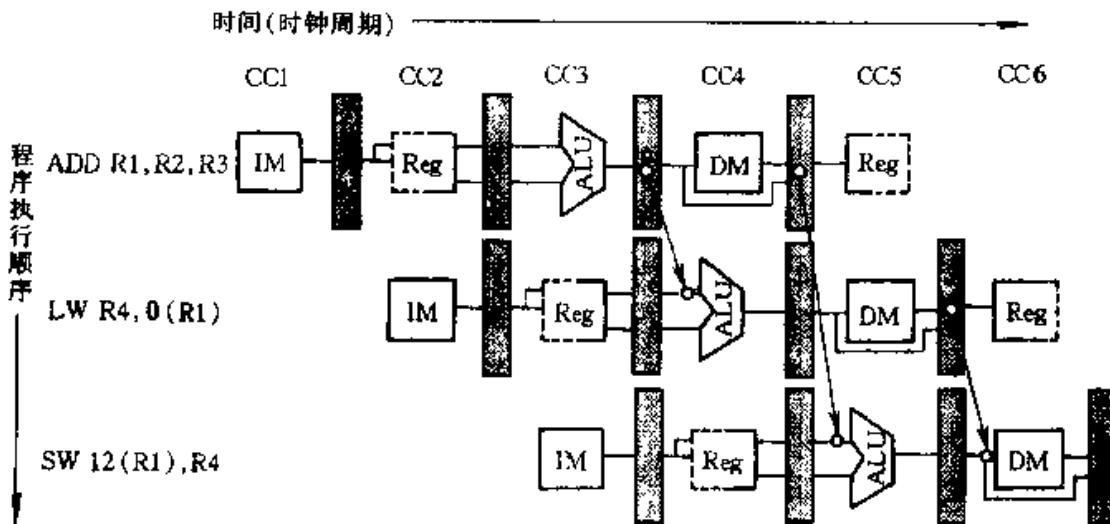


图 3.22 到数据存储器和 ALU 单元的定向路径

在 DLX 中,任何流水线寄存器到任何功能单元的输入都可能需要定向路径。由于 ALU 和数据存储器均要接收操作数,所以需要设置从寄存器文件 EX/MEM 和 MEM/WB 到这两个单元输入的定向路径。除此之外,DLX 的零检测单元在 EX 周期完成分支条件检测操作,当然也需要设置到该单元的定向路径。在后面的有关内容中,将进一步说明 DLX 流水线所有必需的定向路径,并讨论对这些定向路径的控制方法。

前面的一些数据相关的实例均是有关寄存器操作数的,但是数据相关也有可能发生在一对指令对存储器同一单元进行读写的时候。比如,如果 DLX 流水线允许某条指令访问 Cache 失效时,继续向前流水该指令之后的指令,那么可能会引起存储器访问顺序的混乱。所以,DLX 流水线规定:当某条指令访问 Cache 失效时,暂停该指令之后的所有指令,以保证访问存储器的正确顺序。

下一章将讨论一种允许存储器访问顺序不同于其在非流水实现中访问顺序的流水线机器,而这一章讨论的所有数据相关仅涉及 CPU 内部的寄存器。

3. 数据相关的分类

根据指令对寄存器的读写顺序,可以将数据相关分为如下三种类型。习惯上,这些相关是根据流水线所必须保持的访问顺序来命名的。考虑流水线中的两条指令 i 和 j ,且 i 在 j 之前进入流水线,由此可能带来的数据相关有:

(1) 写后读相关(RAW: Read After Write): j 的执行要用到 i 的计算结果,但是当其在流水线中重叠执行时, j 可能在 i 写入其计算结果之前就先行对保存该结果的寄存器进行读操作,从而得到错误的值。这是最常见的一种数据相关,图 3.21 和图 3.22 中采用定向技术消除的数据相关就属于这种类型。

(2) 写后写相关(WAW: Write After Write): j 和 i 的目的操作数一样,但是当其在流水线中重叠执行时, j 可能在 i 写入其计算结果之前就先行对保存该结果的寄存器进行写操作,从而导致写入顺序错误,在目的寄存器中留下的是 i 写入的值,而不是 j 写入的值。

如果在流水线中不只一个段可以进行写操作,或者当流水线暂停某条指令时,允许该指令之后的指令继续前进,就可能会产生这种类型的数据相关。由于 DLX 流水线只在 WB 段写寄存器,所以在 DLX 流水线中执行的指令不会发生这种类型的数据相关。

如果对 DLX 流水线作如下改变,在 DLX 流水线中执行的指令就有可能发生 WAW 相关。首先,将 ALU 运算结果的写回操作移到 MEM 段进行,因为这时计算结果已经有效;其次,假设访问数据存储器占两个流水段。下面是两条指令在修改后的 DLX 流水线中执行的情况:

LW	R1,	0	(R2)	IF	ID	EX	MEM1	MEM2	WB
ADD	R1,	R2,	R3		IF	ID	EX	WB	

可以看出,在修改后的 DLX 流水线中执行上述指令序列后,寄存器 R1 中的内容是第一条指令(LW)的写入结果,而不是 ADD 指令的写入结果。这就是由于 WAW 相关所带来的错误执行结果。

(3) 读后写相关(WAR: Write After Read): j 可能在 i 读取某个源寄存器的内容之前就先对该寄存器进行写操作,导致 i 后来读取到的值是错误的。

由于 DLX 流水线在 ID 段完成所有的读操作, 在 WB 段完成所有的写操作。所以, 在 DLX 流水线中不会产生这种类型的数据相关。

基于上面修改后的 DLX 流水线, 考察下面两条指令的执行情况:

SW R2, 0(R5)	IF	ID	EX	MEM1	MEM2	WB
ADD R2, R3, R4		IF	ID	EX	WB	

如果 SW 指令在 MEM2 段的后半部分读取寄存器 R2 的值, ADD 指令在 WB 段的前半部分将计算结果写回寄存器 R2, 则 SW 将读取错误的值, 将 ADD 指令的计算结果写入存储器中。

值得注意的是, 在读后读(RAR: Read After Read)的情况下, 不存在数据相关问题。

4. 需要暂停的数据相关

前面讨论了如何利用定向技术消除由于数据相关带来的暂停。但是, 并不是所有数据相关带来的暂停都可以通过定向技术消除。考虑以下指令序列:

LW	R1,	0(R2)	
SUB	R4,	R1,	R5
AND	R6,	R1,	R7
OR	R8,	R1,	R9

图 3.23 给出了该指令序列在流水线中执行时所需要的定向路径。可以看出, LW 指令要到第四个时钟周期末尾才能够从存储器中读出数据, 而 SUB 指令在第四个时钟周期开始就需要这一数据。显然, 简单的定向并不能解决该数据相关问题。

为了保证流水线正确执行上述指令序列, 可以设置一个称为“流水线互锁”(pipeline interlock)的功能部件。一旦流水线互锁检测到上述数据相关, 流水线暂停执行 LW 指令之后的所有指令, 直到能够通过定向解决该数据相关为止(如图 3.24 所示)。

从图 3.24 可以看出, 流水线互锁在流水线中插入了暂停和气泡。由于气泡的插入, 使得上述指令序列的执行时间增加了一个时钟周期。在这种情况下, 暂停的时钟周期数称为“载入延迟”。流水线在第四个时钟周期没有启动任何指令, 在第六个时钟周期没有流出执行完毕的指令。图 3.25 是加入暂停前后的流水线时空图。

例 3.5 假设某指令序列中 20% 的指令是 Load 指令, 并且紧跟在 Load 指令之后的半数指令需要使用到载入的结果, 如果这种数据相关将产生一个时钟周期的延迟。理想流水线(没有任何延迟, CPI 为 1)的指令执行速度要比这种真实流水线的快多少?

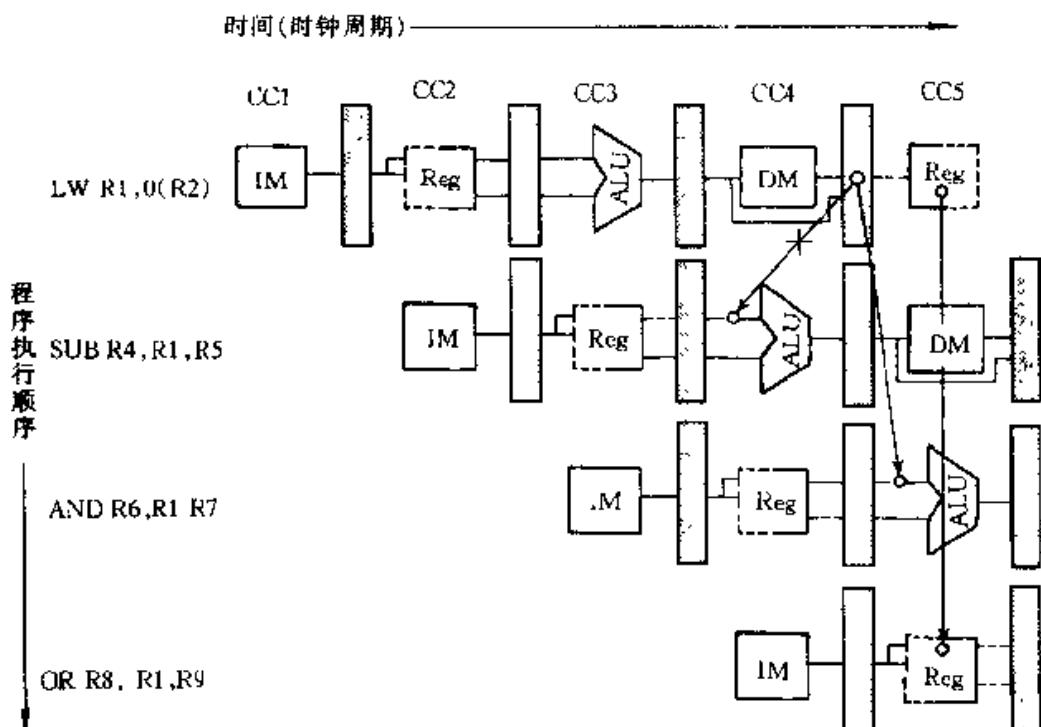


图 3.23 LW 指令不能将结果定向到 SUB 指令

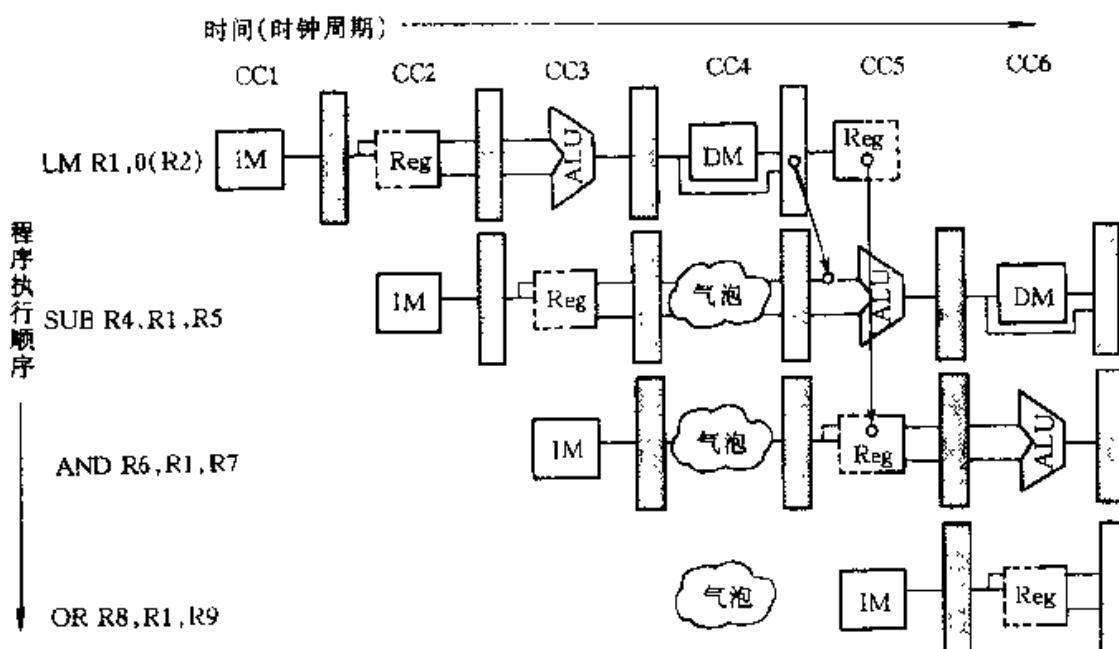


图 3.24 流水线互锁插入暂停后的流水线数据通路

解 我们可以利用 CPI 作为衡量标准。对于真实的流水线而言,由于 Load 指令之后的半数指令需要暂停,所以这些被暂停指令的 CPI 是 2。又知 Load 指令占全部指令的 20%,所以真实流水线的实际 CPI 为: $(0.9 \times 1 + 0.1 \times 2) = 1.1$, 这表示理想流水线的指令执行速度是其执行速度的 1.1 倍。

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB	
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB
OR R8, R1, R9				stall	IF	ID	EX	MEM

图 3.25 插入暂停(stall)前后的流水线时空图

下面讨论如何利用编译器技术来减少这种必须的暂停,然后论述如何在流水线中实现数据相关检测和定向。

5. 对数据相关的编译器调度方法

流水线常常会遇到许多种类型的暂停。比如,采用典型的代码生成方法对 $A = B + C$ 这种常用的表达式进行处理,可以得到如图 3.26 所示的指令序列。从图 3.26 可以看出,在 ADD 指令的流水过程中必须插入一个暂停时钟周期,以保证变量 C 的读入值有效。既然定向无法消除指令序列中所包含的这种暂停,那么能否让编译器在进行代码生成时就消除这些潜在的暂停呢?

LW R1, B	IF	ID	EX	MEM	WB				
LW R2, C		IF	ID	EX	MEM	WB			
ADD R3, R1, R2			IF	ID	stall	EX	MEM	WB	
SW A, R3				IF	stall	ID	EX	MEM	WB

图 3.26 $A = B + C$ 的 DLX 代码序列及其流水线实现时空图表示

实际上,编译器的确可以通过重新组织代码顺序来消除这种暂停。通常称这种重新组织代码顺序消除暂停的技术为“流水线调度”(pipeline scheduling)或“指令调度”(instruction scheduling)。

例 3.6 请为下列表达式生成没有暂停的 DLX 代码序列。假设载入延迟为 1 个时钟周期。

$$a = b + c;$$

$$d = e - f;$$

解 调度前后的指令序列如表 3.2 所示。可以看出,两条 ALU 指令(ADD Ra, Rb, Rc 和 SUB Rd, Re, Rf)分别和两条 Load 指令(LW Rc, c 和 LW Rf, f)之间存在数据相关。为了保证流水线正确执行调度前的指令序列,必须在指令执行过程中插入两个时钟周期的暂停。但是考察调度后的指令序列不难发现,由于流水线允许定向,就不必在指令执行过程中插入任何暂停周期。

表 3.2 调度前后的代码序列

调度前代码		调度后代码	
LW	Rb,b	LW	Rb,b
LW	Re,c	LW	Re,c
ADD	Ra,Rb,Re	LW	Re,e // 避免暂停 ADD 指令；
SW	a,Ra	ADD	Ra,Rb,Re
LW	Re,e	LW	Rf,f
LW	Rf,f	SW	a,Ra // 避免暂停 SUB 指令；
SUB	Rd,Re,Rf	SUB	Rd,Re,Rf
SW	d,Rd	SW	d,Rd

6. 对 DLX 流水线控制的实现

让一条指令从流水线的指令译码段(ID)移动到执行段(EX)的过程通常称为指令发射(instruction issue)，而经过了该过程的指令为已发射的(issued)指令。

对于 DLX 标量流水线而言，所有的数据相关均可以在流水线的 ID 段检测到，如果存在数据相关，指令在其发射之前就会被暂停。这样，我们可以在 ID 段决定需要什么样的定向，然后设置相应的控制。在流水线中较早地检测到相关，可以降低实现流水线的硬件复杂度，因为这样不必被迫在流水过程中将一条已经改变了机器状态的指令挂起。另外一种方法是在使用一个操作数的时钟周期开始(DLX 流水线的 EX 和 MEM 段的开始)检测相关，确定必需的定向。

为了说明这两种方法的不同，我们将以 Load 指令所引起的 RAW 相关为例，论述如何通过在 ID 段的检测来实现流水线控制，其中到 ALU 输入的定向路径可以在 EX 段实现。表 3.3 列出了流水线相关检测硬件可以检测到的各种相关情况。

表 3.3 流水线相关检测硬件可以检测到的各种相关情况

相关情况	指令序列范例	动作
没有相关	LW R1,45(R2)	
	ADD R5,R6,R7	因为紧跟着的三条指令和 R1 之间没有相关，所以不需要暂停 LW 指令之后的指令
	SUB R8,R6,R7	
	OR R9,R6,R7	

续表

相关情况	指令序列范例	动作
需要暂停的相关	LW R1,45(R2) ADD R5,R1,R7 SUB R8,R6,R7 OR R9,R6,R7	比较器检测到 ADD 指令中使用了寄存器 R1，并在 ADD 指令开始 EX 周期之前暂停 ADD 指令（同时也暂停了 SUB 指令和 OR 指令）
通过定向消除的相关	LW R1,45(R2) ADD R5,R6,R7 SUB R8,R1,R7 OR R9,R6,R7	比较器检测到 SUB 指令中使用了寄存器 R1，并在 SUB 指令的 EX 周期开始时将载入的结果定向到 ALU
按顺序访问的相关	LW R1,45(R2) ADD R5,R6,R7 SUB R8,R6,R7 OR R9,R1,R7	不需要任何动作，因为 OR 指令在 ID 段的后半个周期读 R1，LW 指令已经在 WB 段的前半个周期将载入数据写入了 R1

现在来看看如何实现流水线互锁。如果某条指令和 Load 指令有一个 RAW 相关时，该指令处于 ID 段，Load 指令处于 EX 段。我们可以用表 3.4 来描述此时所有可能的相关情况。

表 3.4 指令在 ID 段为检测是否需启动流水线锁而进行的三种比较

ID/EX 的操作码域 (ID/EX, IR _{0..5})	IF/ID 的操作码域 (IF/ID, IR _{0..5})	匹配操作数域
Load	R-R ALU	ID/EX, IR _{11..15} = IF/ID, IR _{6..10}
Load	R-R ALU	ID/EX, IR _{11..15} = IF/ID, IR _{11..15}
Load	Load、Store、ALU 立即值或分支	ID/EX, IR _{11..15} = IF/ID, IR _{6..10}

一旦硬件检测到上述 RAW 相关，流水线互锁必须在流水线中插入暂停周期，使正处于 IF 和 ID 段的指令不再前进。如同 3.2 节中所说，在流水线寄存器文件中附带了所有的控制信息，所以当硬件检测到一个相关，只需将 ID/EX 流水线寄存器组中的控制寄存器改为全 0 即可，全 0 表示不进行任何操作。另外，还必须暂停向前传送 IF/ID 寄存器组的内容，使得流水线能够保持被暂停的指令。

对定向而言，虽然可能要考虑许多情况，但是定向逻辑的实现方法是类似的。实现定向逻辑的关键是，流水线寄存器不仅包含了被定向的数据，而且包含了目标

和源寄存器域。从上面的讨论可知,所有定向都是从 ALU 或数据存储器的输出到 ALU、数据存储器或 O 检测单元的输入的定向,我们可以分别将 EX/MEM 和 MEM/WB 段的寄存器 IR 同 ID/EX 和 EX/MEM 段中的寄存器 IR 相比较,决定是否需要定向,从而实现必需的定向控制。比如,当定向目标是 EX 段的 ALU 输入时,定向硬件需要进行的一些比较和定向操作如表 3.5 所示。

表 3.5 定向目标是 ALU 输入时,所需进行的比较和定向操作

包含定向 源的流水 线寄存器	定向源相 应指令的 操作码	包含定向 目标的流 水线寄存器	定向目标相应指令的操作码	定向的目标	比较操作(如 果相等就定 向)
EX/MEM	R-R ALU	ID/EX	R-R ALU、ALU 立即值、 Load、Store、分支	ALU 上面的输入	EX/MEM.IR _{46..30} = ID/EX.IR _{6..10}
EX/MEM	R-R ALU	ID/EX	R-R 类型的 ALU	ALU 下面的输入	EX/MEM.IR _{46..30} = ID/EX.IR _{11..15}
MEM/WB	R-R ALU	ID/EX	R-R ALU、ALU 立即值、 Load、Store、分支	ALU 上面的输入	MEM/WB.IR _{46..30} = ID/EX.IR _{6..10}
MEM/WB	R-R ALU	ID/EX	R-R ALU	ALU 下面的输入	MEM/WB.IR _{46..30} = ID/EX.IR _{11..15}
EX/MEM	ALU 立即值	ID/EX	R-R ALU、ALU 立即值、 Load、Store、分支	ALU 上面的输入	EX/MEM.IR _{41..15} = ID/EX.IR _{6..10}
EX/MEM	ALU 立即值	ID/EX	R-R ALU	ALU 下面的输入	EX/MEM.IR _{41..15} = ID/EX.IR _{11..15}
MEM/WB	ALU 立即值	ID/EX	R-R ALU、ALU 立即值、 Load、Store、分支	ALU 上面的输入	MEM/WB.IR _{41..15} = ID/EX.IR _{6..10}
MEM/WB	ALU 立即值	ID/EX	R-R ALU	ALU 下面的输入	MEM/WB.IR _{41..15} = ID/EX.IR _{11..15}
MEM/WB	Load	ID/EX	R-R ALU、ALU 立即值、 Load、Store、分支	ALU 上面的输入	MEM/WB.IR _{41..15} = ID/EX.IR _{6..10}
MEM/WB	Load	ID/EX	R-R ALU	ALU 下面的输入	MEM/WB.IR _{41..15} = ID/EX.IR _{11..15}

定向的控制硬件除了需要用比较器和组合逻辑来确定什么时候打开哪一条

定向路径之外,还需要在 ALU 输入端采用具有多个输入的多路器,并增加相应的定向路径连接通路。改进图 3.12 中的相关硬件,可以得到图 3.27,图中的虚线表示流水线所增设的定向路径。

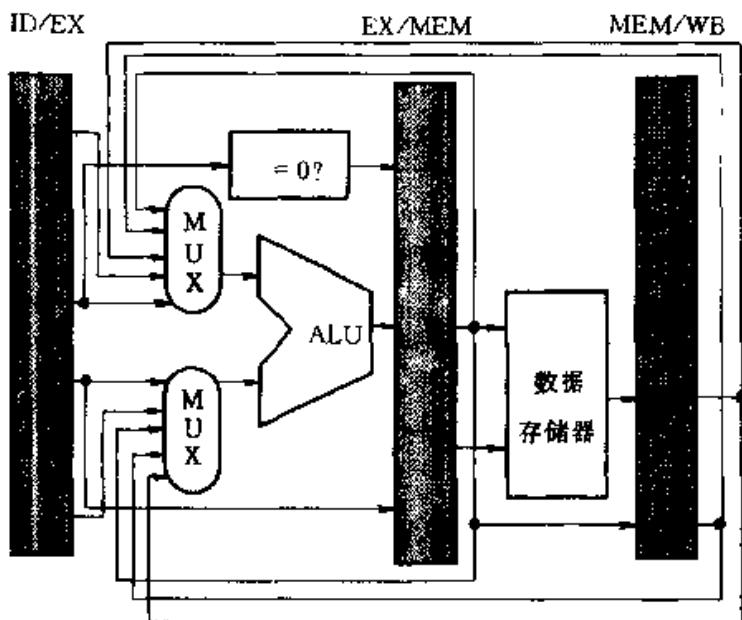


图 3.27 流水线增设的定向路径

3.3.3 流水线的控制相关

在 DLX 流水线上执行分支指令时,PC 值有两种可能的变化情况。一种是 PC 值发生改变(为分支转移的目标地址);一种是 PC 值保持正常(等于其当前值加 4)。如果一条分支指令将 PC 值改变为分支转移的目标地址,那么我们称分支转移“成功”;如果分支转移条件不成立,PC 值保持正常,我们称分支转移“失败”。

处理分支指令最简单的方法是:一旦在流水线中检测到某条指令是分支指令,就暂停执行该分支指令之后的所有指令,直到分支指令到达流水线的 MEM 段,确定了新的 PC 值为止。我们当然不希望在流水线还没有确定某条指令是分支指令之前就暂停执行指令,所以对分支指令而言,在流水线完成其译码操作(ID 段)之后才会暂停执行其后继指令。

根据上述处理分支指令的方法,可以得到如图 3.28 所示的流水线时空图。从图中可以看出,在流水线中插入了两个暂停周期,当分支指令在 MEM 段确定新的 PC 值后,流水线作废分支直接后继指令的 IF 周期(相当于一个暂停周期),按照新的有效 PC 值取指令。显然,分支指令给流水线带来了三个时钟周期的暂停。

在暂停周期中,可以通过设置 IF/ID 寄存器文件的 IR 域为 0 (noop 操作) 来实现上述暂停。你可能已经注意到: 如果图 3.28 中的分支指令是失败的,那么重复分支直接后继指令的 IF 周期就毫无必要,因为流水线实际上已经取出了正确的指令。后面将充分利用这一事实来改善流水线处理分支指令的性能。

分支指令	IF	ID	EX	MEM	WB					
分支后继指令		IF	stall	stall	IF	ID	EX	MEM	WB	
分支后继指令+1						IF	ID	EX	MEM	WB
分支后继指令+2							IF	ID	EX	MEM
分支后继指令+3								IF	ID	EX
分支后继指令+4									IF	ID
分支后继指令+5										IF

图 3.28 简单处理分支指令的方法及其流水线时空图

如前所述,如果流水线处理每条分支指令,都要暂停三个时钟周期,这必然会给流水线的性能带来相当大的损失。比如,假设分支指令在目标代码中出现的频率是 30%,流水线理想的 CPI 为 1,那么具有上述暂停的流水线只能达到理想加速比的一半,所以降低分支损失对充分发挥流水线的效率十分关键。

减少流水线处理分支指令时的暂停时钟周期数有如下两种途径:

1. 在流水线中尽早判断出分支转移是否成功;
2. 尽早计算出分支转移成功时的 PC 值(即分支的目标地址)。

为了优化处理分支指令,在流水线中应该同时采用上述两条途径,缺一不可。即使知道分支转移的目标地址,而不知道分支转移是否成功对减少暂停是徒劳的; 知道分支转移是否成功,而不知道分支转移的目标地址,同样对降低分支损失毫无帮助。下面让我们看看如何基于这些思想,从硬件上改进 DLX 流水线,达到降低分支损失的目的。

在 DLX 流水线中,分支指令(BEQZ 和 BENZ)需要测试分支条件寄存器的值是否为 0,所以可以把测试分支条件寄存器的操作移到 ID 段完成,从而使得在 ID 周期末就完成分支转移成功与否的检测。

另外,由于要尽早计算出两个 PC 值(分支转移成功和失败时的 PC 值),也可以将计算分支目标地址的操作移到 ID 段完成。为此,需要在 ID 段增设一个加法器(注意,为了避免结构相关,不能用 EX 段的 ALU 功能部件来计算分支转移目标地址)。图 3.29 是对 DLX 流水线进行上述改进后的流水线数据通路。容易看出,基于上述改进后的流水线数据通路,处理分支指令只需要一个时钟周期的暂停。表 3.6 列出了在改进后的流水线数据通路上处理分支指令的一些操作。

表 3.6 改进后流水线的分支操作

流水段	分支指令操作
IF	IF/ID. IR \leftarrow Mem[PC]; IF/ID. NPC, PC \leftarrow (f ID/EX. cond {ID/EX. NPC} else {PC + 4});
ID	ID/EX. A \leftarrow Regs[IF/ID. IR _{6..10}]; ID/EX. B \leftarrow Regs[IF/ID. IR _{11..15}]; ID/EX. NPC \leftarrow IF/ID. NPC + (IR ₁₆) ¹⁶ # # IR _{16..31} ; ID/EX. IR \leftarrow IF/ID. IR; ID/EX. cond \leftarrow (Regs[IF/ID. IR _{6..10}] op 0); ID/EX. Imm \leftarrow (IR ₁₆) ¹⁶ # # IR _{16..31} ;
EX	
MEM	
WB	

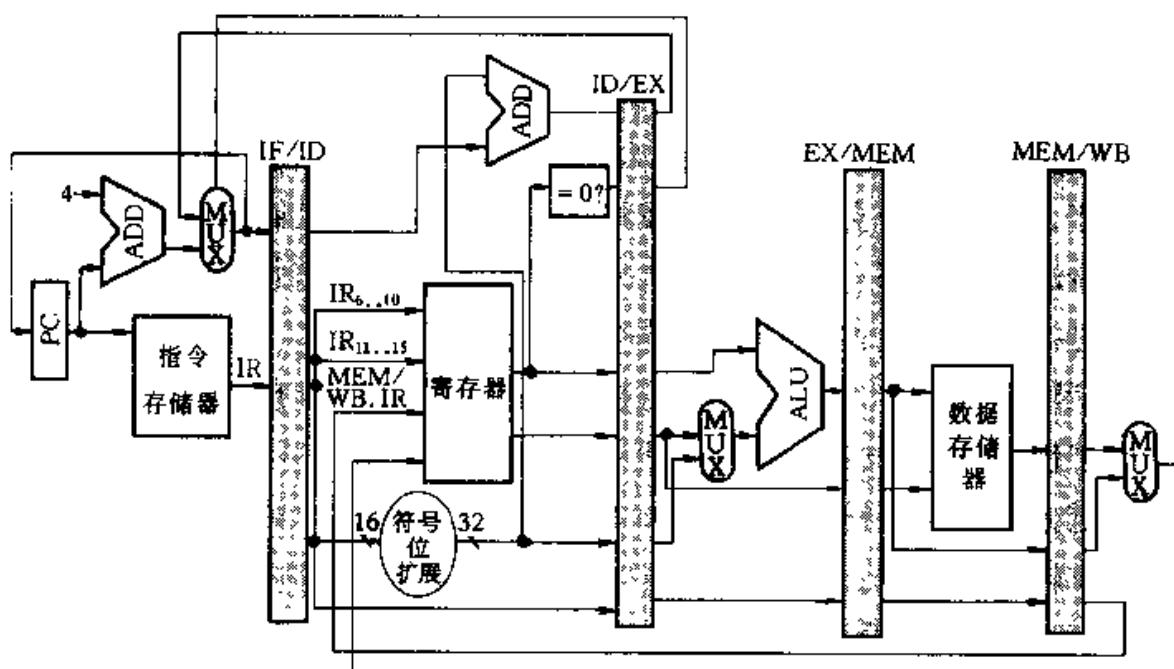


图 3.29 为降低分支损失,对图 3.12 进行改进后的流水线数据通路

对某些机器的流水线而言,由于检测分支条件和计算分支转移目标地址需要更长的时间,所以处理分支指令所带来的控制相关可能需要更多的时钟周期。一般来说,流水线越深,处理分支指令所带来的控制相关所需要的时钟周期数就越多。

在讨论各种减少分支损失的方法之前,首先看看程序中分支的行为特点。

1. 程序中分支的行为特点

既然分支指令对流水线的性能有非常大的影响,所以对程序中分支行为特点的研究,有助于总结分支指令的行为规律,从中获取一些减少流水线性能损失的思想和依据。基于 SPEC 基准程序集测量各种能够改变 PC 值的指令的执行频率,可以得到如图 3.30 所示的统计结果。

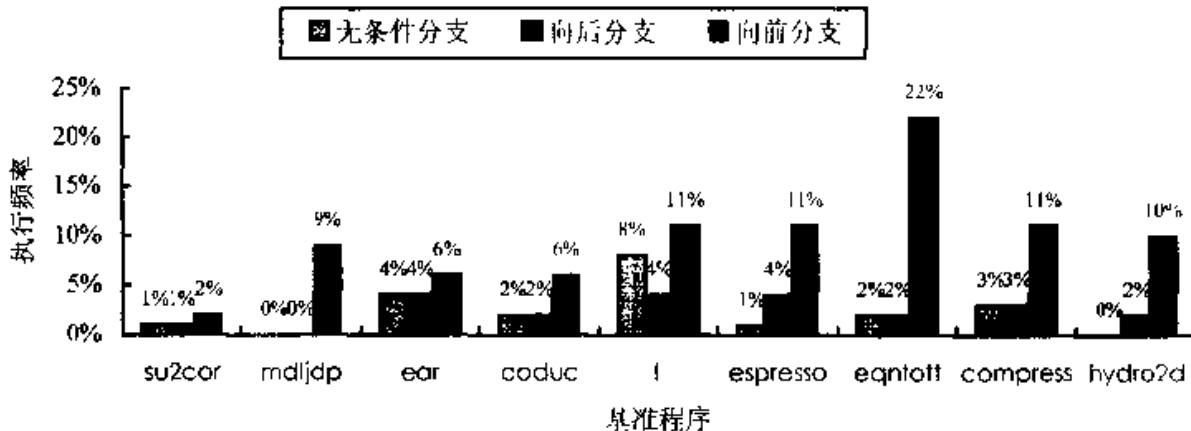


图 3.30 各种能够改变 PC 值的指令的执行频率

需要说明的是,图中的“无条件分支”类型包括无条件分支指令、跳转指令、过程调用和返回指令。从图 3.30 可以看出,整型基准程序的条件分支指令执行频率约为 14%~16%,而无条件分支指令的执行频率却非常低(li 基准程序的无条件分支指令的执行频率比较高,主要是因为 li 基准程序包含大量的过程调用);对于浮点基准程序而言,其条件分支指令的执行频率在 3%~12% 之间,但是从整体来看,其条件分支和无条件分支指令的执行频率都要比整型基准程序的低。另外,向前(forward)分支指令执行频率大约是向后(backward)分支指令执行频率的 1~3 倍。

除此之外,条件分支是否成功对流水线的性能也有较大影响。图 3.31 是用 SPEC 基准程序对条件分支成功的概率进行测量统计的结果。从图中可以看出,平均大约有 67% 的条件分支指令都是成功的。

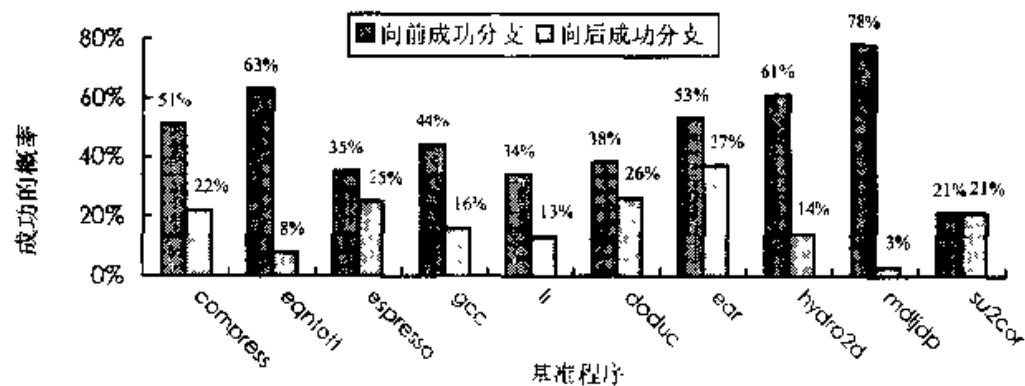


图 3.31 条件分支指令分支转移成功的概率

结合图 3.30 和图 3.31 的统计结果,可以估算出:向前条件分支指令分支转移成功的概率约为 60%,向后条件分支指令分支转移成功的概率约为 85%。向后条件分支指令分支转移成功的概率比向前条件分支指令的高。由于向后条件分支指令一般和程序中的循环相对应,所以它具有较高的分支转移成功概率,这

对降低流水线损失是非常有利的。

2. 降低流水线分支损失的方法

降低流水线分支损失的方法有许多种。前面论述了改进流水线硬件减少流水线暂停周期的方法。这里主要从编译技术的角度,论述四种降低流水线分支损失的简单方法。首先需要说明的是,这些方法对分支转移成功与否进行的预测都是静态的,并在整个程序的执行过程中保持这种预测结论,即:要么总是认为分支转移成功,要么总是认为分支转移失败。

(1) “冻结”(freeze)或“排空”(flush)流水线的方法

在流水线中,处理分支最简单的方法是“冻结”或“排空”流水线,保持或清除流水线在分支指令之后读入的任何指令,直到知道分支指令的目标地址以及分支转移是否成功为止。这种方法优点在于其对硬件和软件的要求都十分简单,我们前面采用的就是这种方法,这里不再赘述。

(2) “预测分支失败”的方法

如果流水线采用“预测分支失败”的方法处理分支指令,那么当流水线译码到一条分支指令时,流水线继续取指令,并允许该分支指令后的指令继续在流水线中流动。当流水线确定分支转移成功与否以及分支的目标地址之后,如果分支转移成功,流水线必须将在分支指令之后取出的所有指令转化为空操作,并在分支的目标地址处重新取出有效的指令;如果分支转移失败,那么可以将分支指令看作是一条普通指令,流水线正常流动,无需将在分支指令之后取出的所有指令转化为空操作。

流水线采用“预测分支失败”方法处理分支指令的时空图如图 3.32 所示。

失败的分支指令	IF	ID	EX	MEM	WB
指令 $i+1$	IF	ID	EX	MEM	WB
指令 $i+2$		IF	ID	EX	MEM
指令 $i+3$			IF	ID	WB
指令 $i+4$				IF	ID

成功的分支指令	IF	ID	EX	MEM	WB
指令 $i+1$	IF	idle	idle	idle	idle
分支目标		IF	ID	EX	MEM
分支目标+1			IF	ID	WB
分支目标+2				IF	ID

图 3.32 基于“预测分支失败”方法的流水线时空图

(3) “预测分支成功”的方法

另一种降低流水线分支损失的方法便是“预测分支成功”方法,一旦流水线译码到一条指令是分支指令,且完成了分支目标地址的计算,我们就假设分支转

移成功，并开始在分支目标地址处取指令执行。

对 DLX 流水线而言，因为在知道分支转移成功与否之前，无法知道分支目标地址（*O* 检测和计算分支目标地址均在 ID 段完成），所以这种方法对降低 DLX 流水线分支损失没有任何好处。而在某些流水线中，特别是那些具有隐含设置条件码或分支条件更复杂指令的流水线机器中，在确定分支转移成功与否之前，便可以知道分支的目标地址，这时采用这种方法便可以降低这些流水线的分支损失。

(4) “延迟分支”(delayed branch)方法

为降低流水线分支损失而采用的第四种方法就是“延迟分支”方法。其主要思想是从逻辑上“延长”分支指令的执行时间。延迟长度为 *n* 的分支指令的执行顺序是：

```

    分支指令
    顺序后继指令 1
    .....
    顺序后继指令 n
    分支目标处指令(如果分支成功)

```

所有顺序后继指令都处于“分支延迟槽”(branch-delay slots)中，无论分支成功与否，流水线都会执行这些指令。具有一个分支延迟槽的 DLX 流水线的时空图如图 3.33 所示。

失败的分支指令	IF	ID	EX	MEM	WB
延迟分支指令 (<i>i</i> +1)	IF	ID	EX	MEM	WB
指令 <i>i</i> +2		IF	ID	EX	MEM
指令 <i>i</i> +3			IF	ID	EX
指令 <i>i</i> +4				IF	ID
成功的分支指令	IF	ID	EX	MEM	WB
延迟分支指令 (<i>i</i> +1)	IF	ID	EX	MEM	WB
分支目标指令 <i>j</i>		IF	ID	EX	MEM
分支目标指令 <i>j</i> +1			IF	ID	EX
分支目标指令 <i>j</i> +2				IF	ID

图 3.33 基于“延迟分支”方法的流水线时空图

从该图可以看出，基于“延迟分支”方法，无论分支成功与否，其流水线时空图所描述的流水线的行为是类似的，流水线中均没有插入暂停周期，从而极大地降低了流水线分支损失。从该图也可以看出，实际上是处于分支延迟槽中的指令“掩盖”了流水线原来所必须插入的暂停周期。那么将什么指令放入分支延迟槽中呢？

选择放入分支延迟槽中的指令必须遵循有效和有用两个原则,这也是一种指令调度技术。三种调度分支延迟指令的常用方法如图 3.34 所示,表 3.7 说明了这三种指令调度方法的特点及其局限性。

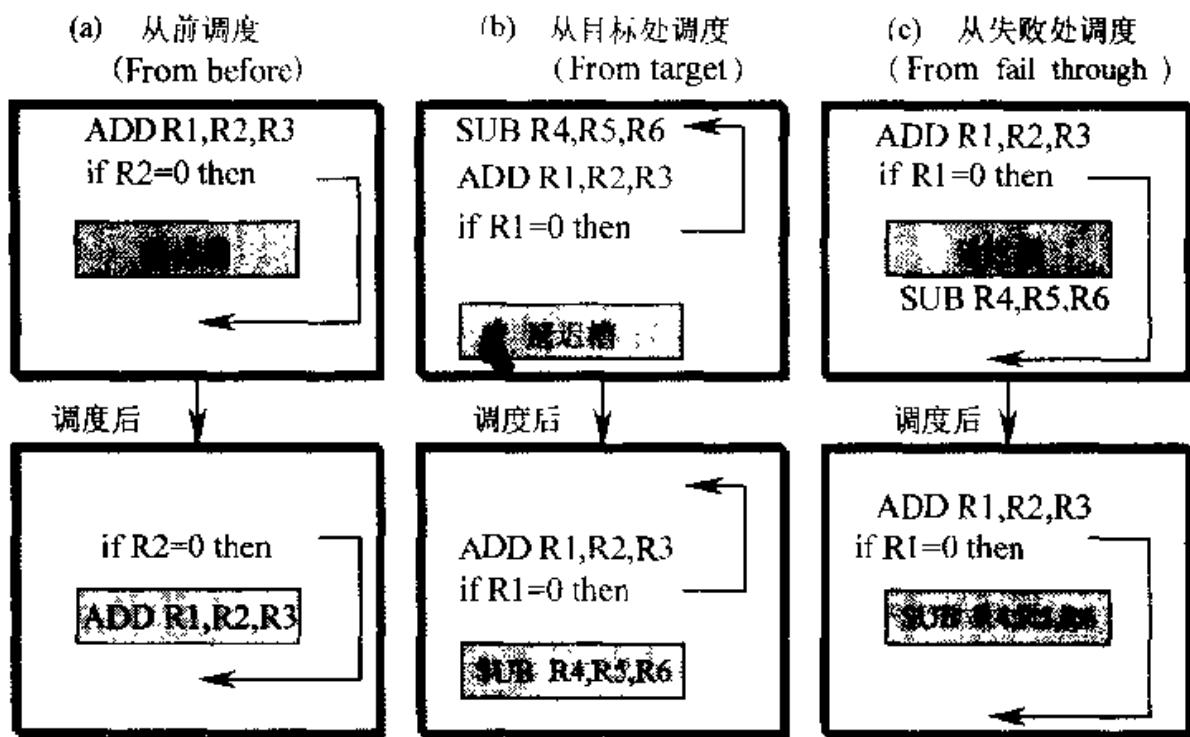


图 3.34 调度分支延迟指令的三种常用方法

表 3.7 调度分支延迟指令的三种常用方法的特点及其局限性

调度策略	对调度的要求	对流水线性能改善的影响
从前调度	分支必须不依赖于被调度的指令	总是可以有效提高流水线性能
从目标处调度	如果分支转移失败,必须保证被调度的指令对程序的执行没有影响,可能需要复制被调度指令	分支转移成功时,可以提高流水线性能。但由于复制指令,可能加大程序空间
从失败处调度	如果分支转移成功,必须保证被调度的指令对程序的执行没有影响	分支转移失败时,可以提高流水线性能

从图 3.34 和表 3.7 可以看出,调度延迟分支指令的方法可以降低流水线分支损失,但是它受限于被调度进入分支延迟槽中的指令,以及编译器预测分支转移是否成功的能力。

为了提高编译器填充分支延迟槽的能力,许多流水线机器引入了“取消”(canceling)或“作废”(nullifying)分支。对取消分支而言,分支指令包含了预测的分支方向,当实际分支方向和事先所预测的一样,执行分支延迟槽中的指令,

流水线正常工作；当实际分支方向和事先所预测的不同，就将分支延迟槽中的指令转化成一个空操作。图 3.35 给出了取消分支在分支转移成功和失败两种情况下的行为。

失败的分支指令	JF	ID	EX	MEM	WB				
分支延迟指令 $i+1$		IF	ID	idle	idle	idle			
指令 $i+2$			IF	ID	EX	MEM	WB		
指令 $i+3$				IF	ID	EX	MEM	WB	
指令 $i+4$					IF	ID	EX	MEM	WB

成功的分支指令	IF	ID	EX	MEM	WB				
分支延迟指令 $i+1$		IF	ID	EX	MEM	WB			
分支目标			IF	ID	EX	MEM	WB		
分支目标+1				IF	ID	EX	MEM	WB	
分支目标-2					IF	ID	EX	MEM	WB

图 3.35 取消分支的行为依赖于实际的分支方向

3. 各种处理分支方法的性能

现在来看看上述各种减少流水线分支损失方法对流水线性能的影响。假设流水线理想的 CPI 为 1，那么具有分支损失的流水线加速比可以用下式来计算：

$$S = \frac{D}{1 + C} \quad (3.15)$$

其中， D 为流水线的深度， C 为由于处理分支指令而给程序中每条指令带来的平均暂停时钟周期数。又因为

$$C = f \times P_{\text{分支}} \quad (3.16)$$

其中， f 为程序中分支指令出现的频率， $P_{\text{分支}}$ 为处理分支指令流水线所需要暂停的平均时钟周期数，即平均分支损失。根据式(3.15)和式(3.16)可得

$$S = \frac{D}{1 + f \times P_{\text{分支}}} \quad (3.17)$$

式(3.17)中分支指令出现频率和平均分支损失既来源于无条件分支，也来源于条件分支，但是后者占主要部分。根据 DLX 的一些测量统计结果，可以得到如表 3.8 所示的各种方法降低分支损失的效果（假设 DLX 的理想 CPI 是 1）。

表 3.8 各种减少分支损失方法的效果

调度方法	每条条件分支指令的 分支损失		每条无条件分支指 令的损失	每条分支指令的平均 分支损失		具有分支暂停的实际 CPI	
	整型平均	浮点平均		整型平均	浮点平均	整型平均	浮点平均
暂停流水线	1.00	1.00	1.00	1.00	1.00	1.17	1.15
预测分支成功	1.00	1.00	1.00	1.00	1.00	1.17	1.15
预测分支失败	0.62	0.70	1.00	0.69	0.74	1.12	1.11
延迟分支	0.25	0.35	0.00	0.21	0.30	1.04	1.04

3.4 流水线计算机实例分析(MIPS R4000)

3.4.1 MIPS R4000 整型流水线

R4000 处理器是一种流水线处理器, 它所实现的 MIPS-3 指令集是一种和 DLX 类似的 64 位指令集。但是和 DLX 流水线不同,R4000 的流水线特别考虑了流水访问存储器的操作。

R4000 的 8 段流水线结构如图 3.36 所示。其 8 个流水段的功能分别为: IF 段是取指令的前半部分操作, 主要完成选择 PC 值和访问指令 Cache 的启动工作; IS 段是取指令的后半部分操作, 主要完成访问指令 Cache 的操作; RF 段完成译码、取寄存器和相关检测等操作, 并检测指令 Cache 命中情况; EX 段为执行段, 包括计算有效地址、ALU 操作、计算分支目标地址和检测分支条件; DF 段完成取数据操作, 是访问数据 Cache 的前半部分; DS 段为访问数据 Cache 的后半部分; TC 段完成 Cache 标记检测, 以确定访问数据 Cache 是否命中; WB 段写回读出的数据或寄存器 - 寄存器操作的结果。

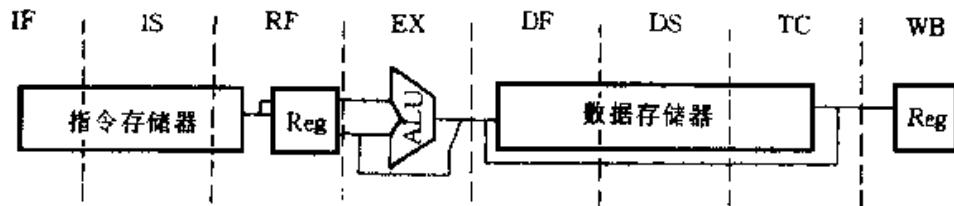


图 3.36 R4000 流水线结构

由于 R4000 的流水线段数较多, 所以其时钟速率可达 100 ~ 200 MHz。有时也称这种类型的流水是“超级流水”(superpipelining)。

指令序列在该流水线中重叠执行情况如图 3.37 所示。值得注意的是, 尽管

访问指令存储器和数据存储器在流水线中占据多个流水周期,但是这些访问存储器的操作是全流水的,所以 R4000 流水线可以在每个时钟周期启动一条新的指令。

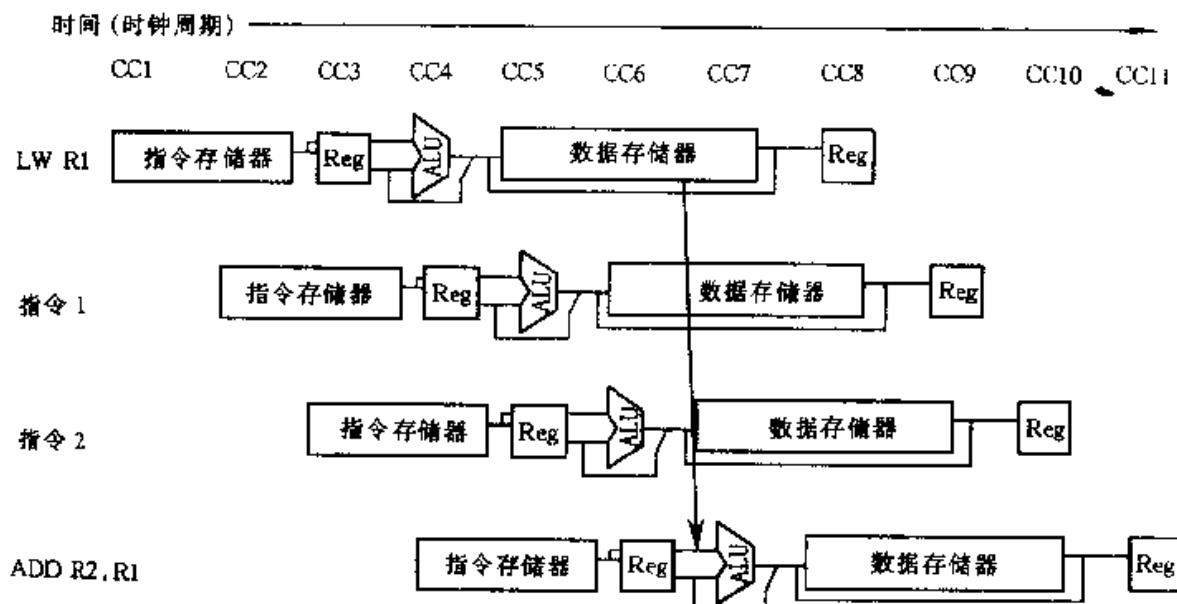


图 3.37 指令在 R4000 流水线中重叠执行

从图 3.37 可以看出,由于从存储器中读入的数据在 DS 段的末尾才会有效,所以其载入延迟是 2 个时钟周期,由此可见 R4000 的流水线具有较长的载入和分支延迟。另外,考虑如图 3.38 所示的实例,可以看出,由于紧跟 Load 指令之后的指令要使用 Load 指令从存储器中读出的数据,为了保证 ADD 指令能够使用正确的载入数据,必须在流水线中插入两个暂停周期,并采用定向技术将读出的数据直接定向到 ADD 指令的 ALU 输入端。对于 SUB 指令来说,也需定向 Load 指令读出的数据,而 OR 指令则可直接从寄存器 R1 中读取所需要的值。

指令序列	时钟周期								
	1	2	3	4	5	6	7	8	9
LW R1	IF	IS	RF	EX	DF	DS	TC	WB	
ADD R2, R1		IF	IS	RF	stall	stall	EX	DF	DS
SUB R3, R1			IF	IS	stall	stall	RF	EX	DF
OR R4, R1				IF	stall	stall	IS	RF	EX

图 3.38 指令序列在 R4000 流水线中的执行时空图

因此,对 R4000 的流水线来说,定向是十分重要的。实际上,其流水线的定向路径比 DLX 流水线要多。在 R4000 的流水线中,到 ALU 输入有四个定向源: EX/DF/DS, DS/TC 和 TC/WB, 所以其对定向的控制也要比 DLX 流水线复杂得多。

从图 3.39 可以看出,由于在 R4000 的流水线中,在 EX 段完成分支条件的计算,所以其基本的分支延迟是三个时钟周期。为了降低分支延迟损失,MIPS 结构采用了单周期延迟分支技术,并且延迟分支调度是基于预测失败策略(从失败处调度策略)。图 3.40 为基于这种技术的 R4000 流水线对分支指令处理的时空图。

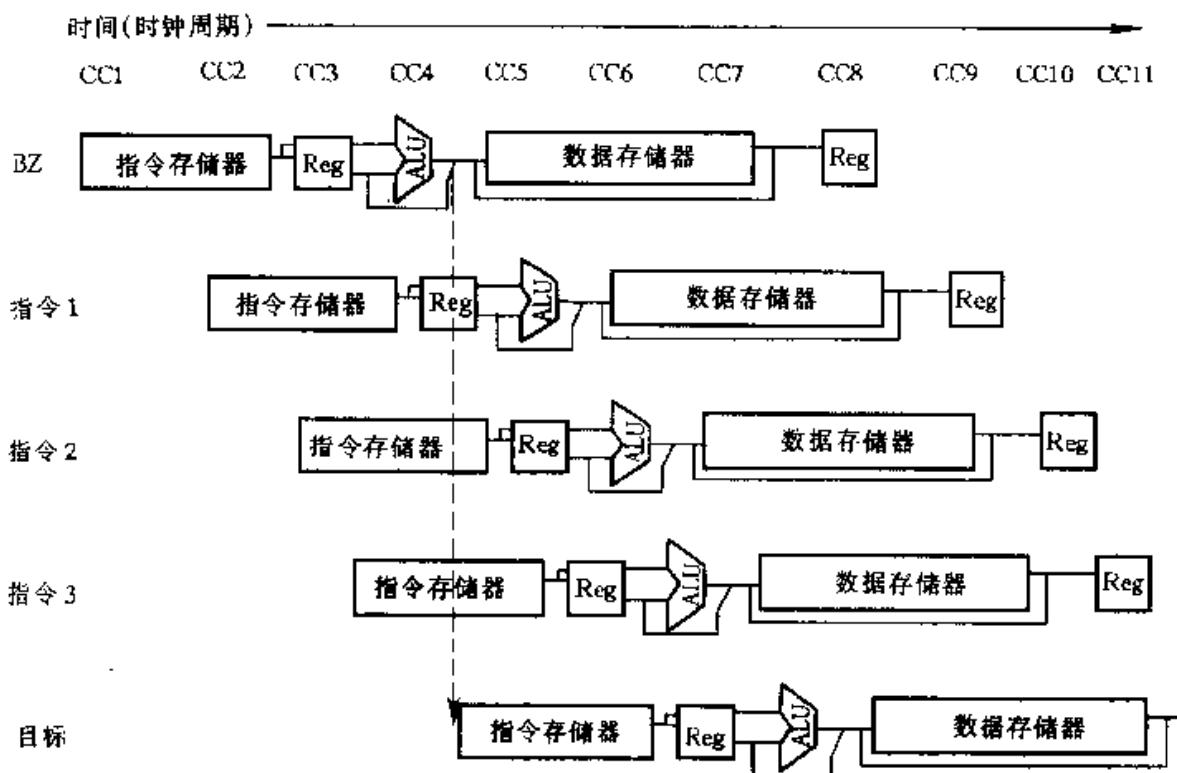


图 3.39 R4000 流水线的基本分支延迟为 3 个时钟周期

指令序列	时钟周期									
	1	2	3	4	5	6	7	8	9	
分支指令	IF	IS	RF	EX	DF	DS	TC	WB		
延迟槽		IF	IS	RF	EX	DF	DS	TC	WB	
暂停			stall							
暂停				stall	stall	stall	stall	stall	stall	
分支目标					IF	IS	RF	EX	DF	

指令序列	时钟周期									
	1	2	3	4	5	6	7	8	9	
分支指令	IF	IS	RF	EX	DF	DS	TC	WB		
延迟槽(分支指令+1)		IF	IS	RF	EX	DF	DS	TC	WB	
分支指令+2			IF	IS	RF	EX	DF	DS	TC	
分支指令+3				IF	IS	RF	EX	DF	DS	

图 3.40 基于单周期延迟分支方法,R4000 流水线处理分支指令的时空图

3.4.2 MIPS R4000 浮点流水线

R4000 的浮点部件由一个浮点除法器、一个浮点乘法器和一个浮点加法器组成,其浮点功能部件流水线可以认为是由表 3.9 中所列出的 8 个不同的段组成:

表 3.9 R4000 浮点流水线中 8 个流水段

流水段	功能部件	描述
A	浮点加法器	尾数加流水段
D	浮点除法器	除法流水段
E	浮点乘法器	例外测试段
M	浮点乘法器	乘法器第一个流水段
N	浮点乘法器	乘法器第二个流水段
R	浮点加法器	舍入段
S	浮点加法器	操作数移位段
U		展开浮点数

R4000 的浮点流水线是一种多功能非线性流水线,不同的指令对该流水线不同段的使用情况不尽相同,顺序也互不相同。有的指令可能要多次使用同一流水段,有的指令可能根本不会用到某些流水段。表 3.10 列出了最常用的双精度浮点操作指令在该流水线中的延迟时钟周期数、启动时钟周期间隔及其所使用的流水段。

表 3.10 双精度浮点操作指令延迟、初始化间隔和流水段的使用情况

浮点指令	延 迟	初始化间隔	使用的流水段
加、减	4	3	U,S-A,A+R,R+S
乘	8	4	U,E+M,M,M,M,N,N+A,R
除	36	35	U,A,R,D ²⁸ ,D+A,D+R,D+A,D+R,A,R
求平方根	112	111	U,E,(A+R) ¹⁰⁸ ,A,R
取反	2	1	U,S
求绝对值	2	1	U,S
浮点比较	3	2	U,A,R

表 3.10 中的“+”表示在一个时钟周期内要用到相应的两个流水段,而流水段标记的上标表示该流水段要重复使用的次数。根据表 3.10,我们也可以确定发射不同的浮点操作指令是否会引起流水线暂停,如果浮点操作指令在流水线

中出现了流水段冲突,那么就必须在流水线中插入暂停周期,以消除这种相关。

3.4.3 MIPS R4000 流水线的性能分析

对 R4000 流水线而言,主要有如下四个方面的因素可能会引起在流水线中插入暂停周期:

1. 载入暂停: 将使用载入结果的指令从 Load 指令开始向后推迟 1 到 2 个时钟周期;
2. 分支暂停: 每个成功分支,以及排空或取消分支延迟槽所需的 2 个暂停时钟周期;
3. 浮点结果暂停: 由于浮点操作数的 RAW 相关造成的暂停;
4. 浮点结构性暂停: 由于浮点流水线的功能单元冲突限制指令发射而带来的暂停。

在 R4000 流水线上运行 SPEC92 的 10 个基准程序,考察上述四个因素引起的暂停对流水线 CPI 的影响,可以得到如表 3.11 所示的结果。

表 3.11 暂停对 R4000 流水线 CPI 的影响

基准程序	流水线 CPI	载入暂停时钟周期数	分支暂停时钟周期数	浮点结果暂停时钟周期数	浮点结构性暂停时钟周期数
compress	1.20	0.14	0.06	0.00	0.00
eqntort	1.88	0.27	0.61	0.00	0.00
espresso	1.42	0.07	0.35	0.00	0.00
gcc	1.56	0.13	0.43	0.00	0.00
li	1.64	0.18	0.46	0.00	0.00
整数平均	1.54	0.16	0.38	0.00	
doduc	2.84	0.01	0.22	1.39	0.22
mdljdp2	2.66	0.01	0.31	1.20	0.15
ear	2.17	0.00	0.46	0.59	0.12
hydro2d	2.53	0.00	0.62	0.75	0.17
su2cor	2.18	0.02	0.07	0.84	0.26
浮点平均	2.48	0.01	0.33	0.95	0.18
总平均	2.00	0.10	0.36	0.46	0.09

根据上述测试统计结果,可以看出 R4000 流水线的分支延迟比 DLX 流水线的要长,特别是对于整型程序而言,由于分支指令执行频率较高,所以 R4000 流水线处理分支指令所需的时钟周期数也较多。

另外,还可以看到由于 R4000 浮点流水线受浮点指令发射时间间隔以及浮点功能部件冲突的限制,使得浮点指令延迟对流水线 CPI 有很大影响,所以对 R4000 流水线来说,降低浮点操作延迟是提高其性能的基础。

前面主要是结合 DLX 流水线阐述了流水线的基本原理、基本结构、相关和流水线设计中的若干问题和一些关键技术。这些思想和技术对向量处理机也是适用的,但向量处理机还有自己的特点。为了能更深入认识这些特点,下面对向量处理机作一简要介绍。

3.5 向量处理机

从前面的分析中我们知道,如果输入到流水线中的指令既无数据相关,也无控制相关和结构相关,则流水线就有可能装满,从而使流水线获得很高的效率和吞吐率。而在科学计算中,往往有大量不相关的数据进行同一种运算,这正好适合于流水线的特点。因此,就出现了设有向量数据表示和相应向量指令的向量流水线处理机。由于这种机器能较好地发挥流水线技术的特性,因此可以达到较高的速度(一般可达 1 亿次浮点运算/s 的数量级)。一般也称向量流水处理机为向量处理机(vector processor)。

本节首先介绍向量处理方式及其对处理机结构的要求,然后以 CRAY-1 为例,介绍向量处理机的结构及其特点,以及向量处理中的一些关键技术。

3.5.1 向量处理方式和向量处理机

这里,举一个简单的例子来说明向量处理方式。例如,以下是用 FORTRAN 语言写的一个循环程序:

```
DO 10 i=1,N
10 d[i]=a[i] * (b[i] + c[i])
```

对此可以有如下几种处理方式。

1. 水平(横向)处理方式

水平处理方式也就是逐个求 $d[i]$ 的方式,为此,先计算: $d[1] = a[1] * (b[1] + c[1])$; 再计算: $d[2] = a[2] * (b[2] + c[2])$; ……; 最后计算: $d[N] = a[N] * (b[N] + c[N])$ 。一般的计算机就是采用这种方式组成循环程序进行处理的。在每次循环中,至少要用到如下几条机器指令:

$$k_i = b_i + c_i$$

$$d_i = k_i * a_i$$

.....

BE(等于“0”分支成功)

上面程序计算共需 N 次循环, 其中 $N-1$ 次分支成功, 在每次循环中有一次数据相关。如果用静态流水线, 则要进行 2 次乘和加的功能转换, 所以共出现 N 次数据相关和 $2N$ 次功能切换。因此, 这种水平处理方式不适合于对向量进行流水处理。

实际上, 我们可以认为 A 、 B 、 C 、 D 是长度为 N 的向量 $A=(a_1 a_2 \cdots a_N)$, $B=(b_1 b_2 \cdots b_N)$, $C=(c_1 c_2 \cdots c_N)$, $D=(d_1 d_2 \cdots d_N)$ 。因此, 上述 DO 循环可以写成如下向量运算的形式:

$$D = A * (B + C)$$

基于该向量表示形式, 还可以有下面两种处理方式。

2. 垂直(纵向)处理方式

垂直处理方式是将整个向量按相同的运算处理完之后, 再去执行别的运算。对于上式, 则有

$$K = B + C$$

$$D = K * A$$

可以看出, 这种处理方式仅用了两条向量指令, 且处理过程中没有出现分支指令, 每条向量指令内无相关, 两条向量指令间仅有 1 次数据相关。如果仍用静态多功能流水线, 也只需 1 次功能切换, 所以这种处理方式适合于对向量进行流水处理。

下面分析这种处理方式对处理机结构的要求。由于向量长度 N 是不受限制的, 无论 N 有多大, 相同的运算都用一条向量指令完成。因此, 向量运算指令的源向量和目的向量都存放在存储器内, 这使这种处理机流水线运算部件的输入、输出端都直接(或经向量数据缓冲器)与存储器相联, 从而构成所谓存储器-存储器型操作的运算流水线, 其结构如图 3.41 所示。CDC 公司的 STAR-100、CYBER-205 等中央处理机都是采用这种结构。

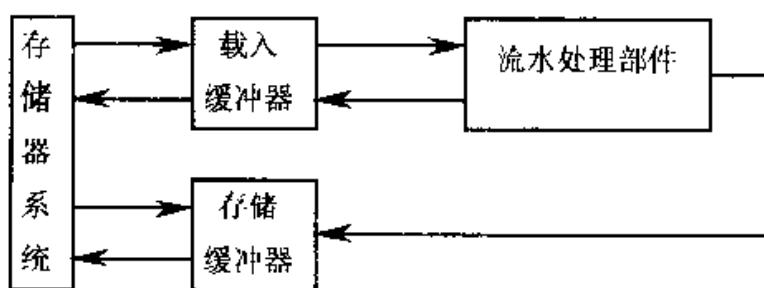


图 3.41 存储器-存储器型操作的运算流水线

这种结构要求极大地提高存储系统和流水处理机之间的通信带宽,使得在流水线的输入端能每拍从存储器取得一对元素,并能向存储器写回一个结果,这样才能保证流水线的平稳流动。

3. 分组(纵横)处理方式

分组处理方式是把长度为 N 的向量,分成若干组,每组长度为 n ,组内按纵向方式处理,依次处理各组。若

$$N = s \cdot n + r$$

其中 r 为余数,也作为一组处理,则共有 $s+1$ 组,其运算过程为:先算第 1 组;再算第 2 组……最后算第 $s+1$ 组。

为了减少循环的影响,每组内各用两条向量指令,各组内仅有一次向量指令的数据相关。如果也用静态多功能流水线,则各组需 2 次功能切换,比水平处理方式要少很多。所以,这种处理方式也适合于对向量进行流水处理。

这种处理方式对向量长度 N 的大小亦不限制,但是,每一组的长度最大不能超过 n 。因此,可设置长度为 n 的向量寄存器,使得每组向量运算的源向量和目的向量都在向量寄存器中,运算流水线的输入、输出端都与向量寄存器相联,从而构成所谓寄存器-寄存器型操作的运算流水线,如图 3.42 所示。不少机器如 CRAY-1、YH-1 的中央处理器都是采用这种结构。

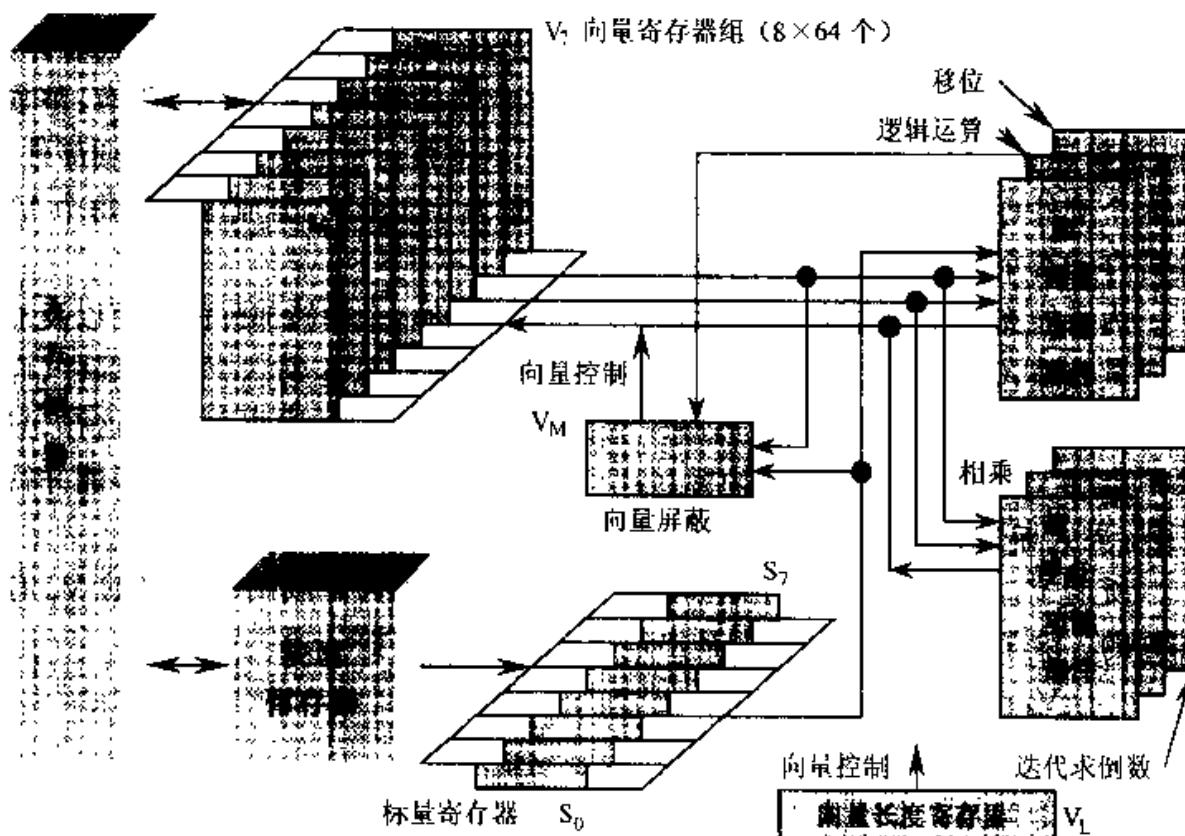


图 3.42 寄存器-寄存器型操作的运算流水线(CRAY-1 简图)

这种结构要求有容量足够大的向量寄存器组。它们不但能存放源向量,而且能保留中间结果,从而大大减少访问存储器的次数;此外,可降低对存储器带宽的要求,亦可减少因存储器访问冲突而引起的等待时间,从而提高处理速度。

最后,讨论一下向量处理机的速度评价方法。在标量处理机中,执行一条运算指令,一般可以得到一个运算结果。因此,通常用每秒执行多少指令 MIPS (Million Instructions Per Second) 来衡量机器的运算速度。而向量处理机则完全不同,执行一条向量指令往往可以取得几十个或更多的运算结果。显然,再用上述指标来衡量机器速度就不合适了。在科学计算中,常常用每秒取得多少个浮点运算结果表示机器速度,以 MFLOPS(Million of Floating Point Per Second)作为测量单位。必须注意,这一指标不能直接和标量处理机中所用的 MIPS 相比。因为计算机执行的指令,除运算指令外,还有更多的服务性指令(如 Load、Store、测试、分支等)。在每秒执行多少条指令的速度指标中,是把这些服务性指令都考虑在内的;而在每秒取得多少个浮点运算结果的速度指标中,则不考虑这些指令。

3.5.2 向量处理机实例分析

20世纪70年代中期问世的CRAY-1向量机是向量处理机的典型代表,其向量流水处理部件简图如图3.42所示。可为向量运算使用的功能部件有:整数加、逻辑运算、移位、浮点加、浮点乘、浮点迭代求倒数。它们都是流水处理部件,且六个部件可并行工作。向量寄存器组的容量为512个字,分成8块。每个 V_i 块可存元素个数达64的一个向量。

为了能充分发挥向量寄存器组和可并行工作的六个功能部件的作用以及加快向量处理,CRAY-1设计成每个 V_i 块都有单独总线可连到六个功能部件,而每个功能部件也各自都有把运算结果送回向量寄存器组的输出总线。这样,只要不出现 V_i 冲突和功能部件冲突,各个 V_i 之间和各个功能部件之间都能并行工作,大大加快了向量指令的处理,这是CRAY-1向量处理的显著特点。

所谓 V_i 冲突指的是并行工作的各向量指令的源向量或结果向量的 V_i 有相同的。除了相关情况之外,就是出现源向量冲突,例如:

$$V_4 = V_1 + V_2$$

$$V_5 = V_1 \wedge V_3$$

这两条向量指令不能同时执行,需在第一条向量指令执行完,释放 V_1 后,第二条指令才能执行。这是因为这两条指令的源向量之一虽然都是取自 V_1 ,但二者的首元素下标可能不同,向量长度也可能不同,难以由 V_1 同时提供两条指

令所需的源向量。这种冲突和前面所讨论的结构相关是一样的。所谓功能部件冲突指的是同一个功能部件被一条以上的并行工作向量指令所使用。例如：

$$V_4 = V_2 * V_3$$

$$V_5 = V_1 * V_6$$

这两条向量指令都需用到浮点相乘部件，因此在第一条指令执行完毕，功能部件释放后，第二条指令才能执行。

CRAY-1 有如图 3.43 所示的四种向量指令。第 1 种指令每拍从 V_i 、 V_j 块顺序取得一对元素送入功能部件。其输出也是每拍送进 V_k 块一个结果元素。元素对的个数由 V_L (向量长度)寄存器指明。向量屏蔽寄存器(V_M)为 64 位，每位对应 V 的一个元素。在向量合并或测试时，由 V_M 控制对哪些元素进行合并和测试。一条指令至多只能处理 64 对元素(对应每块的容量)。若向量的长度大于 64，需用向量循环程序将其分段，各段中向量元素个数要小于或等于 64，然后以段为单位从存储器中调入并进行处理。第 2 种指令和第 1 种指令的差别只在于它的一个操作数取自目标量寄存器 S_i 。大多数向量指令都属这两种。由于它们不是由存储器而是由向量寄存器取得操作数，所以流水速度可很高，CRAY-1 的时钟周期时间为 12.5 ns(一拍)。第 3、4 种指令是控制存储器与向量寄存器之间的数据传送。

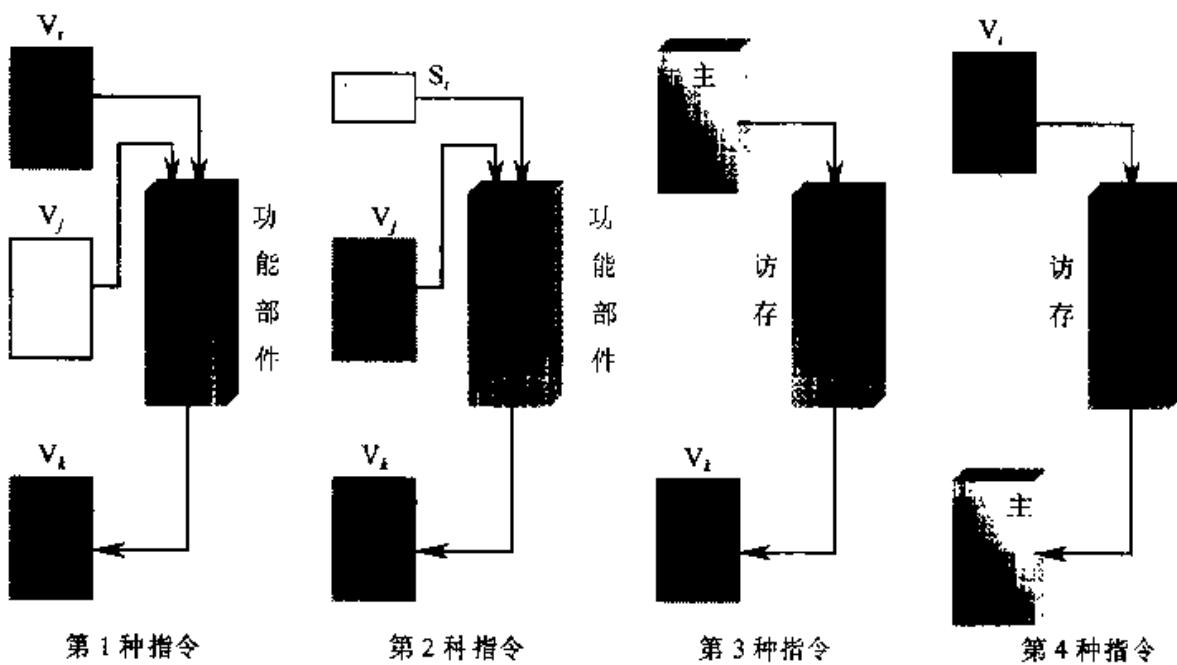


图 3.43 CRAY-1 的四种向量指令

CRAY-1 向量处理的另一个显著特点是，只要不出现功能部件冲突和源向量冲突，通过链接结构可使“写后读”相关的向量指令也能并行处理。例如，对上述向量运算 $D = A * (B + C)$ ，若 $N \leq 64$ ，向量为浮点数，则在 B, C 取到 V_0, V_1 后，就可用以下三条向量指令求解：

1. $V_3 \leftarrow$ 存储器(访存, 取 A)
2. $V_2 \leftarrow V_0 + V_1$ (浮点加)
3. $V_4 \leftarrow V_2 * V_3$ (浮点乘, 存 D)

第 1、2 条指令无任何冲突, 可以并行执行。第 3 条指令与第 1、2 条指令之间存在数据相关, 不能并行执行, 但是如果能够将第 1、2 条指令的结果元素直接链接到第 3 条指令所用的功能部件, 那么第 3 条指令就能与第 1、2 条指令并行执行。其链接过程如图 3.44 所示。

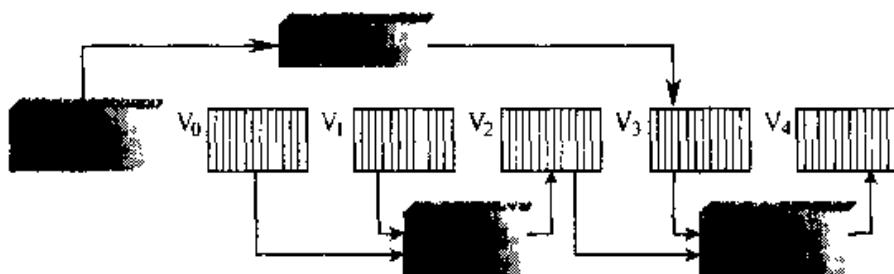


图 3.44 通过链接技术实现指令重叠执行

由此可见, 所谓链接特性, 实质上是把流水线“定向”的思想引入到向量执行过程的结果。

CRAY-1 在把元素从 V_i 送往功能部件及把结果存入 V_j 时都需一拍。由于第 1、2 条指令之间没有任何冲突, 可以同时执行, 而“访存”拍数正好与“浮加”的一样, 因此, 从访存开始, 直至把第一个结果元素存入 V_4 , 所需拍数(亦称为链接流水线的流水时间)为

$$1(\text{送}) + 6(\text{访存}) + 1(\text{入}) + 1(\text{送}) + 7(\text{浮乘}) + 1(\text{入}) = 17 \text{ 拍}$$

此后, 就是每拍取得一个结果元素存入 V_4 。显然, 这要比第 1、2 条指令全执行完, 所有元素全进入 V_2 、 V_3 后, 才开始执行第 3 条指令要快得多。

通过这种链接技术使得 CRAY-1 流水线能灵活组织, 从而更能发挥流水技术的效能。

CRAY-1 的向量指令还可做到“源 V”和“结果 V”是同一个, 这种向量递归操作和前述的链接特性对于实现诸如求向量点积等是很有好处的。

上面结合 CRAY-1 介绍了向量流水机器的结构特点。然而, 要使软件能充分发挥硬件所提供的这些特点却是很不容易的, 它必然要对语言结构和编译程序提出新的要求。例如, 它希望高级语言能增设向量运算符(如向量加、向量乘等), 不然, 程序设计者在编制高级语言程序时, 要把向量运算通过 DO LOOP 实现, 而编译程序反过来却又要把 DO LOOP 型语句变换成向量型的机器语言去执行。例如:

$$C = A + B$$

的向量运算,程序设计者是用

```
DO 20 I=1,N  
20      C(I)=A(I)+B(I)
```

实现,但编译程序却又要把它编译成

```
VECT-BEGIN  
A,B,C=VECTOR(1…N)  
C=A+B  
VECT-END
```

去执行。

另外,优化的目标程序必然要和向量流水机器的具体结构特点密切相关,这会使编译程序的设计复杂化。例如,对图 3.5 所示的 ASC 结构,那就要求把目标程序中的相加指令和相乘指令分别集中在一起,以减少流水线功能切换所费时间。然而,CRAY-1 却无此要求,因为它有独立的能并行执行的相乘、相加功能部件。但是,它却要求编译程序能充分发挥多功能部件和上述链接能力所能提供的多条向量指令可并行执行的特点。

3.6 小结

流水线是提高处理器性能的一种最重要技术。在 20 世纪 50 年代后期到 60 年代中期,许多早期的计算机设计者也致力于通过流水线来提高性能。但是,在 60 年代后期到 70 年代后期,计算机体系结构设计者却在关注其他一些事情,如计算机的费用、大小和可靠性等。在这个时期,流水线扮演的是一个配角。正是由于当时流水线没有引起人们的充分重视,所以那时设计的多数指令集都很难对其进行流水处理,也许 VAX 指令集结构就是一个最好的例证。

在 70 年代后期和 80 年代初,一些研究者意识到指令的复杂性和实现的容易程度,特别是与流水处理的容易程度之间紧密相关,从而提出了 RISC 计算机的概念。RISC 技术的出现极大地简化了指令集,使流水线技术的开发取得了长足的进展,当前这些实现技术已经变得非常成熟。

本章介绍了流水线技术中的一些基本思想,以及提高处理性能的简单编译策略,它们是以 R4000 为代表的 80 年代微处理器的基础。为了进一步提高性能,当前多数微处理器都引入了许多新的设计方法,如基于硬件的流水线调度、动态分支预测,多指令发射以及更加强大的编译技术。下一章将对这些更加先进的技术进行详细的论述。

习 题 三

3.1 请解释下列术语

流水线 单功能流水线 多功能流水线 静态流水线 动态流水线
 部件级流水线 处理机级流水线 处理机间流水线 线性流水线
 非线性流水线 标量流水处理机 向量流水处理机 结构相关
 数据相关 控制相关 定向 写后读相关 读后写相关 写后写相关

3.2 简述流水线技术的特点。

3.3 请画出 DLX 基本流水线，并简述其工作原理。

3.4 你如何看待指令单周期实现和多周期实现。

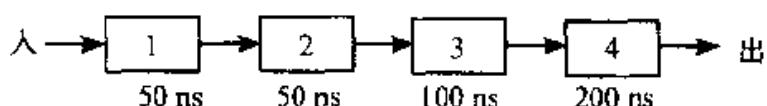
3.5 解决流水线结构相关的方法有哪些？

3.6 降低流水线分支损失的方法有哪些？

3.7 请对延迟分支方法中的三种调度策略进行评价。

3.8 简述三种向量处理方式，它们对向量处理机的结构要求有何不同？

3.9 有一指令流水线如下所示。



(1) 求连续输入 10 条指令，该流水线的实际吞吐率和效率；

(2) 该流水线的“瓶颈”在哪一段？请采取三种不同的措施消除此“瓶颈”。对于你所给出的三种新的流水线，仍计算连续输入 10 条指令时，其实际吞吐率和效率。

3.10 有一个流水线由 4 段组成，其中每当流经第 3 段时，总要在该段循环一次，然后才能流到第 4 段。如果每段经过一次所需要的时间都是 Δt ，问：

- (1) 当在流水线的输入端连续地每 Δt 时间输入任务时，该流水线会发生什么情况？
- (2) 此流水线的最大吞吐率为多少？如果每 $2\Delta t$ 输入一个任务，连续处理 10 个任务时的实际吞吐率和效率是多少？
- (3) 当每段时间不变时，如何提高该流水线的吞吐率？仍连续处理 10 个任务时，其吞吐率提高多少？

3.11 流水线有 m 段，各段的处理时间分别是 t_i ($i = 1, 2, \dots, m$)，现有 n 个任务需要完成，且每个任务均需流水线各段实现，请计算：

- (1) 流水线完成这 n 个任务所需要的时间；
- (2) 和非流水实现相比，这 n 个任务流水实现的加速比是多少？加速比的峰值是多少？

3.12 在改进的 DLX 流水线上运行如下代码序列：

```

LOOP:   LW    R1,  0  (R2)
        ADDI  R1,  R1,  #1
        SW    R1,  0  (R2)
        ADDI  R2,  R2,  #4
  
```

```
SUB R4, R3, R2
BNZ R4, LOOP
```

其中, R3 的初值是 R2 + 396。假设:在整个代码序列的运行过程中,所有的存储器访问都是命中的,并且在一个时钟周期中对同一个寄存器的读操作和写操作可以通过寄存器文件“定向”,问:

- (1) 在没有任何其他定向(或旁路)硬件的支持下,请画出该指令序列执行的流水线时空图。假设采用排空流水线的策略处理分支指令,且所有的存储器访问都可以命中 Cache,那么执行上述循环需要多少个时钟周期?
- (2) 假设该 DLX 流水线有正常的定向路径,请画出该指令序列执行的流水线时空图。假设采用预测分支失败的策略处理分支指令,且所有的存储器访问都可以命中 Cache,那么执行上述循环需要多少个时钟周期?
- (3) 假设该 DLX 流水线有正常的定向路径和一个单周期延迟分支,请对该循环中的指令进行调度。注意可以重新组织指令的顺序,也可以修改指令的操作数,但是不能增加指令的条数。请画出该指令序列执行的流水线时空图,并计算执行上述循环所需要的时钟周期数?

3.13 假设各种分支占所有指令数的百分比如下表所示:

条件分支	20% (其中的 60% 是成功的)
跳转和调用	5%

现有一深度为 4 的流水线(流水线有 4 段),无条件分支在第二个时钟周期结束时就被解析出来,而条件分支要到第三个时钟周期结束时才能够被解析出来。第一个流水段是完全独立于指令类型的,即所有类型的指令都必须经过第一个流水段的处理,请问在没有任何结构相关的情况下,该流水线相对于存在上述结构相关情况下的加速比是多少?

3.14 在 CRAY-1 机器上,按照链接方式执行下述 4 条向量指令(括号中给出了相应功能部件时间),如果向量寄存器和功能部件之间的数据传送需要 1 拍,试求此链接流水线的通过时间是多少拍?如果向量长度为 64,则需多少拍才能得到全部结果。

$V_0 \leftarrow$ 存储器	(从存储器中取数: 7 拍)
$V_2 \leftarrow V_0 + V_1$	(向量加: 3 拍)
$V_3 \leftarrow V_2 \ll A_3$	(按(A_3)左移: 4 拍)
$V_5 \leftarrow V_3 \wedge V_4$	(向量逻辑乘: 2 拍)

3.15 某向量处理机有 16 个向量寄存器,其中 $V_0 \sim V_5$ 中分别放有向量 A、B、C、D、E、F,向量长度均为 8,向量各元素均为浮点数;处理部件采用两个单功能流水线,加法功能部件时间为 2 拍,乘法功能部件时间为 3 拍。采用类似于 CRAY-1 的链接技术,先计算 $(A \cdot B) * C$,在流水线不停流的情况下,接着计算 $(D + E) * F$ 。

- (1) 求此链接流水线的通过时间为多少拍?(设寄存器入、出各需 1 拍)

- (2) 假如每拍时间为 50 ns, 完成这些计算并把结果存进相应寄存器, 此处理部件的实际
吞吐率为多少 MFLOPS?

第四章 指令级并行

4.1 指令级并行的概念

从第三章可以知道,当指令不相关时,它们在流水线中是重叠执行的。这种指令序列中存在的潜在并行性称为指令级并行(Instruction-Level Parallelism,简记为 ILP)。本章研究如何通过各种可能的技术,获得更多的指令级并行性。

在流水线技术中,建立了下面的公式:

$$\begin{aligned} \text{CPI}_{\text{流水线}} = & \text{ CPI}_{\text{理想}} + \text{停顿}_{\text{结构相关}} + \text{停顿}_{\text{先写后读}} \\ & + \text{停顿}_{\text{先读后写}} + \text{停顿}_{\text{写后写}} + \text{停顿}_{\text{控制相关}} \end{aligned}$$

流水线的理想 CPI 是流水线的最大流量;结构相关停顿是由于两条指令使用同一个功能部件而导致的停顿;控制相关停顿是由于指令流的改变(如分支指令)而导致的停顿;先写后读(RAW)停顿、先读后写(WAR)停顿和写后写(WAW)停顿是由数据相关造成的。表 4.1 中列出了要研究的技术以及它们所克服的停顿。

表 4.1 本章要研究的技术以及它们所克服的停顿

技术	主要克服的停顿
循环展开	控制相关停顿
基本流水线调度	数据先写后读停顿
指令动态调度	各种数据相关停顿
分支预测	控制相关停顿
推断(Speculation)	所有数据/控制相关停顿
多指令流出	提高理想 CPI

指令级并行研究的是一个基本程序块中指令序列之间存在的并行性。最常见的基本块是循环体。循环体中指令之间的并行性称为循环级并行性,这是指令级并行研究的重点之一。在开发循环级并行的各种技术中,最基本的技术有循环展开(loop unrolling)技术和重命名(renameing)技术。

表 4.2 本章使用的浮点流水线的延迟

产生结果指令	使用结果指令	延迟时钟周期数
浮点计算	另外的浮点计算	3
浮点计算	浮点存操作(SD)	2
浮点取操作(LD)	浮点计算	1
浮点取操作(LD)	浮点存操作(SD)	0

循环展开是展开循环体若干次,将循环级并行性转化为指令级并行的技术。这个过程既可以通过编译器静态完成,也可以通过硬件动态进行。开发循环级并行性的另外一个重要技术是向量处理技术。具有向量处理指令的典型机器是向量计算机,有关向量处理和向量计算机的内容在第三章中已有介绍,本章不作讨论。与前面一致,本章中的分支指令就是指条件转移指令。

4.1.1 循环展开调度的基本方法

要保证流水线的效率,就必须保持流水线长时间充满。通过开发无关指令序列,使它们重叠执行,可以使流水线长时间地充满。所谓指令调度就是通过改变指令在程序中的位置,将相关指令之间的距离加大到不小于指令执行延迟的时钟数,这样就可以将相关指令转化为实际上无关指令。指令调度是循环展开的技术基础。

编译器在完成这种指令调度时,受限于以下两个特性:一是程序固有的指令级并行性,二是流水线功能部件的执行延迟。本章中使用的浮点流水线的延迟如表 4.2 所示。整数流水线采用标准 DLX 整数流水线,除了分支操作有一个时钟周期的延迟,其他操作没有延迟。下面通过一个实例对循环展开技术进行研究和性能分析。

例 4.1 对于下面的源代码,转换成 DLX 汇编语言,在不进行指令调度和进行指令调度两种情况下,分析代码一次循环的执行时间。

```
for (i=1; i<=1000; i++)
    x[i] = x[i] + s;
```

解 按照编译技术,首先给变量分配寄存器:整数寄存器 R1 用作循环计数器,初值为向量中最高端地址元素的地址,浮点寄存器 F2 用于保存常数 s。下面将程序转换成 DLX 汇编语言程序。

```
LOOP:   LD      F0,0(R1)    ;F0 中为向量元素
        ADDD   F4,F0,F2    ;加常数
        SD     0(R1),F4    ;保存结果
        SUBI   R1,R1,#8    ;修改指针,偏移 8 个字节
```

BNEZ R1,LOOP ;循环控制

不进行指令调度的情况下,根据表 4.2 中指令执行延迟的定义,程序执行情况如下:

指令流出时钟			
LOOP:	LD	F0,0(R1)	1
	(空转)		2
	ADDD	F4,F0,F2	3
	(空转)		4
	(空转)		5
	SD	0(R1),F4	6
	SUBI	R1,R1,#8	7
	BNEZ	R1,LOOP	8
	(空转)		9

可以看出,每次循环需要 9 个时钟周期,其中有 4 个是空转周期。在对上述程序进行指令调度以后,程序的执行情况如下:

指令流出时钟			
LOOP:	LD	F0,0(R1)	1
	(空转)		2
	ADDD	F4,F0,F2	3
	SUBI	R1,R1,#8	4
	BNEZ	R1,LOOP	5
	SD	#8(R1),F4	6

最后是将保存结果的指令放在分支指令的转移空槽中,一次循环的操作时间从 9 个时钟周期减少到 6 个时钟周期,其中只有 1 个空转周期。

这个例子中的指令调度是由编译器完成的。它首先在指令 ADDD 和指令 SD 之间的空转周期中调入后续指令序列中无关的指令,这种“调动”可能导致一些指令中操作数地址的偏移量发生变化。一个“聪明”的编译器可以找出指令块中的无关指令,对它们的位置进行调整,并根据调整的结果,对其中发生变化的位置(地址)进行修改,如例子中需要将存指令(SD)中目标操作数地址的偏移量由 0 变为 8。

进一步分析上面的例子可以发现,虽然一次循环的执行时间从 9 个时钟周期减少到 6 个时钟周期,但是其中只有 LD、ADDD 和 SD 这 3 条指令是有效操作,占用 3 个时钟周期,而空转、SUBI 和 BNEZ 这 3 个时钟周期都是为了控制循环而附加的。减少这种附加开销的最简单办法就是运用循环展开技术。

循环展开就是通过多次复制循环体并改变结束条件来相对增加有效操作时间。这种技术也给编译器进行指令调度带来了方便。通过下面的例子，我们可以看到循环展开技术的特点。

例 4.2 将上述例子中的循环展开，复制 4 个副本，在对展开后的指令序列不调度和调度两种情况下，分析代码的性能。

解 首先分配寄存器。F0、F4 已经用于第 1 个副本，F2 用于保存常数，F6 和 F8 用于第 2 个副本，F10 和 F12 用于第 3 个副本，F14 和 F16 用于第 4 个副本。展开后没有调度的代码如下：

指令流出时钟		
LOOP:	LD	F0,0(R1) 1
	(空转)	2
	ADDD	F4,F0,F2 3
	(空转)	4
	(空转)	5
	SD	0(R1),F4 6
	LD	F6,-8(R1) 7
	(空转)	8
	ADDD	F8,F6,F2 9
	(空转)	10
	(空转)	11
	SD	-8(R1),F8 12
	LD	F10,-16(R1) 13
	(空转)	14
	ADDD	F12,F10,F2 15
	(空转)	16
	(空转)	17
	SD	-16(R1),F12 18
	LD	F14,-24(R1) 19
	(空转)	20
	ADDD	F16,F14,F2 21
	(空转)	22
	(空转)	23
	SD	-24(R1),F16 24
	SUBI	R1,R1,±32 25
	BNEZ	R1,LOOP 26

(空转)

27

这个循环共使用了 27 个时钟周期, 平均每次循环使用 $27/4 \approx 6.7$ 个时钟周期。与原始的 9 个时钟周期相比较, 节省的时间主要是从减少循环控制开销中获得的。

循环展开后, 实际指令只有 14 条, 其他 13 个周期都是空转, 可见效率并不高。对指令序列进行调度, 减少空转周期后, 结果如下:

		指令流出时钟
LOOP:	LD	F0,0(R1) 1
	LD	F6,-8(R1) 2
	LD	F10,-16(R1) 3
	LD	F14,-24(R1) 4
	ADDD	F4,F0,F2 5
	ADDD	F8,F6,F2 6
	ADDD	F12,F10,F2 7
	ADDD	F16,F14,F2 8
	SD	0(R1),F4 9
	SD	-8(R1),F8 10
	SD	-16(R1),F12 11
	SUBI	R1,R1,#32 12
	BNEZ	R1,LOOP 13
	SD	8(R1),F16 14

这个循环由于没有数据相关引起的空转等待, 整个循环仅仅使用了 14 个时钟周期, 平均每次循环使用 $14/4=3.5$ 个时钟周期。

从上述例子中可以知道, 由编译器完成的循环展开和静态调度, 能够有效地提高循环级并行性。仔细分析代码结构和执行过程, 可以发现这种循环级并行性的提高是通过指令级并行来达到的。

从上述例子中还可以看出, 循环展开和指令调度时要注意以下几个方面:

(1) 保证正确性。在循环展开和调度过程中尤其要注意两个地方的正确性, 一是循环控制, 二是操作数偏移量的修改。

(2) 注意有效性。只有能够找到不同循环次之间的无关性, 才能够有效地使用循环展开。

(3) 使用不同的寄存器。如果使用相同或者较少数量的寄存器, 就可能导致新的冲突。

(4) 尽可能减少循环控制中的测试和分支。

(5) 注意对存储器数据的相关性分析。对于存储器取指令(Load)和存储器

存指令(Store),需要确定它们中哪些地址是相同的,这种分析对于跨循环次的访存指令尤其重要。

(6) 注意新的相关性。由于展开前不同次的循环在展开后都到了同一次循环中,因此可能带来新的相关性。

实现循环展开的关键是分析清楚代码中指令的相关性,然后通过指令调度来消除相关。这也就是本章的研究内容。

4.1.2 相关性

研究程序代码中的相关性,不但可以明确指令的调度,而且可以明确代码固有的并行性和可以获得的并行性。我们知道,相关的两条指令不但不能同时执行,改变指令的顺序也会引起问题。但是,相关是否导致流水线的空转,还与流水线的组织与结构有关,理解和处理这两点之间的关系是开发指令级并行的关键。

在程序基本块的范围内,相关主要有以下三种:数据相关、名相关和控制相关。

1. 数据相关(data dependence)

对于指令 i 和指令 j ,如果

- (1) 指令 j 使用指令 i 产生的结果,或者
- (2) 指令 j 与指令 k 数据相关,指令 k 与指令 i 数据相关

则指令 j 与指令 i 数据相关。

第二个条件表明,数据相关具有传递性。数据相关是两条指令之间存在一个先写后读相关链,这种相关链是导致流水线停顿的原因之一。分析数据相关的主要工作是:

- (1) 确定指令的相关性
- (2) 确定数据的计算顺序
- (3) 确定最大并行性

这些因素决定了是否可获得程序中存在的指令级并行。数据相关是程序相关性中最本质的相关之一。

2. 名相关(name dependence)

指令使用的寄存器或存储器称为名。如果两条指令使用相同的名,但是它们之间并没有数据流,则称之为名相关。指令 j 与指令 i 之间名相关有以下两种:

- (1) 反相关(anti-dependence):指令 i 先执行,指令 j 写的名是指令 i 读的名。反相关指令之间的执行顺序是必须保证的。反相关就是先读后写相关。
- (2) 输出相关(output dependence):指令 j 和指令 i 写相同的名。输出相关

指令的指令顺序是不允许颠倒的。输出相关就是写后写相关。

与数据相关比较,名相关的指令之间没有数据流。一条指令中的名改变了,并不影响相关的另外一条指令的执行,因此可以通过改变指令中操作数的名来消除名相关,这就是重命名(renameing)技术。对于寄存器操作数进行重命名称为寄存器重命名(register renaming),重命名过程既可以用编译器静态完成,也可以用硬件动态完成。

3. 控制相关(control dependence)

控制相关是指由分支指令引起的相关。它需要根据分支指令执行的结果来确定后续指令执行的顺序。控制相关与分支成功和分支不成功两个基本程序块的执行有关,典型的程序结构是“if-then-else”结构。处理控制相关有以下两个原则:

- (1) 与控制相关的指令不能移到分支指令之前;
- (2) 与控制无关的指令不能移到分支指令之后。

由于控制相关是分支指令引起的,所以除去分支指令可以减少或消除控制相关。在前面讨论的循环展开技术中,消除了若干条分支指令,这就是消除了程序中的控制相关。

4.2 指令的动态调度

在基本流水线中,当流水线取一条指令时,只要指令与正在流水线中运行的指令不存在数据相关,或者存在可以通过相关专用通路机制来克服的相关,就可以流出这条指令。相关专用通路机制降低了流水线的实际延迟,因而某些数据相关不会导致数据阻塞,这个过程就是相关隐藏。如果某数据相关不能被隐藏,阻塞检测硬件机制会从使用结果的指令开始,暂停流水线,一直到数据相关消失以后,再流出新的指令。

早期的几种处理器采用了另外一种方法,称为动态调度。它通过硬件重新安排指令的执行顺序,来调整相关指令实际执行时的关系,减少处理器空转。它可以处理一些编译时未发现的相关(比如涉及到存储器访问的相关),从而简化了编译器。当初发明这些技术的目标是为了实现在一种流水线上编译的代码也可以在另外一种流水线上有效地运行。和其他技术一样,这些优点均是以硬件复杂性的显著增加来换取的。

尽管动态调度并不能真正消除数据相关,但它能在出现数据相关时尽量避免处理器空转。相反,静态流水线调度则是通过调度导致阻塞的相关指令,减少处理器空转。当然,也可以对运行于某种动态调度处理器上的代码进行静态调度。下面将讨论两种策略,第一种是解决写后写和先读后写相关引起的数据阻

塞;第二种是对第一种的扩展,它还可解决先写后读数据阻塞。

4.2.1 动态调度的原理

到目前为止所使用的流水线的最大局限性在于:指令是顺序流出的。也就是说,如果一条指令在流水线中,与之相关的指令及其后面的指令都不能进行处理。因而,如果流水线中比较近的两条指令相关,流水线就会长时间停止。如果有多个功能部件,它们就会处于空闲状态。也就是说,如果指令 j 与当前流水线中运行时间较长的指令 i 相关,那么 j 后面所有的指令都会停止直到 i 执行完毕。例如下面的代码:

```
DIVD F0,F2,F4      ;S1
ADDD F10,F0,F8     ;S2 S2 对 S1 数据相关,S2 不能执行
SUBD F12,F8,F14    ;S3 - S3 与 S1、S2 都没有相关,但不能执行
```

因为 ADDD 和 DIVD 两条指令相关导致流水线空转,所以 ADDD 不能执行;但是,虽然 SUBD 指令与流水线中的所有指令均无关,它仍然不能够执行。这就是由于指令顺序流出而带来的局限性。

在上一章讨论的基本流水线中,结构阻塞和数据阻塞均在译码阶段(ID)进行检查。要使上面指令序列中的 SUBD 执行下去,必须将指令的流出分为两部分:结构阻塞检查和数据阻塞检查。我们希望的是只要指令的操作数就绪就执行,即指令要乱序执行,而且指令的结束也是乱序的。

指令乱序结束带来的最大问题就是异常处理比较复杂。目前在动态调度的处理器中,异常处理是不精确的,因为后面的指令可能在导致异常的指令前面流出,或者在导致异常的指令之前结束。异常出现以后,是难以确定和恢复现场的。本节不讨论这个问题,在后面将通过推断(speculation)的方法来实现精确异常处理。

为了允许乱序执行,将基本流水线的译码阶段再分为两个阶段:

- (1) 流出(Issue, IS):指令译码,检查结构阻塞。
- (2) 读操作数(Read Operands, RO);数据阻塞检查,当没有数据相关引发的阻塞时就读操作数。

指令流出之前先被取至指令队列中,一旦满足流出条件,指令就从队列中流出。这样分段处理,就可以同时执行多条指令。执行阶段紧跟在读操作数之后,这和基本流水线的结构以及工作过程相同。在浮点流水线中,对于不同的运算类型,指令的执行可能需要不同的时钟周期。因而要明确一条指令何时开始执行,何时执行结束,这两者之间就是指令的执行时间。

4.2.2 动态调度算法之一：记分牌

在动态调度流水线中，所有的指令在 IS 阶段顺序流出 (in-order issue)。但是在第二阶段 RO 中，由于采用了相关专用通路机制，所以只要指令运行所需的资源满足并且没有数据相关，就应该允许指令乱序执行，同时记录下这些指令的运行状态。记分牌(scoreboard)技术就是这样一种方法，它的命名起源于具有此功能的 CDC 6600 的记分牌。

在介绍记分牌在基本流水线中如何工作之前，先来分析一下先读后写数据阻塞。这种阻塞在以前讨论过的浮点流水线或整数流水线中均不存在，但当指令乱序执行时就会出现。假设以前的例子中 SUBD 的目的寄存器是 F8，即代码序列为

```
DIVD F0,F2,F4  
ADDD F10,F0,F8  
SUBD F8,F8,F14
```

在 ADDD 和 SUBD 指令之间存在着反相关：如果流水线在 ADDD 读出 F8 之前 SUBD 执行结束，就会出现问题。同样道理，为避免输出相关，必须检测写后写数据相关（比如 SUBD 的目的寄存器也是 F10）。我们将会看到，记分牌技术通过将相关的后一条指令暂停来克服这些相关的。

记分牌技术的目标是尽可能早地执行无关指令，在没有数据相关的情况下，达到每个时钟周期执行一条指令。如果某条指令被暂停，而后面的指令与流水线中正在执行的或被暂停的指令不相关，那么后面的指令就可以流出。记分牌电路全盘负责指令的流出、执行以及相关检测。

要发挥指令乱序执行的好处，必须有多条指令同时处于执行阶段，这就要求有多个功能部件或功能部件流水化或者两者兼有。这里假设处理器采用多个功能部件。CDC 6600 具有 16 个功能部件：4 个浮点部件，5 个存储器访问部件，7 个整数操作部件。对于 DLX，记分牌技术主要用于浮点部件，因为其他部件的操作延迟很小，无所谓同时执行。我们假设有 1 个浮点乘法器、1 个浮点加法器、1 个浮点除法部件和 1 个整数部件，整数部件用来处理所有的存储器访问、分支处理和整数操作。尽管这个例子比 CDC 6600 简单，但它足以阐明记分牌的基本工作原理。图 4.1 给出了记分牌处理器的基本结构。

每条指令均经过记分牌，并记录下有关数据相关的信息，这一步对应于指令流出，部分取代了 DLX 的指令译码 (ID) 阶段的功能。然后记分牌就需要判断什么时候指令可以读操作数并开始执行。如果记分牌判断某条指令不能立即执行，它就检测硬件的变化从而决定何时能够执行。记分牌还控制指令写目标寄存器的时机。因而，阻塞及其解除的监测全都集中在记分牌上。后面我们会看到记分牌的结构图，但首先需要了解一下流水线中指令流出和执行的步骤。

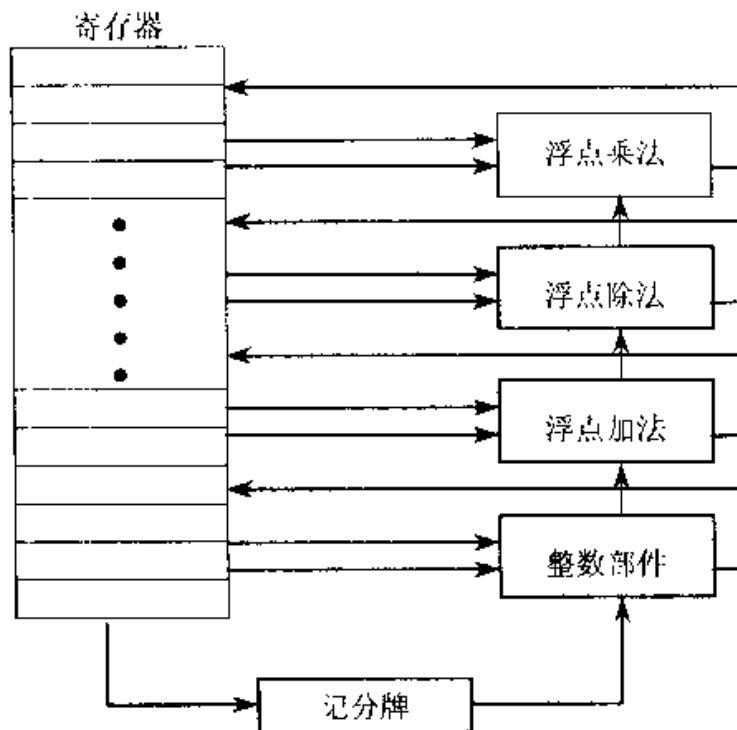


图 4.1 具有记分牌的 DLX 处理器基本结构

每条指令的执行可分为四步(既然主要考虑浮点操作,就不用关心存储器访问这一步)。下面首先介绍一下这四步的主要功能,然后再看记分牌是如何保存必要的信息,从而能决定下一步进行的时间的。这四步代替了 DLX 的指令译码、执行和写结果:

1. 指令流出(Issue, 记为 IS): 如果指令所需的功能部件空闲, 并且其他正在执行的指令使用的目的寄存器与该指令的不同, 记分牌就向相应功能部件流出该指令, 并改变记分牌内部的数据结构。

这一步替代 DLX 流水线中指令译码(ID)阶段的一部分。通过确保正在流水线中执行的指令中没有与所要流出的指令有相同的目的寄存器, 从而可避免出现写后写数据相关。如果存在结构相关或写后写相关, 这条指令就会被停止流出, 并且后面指令的流出也将停止, 直到阻塞消失。

2. 读操作数(Read Operand, 记为 RO): 如果前面已流出的正在运行的指令不对本指令的源操作数寄存器进行写操作, 或者一个正在工作的功能部件已经完成了对这个寄存器的写操作, 那么此操作数有效。这个检测工作由计分牌完成。当操作数有效后, 计分牌告诉功能部件读操作数并开始执行。

这个过程解决了数据的先写后读(RAW)相关。通过以上步骤, 计分牌动态解决了相关阻塞, 指令可能乱序流出。读操作数与指令流出加在一起相当于简单的 DLX 流水线的指令译码(ID)。

3. 执行(Execution, 记为 EX): 取到操作数后就开始执行指令。这一步相当于 DLX 流水线中的 EXE, 并且在流水线中要占用多个时钟周期。

4. 写结果(Write Result, 记为 WR): 记分牌知道指令执行完毕后, 如果目标寄存器空闲, 就将结果写入到目标寄存器中, 然后释放本指令使用的所有资源。

这里将检测先读后写相关, 如果有必要, 计分牌将暂停此指令写结果到目的寄存器, 直到相关消失。一般, 出现以下的情况就不允许指令写结果:

- (1) 前面的某条指令(按顺序流出)还没有读取操作数;
- (2) 其中某个操作数寄存器与结束的指令的目的寄存器相同。

先读后写阻塞不存在或消失以后, 记分牌将通知功能部件将结果存到结果寄存器。这一步替代了 DLX 流水线中的 WB 一步。

只有所有的操作数全都准备好以后才算指令的操作数准备就绪, 因而记分牌只有所有的寄存器全准备好才可以读。记分牌根据它自己的数据结构, 通过与功能部件的通信来控制指令处理的每一步。然而还存在一个问题, 就是操作数和结果到寄存器文件的总线是有限的, 这会导致结构阻塞。记分牌必须保证进入流水线第 2 步和第 4 步的功能部件的总数不能超过可用总线的数目。在此不详细讨论这个问题。

下面详细分析一下一个具有五个功能部件的 DLX 流水线所要维护的数据结构。图 4.2 给出下列代码运行过程中记分牌保存的信息。

LD	F6,34(R2)
LD	F2,45(R3)
MULTD	F0,F2,F4
SUBD	F8,F6,F2
DIVD	F10,F0,F6
ADDD	F6,F8,F2

记分牌的信息分为三部分:

- (1) 指令状态表: 表示正在执行的各条指令处于四步中的哪一步。
- (2) 功能部件状态表: 指示出功能部件的状态。每个功能部件状态表有九个域:

Busy: 指示功能部件是否在工作

Op: 功能部件当前执行的操作

Fi: 目的寄存器编号

Fj, Fk: 源寄存器编号

Qj, Qk: 向 Fj, Fk 中写结果的功能部件

Rj, Rk: Fj, Fk 就绪并且还没有被使用标志

- (3) 结果寄存器状态表: 标示每个寄存器如果是当前运行的指令目的寄存器, 它由哪个功能部件写结果。

下面看一看图 4.2 中的指令序列如何往下执行。

例 4.3 假设浮点流水线中执行的延迟如下: 加法需 2 个时钟周期, 乘法需 10 个时钟周期, 除法需 40 个时钟周期。代码段和起始点状态如图 4.2。分别给出 MULTD 和 DIVD 准备写结果之前的记分牌状态。

指令	指令状态表			
	IS	RO	EX	WR
LD F6,34(R2)	√	√	√	√
LD F2,45(R3)	√	√	√	
MULTD F0,F2,F4	√			
SUBD F8,F6,F2	√			
DIVD F10,F0,F6	√			
ADDD F6,F8,F2				

部件名称	功能部件状态表								
	Busy	Op	F _i	F _j	F _k	Q _j	Q _k	R _j	R _k
整数	yes	LD	F2	R3				no	
乘法	yes	MULTD	F0	F2	F4	整数		no	yes
加法	yes	SUBD	F8	F6	F2		整数	yes	no
除法	yes	DIVD	F10	F0	F6	乘法		no	yes

部件名称	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
部件名称	乘法	整数			加法	除法		

图 4.2 记分牌信息组成和记录的信息

解 第二个 LD 指令与 MULD 和 SUBD 之间、MULTD 和 DIVD 之间以及 SUBD 和 ADDD 之间存在着先写后读相关; DIVD 和 ADDD 之间存在着先读后写相关; ADDD 指令对于 SUBD 指令关于浮点加法部件还存在着结构相关。图 4.3 和图 4.4 分别给出 MULTD 和 DIVD 指令将要写结果时记分牌的状态。

现在来详细分析讨论一下记分牌是如何控制指令执行的。操作在流水线中前进时, 记分牌必须记录与操作有关的信息, 如寄存器名等。下面是每条指令在流水线中前进的条件和记分牌的记录。为区分寄存器的名字和寄存器的值, 约定将寄存器的名字加'', 例如 $F_j(FU) \leftarrow 'S1'$ 表示将寄存器 S1 的名字送入 $F_j(FU)$, 而不是它的内容。FU 表示指令使用的功能部件; D 表示目的寄存器的名字, S1 和 S2 表示源操作数的名字, Op 是要进行的操作; $F_j(FU)$ 表示功能部件 FU 的 F_j 数据项; result('D') 表示结果寄存器状态表中对应于寄存器 D 的内容, 为产生结果 D 的功能部件名。

1. 指令流出 (IS)

(1) 进入条件:

not Busy(FU) and not result('D'); //判断结构阻塞和写后写

(2) 记分牌记录内容:

指令	指令状态表			
	IS	RO	EX	WR
LD F6,F4(R2)	√	√	√	√
LD F2,F4(R3)	√	√	√	√
MULTD F0,F2,F4	√	√	√	
SUBD F8,F6,F2	√	√	√	√
DIVD F10,F0,F6	√			
ADDD F6,F8,F2	√	√	√	

部件名称	功能部件状态表								
	Busy	Op	F _i	F _j	F _k	Q _j	Q _k	R _j	R _k
整数	no								
乘法	yes	MULTD	F0	F2	F4			no	no
加法	yes	ADDD	F6	F8	F2			no	no
除法	yes	DIVD	F10	F0	F6	乘法		no	yes

部件名称	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
部件名称	乘法			加法		除法		

图 4.3 程序段执行到 MULTD 将要写结果时记分牌的状态

```

Busy(FU)←yes;
Op(FU)←Op;
Fi(FU)←'D';
Fj(FU)←'S1';
Fk(FU)←'S2';
Qj←result('S1');           //产生'S1'的FU
Qk←result('S2');           //产生'S2'的FU
Rj←not Qj;                  //Rj是否可用?
Rk←not Qk;                  //Rk是否可用?
result('D')←FU;             //‘D’被FU用作结果Reg.

```

2. 读操作数(RO)

(1) 进入条件:

Rj·Rk; //解决先写后读

(2) 记分牌记录内容:

Rj←no; //已经取走了就绪的数据 Rj

Rk←no; //已经取走了就绪的数据 Rk

3. 执行(EX)

结束条件:

功能部件操作结束

4. 写结果(WR)

(1) 进入条件:

 $\forall f((F_j(f) \neq F_i(FU) \text{ or } R_j(f) = no))$ and $(F_k(f) \neq F_i(FU) \text{ or } R_k(f) = no)); //\text{检查是否存在先读后写}$

(2) 记分牌记录内容:

 $\forall f(if Q_j(f) = FU \text{ then } R_j(f) \leftarrow yes); //\text{有等结果的指令,则数据可用}$ $\forall f(if Q_k(f) = FU \text{ then } R_k(f) \leftarrow yes);$ result($F_i(FU)$) = no; $//F_i(FU)$ 空闲busy(FU) = no

指令	指令状态表			
	IS	RO	EX	WR
LD F6,34(R2)	√	√	√	√
LD F2,45(R3)	√	√	√	√
MULTD F0,F2,F4	√	√	√	√
SUBD F8,F6,F2	√	√	√	√
DIVD F10,F0,F6	√	√	√	
ADDD F6,F8,F2	√	√	√	√

部件名称	功能部件状态表								
	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
整数	no								
乘法	no								
加法	no								
除法	yes	DIVD	F10	F0	F6			no	no

部件名称	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
						除法		

图 4.4 序段执行到 DIVD 将要写结果时记分牌的状态

CDC 6600 设计者测试的结果是:对于 FORTRAN 语言,性能提高 1.7 倍;而对于手写的汇编程序性能提高 2.5 倍。CDC 6600 上记分牌的逻辑电路相当于一个功能部件,耗费惊人地低。耗费最大的地方是大量的总线——大约是顺序执行指令(假设执行周期流出一条指令)的处理器中总线的四倍。记分牌技术

通过指令级并行来减少程序中因为数据相关引起的流水线停顿,它的性能受限于以下几个方面:

- (1) 程序指令中可开发的并行性,即是否存在可以并行执行的不相关的指令。如果每条指令均与前面的指令相关,那么任何动态调度策略均无法解决流水线停顿的问题。如果指令级并行性仅仅从同一个基本程序段中开发(像 CDC 6600 那样),并行性也不会太高。
- (2) 记分牌容量:记分牌的容量决定了流水线能在多大范围内寻找不相关指令。
- (3) 功能部件的数目和种类:它决定了结构冲突的严重程度。采用动态调度后结构冲突会更加频繁。
- (4) 反相关和输出相关:引起记分牌中先读后写和写后写阻塞。

第 2 和第 3 个问题可通过增加记分牌的容量和功能部件的数量来解决,这会导致处理器成本增加,并可能影响系统时钟周期时间。在采用动态调度的处理器中,写后写和先读后写阻塞会增多,因为乱序流出的指令在流水线中会引起更多的名相关。如果在动态调度中采用分支预测技术,就会出现循环的多个迭代同时执行,名相关将更加严重。

下面将要讨论的是寄存器重命名(register renaming)技术,它动态地消除名相关,从而避免先读后写和写后写冲突。寄存器重命名技术使用大量的虚拟寄存器代替源代码中的寄存器,它也是相关专用通路技术实现的基础。

4.2.3 动态调度算法之二: Tomasulo 算法

IBM 360/91 浮点部件首先采用了这种机制,它允许指令发生冲突后还可以继续执行。它是由 Robert Tomasulo 发明的,因而以他的名字命名,称为 Tomasulo 算法。Tomasulo 算法将记分牌的关键部分和寄存器重命名技术结合在一起,尽管此机制的实现中有多种变化,但其基本核心都是通过寄存器重命名来解决写后写和先读后写冲突。

IBM 360/91 比 CDC 6600 晚三年推出,恰巧在商业计算机使用 Cache 之前。IBM 的目标是想在整个 360 系列仅仅设计一个指令系统和一个编译器,使之在各种档次的计算机上都能达到很高的性能。360/91 要求具有很高的浮点性能,但不是通过高端机器的专用的编译器实现;并且 360 只有四个双精度浮点寄存器,编译器调度的有效性受到很大限制,这也是使用 Tomasulo 算法的原因;360/91 的访存时间和浮点计算时间都很长,这也是 Tomasulo 算法要克服的问题;另外,Tomasulo 算法还可支持循环的多次迭代重叠执行。

下面的讨论是基于 DLX 的浮点流水线功能部件。DLX 和 IBM 360/91 主要的不同是 360/91 使用寄存器 - 存储器型指令,但是 360/91 使用了一个取

(Load)功能部件,所以加入寄存器-存储器寻址模式也无需很大的变动。360/91使用的是流水功能部件,而不是多个功能部件。现在假设 DLX 是多个功能部件环境。IBM 360/91 同时支持三个浮点加操作、两个浮点乘操作、六个浮点取操作和三个浮点存操作,取和存分别通过取数缓冲和存数缓冲。

前面讨论过,通过寄存器重命名,编译器可以解决数据写后写相关和先读后写相关。在 Tomasulo 算法中,寄存器重命名是通过保留站(reservation station)来实现,它保存等待流出和正在流出指令所需要的操作数。Tomasulo 算法的基本思想是只要操作数有效,就将其取到保留站,避免指令直接从寄存器中取数据。即将执行的指令从相应的保留站中取得操作数,并将执行结果直接送到等待数据的其他保留站。因而,对于连续的寄存器写,只有最后一个才真正更新寄存器中的内容。一条指令流出时,存放操作数的寄存器被重新命名为对应于该寄存器保留站的名称(编号),以上过程就是寄存器重命名(register renaming)。指令流出逻辑和保留站相结合实现寄存器重命名,从而消除数据写后写和先读后写相关。这是 Tomasulo 算法和记分牌在概念上最大的不同。因为保留站的数目远多于实际的寄存器,因而可以消除一些编译技术所不能解决的相关。在下面对 Tomasulo 算法的讨论中,我们还会回到寄存器重命名这个问题上,来看它具体的过程以及如何消除冲突。

除了寄存器重命名技术,Tomasulo 算法和记分牌在结构上还有两处显著的不同。第一,冲突检测和指令执行控制机制分开。一个功能部件的指令何时开始执行,由每个功能部件的保留站控制,而记分牌则是集中控制。第二,计算的结果通过相关专用通路直接从功能部件进入对应的保留站进行缓冲,而不一定是写到寄存器。这个相关专用通路通过一条公用数据总线(因为在 IBM 360/91 中称为公共数据总线 Common Data Bus,简称 CDB)来实现,所有等待这个结果的功能部件(指令)可同时读取。与之相比,记分牌将结果写到寄存器,等待功能部件来相互竞争。

记分牌和 Tomasulo 算法中结果总线的数目均可以变化。实际的机器中,CDC 6600 有多条结果总线(浮点部件有两条),而 IBM 360/91 只有一条。

图 4.5 是采用 Tomasulo 算法的 DLX 的浮点部件的基本结构,图中不包括记录和控制指令执行过程中使用的各种表格。保留站保存已流出到本功能部件的等待执行的指令;如果该操作的源操作数已经存在,则将它取来保存到保留站中;如果操作数还没有计算出来,则保留站中记录它的源在何处,即指明它由哪个功能部件产生。取缓冲和存缓冲保存的是读写存储器的数据或地址。浮点寄存器通过一对总线连到功能部件,并通过其中一条总线连到存缓冲。功能部件的结果和从存储器读取的数据都送到公共数据总线上,除了取缓冲的输入和存缓冲的输出以外,所有部分均与公共数据总线相连。所有的缓冲和保留站全有

用于阻塞控制的标志位。浮点加法器完成加法和减法操作，浮点乘法器完成乘法和除法操作。

与讨论记分牌时一样，在详细研究保留站及其算法之前，先来看一下指令流水线的分站情况。因为操作数的传输过程与记分牌不同，使用 Tomasulo 算法的指令流水线仅需三站：

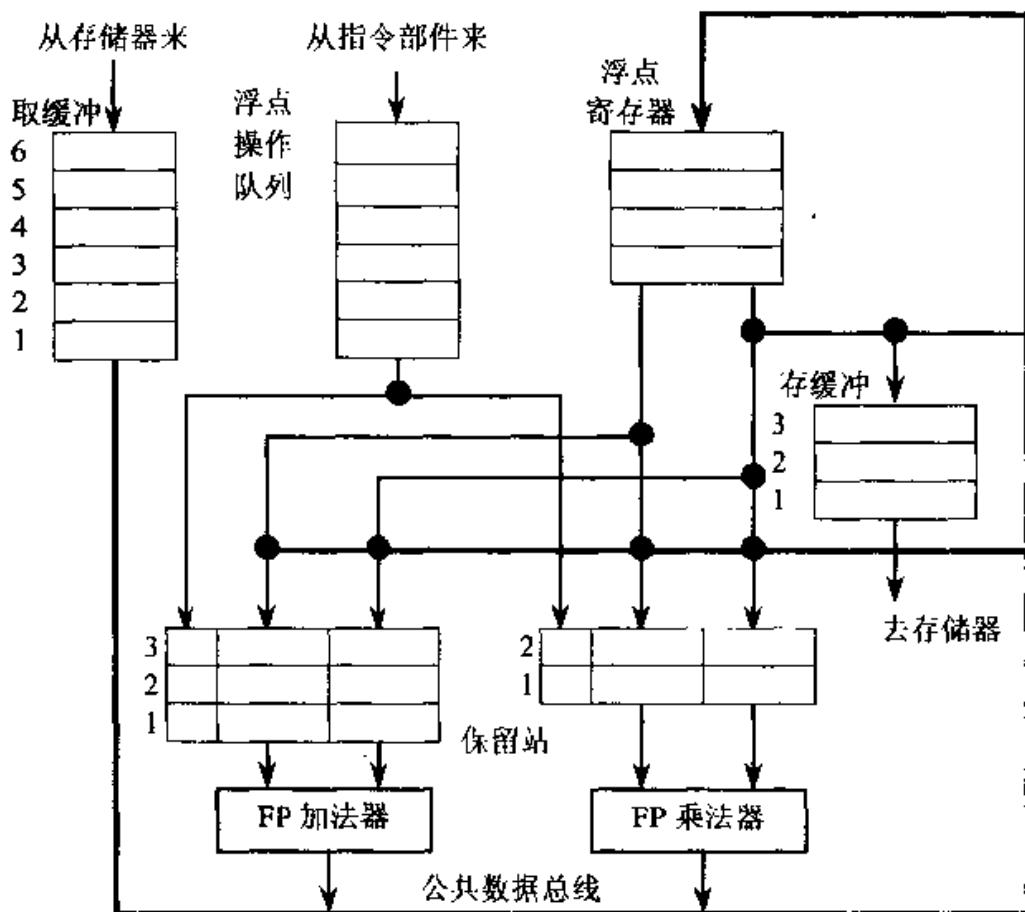


图 4.5 用 Tomasulo 算法的 DLX 浮点部件的结构

(1) 流出(Issue): 从浮点操作队列中取一条指令。如果是浮点操作并且有空的保留站就流出。如果操作数在寄存器中就将其送入保留站。如果是访存指令，只要有空的缓冲，指令就流出。如果没有空的保留站或空缓冲，这就发生了结构冲突，指令就停止流出，直至需要的保留站或缓冲为空。这一步还进行寄存器重命名处理。

(2) 执行(Execute): 如果有操作数未就绪，监视公共数据总线等待所需寄存器的计算结果。某个操作数计算完毕后就被放入等待该结果的所有保留站。当两个操作数都就绪后开始执行指令。这一步检查先写后读数据冲突。

(3) 写结果(Write Result): 结果计算完毕后，将其写入公共数据总线，从而传输至等待此结果的功能部件和寄存器。

尽管这些步骤和记分牌基本上类似，但有三处显著的不同之处：第一，没有

检查数据写后写和先读后写相关的过程，在指令流出过程中操作数寄存器重命名已将其消除。第二，通过公共数据总线来广播结果，将结果送到所有等待此结果的保留站，目标寄存器也相当于一个保留站。第三，存储器存和取都作为基本的功能部件。另外，由于保留站技术有效地解决先写后读，而无需特殊处理，因此，记分牌中判断先写后读的取操作数这一步也被消除。

用于检测和消除阻塞的数据结构附加在保留站、寄存器文件和存/取缓冲上。不同的部件附加的信息不同。除了取缓冲之外，各部件的每一项均由一个标志域，它保存寄存器重命名所使用的虚拟寄存器的名称。在 DLX 中，采用 4 位数字表示 5 个保留站和 6 个取缓冲中的某一个部件，5 个保留站和 4 个寄存器就等同于有 9 个结果寄存器（而 IBM 360/91 中只有 4 个双精度寄存器）。我们希望通过寄存器重命名来获得更大的虚拟寄存器空间。标志域指出哪个保留站中指令运行的结果是本部件的操作数。一旦某条指令流出后在等待操作数，它将用产生这个操作数的保留站号来标示操作数，而不是等待寄存器中的结果。特殊编号 0 表示寄存器中的操作数有效。

保留站的数目多于实际的寄存器，通过使用保留站将寄存器重命名，就消除了数据的写后写和先读后写相关。在 Tomasulo 算法中，保留站就是扩展的虚拟寄存器，别的方法还可能采用加入额外的寄存器，如后面将要讲的再定序（reorder）缓冲寄存器等。

下面定义一下有关的术语和数据结构。在不会引起二义性的情况下尽量采用记分牌中的术语，另外还沿用了 IBM 360/91 的一些术语。这里再强调一下：Tomasulo 算法中所讲的标志(tags)是指缓冲或产生结果的功能部件；当一条指令流出到保留站以后，原来操作数的寄存器名将不再引用。

每个保留站有 6 个域：

Op: 对操作数 S1 和 S2 所进行的操作。

Q_j, Q_k: 产生结果的保留站号。等于 0，表示操作数在 V_j 和 V_k 中，或不需要操作数。

V_j, V_k: 两个源操作数的值。操作数项中，V 或 Q 域最多只有一个有效。

Busy: 标示本保留站和相应的功能部件是否空闲。

寄存器文件和存缓冲每项还有一个 Q_i 域：

Q_i: 操作结果要存入本寄存器或存储器的指令的保留站号。如果 Q_i 空，表示当前没有指令要将结果写入此寄存器或存储器。

存和取缓冲还各有一个 Busy 域和一个 Address 域：

Busy: 标示缓冲是否完成相应的存/取操作或空闲。当寄存器空闲时，Q_i 域空。

Address: 地址域，用于记录存或取的地址。

此外，存缓冲也有一个 V 域，保存要存入存储器的数据。

在讨论具体的算法之前,先看一下对于下列代码,保留站的信息是怎样的。

LD	F6,34(R2)	;S1
LD	F2,45(R3)	;S2
MULTD	F0,F2,F4	;S3
SUBD	F8,F6,F2	;S4
DIVD	F10,F0,F6	;S5
ADDD	F6,F8,F2	;S6

前面已看过当上述代码 S1 写入结果后记分牌的信息,现在图 4.6 给出的是 Tomasulo 算法中此时保留站、存缓冲、取缓冲和寄存器的标志等信息。ADD1 标志表示是第一个加法功能部件,MULTD1 标志表示是第一个乘法功能部件,依此类推。图中列出的指令状态表,仅仅是为了帮助理解,实际上并不是硬件的一部分,每条指令的状态都保存在保留站中。

图中所有的指令均已流出,但只有第一条 LD 指令执行完毕并将结果写到公共数据总线上。V_j 和 V_k 中用类似硬件描述语言的方法给出操作数的值,存/取缓冲的状态没有给出。取缓冲 2 是此时唯一工作的单元,它执行的是代码序列中的 S2,从地址为 R3 + 45 的存储器单元中取操作数。一个操作数在任何时间只能由 Q 域或 V 域之一来标示。

指 令	指令状态表				
	流 出	执 行	写 结 果		
LD F6,34(R2)	√	√			√
LD F2,45(R3)	√	√			
MULTD F0,F2,F4	√				
SUBD F8,F6,F2	√				
DIVD F10,F0,F6	√				
ADDD F6,F8,F2	√				

名称	保 留 站					
	Busy	Op	V _j	V _k	Q _j	Q _k
ADD1	yes	SUBD	MEM[34+REGS[R2]]			LOAD2
ADD2	yes	ADDD			ADD1	LOAD2
ADD3	no					
MULT1	yes	MULTD		REG[F4]	LOAD2	
MULT2	yes	DIVD		MEM[34+REGS[R2]]	MULT1	

域	寄 存 器 状 态 表							
	F0	F2	F4	F6	F8	F10	...	F30
Qi	MULT1	LOAD2		ADD2	ADD1	MULT2		

图 4.6 第一条 LD 指令结束后保留站和寄存器的标志

直观上看,这个表中的信息与记分牌有两个显著的不同之处。第一,操作数一旦有效,它的值就被存入保留站的一个 V 域中,而不是从寄存器文件或计算结果的保留站中取,实际上保留站根本不保存计算结果。第二,指令 ADDD 在记分牌中由于存在先读后写数据相关,在写结果阶段阻塞,而在 Tomasulo 算法中由于消除了先读后写,它能够在 DIVD 指令执行前执行完毕。Tomasulo 算法主要的优点是:

- (1) 分布的阻塞检测逻辑机制;
- (2) 消除了数据写后写和先读后写相关导致的阻塞。

第一条优点带来的好处是如果多条指令都在等待一个结果,且指令的另一个操作数已准备就绪,公共数据总线广播这个被等待的结果后,这些指令均可以执行。而在记分牌中,等待的指令必须在寄存器总线就绪以后才能从寄存器中读取结果。

通过使用保留站进行寄存器重命名,Tomasulo 算法解决了数据写后写和先读后写相关导致的阻塞。例如,在上面的代码中 DIVD 和 ADDD 指令尽管由于 F6 存在先读后写数据阻塞,但也都可以流出。阻塞通过下列两种途径之一消除:一,产生 DIVD 指令操作数 F6 的指令(第一个 LD)一旦执行完,DIVD 指令对应保留站的 Vk 域就保存这个结果,这样 F6 不再有先读后写相关(DIVD 与 ADDD 之间),ADDD 指令就执行下去。二,如果第一个 LD 指令还未执行完毕,MULT2 的 Qk 将指向 LOAD1 保留站,而不是 F6。所以,无论哪种情况 ADDD 均可以流出并开始执行。实际上,所有使用第一条 LD 结果的指令情况均指向其保留站 LOAD1,从而允许 ADDD 指令执行并将结果存入结果寄存器而不影响 DIVD 的执行。一会儿我们将看一个消除写后写的例子,这里先看一下前面例子的运行情况。

例 4.4 假设浮点部件的延迟为:加法 2 个时钟周期,乘法 10 个时钟周期,除法 40 个时钟周期。代码同上,给出 MULT2 准备写结果时的状态表的信息。

解 结果如图 4.7 所示。与记分牌不同,因为 DIVD 的操作数已有副本,ADDD 可以执行完毕,从而克服了先读后写数据阻塞。要注意的是,即使对 F6 的取操作被延迟,对 F6 的加操作仍可以进行而不会导致写后写数据相关。

下面研究 Tomasulo 算法中指令执行的步骤。其中取(Load)操作和存(Store)操作指令稍有特殊之处。只要有取缓冲可用,取操作指令就可以执行;执行完毕后,就与其他功能部件一样,一旦公共数据总线空闲,就将结果放在公共数据总线上。存操作从公共数据总线或寄存器文件中取结果后自动执行;完毕后将这个保留站的 Busy 域置为空闲,以表示可供其他指令使用。

下面给出 Tomasulo 算法中指令执行的主要条件、步骤和记录。其中 D 是目的

寄存器, S1 和 S2 分别是操作数寄存器号,R 指定给 D 的保留站或缓冲号,RS 是保留站数据结构,由保留站或取缓冲返回的值为 result, reg 代表寄存器的数据结构(不是寄存器文件),store 是存缓冲的数据结构。另外不要忘了‘Ri’代表寄存器 Ri 的名字而非其内容。一条指令流出后,目的寄存器的 Qi 域置为指令流出到的保留站或缓冲号。如果寄存器中的操作数已就绪,将其存入 V 域。否则将 Q 域置为计算操作数的保留站号。指令在保留站中一直等到两个操作数全都就绪,即 Q 域全为零。Q 域在指令流出时置零或在与本指令相关的指令执行完毕并回写结果时置零。指令执行完毕后且公共数据总线空闲,就进行结果回写。所有的缓冲、寄存器和保留站的 Qj 或 Qk 域如果与执行完毕的保留站相同,就从公共数据总线上读取数据并标记 Q 域以表示操作数的值已取到。这样,公共数据总线就将结果在一个时钟周期内广播到多个目的地。如果等待操作数的指令就绪,它们全都可以在下一个时钟周期开始执行。

指 令	指令状态表					
	流 出	执 行	写结果			
LD F6,34(R2)	√	√				√
LD F2,45(F3)	√	√				√
MULTD F0,F2,F4	√	√				
SUBD F8,F6,F2	√	√				√
DIVD F10,F0,F6	√					
ADDD F6,F8,F2	√	√				√

名 称	保 留 站					
	Busy	Op	Vj	Vk	Qj	Qk
ADD1	no					
ADD2	no					
ADD3	no					
MULT1	yes	MULTD	MEM[45+REGS[R3]]	REG[F4]		
MULT2	yes	DIVD		MEM[34+REGS[R2]]	MULT1	

域	寄 存 器 状 态 表							
	F0	F2	F4	F6	F8	F10	...	F30
Qi	MULT1					MULT2		

图 4.7 MULTD 准备写结果时的状态表的信息

1. 指令流出(Issue)

(1) 进入条件:

保留站或缓冲有空闲

(2) 记录内容:

if(reg['S1'].Qi ≠ 0)

```

    {RS[R].Qj ← reg['S'].Qi} //寄存器重命名过程
else
    {RS[R].Vj ← S1;           //把操作数取到保留站
     RS[R].Qj ← 0};          //数据 Vj 有效
if(reg['S2'].Qi ≠ 0)
    {RS[R].Qk ← reg['S1'].Qi} //第二操作数
else
    {RS[R].Vk ← S2;
     RS[R].Qk ← 0};
RS[R].busy ← yes;           //本保留站忙
reg['D'].Qi ← R;           //寄存器D的本指令的目标寄存器
RS[R].Op ← Op;             //设置操作类型

```

2. 执行(Execution)

(1) 进入条件:

$(RS[R].Qj = 0) \text{ and } (RS[R].Qk = 0)$; //两个源操作数就绪

(2) 记录内容:

V_j 和 V_k 中不再有操作数

3. 写结果(Write Result)

(1) 进入条件:

保留站 R 执行结束, 且公共数据总线(CDB)可用(空闲)

(2) 记录内容:

```

 $\forall x (\text{if}(reg[x].Qi = R)$ 
    {  $F_x \leftarrow \text{result}$ ;           //向浮点寄存器写结果(所有的  $F_x$ )
       $reg[x].Qi \leftarrow 0$ });        //相应的目标寄存器中结果有效

 $\forall x (\text{if}(RS[x].Qj = R)$ 
    { $RS[x].Vj \leftarrow \text{result}$ ;   //所有使用本结果作为第一操作数的
     //保留站可以获得数据
       $RS[x].Qj \leftarrow 0$ });
 $\forall x (\text{if}(RS[x].Qk = R)$ 
    { $RS[x].Vk \leftarrow \text{result}$ ;   //所有使用本结果作为第二操作数的
     //保留站可以获得数据
       $RS[x].Qk \leftarrow 0$ });
 $\forall x (\text{if}(store[x].Qi = R)$ 
    { $store[x].V \leftarrow \text{result}$ ; //所有使用本结果的
      }

```

```

        //存操作缓冲数据有效
store[x].Qi ← 0 });
RS[R].Busy ← no;      //释放保留站，置保留站空闲

```

通过一个循环的例子,可以全面理解动态寄存器重命名是如何解决写后写和先读后写数据相关导致的阻塞。下面的循环是一个将数组中的元素与 F2 中的标量相乘的代码:

LOOP:	LD	F0,0(R1)
	MULTD	F4,F0,F2
	SD	0(R1),F4
	SUBI	R1,R1,#8
	BNEZ	R1,LOOP ;branches if R1≠0

假定转移成功,使用保留站将会使循环的多个迭代同时执行,而不需要事先进行循环展开,这实际上是通过硬件进行动态的循环展开。前面我们已经看到,循环展开和指令调度,需要使用大量的寄存器。在 IBM 360 的体系结构中,只有四个浮点寄存器,这大大限制了循环展开,因为展开会导致大量写后写和先读后写冲突。Tomasulo 算法通过使用大量的虚拟寄存器(保留站、缓冲),仅使用少数的寄存器就可以使循环的多个迭代同时运行。保留站通过重命名处理,扩展了实际的寄存器组。

例 4.5 现在假设将连续两个迭代的所有指令全都流出,但所有的浮点存取及操作全都没完成。给出状态表的信息。

解 保留站、寄存器状态表以及此时的存取缓冲如图 4.8 所示。忽略整数部件 ALU 的操作,并假设转移成功且设乘法操作可在 4 个时钟周期内完成(乘法执行段用 3 拍),一旦系统达到此状态则两个循环同时执行,CPI 将达到接近 1.0。如果忽略循环的开销,且假设有足够的寄存器,那么性能可达到通过编译器循环展开并调度后的水平。

图中存、取缓冲中的 Address 域表示取数或存数的地址。取操作在取缓冲中,乘法保留站中的有关项表示取的结果是其操作数。存缓冲表示乘法结果的目的是存操作的存入地址。

在此例中有 Tomasulo 算法的另一个关键技术——存储器访问地址分析和判别。第二层的取操作在第一层的存操作完成前就可以执行,这与正常的顺序是不同的。实际上,只要存取数据的存储器地址不同,存取就可以正确无误地乱序执行。因此在流出取操作指令之前,需要先检查存缓冲中的所有地址,如果取操作指令的地址与存缓冲中的某个地址相同,取操作指令就必须暂停等待,直到

存缓冲中没有与取操作具有相同地址的操作,才可以执行取操作指令。这种动态的存储器地址判别技术在编译器中调度取和存操作时也经常使用。

如果转移的开销不大(下一节的问题),这种动态调度机制能得到非常高的性能。这种方法的主要缺点是 Tomasulo 算法的复杂性,实现它需要大量的硬件,尤其是保存大量相关中间结果的高速缓存和复杂的控制逻辑。此外,性能还受限于公共数据总线,因为它是单总线,如果增加公共数据总线的数量,与总线连接的缓冲和保留站的接口硬件将会加倍。

指 令		指令状态表			
		循环层次	流出	执行	写结果
LD	F0,0(R1)	1	√	√	
MULTD	F4,F0,F2	1	√		
SD	0(R1),F4	1	√		
LD	F0,0(R1)	2	√	√	
MULTD	F4,F0,F2	2	√		
SD	0(R1),F4	2	√		

名称	保 留 站					
	Busy	Op	Vj	Vk	Qj	Qk
ADD1	no					
ADD2	no					
ADD3	no					
MULT1	yes	MULTD		REGS[F2]	LOAD1	
MULT2	yes	MULTD		REGS[F2]]	LOAD2	

域	寄存器状态表							
	F0	F2	F4	F6	F8	F10	...	F30
Qi	LOAD2		MULT2					

域	取 缓 冲		
	LOAD1	LOAD2	LOAD3
Address	REGS[R1]	REGS[R1]-8	
Busy	yes	yes	no

域	存 缓 冲		
	STORE1	STORE2	STORE3
Qi	MULT1	MULT2	
Busy	yes	yes	no
Address	REGS[R1]	REGS[R1]-8	

图 4.8 两个循环同时执行而无指令执行完毕时的状态表信息

Tomasulo 算法结合了两种不同的技术:一是寄存器重命名,这样可以获得

一个更大的虚拟寄存器空间；二是对寄存文件中源操作数缓存，它解决了寄存器中的有效操作数名相关引起的阻塞。这种寄存器重命名和一串链式的结果数缓存技术，在后面讨论硬件分支预测时还会用到。

Tomasulo 算法提高指令级并行的关键技术是：动态调度、寄存器重命名和动态存储器地址判别。

对应于数据相关的动态调度技术，分支也有动态处理技术，称为动态分支预测。动态分支预测技术有两种目标：预测转移是否成功和尽早得到转移目标地址。这就是下一节讨论的问题。

4.3 控制相关的动态解决技术

前面讨论了动态解决数据相关的一些技术，而分支指令导致的控制相关也可能导致流水线停顿。实际上，处理器可达到的指令级并行度越高，控制相关的影响越大。本节中介绍的技术，对于每个时钟周期流出一条指令的处理器来讲有一定的帮助，但处理器一个时钟周期流出多条指令还受到以下两个因素的限制：第一，流出 n 条指令的处理器中，遇到分支指令的速度也快了 n 倍；第二，根据 Amdahl 定律可知，随着机器 CPI 的降低，控制相关对性能的影响越来越大。

上一章中讨论了几种静态处理分支的机制，这些机制的动作不依赖于分支的动态行为。还讨论过分支延迟机制，它通过编译的调度来优化分支。本节着重通过硬件，动态地进行分支处理，对程序运行时分支的行为进行预测，提前对分支操作做出反应，加快分支处理的速度。

我们先从简单的分支预测机制开始，然后研究能提高预测准确性的分支预测方法，最后再研究一些更精细的机制，用它们来提前找到分支以后的指令。所有这些机制的目的是尽可能早地知道分支以后的指令，避免由控制相关导致的流水线停顿。分支预测的效果不仅取决于其准确性，而且与分支预测时的开销密切相关。分支转移的最终延迟取决于流水线的结构、预测的方法和预测错误后恢复所采取的策略。

4.3.1. 减少分支延迟：分支预测缓冲技术

动态分支预测技术是一种基于历史记录的分支预测，它必须解决好一个问题的两个方面，一方面是记录一个分支指令的历史，另一方面是决定预测的分支。

记录历史的方法有以下几种：仅仅记录最近一次或几次的分支历史；记录分支成功的地址；记录分支历史和分支目标，相当于前面两种方式的结合；记录分支目标的一条或若干条指令。下面根据这些历史记录的方法，分别讨论分支是如何预测的。

最简单的动态分支预测技术是分支预测缓冲技术(Branch Prediction Buffer, BPB, 或 Branch History Table), 它仅仅使用一片缓冲区记录最近一次或几次的分支历史。分支预测缓冲用分支指令地址的低位来索引, 缓冲的存储区为 1 位的分支历史记录位, 称为预测位, 记录该指令最近一次分支是否成功(假定 1 为成功, 0 为不成功), 缓冲区没有其他任何标志位。它的状态转换图如图 4.9 所示。

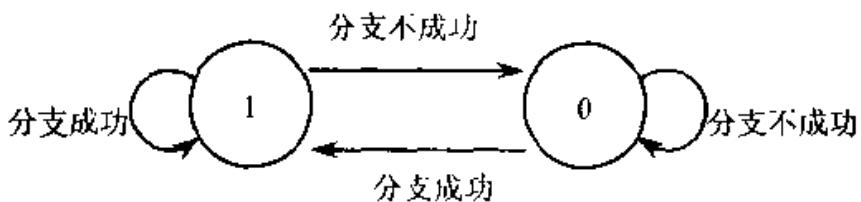


图 4.9 只有一个预测位的分支预测缓冲状态转换图

分支预测缓冲技术包括两个方面, 其一为分支预测, 其二为预测位修改。分支预测与预测位当前的状态有关: 如果当前缓冲记录的预测位为 1, 则预测下一次分支为成功; 当预测位为 0 时, 预测下一次分支不成功。预测位的修改与分支指令的执行结果有关: 如果当前分支成功, 则预测位置 1; 如果当前分支不成功, 则预测位清 0。

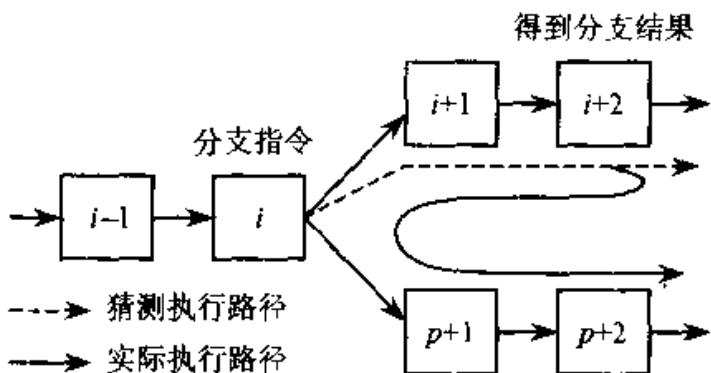


图 4.10 分支预测执行不成功和重新执行过程

分支预测缓冲技术只有在转移延迟大于修改 PC 的时间的情况下, 才能减小分支成功时的时间开销, 使得程序计数器(PC)修改成分支成功目标地址的时间达到一般程序计数器修改所需的时间(即 $PC + 1$ 的时间)。实际上, 预测并不知道分支是否成功, 而且记录的标志有可能被别的低位地址相同的分支指令更改, 但要保证程序执行不会出错。因为预测仅仅预示着分支可能成功, 从而从预测的方向开始取指令, 现场将被保留; 一旦后来证明预测错误, 预测位就被修改, 并且恢复现场, 程序从分支指令处重新执行, 如图 4.10 所示。显然, 如果每次预测都正确, 这种缓冲区是非常有效的。缓冲区的性能与缓冲区中缓存的分支转移指令执行的频度和匹配后预测的准确性有关。这种简单的一位预测机制有一个缺点: 对于循环程序而言, 只要预测出错, 就会连续出错两次而不是一次。

例 4.6 一个循环共循环 10 次, 它将分支成功 9 次, 1 次不成功。假设此分支的预测位始终在缓冲区中。那么分支预测的准确性是多少?

解 这种固定的预测将会在第一次和最后一次循环中出现预测错误。第一次预测错误是源于上次程序的执行, 因为上一次程序最后一次分支是不成功的。最后一次预测错误是不可避免的, 因为前面的分支总是成功的, 共 9 次。因此, 尽管分支成功的比例率是 90%, 但分支预测的准确性为 80% (2 次不正确, 8 次正确)。通常这种由循环形成的分支都是进行多次成功, 最后一次不成功, 因而一位的分支预测将错 2 次。

对于这种比较规范的分支转移, 理想情况下预测的准确率与分支转移成功比例几乎相同。

为了解决这种预测错误, 可以采用两个预测位的预测机制。在两个预测位的分支预测中, 更改对分支的预测必须有两次连续预测错误。图 4.11 给出两位分支预测的状态转换图。

两位分支预测机制是 N 位分支预测的一个特例。 N 位分支预测缓冲采用 N 位计数器, 则计数器的值在 0 到 $2^N - 1$ 之间: 当计数器的值大于或等于最大值的一半 (2^{N-1}) 时, 则预测下一次分支成功; 否则预测下一次分支不成功。预测位的修改和两位预测时相同: 当分支成功时计数器的值加 1, 不成功时减 1。研究表明, 实际应用中 N 位分支预测的性能和两位分支预测差不多, 因而大多数处理器都只采用两位分支预测。

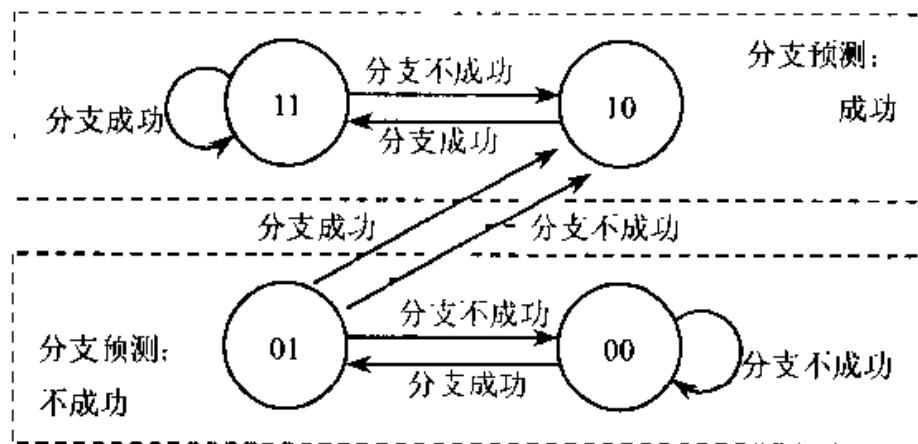


图 4.11 具有两个分支预测位的分支预测缓冲状态转换机制

对大多数流水线, 分支判定依赖的结果往往需要到分支转移指令执行阶段 (EX) 的后期才产生, 现在将分支指令执行阶段进行的分支判定提前到了指令译码阶段 (ID), 只是这时进行的不是分支判定, 而是分支预测。但是, 在改进的 DLX 流水线中, 分支指令在指令译码 (ID) 阶段将寄存器与零比较, 而此时分支目标地址

也已经计算出来了,所以只要分支指令所使用的条件寄存器不发生阻塞,分支预测与计算出分支目标地址几乎是同时完成的,所以这种机制对于改进的 DLX 简单的流水线来讲并没什么帮助。后面我们将讨论一种有助于改进后的 DLX 流水线的机制,现在来看一下通常情况下分支预测的工作过程。

对于真实的应用程序,两位分支预测的准确率可达到多少呢?根据对 IBM Power 体系结构的研究,在使用 4 096 个记录项的预测缓冲区的情况下,SPEC89 标准测试程序中的分支预测准确率可达到 82%~99%,即错误率为 1%~18% 并且通过各种结果分析,4 K 的缓冲区是恰当的,它的性能与无穷大的缓冲区性能几乎相同,而减少缓冲区,性能将下降。

4.3.2 进一步减少分支延迟: 分支目标缓冲

为了进一步减少 DLX 流水线的分支延迟,就需要在新 PC 形成之前,即取指令阶段(IF)后期,知道在什么地址取下一条指令。这就意味着必须知道还没译码的指令是否是分支指令,如果是,还要知道可能的分支目标指令的地址是什么。如果下一条指令是分支指令而且已知它的目的地址,则分支的开销可以降为零。如何实现这个目标呢?我们的做法是:将分支成功的分支指令的地址和它的分支目标地址都放到一个缓冲区中保存起来,缓冲区以分支指令的地址作为标示;取指令阶段,所有指令地址都与保存的标示作比较,一旦相同,就认为本指令是分支指令,且认为它转移成功,并且它的分支目标(下一条指令)地址就是保存在缓冲区中的分支目标地址。这个缓冲区就是分支目标缓冲区(Branch Target Buffer,BTB,或 Branch Target Cache)。图 4.12 是它的结构,图 4.13 是它的处理过程。

工作过程中,当前指令的地址与第一栏中的地址标示集合相比较,缓冲区地址标示中所有的地址为分支转移成功的指令地址。如果指令的 PC 值与某一项匹配,则认为当前指令是成功的分支指令;第二栏,即分支目标 PC 的值,将作为下一指令的地址送往 PC 寄存器。第五章中将详细讨论 Cache,到时候会发现实现分支目标缓冲区的硬件与 Cache 的硬件基本相同。

下面是分支目标缓冲的工作流程以及执行的各个阶段在流水线中的分配。如果当前指令的地址与缓冲区中的标示匹配,那么此指令必为分支转移成功指令,且下一条指令的 PC 值在分支目标缓冲的分支目标 PC 域中,因此在本指令指令译码阶段(ID)阶段开始从预测指令的 PC 处开始取下一条指令。如果分支指令地址没有匹配而指令发生了转移,则分支目标地址会在指令译码阶段(ID)阶段末知道,并将其本身的地址和目的地址加入缓冲区中。如果在分支目标缓冲中找到了当前指令地址,而指令当前分支不成功,则将此项从分支目标缓冲中删去。可以看出,如果是分支指令,并且预测正确则不会有任何延迟;如果预测

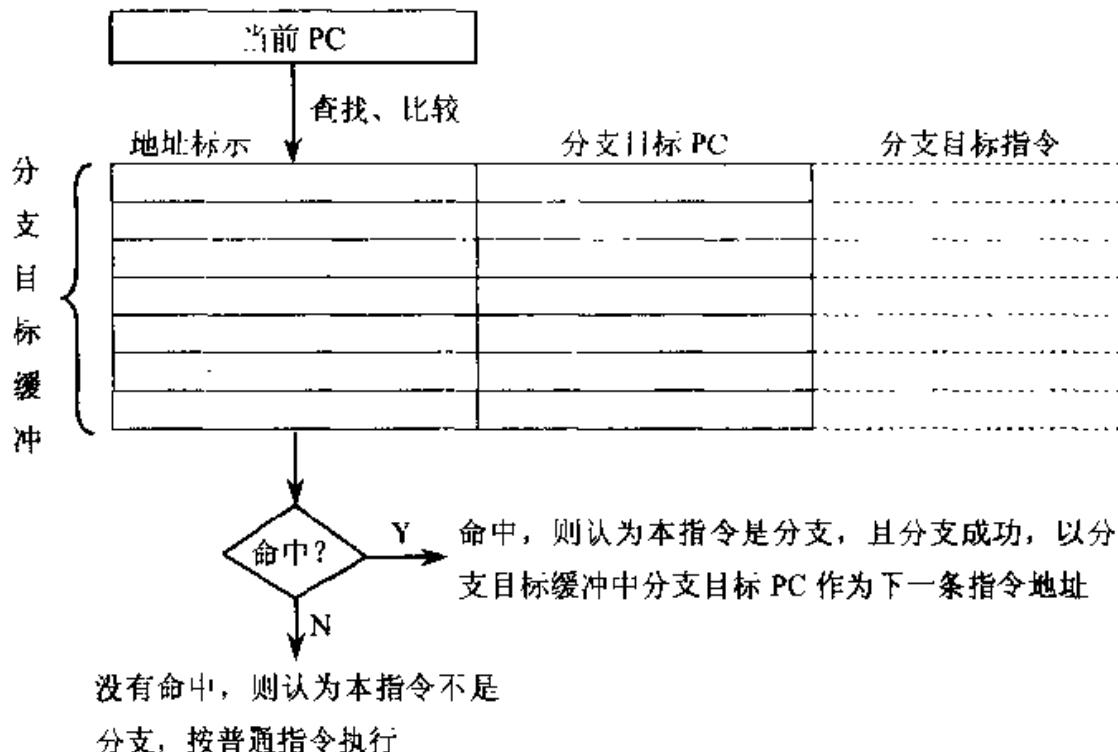


图 4.12 分支目标缓冲的结构和工作过程

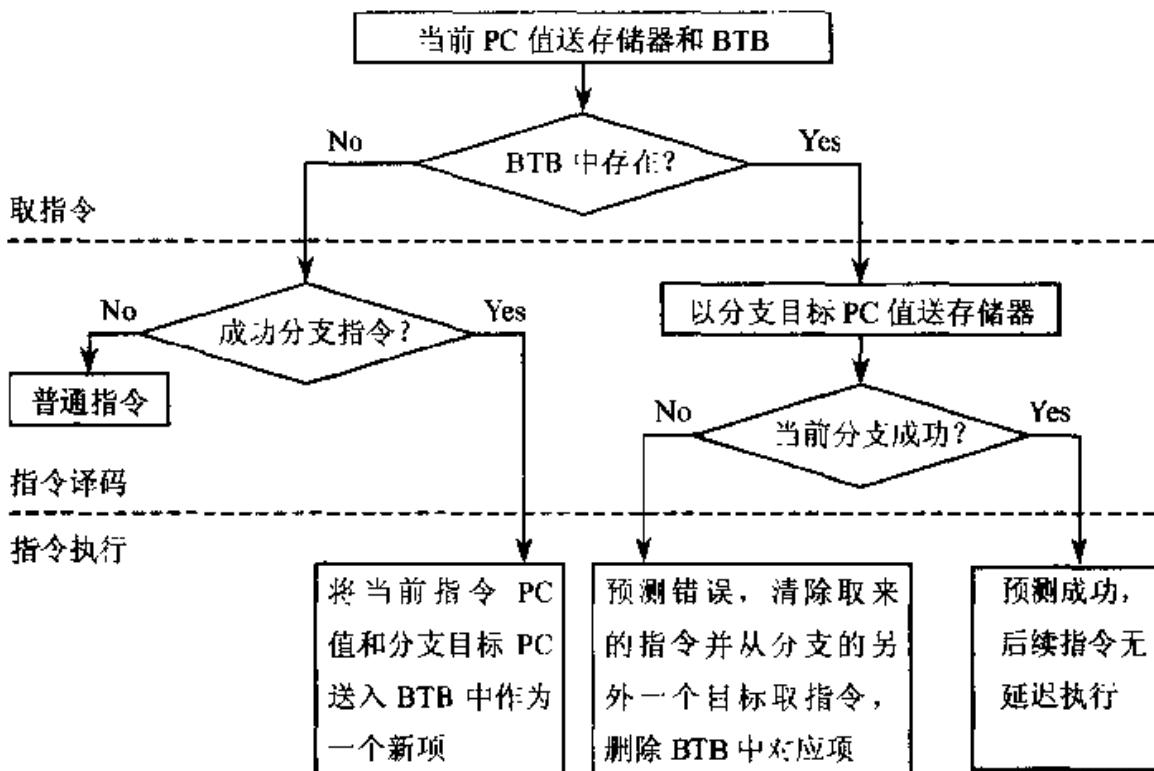


图 4.13 分支目标缓冲处理的步骤

错误，则会耗费一个时钟周期来取错误的指令，并在一个时钟周期后重新取正确

指令。如果转移指令不在缓冲区中,则将其当成是不成功的分支指令处理,耗费延迟的大小取决于指令是否转移成功。实际实现中,不命中或者预测错误时的延迟会更大,因为分支目标缓冲必须更新。解决预测错误或不命中时产生的延迟是一个具有挑战性的问题,因为通常情况下,重写缓冲区时将停止流水线。

在评价分支目的缓冲工作状况之前,我们必须定好各种可能情况下的延迟,参见表 4.3。

例 4.7 按表 4.3 计算分支转移总的延迟,根据下面的假设,计算分支目标缓冲的性能。

- (1) 预测准确率 90%
- (2) 缓冲区命中率 90%
- (3) 假设分支转移成功的比例为 60%

表 4.3 采用 BTB 技术时指令在各种情况下的延迟

指令在 BTB 中?	预测结果	实际的动作	延迟周期
是	成功	成功	0
是	成功	不成功	2
不是		成功	2

解 根据以上分类,性能计算包括 3 个部分:

- (1) 在 BTB 中,预测成功,实际成功;
- (2) 在 BTB 中,预测成功,实际不成功;
- (3) 不在 BTB 中,实际成功。

因为在 BTB 中,预测成功,实际成功时的延迟为 0,所以

$$\begin{aligned}
 \text{分支延迟} &= \text{BTB 命中率} \times \text{预测错误率} \times 2 + (1 - \text{BTB 命中率}) \\
 &\quad \times \text{分支转移成功率} \times 2 \\
 &= (90\% \times 10\% \times 2) + (10\% \times 60\% \times 2) \\
 &= 0.18 + 0.12 \\
 &= 0.30 \text{ 时钟周期}
 \end{aligned}$$

可以与标准 DLX 流水线中分支延迟的性能进行对比,它是每个分支转移约为 0.5 个时钟周期。请注意,随着分支延迟的增大,动态分支预测的性能将会有更大的提高;而且,好的分支预测机制将会提高更多的性能。

对分支预测机制的一种改进是在缓冲区中不仅存入目的地址,而且还有入一个或多个目标指令,参见图 4.12 中的虚线部分。这种改动有两种潜在的好

处：第一，在连续取指令之前，可以较长时间地访问缓冲区，这时的分支目的缓冲区较大；第二，将目的指令进行缓冲时可以进行一种称为分支目的指令缓冲（branch folding）的优化。分支目的指令缓冲可使无条件分支的延迟达到零，甚至有的条件分支也可达到零延迟。

另外已经研究并在最新的处理器中采用的一种技术是预测非直接分支，即分支的目的地址随着运行时间而改变。高级程序语言中会有这种跳转，如间接过程调用、CASE 语句、FORTRAN 中的 GOTO 语句等，其中大部分是过程调用中的 RETURN 语句引起的间接跳转。例如，SPEC 标准测试程序中平均有 85% 的间接跳转是 RETURN 语句。这些问题这里不作进一步研究。

4.3.3 基于硬件的推断执行

为了得到更高的并行性，设计者研究出一种称为推断（Speculation）的方法，它允许在处理器还未判断指令是否能执行之前就提前执行（即克服了控制相关）。这种推断执行过程带有明显的投机性质，如果推断正确，它可以消除所有附加延迟；所以在大多数推断正确的前提下，推断就可以有效地加快分支处理的速度。

基于硬件的推断执行结合了三种思想：动态的分支预测来决定执行哪条语句；在控制相关消除之前指令推断执行；对基本块采用动态调度。基于硬件的推断是根据动态的数据相关性来选择指令的执行时间。这种程序运行的方法实质上是数据流运行（data flow execution）：只要操作数有效，指令就执行。

将 Tomasulo 算法的实现硬件加以扩展就可支持指令推断执行。要推断执行，就要将推断执行的结果供给其他指令使用，同时又要明确这些结果不是实际完成的结果。这些推断执行产生的结果直到指令处于非推断执行状态时才能确定为最终结果，这时才允许写到寄存器或存储器中去。我们将这一步加在指令执行阶段以后，称为指令确认（instruction commit）。

实现推断的关键思想是允许指令乱序执行但顺序确认，只有确认以后的结果才是最终的结果，从而避免不可恢复的行为（如更新状态或发生异常）。在简单的单流出 DLX 流水线中，将写结果阶段放在流水线的最后，保证在顺序地检测完指令可能引发的异常情况之后再确认指令结果。加入推断后，需要将指令的执行和指令的确认区分开，因为指令在确认之前可能早就执行完了。所以，加入指令确认阶段需要一套额外的硬件缓冲，来保存那些执行完毕但未经过确认的指令及其结果。这种硬件的缓冲称为再定序（reorder）缓冲区，同时还用它来在推断执行的指令之间传送结果。

再定序缓冲和 Tomasulo 算法中的保留站一样，提供了额外的虚拟寄存器，扩充了寄存器的容量。再定序缓冲保存指令执行完毕到指令得到确认之间的所

有指令及其结果，所以再定序缓冲像 Tomasulo 算法中的保留站一样是后续指令操作数的来源之一。主要的不同点是在 Tomasulo 算法中，一旦指令结果写到目的寄存器，下面的指令就会从寄存器文件中得到数据。而对于推断执行，直到指令确认后（即明确地知道指令应该执行）才更新寄存器文件，因而必须由再定序缓冲提供在指令执行完毕和确认之间的结果，作为其他指令的操作数。再定序缓冲和 Tomasulo 算法中的存操作缓冲不一样，为了简单起见我们可以将存（Store）缓冲的功能集成到再定序缓冲。因为结果在写入寄存器之前由再定序缓冲保存，所以再定序缓冲区还可以取代取（Load）缓冲区。

再定序缓冲的每个项包含三个域：指令类型、目的地址和值域。指令类型是否是分支（尚无结果）、存操作（目的地址为存储器）或寄存器操作（ALU 操作或目的地址是寄存器的取操作）。目的地址域给出结果应写入的目的寄存器号（对于取操作和 ALU 指令）或存储器的地址（存操作）。值域用来保存结果直到指令得到确认。后面可以看到再定序缓冲区项的结构示意。

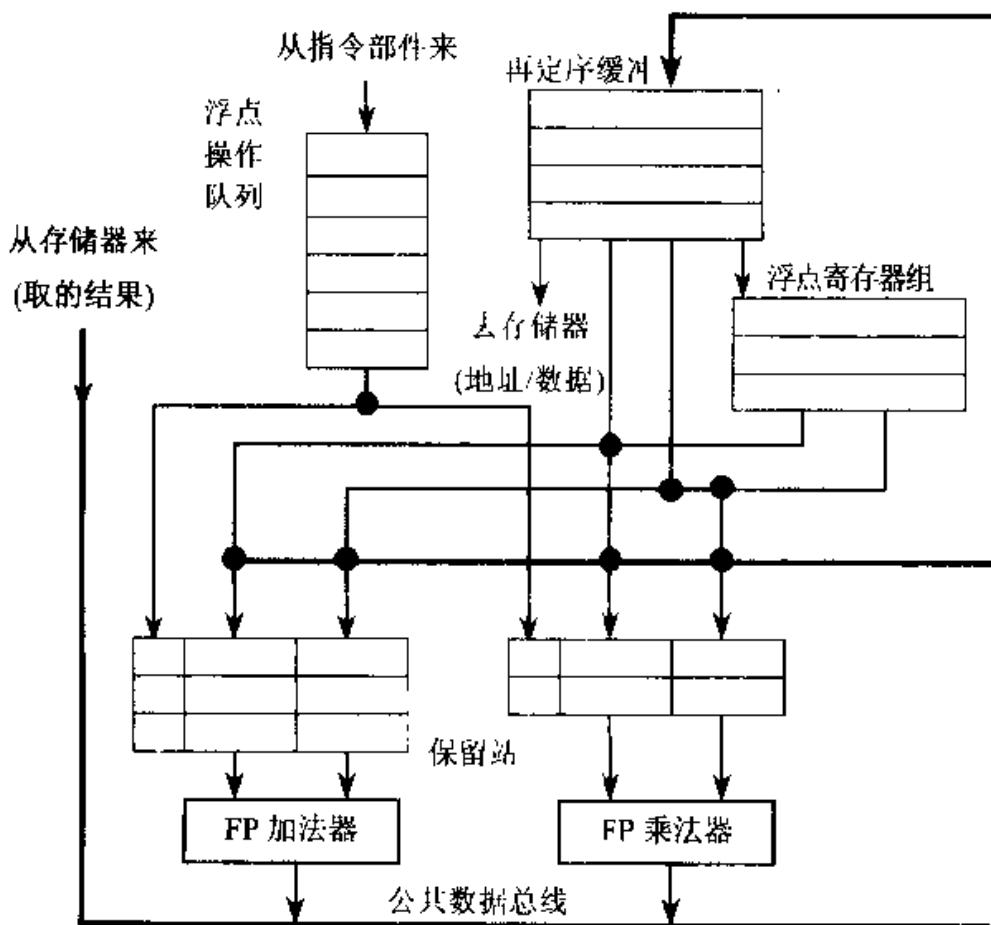


图 4.14 采用 Tomasulo 算法并支持推断执行的 DLX 浮点部件的结构

图 4.14 给出使用再定序缓冲的处理器的硬件结构。再定序缓冲区彻底代替了存储器取和存缓冲区。尽管保留站的重命名功能被再定序缓冲代替了，但

是在指令流出和指令开始执行之间仍需要有地方保存指令代码,这个功能仍由保留站提供。因为每条指令在指令确认之前在再定序缓冲中都占有的一项,所以我们用再定序缓冲项的编号而非保留站号来标志结果(即重命名的寄存器号)。保留站登记的是相应的分配给该指令的再定序缓冲的编号。使用再定序技术,DLX 浮点指令的执行包含以下四步:

(1) 指令流出(Issue):从浮点指令队列中取一条指令。如果有空的保留站和空的再定序缓冲就进行取指令。如果操作数在寄存器或在再定序缓冲中,则将它送入保留站,并更新再定序缓冲的控制域,表示它的结果正在被使用。分配保存此结果的再定序缓冲的编号也要送入保留站,从而当这个结果放到 CDB 上时用它来标示。如果保留站或再定序缓冲全满,则停止流出指令,直到两者均有空项。

(2) 执行(Execution):如果有~一个或多个操作数无效,就等待并不断检测 CDB。这一步检测先写后读相关。当保留站中的两个操作数全有效后就可以执行这个操作。一些动态调度的处理器称这步为流出(Issue),但我们这里使用的是 CDC 6600 的术语。

(3) 写结果(Write Result):结果有效后将其写到 CDB 上(附带本指令流出时分配的再定序缓冲项号),然后从 CDB 写到再定序缓冲以及等待此结果的保留站。保留站也可以从再定序缓冲直接读到结果而不需要到 CDB,就像记分牌直接从寄存器读结果而不是进行总线竞争。然后释放保留站。

(4) 确认(Commit):当一条指令不是预测错误的分支转移指令,到达再定序缓冲区的出口且结果有效时,将结果回写到目的寄存器(如果是存操作,则将结果写存储器),指令的推断执行过程结束,然后将指令从再定序缓冲中清除。当预测错误的分支指令到达再定序缓冲区的出口时,将指出推断执行错误;清除再定序缓冲并从分支的正确人口重新开始执行。如果分支预测正确则此分支执行完毕。一些机器称这个过程为完成(Completion 或 Graduation)。

一旦指令得到确认,它在再定序缓冲中占用的项就更新为空,目的寄存器和存储器就可以更新。为了避免再定序缓冲大量使用存储空间,再定序缓冲区一般采用环形队列机制。当再定序缓冲区满时,就简单地停止指令的流出,直到有空项。下面看一下前面在 Tomasulo 算法中如何实现再定序缓冲机制。

例 4.8 假设浮点功能单元的延迟为加法是 2 个时钟周期,乘法是 10 个时钟周期,除法 40 个时钟周期。给出下面的代码段当指令 MULTD 要确认时的状态。

LD	F6,34(R2)
LD	F2,45(R3)
MULTD	F0,F2,F4
SUBD	F8,F6,F2

DIVD F10,F0,F6
ADDI F6,F8,F2

解 状态如图 4.15 中的三个表所示。要注意的是指令 SUBD 尽管已经执行完毕，但直到 MULTD 得到确认后才能得到确认。另外 Qi 和 Qk 域以及寄存器状态域全由再定序缓冲的编号所代替，目的寄存器域给出产生结果的再定序缓冲号。 $\#x$ 表示再定序缓冲 x 项值域中的值。

这个例子充分反映了采用推断执行的处理器和采用动态调度的处理器之间的区别。将图 4.15 和基本 Tomasulo 算法的状态图(图 4.7)比较一下，上面例子中的主要不同是指令是顺序执行完毕的。相反，图 4.7 中指令 SUBD 和 ADDD 指令是乱序完成的。

名字	保留站状态						
	忙	操作	Vj	Vk	Qj	Qk	目的
ADD1	no						
ADD2	no						
ADD3	no						
Mult1	no	MULTD	Mem[45+Regs[R2]]	Regs[F4]			#3
Mult2	yes	DIVD		Mem[34+Regs[R2]]	#3		#5

项号	再定序缓冲区					
	忙	指令		状态	目的	值
1	no	LD	F6,34(R2)	确认	F6	Mem[34+Regs[R2]]
2	no	LD	F2,45(R3)	确认	F2	Mem[45+Regs[R3]]
3	yes	MULTD	F0,F2,F4	写结果	F0	#2×Regs[F4]
4	yes	SUBD	F8,F6,F2	写结果	F8	#1-#2
5	yes	DIVD	F10,F0,F6	执行	F10	
6	yes	ADDD	F6,F8,F2	写结果	F6	#4+#2

域	浮点寄存器状态							
	F0	F2	F4	F6	F8	F10	...	F30
再定序号	3			6	4	5		
忙	yes	no	no	yes	yes	yes	...	no

图 4.15 推断执行，MULTD 确认前的保留站和再定序缓冲的状态

通过再定序缓冲，可以在进行精确异常处理的同时进行动态指令调度。在上例中，如果指令 MULTD 引起异常，可简单地等到它到达再定序缓冲的头，然后处理异常，而将其他等待确认的指令清除。因为指令的确认是顺序的，从而可以精确处理异常。相反，在前面使用 Tomasulo 算法的例子中，SUBD 和 ADDD 指令在 MULTD 产生异常之前早已完成。F8 和 F6(SUBD 和 ADDD 指令的目的寄存器)中的结果已经写回了，从而异常情况处理是不精确的。一些用户和体系结构的设计者认为在高性能的处理器中不精确的异常是可以接受的，因为程序很可能终止，这个问题这里不再深入讨论。其他异常情况如页面故障，如果是不精确异常，就非常

难处理,因为它要求异常处理后,透明地继续执行原来的程序。

尽管我们这里使用这种推断执行的方法是针对浮点的,它可以很容易的推广到整数寄存器和功能单元。实际上,推断执行对于整数程序更有效,因为这些程序的代码中分支预测成功的几率较小。基于硬件的推断和动态调度相结合,可以做到同一种体系结构但实现不同的机器能够使用相同的编译器。尽管这种优点是最难加以量化的,但从长远来看可能是最重要的,这也是 IBM 360/91 的出发点之一。

推断技术存在的一个主要缺点是:支持推断的硬件太复杂,需要大量的硬件资源。

这里研究的这些克服控制相关的方法在商业化处理器中已得到广泛采用(PowerPC 620、MIPS R10000、Intel P6、PII、AMD K5、K6 等),它们都是在 Tomasulo 算法的基础上将推断执行和动态调度相结合,使处理器达到很高的性能。

4.4 多指令流出技术

前两节介绍的技术主要是通过减少数据相关和控制相关,来达到 CPI 为 1 的理想情况。为获得更高的性能,我们希望能够使 CPI 低于 1。我们不能使 CPI 小于 1 的最直接的原因是每次只能流出一条指令,如果每个时钟周期只流出一条指令,CPI 就不可能低于 1,所以多指令流出处理器的目标就是在一个时钟周期内流出多条指令。多指令流出处理器有三种:超标量(Superscalar)、超流水(Super Pipeline)和超长指令字 VLIW(Very Long Instruction Word,简记为 VLIW)。超标量每个时钟周期流出的指令数不定,它既可以通过编译器静态调度,也可以通过记分牌或 Tomasulo 算法等动态调度,本节中我们分别看一下简单的静态调度和动态调度的超标量处理器。超长指令字与之不同,每个时钟周期流出的指令数是固定的,它们构成一条长指令,或说是一个混合指令包。超长指令字的处理器目前只能通过编译静态调度。超流水就是将每个功能部件进一步流水化,使得一个功能部件在一拍中可以处理多条指令。因为超流水可以简单地理解为一种很长的流水线,所以我们这里不作讨论。为了有比较地阐述本节中所讲的技术,我们仍采用前面假设的流水线延迟,并且采用相同的代码,即:将一标量和数组相加:

LOOP:	LD	F0,0(R1)	;F0 = 数组元素
	ADDD	F4,F0,F2	;加上在 F2 中的标量
	SD	0(R1),F4	;存结果
	SUBI	R1,R1,#8	;将指针减少 8(每个 DW)
	BNEZ	R1,LOOP	;R1 不等于 0, 转移

我们先来看一个简单的超标量处理器。

4.4.1 超标量技术

目前，在典型的超标量处理器中，每个时钟周期可流出 1 到 8 条指令。通常情况下，这些指令必须不相关且满足某些限制条件，如每个时钟周期访存不能多于 1 次等，不同处理器的限制是不同的。如果指令流中的指令相关或不满足限制条件，则只能流出这条指令前面的指令，因此超标量处理器流出的指令数是不定的。与之不同，在超长指令字处理器中，编译器负责产生可同时流出的指令包，硬件不负责动态处理。因此我们认为超标量处理器具有动态多流出能力，而超长指令字是静态的多流出。超标量处理器的指令序列可以采用静态调度或动态调度，现在我们先假设是静态调度。

DLX 处理器是怎样超标量的呢？我们假设每个时钟周期流出两条指令。一条指令可以是取（Load）指令、存（Store）指令、分支指令或整数运算操作，另一条指令可以是任意的浮点操作。将整数指令和浮点指令相结合比随意地双流出要简单，要求也低，这种配置和 HP 7100 结构相似。

每个时钟周期流出两条指令就意味着取指令和解码部件都是 64 位。为了使解码简单，编译结果要求指令成对且与 64 位边界对齐，整数部分在前面。也可以先检查指令，在送往整数或浮点数据通道的时候进行调整，但是这样在增加灵活性的同时也增加了检测机制的复杂性。无论哪种情况，都要求只有第一条指令流出后才可以流出第二条指令，这是由硬件动态决定的。如果第二条指令不满足条件就只流出第一条指令。图 4.16 给出两路超标量指令流的示意图。图中没有具体解释浮点指令在执行（EX）阶段执行时间的问题，因为 DLX 在这方面超标量与普通流水线没什么区别，前面的概念可直接运用。

指 令	流水线工作情况							
整数指令	IF	ID	EX	MEM	WB			
浮点指令	IF	ID	EX	MEM	WB			
整数指令		IF	ID	EX	MEM	WB		
浮点指令		IF	ID	EX	MEM	WB		
整数指令			IF	ID	EX	MEM	WB	
浮点指令			IF	ID	EX	MEM	WB	
整数指令				IF	ID	EX	MEM	WB
浮点指令				IF	ID	EX	MEM	WB

图 4.16 两路超标量指令执行示意图

通过对指令流出部件采用流水技术，我们可以极大地提高指令流出的速率，

但必须采用流水功能部件或多个独立的功能部件,否则数据通道会很快成为瓶颈,限制多流出性能的发挥。

并行流出一条整数指令和一条浮点指令,除了一般的冲突监测机制,只需增加很少的硬件,因为在 Load-Store 的体系结构下,整数操作和浮点操作使用不同的寄存器组和不同的功能部件。但是对于浮点访存指令,它将使用整数部件,从而导致结构冲突。由于同时只能流出两条指令,因此监测是否存在结构相关只需检查两条指令的操作码。

另外,一旦整数部分指令是浮点数据访存,这会导致浮点寄存器端口的访问竞争;如果流出的指令对中浮点指令和整数指令相关,也会产生新的数据相关。浮点寄存器端口的问题,可通过限制浮点存取(访存)指令单独执行来解决。这种方法也可以解决浮点数据存取和浮点操作指令同时流出引起的结构相关;它的实现是比较容易的,但从性能上将是一大缺陷。另外一种解决方法是给浮点寄存器设置两个端口:一个读端口,一个写端口。

当指令对中包含浮点取指令,且后面的浮点指令与之相关时,硬件必须能够检测出来,从而限制后面浮点指令的流出。除了这种情况,其他可能存在的相关的检测和单流出的流水线是相同的。另外,还需要添加一些额外的相关数据通道来避免不必要的流水线空转。

另外还存在一个限制超标量流水线的性能发挥的障碍。在简单的流水线中,取操作指令有一个时钟周期的延迟,这就要求使用本结果的指令在一个时钟周期后才可以启动流出。在超标量流水线中,取操作指令的结果不能在本周期或下一个周期使用,所以下面可能有三条相关的指令不能使用其结果。分支延迟也变为三条指令,因为分支指令肯定是指令对的第一条指令。为了能有效地利用超标量处理器的可获得的并行度,需要采用更有效的编译技术及硬件调度技术以及更复杂的指令译码技术。

下面来看一下在表 4.2 中列出的延迟的条件下,超标量处理器如何进行循环展开和指令调度。

例 4.9 下面是前面我们使用的循环程序段,在超标量 DLX 流水线上将如何调度?

LOOP:	LD	F0,0(R1)	;F0 = 数组元素
	ADDD	F4,F0,F2	;加上在 F2 中的标量
	SD	0(R1),F4	;存结果
	SUBI	R1,R1,±8	;将指针减少 8(每个 DW)
	BNEZ	R1,LOOP	;R1 不等于 0, 转移

解 要达到无延迟的指令调度,须将循环展开 5 次。经过展开,循环包括 5 个 LD、ADDD、SD 指令和 1 个 SUBI、1 个 BNEZ 指令。展开并经过调度的指令

序列如图 4.17 所示。

超标量流水线上展开的代码每次循环需 12 个时钟周期,即每个迭代是 2.4 个时钟周期。而在普通的 DLX 流水线上,没有调度的迭代 1 次为 9 个时钟周期,性能提高了 3.75 倍;调度后为 6 个时钟周期,性能提高了 2.5 倍;展开 4 次并调度后每个迭代为 3.5 个时钟周期,性能提高了 1.4 倍。在这个例子中,超标量 DLX 流水线的性能受限之处是整数计算和浮点计算之间的平衡问题。每条浮点指令在整数指令后面流出,但是本例中没有足够的浮点指令来使流水线达到饱和。

总之,理想情况下,超标量处理器将取出两条指令,第一条是整数指令,第二条是浮点指令,并同时将它们流出。如果指令对存在相关,它们就将顺序流出。

超标量处理器与超长指令字处理器相比有两个优点:

第一,超标量结构对代码是透明的,因为处理器能自己检测下一条指令能否流出,从而不需要排列指令来满足指令流出。

	整数指令	浮点指令	时钟周期
LOOP:	LD F0(R1)		1
	LD F6, -8(R1)		2
	LD F10, -16(R1)	ADDD F4,F0,F2	3
	LD F14, -24(R1)	ADDD F8,F6,F2	4
	LD F18, -32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

图 4.17 在 DLX 超标量流水线上展开并调度后的代码

第二,即使是未经过调度的代码或是旧的编译器编译过的代码也可以运行,当然运行的效果不会很好。解决这个问题的方法之一是使用动态调度技术。

4.4.2 多指令流出的动态调度

多条指令流出技术,可以用记分牌技术或 Tomasulo 算法进行动态调度处理。下面将扩展 Tomasulo 算法,支持每个时钟周期流出两条指令,一条是整数指令,另一条是浮点指令。要求指令是按顺序流向保留站,否则信息记录机制会太复杂。另外,将整数数据寄存器和浮点寄存器分开,只要不使用相同的寄存器

就可以同时将一条整数指令和一条浮点指令送到它们的保留站中去。

这种方法需要限制相关指令的并行执行,比如整数指令是浮点取操作,浮点指令是浮点加,就不能在同一时钟周期中执行这两条指令,但是可以将它们同时流出到各自保留站中,以后再顺序执行。如果硬件调度机制不能在同一时钟周期流出相关的两条指令,那么与仅仅由编译完成指令静态调度的机器相比,动态调度的优势就不大了。

有两种方式可进行双指令流出。第一种是将指令流出阶段进一步流水化,使指令流出的速度是基本机器周期的两倍。这就可以在后面指令流出前更改寄存器表,于是两条指令可同时执行。第二种方法是对流出的指令对进行限制,只有浮点的取操作指令或是从整数寄存器将数据送入浮点寄存器的传送操作,才会产生相关而导致两条指令不能同时流出。如果对流出的指令对限制减少,指令对的复杂度增加,则可能出现的相关情况会更多。

使用结果队列可以减少存储器取操作或数据传送操作对保留站的需求量。使用队列还可以使等待操作数的存操作指令提早流出,Tomasulo 算法就是这样处理的。动态调度对数据传送是最有效的,而静态调度对寄存器-寄存器操作的代码序列最有效,所以,对于存储器取操作和数据传送操作,完全可以通过静态调度和队列来消除保留站。现在有多种机器采用这种方法或对它进行了改进。

为了简单起见,假设将指令流出逻辑机制流水化,从而可以同时流出两条相关但使用不同功能部件的指令。下面看一下以前用过的代码是如何工作的。

例 4.10 下面的代码运行于采用 Tomasulo 算法并且能够同时流出两条指令的 DLX 流水线。假设每个时钟周期能流出一条整数指令和一条浮点指令,如果两条指令相关,则先流出整数指令。各指令的执行延迟时间和前者的相同。假设指令流出和结果回写各占用一个时钟周期,并有动态的分支预测硬件机制。列表表示出循环前面两次迭代各个指令的流出、开始执行和结果回写的时间顺序。源代码为

LOOP:	LD	F0,0(R1)	; F0 = 数组元素
	ADDD	F4,F0,F2	; 加上在 F2 中的标量
	SD	0(R1),F4	; 存结果
	SUBI	R1,R1,±8	; 将指针减少 8(每个 DW)
	BNEZ	R1,LOOP	; R1 不等于 0, 转移

解 循环进行动态展开,并且如果可能指令双流出。运行结果如图 4.18。每次循环运行时间为 $8/2=4$ 个时钟周期。

存储器存操作指令和分支指令没有结果回写阶段,因为它们不进行寄存器写。另外此例中假设 CDB 总线较宽,因为第 8 个时钟周期需要多个回写结果的端口。

指令双流出的数目很小,因为每次只有一条浮点指令流出。要提高双流出的相对比例,就需要编译器来将循环展开,通过消除循环头尾的开销来减少指令的数目。通过这种转换,循环的运行性能会接近达到超标量处理器的性能。此外,如果处理器“更宽”,即在一个时钟周期能流出多条整数指令,性能还能得到进一步提高。

迭代层数	指 令	流出的时间	开始执行时间	写结果时间
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2	1	5	8
1	SD 0(R1), F4	2	9	
1	SUBI R1,R1,#8	3	4	5
1	BNEZ R1,LOOP	4	5	
2	LD F0,0(R1)	5	6	8
2	ADDD F4,F0,F2	5	9	12
2	SD 0(R1), F4	6	13	
2	SUBI R1,R1,#8	7	8	9
2	BNEZ R1,LOOP	8	9	

图 4.18 采用 Tomasulo 算法的指令双发是指令流出、执行和结果回写的时间

4.4.3 超长指令字技术

因为格式固定,处理过程简单,所以采用超长指令字的处理器所需硬件量比超标量要少。例如在双流出的超标量流水线中,需要检查两条指令的操作数,最多达 6 个寄存器,然后才能动态地决定是流出一条指令还是两条指令,并将其送到相应功能部件上。同时流出两条指令,所需的硬件还不算太多,并且也可以扩展到多条指令,但是如果不管指令原来的顺序而同时将它们流出,那么它们之间的各种相关关系完全要动态确定,其复杂度将大大增加。

所以另外一种方法是采用长指令字 LIW(Long Instruction Word)或超长指令字 VLIW(Very Long Instruction Word)的体系结构。超长指令字采用多个独立的功能部件,但它并不是将多条指令流出到各个功能单元,而是将多条指令的操作打包,形成一条非常长的指令,超长指令字由此得名。选择同时可流出的多条指令的任务由编译器完成,而在超标量机器中此功能是由硬件完成的,所以超长指令字机器可以节省大量硬件。指令同时可流出的最大数目越大,超长指令字的性能优势就越显著。我们就讨论流出通道较宽的超长指令字处理器。

如果超长指令字的指令包括两个整数操作、两个浮点操作、两个访存操作和一个分支操作,每个操作可能占用 16~24 位,从而指令长度达到 112~168 位。指令中每一个操作字段称为操作槽。超长指令字处理器中的功能部件数量和指令中包含的操作数量是对应的,为了充分利用这些功能部件,要求代码序列必须有较大的并行度来填充有效的操作槽。这个工作由编译器通过循环展开等指令

调度技术完成。现在我们先假设有一种技术可以产生满足超长指令字要求的代码段,这个代码段用来构建超长指令字,下面看一下处理器的操作过程。

例 4.11 假设超长指令字每个时钟周期可同时流出两条访存指令、两条浮点指令和一条整数指令或分支指令。给出在此处理器上数组元素循环加一个标量的展开后的代码序列。尽可能展开循环以消除流水线的停顿,忽略分支指令的转移槽。

解 代码序列如图 4.19 所示。循环展开 5 次可以消除所有的流水线空转,即没有空指令流出。运行时间为 8 个时钟周期,即每个迭代平均 1.6 个时钟周期。

访存指令 1	访存指令 2	浮点指令 1	浮点指令 2	整数/转移指令
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14,-24(R1)			
LD F18,-32(R1)		ADDD F4,F0,F2	ADDD F8,F6,F2	
		ADDD F12,F10,F2	ADDD F16,F14,F2	
		ADDD F20,F18,F2		
SD 0(R1),F4	SD -8(R1),F8			
SD -16(R1),F12	SD -24(R1),F16			SUBI R1,R1,#40
SD 8(R1),F20				BNEZ R1,LOOP

图 4.19 循环展开后的超长指令字的指令代码序列

8 个时钟周期内流出了 17 条指令,即每个时钟周期 2.1 条。8 个时钟周期共有操作槽 $8 \times 5 = 40$ 个,有效槽的比例为 42.5%。

从上面的例子中可以看出,超长指令字处理器要达到高流出率,就要有比普通 DLX 更多的寄存器。在超长指令字处理器上,上面的指令序列展开 5 次,至少需要 6 个浮点寄存器,而性能并没有达到较优的情况。从图 4.19 中可以看出,展开 10 次可以获得比较理想的性能,这最少需要 11 个寄存器;而相同的指令在普通的 DLX 处理器上运行仅需 2 个浮点寄存器,展开 4 次并进行调度也仅需 5 个。而在前面超标量处理器的例子中,展开 5 次最少需要 6 个寄存器。这也是超长指令字的不足之一。

4.4.4 多流出处理器受到的限制

多流出处理器受哪些限制呢?既然每个时钟周期可以流出 5 条指令,为什么不能流出 50 条呢?指令流出能力主要受以下方面的影响:

- (1) 程序内在的指令级并行性
- (2) 硬件实现的限制
- (3) 超标量和超长指令字处理器固有的技术限制

程序内在指令级并行性的限制是最简单的也是最根本的因素。因为对于流

水线处理器,需要有大量可并行执行的操作才能避免流水线的停顿。如果浮点流水线的延迟为 5 个时钟周期,那么要通过调度使浮点流水线不停顿,就必须有 5 条无关的浮点指令。通常情况下,所需要的无关指令数等于流水线的深度乘以可以同时工作的功能部件数,这就意味着要使具有 5 个功能部件的流水线忙起来,大约需要连续 15~25 条无关指令。

第二种限制是多指令流出的处理器需要大量的硬件资源。这是因为每个时钟周期不仅要流出多条指令还要执行它们。每个时钟周期执行多条指令所需的硬件看似十分明显:将整数部件和浮点部件加倍,硬件开销成正比上升。然而这样所需的存储器带宽和寄存器带宽也大大增加了。比如对于分离的浮点和整数寄存器文件,上述超长指令字处理器整数寄存器文件需要 6 个读端口(每个访存指令需要 2 个,整数指令需要 2 个)、3 个写端口(每个非浮点功能单元 1 个),浮点寄存器需要 6 个读端口(每个访存指令 1 个,每个浮点功能单元 2 个)和 4 个写端口(每个访存部件和浮点部件各 1 个)。这样的带宽要求如果不增加寄存器的硅片面积是不可能做到的,而加大面积就会降低时钟频率。这种 5 个功能单元的超长指令字处理器还需要两个存储器读端口,它比寄存器端口的开销大得多。如果要使流出指令的数目加大,就需要增加更多的存储器端口。这时,只增加算术功能部件是没用的,因为处理器受限于存储器的带宽。随着存储器端口的增加,存储系统的复杂性也大大增加。为了并行访问存储器,可以将存储器分成包含不同地址的多个体,并希望一条单独的指令中的操作没有访问冲突;或者是存储器真正具有双端口,这样开销是相当可观的。在 IBM Power2 的设计中采用了另外一种方法:每个时钟周期访存两次。但是即使存储系统的功能再强大,这种方法对于高速的处理器来讲都太慢了。有关存储系统的技术在后续章节中详细讲解。多端口的层次的存储系统的复杂性和访问延迟,可能是超标量或者超长指令字处理器等多流出处理器面临的最严重的硬件限制。

多指令流出所需的硬件量随实现方法的不同有很大的差别。一个极端是动态调度的超标量处理器,无论是采用记分牌技术还是使用 Tomasulo 算法,都需要大量的硬件,而且动态调度也大大增加了设计的复杂性,要提高时钟频率更加困难,给设计的改进工作也增加了难度。另一极端是超长指令字处理器,指令的流出和调度仅需要很少甚至不需要额外的硬件,因为这些工作全都由编译器进行。这两种极端之间是现存的多数超标量处理器,它们将编译器的静态调度和硬件机制结合起来,共同决定下而可同时并行流出多少条指令。设计多流出处理器的主要难点是:对于访存的开销、硬件机制和编译器技术,如何考虑取舍它们对性能的影响。

最后是超标量处理器和超长指令字所面有的限制。前面我们已经讨论了超标量处理器所存在的问题,即指令流出逻辑机制问题。对于超长指令字,无论是

技术方面还是逻辑机制方面均存在着问题。技术上的问题是执行代码体积增长和操作的锁定关联(Lock-step)的限制。两个因素结合在一起大大增长了超长指令字执行代码的长度：首先是要从串行代码段中获取足够的无关操作，就必须将循环展开很多次，从而增加了代码的长度；第二方面是当指令不满时，没有用到的功能部件的操作槽在编码时用空操作填充。在图 4.19 中，可以看到功能部件仅使用了 42.5%，每条指令一半以上是空的。为了解决这个问题，有时采用智能的编码方式。例如可用一个大的立即数域供任何功能单元使用。另外一种方法是在内存中将代码压缩，代码读到 Cache 时被解压或将代码解码。因为超长指令字是静态调度的，且操作是锁定关联的，所以流水线中任何功能部件的停顿都会导致整个处理器的停顿，因此各个功能单元必须是同步的。尽管可以通过调度使重要的功能单元避免停顿，但是访问数据会导致 Cache 停顿，而且难以调度。因而 Cache 停顿会导致整个流水线的停顿。随着指令流出数的增加，访存的次数也增加，这种操作锁定关联的结构就很难有效的利用数据 Cache，从而增加了存储器的复杂性和延迟。

二进制代码的兼容性是超长指令字所面临的最主要的逻辑问题。这种问题存在于每一代处理器之间，即使处理器所实现的基本指令是相同的。问题的根源是不同处理器的指令流出数目和功能单元延迟等都是不同的。和超标量处理器相比，超长指令字机器之间进行代码移植是非常困难的。解决这个问题通常采用目标代码的转换和仿真的方法。代码的兼容性是超长指令字机器面临的最棘手的问题之一。

对于所有多指令流出的处理器来讲，面临的主要问题是如何获得较大数目的指令级并行性。在浮点程序中，通过简单地循环展开获得的并行度，还不如原始代码在向量处理器上运行的效率。目前尚不清楚多指令流出的处理器是否比向量处理器更好，它们的开销基本相同，但向量处理器的速度可能更高。与标量处理器相比，多流出处理器有两方面潜在的优势：第一，多流出处理器有潜力从不规则的代码中发掘更高的并行度；第二，所使用的存储系统开销较小。目前，多指令流出将是利用指令级并行性的主要方法，向量处理技术主要是这些处理器的扩展。

4.5 小结

指令级并行是实现高性能中央处理器的主要手段。本章在介绍指令级并行概念的基础上，重点讨论开发指令级并行的硬件技术和方法，包括如何实现指令的动态调度，如何解决控制相关和多指令流出三部分。

指令的动态调度包括目前最常用的两种硬件策略：记分牌技术和 Tomasulo

算法。本章详细讨论了这两种方法的特点、实现复杂性、实例和算法流程。这两种硬件策略是开发指令级并行技术后续内容的基础。

关于控制相关的硬件解决技术,本章主要介绍了分支预测缓冲技术、分支目标缓冲技术和推断执行技术。这些技术在目前广泛使用的微处理器中被普遍使用。

多流出技术是实现每个时钟周期中流出多条指令的必由之路。日前,它主要包括超标量技术,超流水技术和超长指令字技术。这里着重讨论了超标量技术和超长指令字技术,最后还简单讨论了多流出存在的一些技术问题。

习 题 四

4.1 解释下列术语:

指令级并行	名相关	循环展开	指令调度
控制相关	数据相关	反相关	输出相关
动态调度	乱序流出	乱序执行	记分牌
Tomasulo 算法	保留站	公共数据总线	分支预测缓冲
分支目标缓冲	推断执行	再定序缓冲	超标量
超流水	超长指令字		

4.2 列举出下面循环中的所有相关,包括输出相关、反相关、真相关和循环相关。

```

for (i=2; i<100; i=i+1)
    a[i] = b[i] + a[i];           /* s1 */
    c[i+1] = a[i] + d[i];       /* s2 */
    a[i-1] = 2 * b[i];          /* s3 */
    b[i+1] = 2 * b[i];          /* s4 */

```

4.3 根据需要展开下面的循环并进行指令调度,直到没有任何延迟。指令的延迟如表 4.2。

LOOP:	LD	F0,0(R1)
	MULTD	F0,F0,F2
	LD	F4,0(R2)
	ADDD	F0,F0,F4
	SD	0(R2),F0
	SUBI	R1,R1,8
	SUBI	R2,R2,8
	BNEQZ	R1,LOOP

4.4 假设有一个长流水线,仅仅对条件转移指令使用分支目标缓冲。假设分支预测错误的开销为 4 个时钟周期,缓冲不命中的开销为 3 个时钟周期。假设:命中率为 90%,预测精度为 90%,分支频率为 15%,没有分支的基本 CPI 为 1。

(1) 求程序执行的 CPI。

(2) 相对于采用固定的 2 个时钟周期延迟的分支处理,哪种方法程序执行速度更快?

4.5 假设分支目标缓冲的命中率为 90%, 程序中无条件转移指令的比例为 5%, 没有无条件转移指令的程序 CPI 值为 1。假设分支目标缓冲中包含分支目标指令, 允许无条件转移指令进入分支目标缓冲, 则程序的 CPI 值为多少?

4.6 下面一段 DLX 的汇编程序称为 SAXPY, 用于完成下面公式的计算:

$$Y = a \times X + Y$$

其浮点指令延迟如表 4.2 所示, 整数指令均为 1 个时钟周期完成, 浮点和整数部件均为流水。整数操作之间以及与其他所有浮点操作之间的延迟为 0, 转移指令的延迟为 0

FOO:	LD	f2,0(r1)	;load x[i]
	MULTD	f4,f2,f0	;multiply a * X[i]
	LD	f6,0(r2)	;load Y[i]
	ADDD	f6,f4,f6	;add a * X[i] + Y[i]
	SD	0[r2],f6	;store Y[i]
	ADDI	r1,r1,#8	;increment X index
	ADDI	r2,r2,#8	;increment Y index
	SGTJ	r3,r1,done	;test if done
	BEQZ	r3,FOO	;loop if not done

(1) 对于标准的 DLX 单流水线, SAXPY 循环计算一个 Y 值需要多少时间? 其中有多少空转周期?

(2) 对于标准的 DLX 单流水线, 将 SAXPY 循环顺序展开 4 次, 不进行任何指令调度, 计算一个 Y 值平均需要多少时间? 加速比是多少? 其加速是如何获得的?

(3) 对于标准的 DLX 单流水线, 将 SAXPY 循环顺序展开 4 次, 优化和调度指令, 使 SAXPY 循环处理时间达到最优, 计算一个 Y 值平均需要多少时间? 加速比是多少?

(4) 对于采用如图 4.14 推断执行机制的 DLX 处理器, 处理其中只有一个整数部件。当循环第二次执行到

BEQZ r3, FOO

时, 写出前面所有指令的状态, 包括指令使用的保留站、指令起始节拍、执行节拍和写结果节拍, 并写出处理器当前的状态。

(5) 对于两路超标量的 DLX 流水线, 设有两个指令流出部件, 可以流出任意组合的指令, 系统中的功能部件数量不受限制。将 SAXPY 循环展开 4 次, 优化和调度指令, 使 SAXPY 循环处理时间达到最优。计算一个 Y 值平均需要多少时间? 加速比是多少?

(6) 对于如图 4.19 结构的超长指令字 DLX 处理器, 将 SAXPY 循环展开 4 次, 优化和调度指令, 使 SAXPY 循环处理时间达到最优。计算一个 Y 值平均需要多少时间? 加速比是多少?

第五章 存储层次

存储器是计算机的核心部件之一,其性能直接关系到整个计算机系统性能的高低。如何以合理的价格,设计容量和速度满足计算机系统要求的存储器系统,始终是计算机体系结构设计中的关键问题之一。

计算机软件设计者和计算机用户对于存储器容量的需求是无止境的,他们希望容量越大越好,而且速度还要快,价格要便宜。仅用单一的一种存储器很难达到这一目标。较好的方法是采用存储层次,用多种存储器构成存储器的层次结构。

5.1 存储器的层次结构

5.1.1 从单级存储器到多级存储器

从用户的角度来看,存储器的三个主要指标是:容量、速度和价格(本节中,“价格”均指每位价格)。那么,究竟一个存储器的容量应是多大、速度应多快、价格应是多少才比较合理呢?在存储器的容量方面,似乎是再大也不够用,人们对它的需求越来越大。不管有多大的存储器,人们都能开发出足够大、足够复杂的应用程序,使你觉得存储器还是不够用。在速度方面,答案是比较确定的。存储器的速度应能跟得上CPU的速度,即在CPU执行程序时,能以足够快的速度向CPU提供指令和数据。近几年CPU的速度提高得很快,因此,对存储器速度的要求越来越高。在价格方面显然不能太高,至少与系统中的其他部件相比应是合理的。当然,人们总是希望价格越低越好。

然而,人们对于存储器的“容量大、速度快、价格低”的三个要求是相互矛盾的。综合考虑不同的存储器实现技术,可以发现:

- (1) 速度越快,每位价格就越高;
- (2) 容量越大,每位价格就越低;
- (3) 容量越大,速度越慢。

如果只采用其中的一种技术,存储器设计者就会陷入困境:从实现“容量大、价格低”的要求来看,应采用能提供大容量的存储器技术;但从满足性能需求的角度来看,又应采用昂贵且容量较小的快速存储器。走出这种困境的唯一方法,

是采用多种存储器技术,构成存储层次。

图 5.1 是多级存储层次的示意图。其中, M_1, M_2, \dots, M_n 为用不同技术实现的存储器。它们之间以块或页面为单位传送数据。最靠近 CPU 的 M_1 速度最快,容量最小,每位价格最高;而离 CPU 最远的 M_n 则相反,速度最慢,容量最大,每位价格最低。对于其中任何相邻的两级来说,靠近 CPU 的存储器总是容量小一些,速度快一些,价格高一些。整个存储系统想要达到的目标是:从 CPU 来看,该存储系统的速度接近于 M_1 的速度,而容量和每位价格都接近于 M_n 的容量和价格。

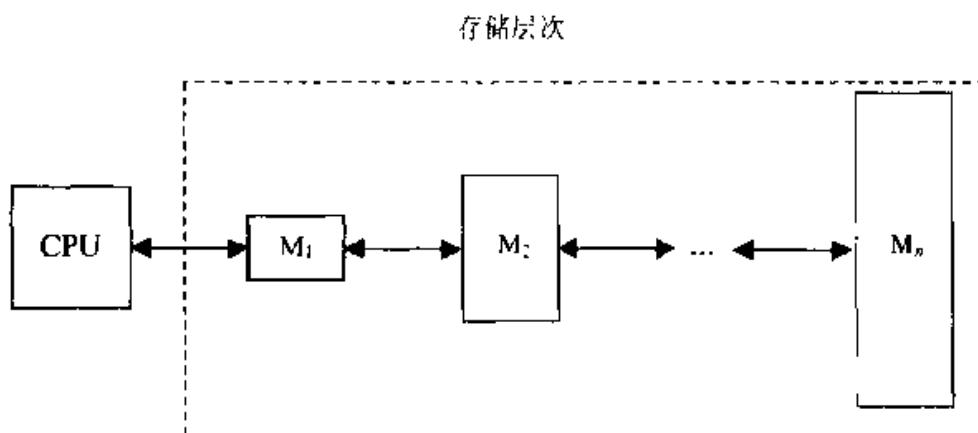


图 5.1 多级存储层次

要实现上述目标,必须做到:存储器若越靠近 CPU,则 CPU 对它的访问频率越高,而且最好大多数的访问都能在 M_1 完成。这是通过利用局部性原理来实现的。局部性原理指出,绝大多数程序访问的指令和数据都是相对簇聚的。我们可以把近期内 CPU 使用的程序和数据放在尽可能靠近 CPU 的存储器中。在存储层次中,任何一层存储器中的数据一般都是其下一层(离 CPU 更远的一层)存储器中数据的子集。CPU 访存时,首先是访问 M_1 ,若在 M_1 中找不到所要的数据,就要访问 M_2 ,将包含所需数据的块或页面调入 M_1 ;若在 M_2 中还找不到,就要访问 M_3 ;依次类推。

5.1.2 存储层次的性能参数

为简单起见,我们仅考虑由 M_1 和 M_2 两个存储器构成的两级存储层次结构。并假设 M_1 的容量、访问时间和每位价格分别为 S_1, T_{A1}, C_1 , M_2 的参数为 S_2, T_{A2}, C_2 。

1. 存储层次的平均每位价格 C

$$C = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}$$

显然,当 $S_1 \ll S_2$ 时, $C \approx C_2$ 。

2. 命中率 H

命中率为 CPU 访问存储系统时, 在 M_1 中找到所需信息的概率。命中率一般用模拟的方法来确定, 也就是通过模拟执行一组有代表性的程序, 分别记录下访问 M_1 和 M_2 的次数 N_1 和 N_2 , 则:

$$H = \frac{N_1}{N_1 + N_2}$$

为了突出反映不命中的情况, 我们还经常使用不命中率或失效概率 F 这个参数。它是指 CPU 访存时, 在 M_1 中找不到所需信息的概率。显然:

$$F = 1 - H$$

3. 平均访问时间 T_A

分两种情况来考虑 CPU 的一次访存:

(1) 当命中时, 访问时间即为 T_{A1} 。 T_{A1} 常称为命中时间(hit-time)。

(2) 当不命中时, 情况比较复杂。在大多数二级存储层次中, 若访问的字不在 M_1 中, 就必须从 M_2 把包含所请求的字的信息块传送到 M_1 , 之后 CPU 才能在 M_1 中访问到这个字。假设传送一个信息块所需的时间为 T_B , 则不命中时的访问时间为

$$T_{A2} + T_B - T_{A1} = T_{A1} + T_M$$

其中 $T_M = T_{A2} + T_B$, 它为从向 M_2 发出访问请求到把整个数据块调入 M_1 中所需的时间。 T_M 常称为失效开销(miss penalty)。

根据以上分析可知

$$\begin{aligned} T_A &= HT_{A1} + (1 - H)(T_{A1} + T_M) \\ &= T_{A1} + (1 - H)T_M \end{aligned}$$

或

$$T_A = T_{A1} + FT_M$$

5.1.3 “Cache – 主存”和“主存 – 辅存”层次

“Cache – 主存”和“主存 – 辅存”层次是常见的两种层次结构, 几乎所有当代计算机都同时具有这两种层次。我们知道, 程序在执行前需先调入主存(在虚拟存储器也是如此, 只是不必一次全部调入, 而是调入一部分执行一部分)。因此, 下面我们将从主存的角度来讨论这两个存储层次。

1. “Cache – 主存”层次

近十多年, CPU 的性能提高得很快, 在 1980 年至 1986 年之间, CPU 以每年 35% 的速度递增, 而从 1987 年开始, CPU 性能则是每年提高 55% (见图 5.2)。但是主存性能的提高却慢得多。例如, DRAM 的速度每年只提高 7%。因此, CPU 和主存之间在性能上的差距越来越大。现代计算机都采用 Cache 来解决这个问题。图 5.3(a)是“Cache – 主存”层次的示意图。这是在 CPU 和主存之间

增加一级速度快、但容量较小且每位价格较高的高速缓冲存储器(Cache)。借助于辅助软硬件,它与主存构成一个有机的整体,以弥补主存速度的不足。这个层次的工作主要由硬件实现。

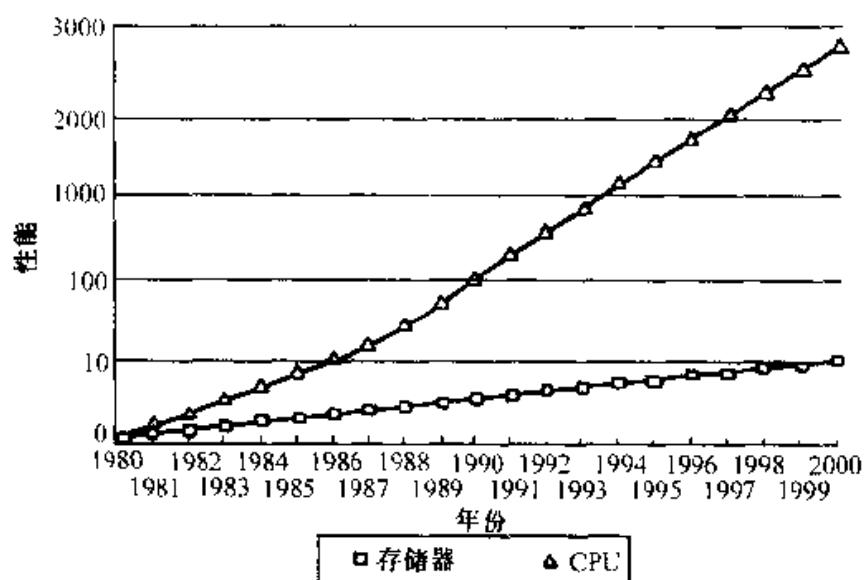
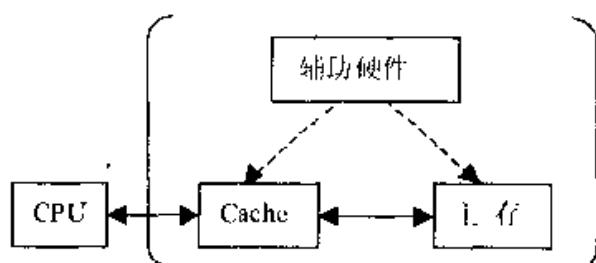
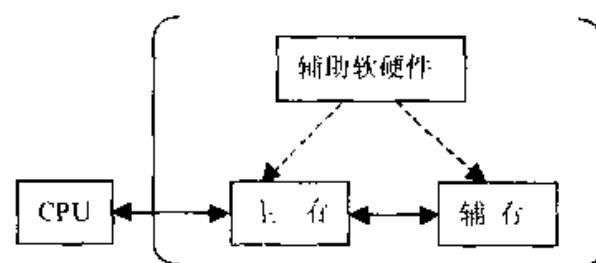


图 5.2 1980 年以来存储器和 CPU 性能随时间而提高的情况(以 1980 年时的性能作为基准)



(a) “Cache – 主存”层次



(b) “主存 – 辅存”层次

图 5.3 两种存储层次

2. “主存 – 辅存”层次

“主存 – 辅存”层次的目的是为了弥补主存容量的不足。它是在主存外面增

加一个容量更大、每位价格更低、但速度更慢的存储器(称为辅存,一般是硬盘)。它们依靠辅助软硬件的作用,构成一个整体,如图 5.3(b)所示。“主存-辅存”层次常被用来实现虚拟存储器,向编程人员提供大量的程序空间。

3. 两者的比较

表 5.1 对“Cache-主存”和“主存-辅存”层次做了一个简单的比较。

表 5.1 “Cache-主存”与“主存-辅存”层次的区别

比较项目 存储层次	“Cache-主存”层次	“主存-辅存”层次
目的	为了弥补主存速度的不足	为了弥补主存容量的不足
存储管理实现	主要由专用硬件实现	主要由软件实现
访问速度的比值 (第一级比第二级)	几比一	几万比一
典型的块(页)大小	几十个字节	几百到几千个字节
CPU 对第二级的访问方式	可直接访问	均通过第一级
失效时 CPU 是否切换	不切换	切换到其他进程

5.1.4 存储层次的四个问题

后面几节将论述“Cache-主存”层次和虚拟存储器(“主存-辅存”)。对于每一个层次,都将讨论以下四个问题:

1. 当把一个块调入高一层(靠近 CPU)存储器时,可以放到哪些位置上?(映象规则)
2. 当所要访问的块在高一层存储器中时,如何找到该块?(查找算法)
3. 当发生失效时,应替换哪一块?(替换算法)
4. 当进行写访问时,应进行哪些操作?(写策略)

搞清楚这些问题,对于理解一个具体存储层次的工作原理以及设计时的考虑是十分重要的。

5.2 Cache 基本知识

如前所述,为了填补 CPU 和主存在速度上的巨大差距,现代计算机都在 CPU 和主存之间设置一个高速、小容量(目前一般为几十 KB 到几百 KB)的缓冲存储器 Cache。Cache 对于提高整个计算机系统的性能有着重要的意义,几乎是一个不可缺少的部件。

不过,必须指出的是,由于利用局部性原理来改进性能目前非常流行,所以只要是用缓冲技术来实现经常出现数据的再利用,就常用到 Cache 这个词。例如文件 Cache,名字 Cache 等。本章中的 Cache 专指 CPU 和主存之间的 Cache。

Cache 是按块进行管理的。Cache 和主存均被分割成大小相同的块。信息以块为单位调入 Cache。相应地,CPU 的访存地址被分割成两部分:块地址和块内位移,如下所示:

主存地址:	块地址	块内位移
-------	-----	------

主存块地址用于查找该块在 Cache 中的位置,块内位移用于确定所访问的数据在该块中的位置。

5.2.1 映象规则

一般来说,主存容量远大于 Cache 的容量。因此,当要把一个块从主存调入 Cache 时,就有个如何放置的问题。这就是映象规则所要解决的。映象规则有以下三种。

1. 全相联映象(fully associative)

全相联是指主存中的任一块可以被放置到 Cache 中的任意一个位置的方法。如图 5.4(a)所示。例如,主存的第 9 块可以放入 Cache 中的任意一个位置(带阴影)。作为一个例子,图中画出了 Cache 大小为 8 块、主存大小为 16 块的情况。实际的 Cache 常包含几百个块,而主存则一般包含上百万个块。

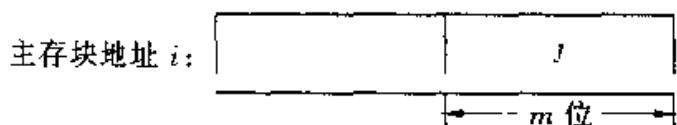
2. 直接映象(direct mapped)

直接映象是指主存中的每一个块只能被放置到 Cache 中唯一的一个位置,如图 5.4(b)所示。图中带箭头的虚线表示映象关系。例如,主存的第 9 块只能放入 Cache 的第 1 块($9 \bmod 8$)的位置。一般地,对于主存的第 i 块(即块地址为 i),设它映象到 Cache 的第 j 块,则

$$j = i \bmod (M)$$

其中 M 为 Cache 的块数。

设 $M = 2^m$,则当表示为二进制数时, j 实际上就是 i 的低 m 位,如下所示:



因此,可以直接用主存块地址的低 m 位去选择直接映象 Cache 中的相应块。

3. 组相联映象(set associative)

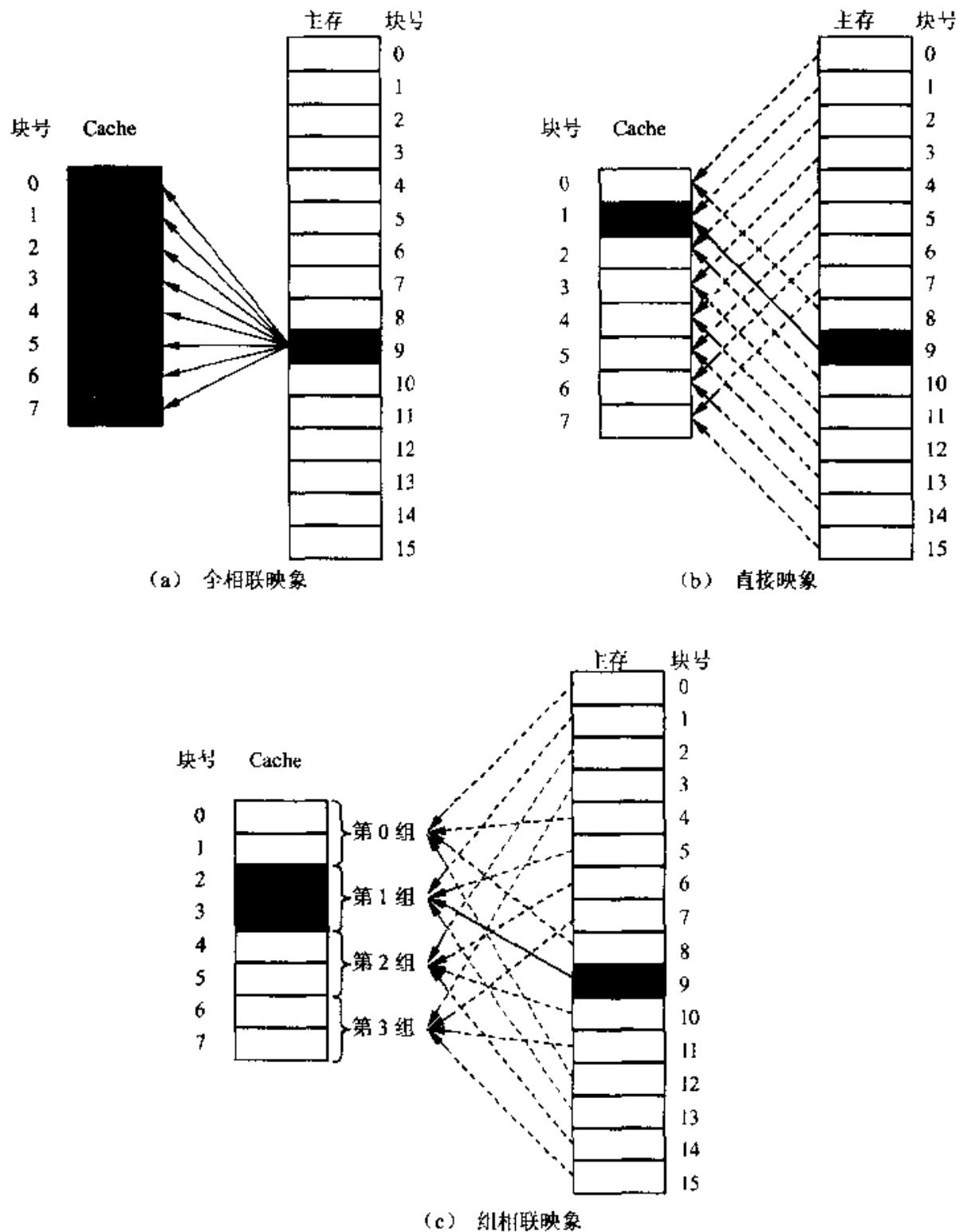


图 5.4 三种映象规则

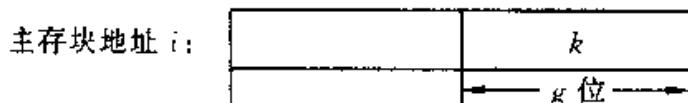
组相联是指主存中的每一块可以被放置到 Cache 中唯一的一个组中的任何一个位置(Cache 被等分为若干组,每组由若干个块构成)。这显然是直接映象和全相联的一种折衷:一个主存块首先是映象到唯一的一个组上(直接映象的特

征),然后这个块可以被放入这个组中的任何一个位置(全相联的特征)。组的选择常采用位选择算法,即:若主存第 i 块映象到 Cache 的第 k 组,则

$$k = i \bmod (G)$$

其中 G 为 Cache 的组数。

设 $G = 2^g$,则当表示为二进制数时, k 实际上就是 i 的低 g 位,如下所示:



因此,可以直接用主存块地址的低 g 位去选择组相联 Cache 中的相应组。这里的低 g 位以及上述直接映象中的低 m 位通常称为索引(index)。

如果每组中有 n 个块($n = M/G$),则称该映象规则为 n 路组相联(n -way set associative)。图 5.4(c)为两路组相联映象的示意图。

n 的不同取值构成了一系列不同相联度(associativity)的组相联。直接映象和全相联实际上是组相联的两种极端情况。表 5.2 中列出了各种情况下,路数 n 和组数 G 的取值。表中 M 为 Cache 的块数。

表 5.2 不同相联度下的路数和组数

	n (路数)	G (组数)
全相联	M	1
直接映象	1	M
其他组相联	$1 < n < M$	$1 < G < M$

相联度越高(即 n 的值越大),Cache 空间的利用率就越高,块冲突概率就越低,因而 Cache 的失效率就越低。块冲突是指一个主存块要进入已被占用的 Cache 块位置。显然,全相联的失效率最低,直接映象的失效率最高。虽然从降低失效率的角度来看, n 的值越大越好,但在后面我们将看到,增大 n 值并不一定能使整个计算机系统的性能提高,而且还会使 Cache 的实现复杂度和代价增大。因此,绝大多数计算机都采用直接映象、两路组相联或四路组相联。特别是直接映像,采用得最多。

5.2.2 查找方法

当 CPU 访问 Cache 时,如何确定 Cache 中是否有所要访问的块?若有的话,如何确定其位置?这是通过查找目录表来实现的。Cache 中设有一个目录表,该表所包含的项数与 Cache 的块数相同,每一项对应于 Cache 中的一个块,

用于指出当前该块中存放的信息是哪个主存块的(有多个主存块映象到该 Cache 块)。它实际上记录了该主存块的块地址的高位部分,称为标识(tag)。每个主存块能唯一地由其标识来确定。

为了指出 Cache 中的块是否包含有效信息,一般是在目录表中给每一项设置一个有效位。例如,当该位为“1”时表示:该目录表项有效,Cache 中相应块所包含的信息有效。当一个主存块被调入 Cache 中某一个块位置时,它的标识就被填入目标表中与该 Cache 块相对应的项中,并且该项的有效位被置“1”。

根据映象规则不同,一个主存块可能映象到 Cache 中的一个或多个 Cache 块位置。为便于讨论,我们把它们称为候选位置。当 CPU 访问该主存块时,必须且只需查找它的候选位置所对应的目录表项(标识)。如果有与所访问的主存块相同的标识,且其有效位为“1”,则它所对应的 Cache 块即是所要找的块。为了保证速度,对各候选位置所对应的标识的检查比较应并行进行。

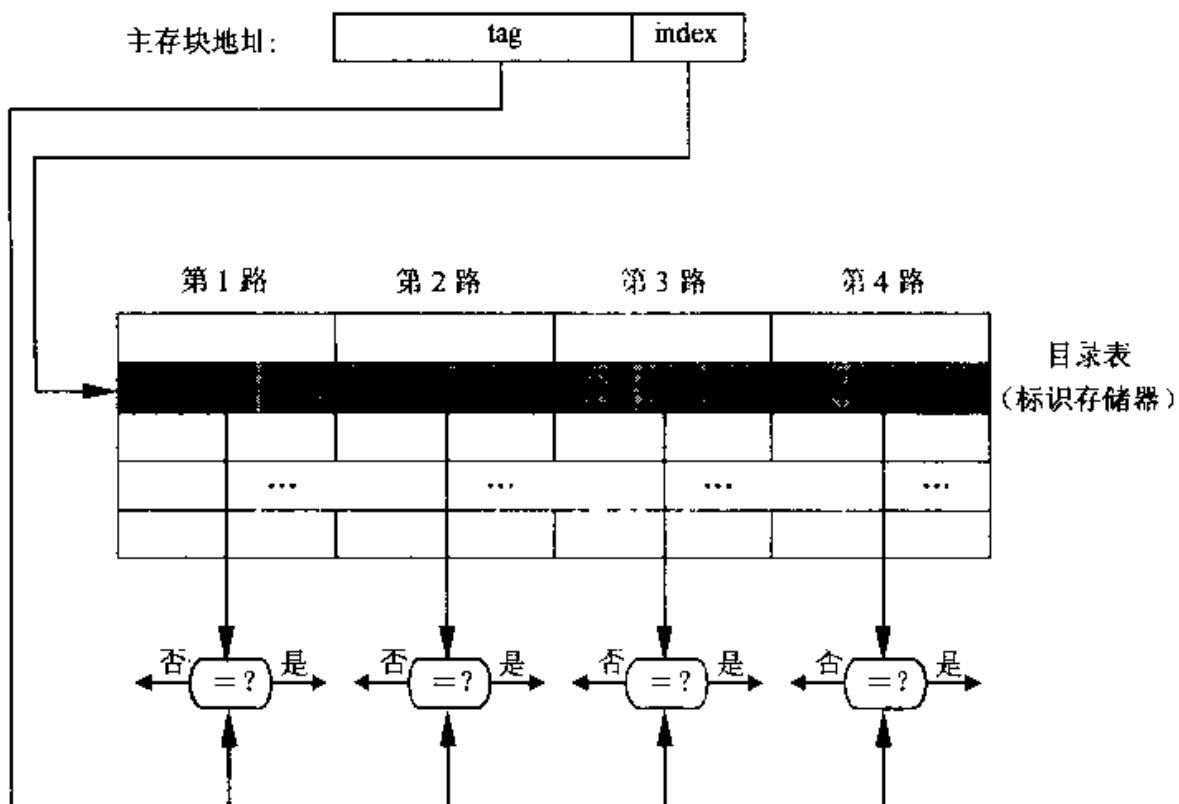


图 5.5 4 路组相联并行标识比较

直接映象 Cache 的候选位置最少,只有一个;全相联 Cache 的候选位置最多,为 M 个;而 n 路组相联则介于两者之间,为 n 个。并行查找的实现方法有两种:(1)用相联存储器实现;(2)用单体多字存储器和比较器来实现。图 5.5 中画出了用第二种方法来实现 4 路组相联的情况。这时需要 4 个比较器。CPU

访存时,用本次访存地址中的索引 index 从标识存储器中选取一行(对应于一组),并从该行读出 4 个标识,然后将它们与本次访存地址中的标识 tag 进行并行比较。根据比较结果确定是否命中以及该组中哪一个块是要访问的块(若命中)。

由图中可以看出, n 越大, 实现查找的机制就越复杂, 代价就越高。直接映象 Cache 的查找最简单: 只需查找一个位置。所访问的块要么就在这个位置上, 要么就不在 Cache 中。

无论是直接映象还是组相联, 查找时只需比较 tag, index 无需参加比较。这是因为 index 已被用来选择要查找的组(或块), 而所有索引相同(且只有索引相同)的块都被映象到该组(块)中。所以, 该组中存放的块的索引一定与本次访存的 index 相同。如果 Cache 的容量不变, 提高相联度会增加每一组中的块数, 从而会减少 index 的位数和增加 tag 的位数。当采用类似于图 5.5 的并行比较方案时, 不仅所需比较器的个数随之增加, 而且比较器的位数也随之增大。在全相联的情况下, index 的位数为 0, 块地址全都作为 tag。

5.2.3 替换算法

由于主存中的块比 Cache 中的块多, 所以当要从主存调入一个块到 Cache 中时, 会出现该块所映象到的一组(或一个)Cache 块已全被占用的情况。这时, 需强迫腾出其中的某一块, 以接纳新调入的块。那么应该替换哪一块呢? 这就是替换算法所要解决的问题。

直接映象 Cache 中的替换很简单, 因为只有一个块, 别无选择。而在组相联和全相联 Cache 中, 则有多个块供选择, 我们当然希望应尽可能避免替换掉马上就要用到的信息。主要的替换算法有以下三种。

1. 随机法

为了均匀使用一组中的各块, 这种方法随机地选择被替换的块。有些系统采用伪随机数法产生块号, 以使行为可再现。这对于调试硬件是很有用的。

这种方法的优点是简单、易于用硬件实现, 但这种方法没有考虑 Cache 块过去被使用的情况, 反映不了程序的局部性, 所以其失效率比 LRU 的高。

2. 先进先出法 FIFO(First-In-First-Out)

这种方法选择最早调入的块作为被替换的块。其优点也是容易实现。它虽然利用了同一组中各块进入 Cache 的顺序这一“历史”信息, 但还是不能正确地反映程序的局部性。因为最先进入的块, 很可能是经常要用到的块。

3. 最近最少使用法 LRU(Least Recently Used)

这种方法本来是指选择近期最少被访问的块作为被替换的块。但由于实现比较困难, 现在实际上实现的 LRU 都只是选择最久没有被访问过的块作为被

替换的块。这种方法所依据的是局部性原理的一个推论：如果最近刚用过的块很可能就是马上要再用到的块，则最久没用过的块就是最佳的被替换者。

LRU 能较好地反映程序的局部性，因而其失效率在上述三种方法中是最低的。但是 LRU 比较复杂，硬件实现比较困难，特别是当组的大小增加时，LRU 的实现代价会越来越高，而且经常只是近似地实现。

表 5.3 给出了 LRU 与随机法在失效率方面的比较。表中的数据是对于一个 VAX 地址流(既包括用户程序，也包括操作系统程序)，在块大小为 16 字节的情况下统计的。在这个例子中，对于大容量 Cache，LRU 和随机法的失效率几乎没什么差别。

表 5.3 LRU 和随机替换法的失效率比较

Cache 容量	相 联 度					
	2 路		4 路		8 路	
	LRU	随机替换	LRU	随机替换	LRU	随机替换
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

LRU 和随机法分别因其失效率低和实现简单而被广泛采用。

5.2.4 写策略

处理器对 Cache 的访问主要是读访问，因为所有对指令的访问都是“读”，而且大多数指令都不对存储器进行“写”。第二章中指出，DLX 程序的 Store 和 Load 指令所占的比例分别为 9% 和 26%，由此可得“写”在所有访存操作中所占的比例为

$$9\% / (100\% + 26\% + 9\%) \approx 7\%,$$

而在访问数据 Cache 操作中所占的比例为

$$9\% / (26\% + 9\%) \approx 25\%.$$

基于上述百分比，特别是考虑到处理器一般是对“读”要等待，而对“写”却不必等待，应该说设计 Cache 要针对最经常发生的“读”进行优化。然而，Amdahl 定律告诉我们，高性能 Cache 的设计不能忽略“写”的速度。

幸运的是，最经常发生的“读”也是最容易提高速度的。访问 Cache 时，在读出标识进行比较的同时，可以把相应的 Cache 块也读出。如果命中，则把该块中所请求的数据立即送给 CPU；若为失效，则所读出的块没什么用处，但也没什么

坏处，置之不理就是了。

然而，对于“写”却不是如此。只有在读出标识并进行比较，确认是命中后，才可对 Cache 块进行写入。由于检查标识不能与写入 Cache 块并行进行，“写”一般比“读”要花费更多的时间。另一个比较麻烦的地方是处理器要写入的数据的宽度不是定长的（通常为 1~8 字节），写入时，只能修改 Cache 块中相应的部分。而“读”则可以多读出几个字节。

按照存储层次的要求，Cache 内容应是主存部分内容的一个副本。但是“写”访问却有可能导致它们内容的不一致。例如，当处理机进行“写”访问，往 Cache 写入新的数据后，则 Cache 中相应单元的内容已发生变化，而主存中该单元的内容却仍然是原来的。这就产生了所谓的 Cache 与主存内容的一致性问题。显然，为了保证正确性，主存的内容也必须更新。至于何时更新，这正是写策略所要解决的问题。

写策略是区分不同 Cache 设计方案的一个重要标志。写策略主要有两种：

1. 写直达法 (write through)

写直达法也称为存直达法 (store through)。它是指在执行“写”操作时，不仅把信息写入 Cache 中相应的块，而且也写入下一级存储器中相应的块。

2. 写回法 (write back)

写回法也称为拷回法 (copy back)，它只把信息写入 Cache 中相应的块。该块只有在被替换时，才被写回主存。

为了减少在替换时块的写回，常采用“污染位”标志。即为 Cache 中的每一块设置一个“污染位”（设在与该块相应的目录表项中），用于指出该块是“脏”的（被修改过）还是“干净”的（没被修改过）。替换时，若被替换的块是“干净”的，则不必写回下一级存储器，因为这时下一级存储器中相应块的内容与 Cache 中的一致。

写回法和写直达法各有特色。两者相比，写回法的优点是速度快，“写”操作能以 Cache 存储器的速度进行。而且对于同一单元的多个写最后只需一次写回下一级存储器，有些“写”只到达 Cache，不到达主存，因而所使用的存储器频带较低。这使得写回法对于多处理机很有吸引力。写直达法的优点易于实现，而且下一级存储器中的数据总是最新的。后一个优点对于 I/O 和多处理机来说是重要的。I/O 和多处理机经常难以在这两种方法之间选择：它们既想用写回法来减少访存的次数，又想用写直达法来保持 Cache 与下一级存储器的一致性。

采用写直达法时，若在进行“写”操作的过程中 CPU 必须等待，直到“写”操作结束，则称 CPU 写等待 (write stall)。减少写等待的一种常用的优化技术是采用写缓冲器 (write buffer)。CPU 一旦把数据写入该缓冲器，就可以继续执行，从而使下一级存储器的更新和 CPU 的执行重叠起来。不过，在后面很快就

会看到,即使有写缓冲器,也可能发生写等待。

由于“写”访问并不需要用到所访问单元中原有的数据。所以,当发生写失效时,是否调入相应的块,有两种选择:

- 按写分配法(write allocate): 写失效时,先把所写单元所在的块调入 Cache,然后再进行写入。这与读失效类似。这种方法也称为写时取(fetch on write)方法。

- 不按写分配法(no-write allocate): 写失效时,直接写入下一级存储器而不将相应的块调入 Cache。这种方法也称为绕写法(write around)。

虽然上述两种方法都可应用于写直达法和写回法,写回法 Cache 一般采用按写分配法(这样,以后对那个块的“写”就能被 Cache 捕获),而写直达法一般采用不按写分配法(因为以后对那个块的“写”仍然还要到达下一级存储器)。

5.2.5 Cache 的结构

为了从根本上理解上述思想,下面介绍 DEC 的 Alpha AXP 21064 微处理器中的内部数据 Cache。图 5.6 为其结构框图。这是一个容量为 8KB 的直接映象 Cache,块大小为 32 字节,共有 256 个块。采用写直达方式,写缓冲器的大小为 4 个块,并且在写失效时不按写分配。

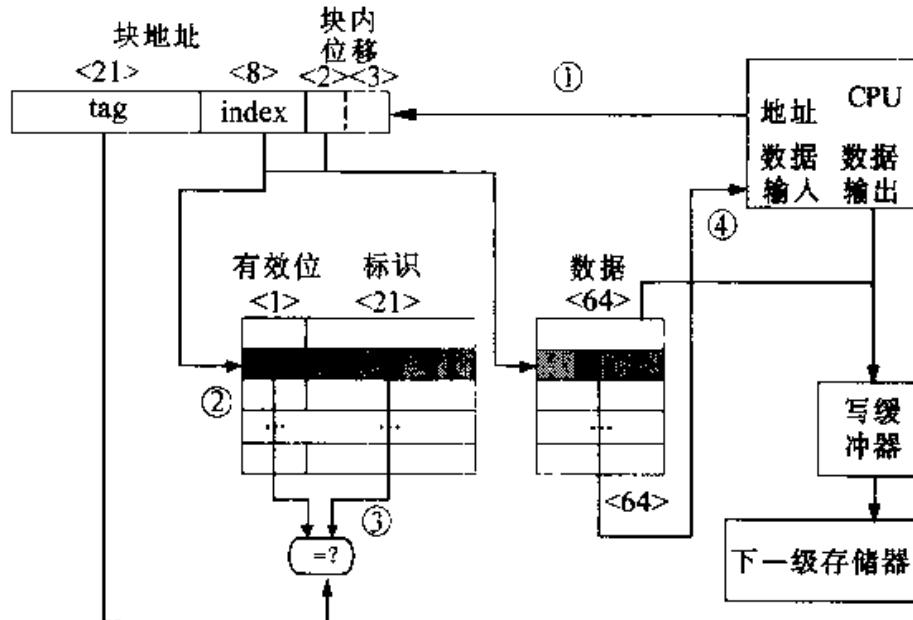


图 5.6 Alpha AXP 21064 微处理器中数据 Cache 的结构

让我们参照图 5.6 来看看写命中的操作步骤(图中带圈的数字表示步骤的顺序)。

21064 微处理器传送给 Cache 的物理地址为 34 位(图中的①)。这个地址被分为两部分:块地址(29 位)和块内偏移量(5 位)。块地址又进一步被分为地址标识(21 位)和 Cache 索引(8 位)。

Cache 索引用来从 256 个 Cache 块中选择一块, 读出数据以及相应的标识。该标识将在下一步用于判断要访问的块是否在 Cache 中(即是否命中)。索引的位数由 Cache 容量、块大小、相联度决定。21064 的 Cache 是直接映象的, 所以相联度为 1, 我们可以按下式来计算索引所需的位数:

$$2^{\text{index}} = \frac{\text{Cache 容量}}{\text{块大小} \times \text{相联度}} = \frac{8192}{32 \times 1} = 256 = 2^8$$

因此, 索引为 8 位, 标识为 $29 - 8 = 21$ 位。

第二步(②)是按索引选择标识和数据。在直接映象的 Cache 中, 读出数据并送往 CPU 与读出标识并进行匹配这两个过程可以并行进行。

标识从 Cache 中读出来以后, 就去和 CPU 送来的物理地址中的标识部分进行比较, 这是图中的第三步(③)。为了保证标识信息有效, 其相应的有效位必须为“1”, 否则比较的结果就是无效的。

如果标识比较的结果是匹配, 且有效位为“1”, 那么最后一步(④)就是发信号通知 CPU 从 Cache 中取走数据。21064 完成这 4 步需要 2 个时钟周期。如果在这个过程中, 指令需要用到本次“读”的结果, 这条指令就只好等待。

和任何 Cache 一样, 在 21064 中对“写”的处理比对“读”的处理更复杂。如果被写的字正好在 Cache 中(写命中), 前三步跟上面是一样的。最后, 在确认标识比较为匹配之后, 才把数据写入。

因为 21064 使用写直达 Cache, 所以到此写过程还未结束, 还应将数据送往写缓冲器。21064 的写缓冲器含有四个块, 每块大小为 4 个字, 缓冲器是按字寻址的(21064 中每个字为 8 字节)。如果此时写缓冲器未满, 那么就把数据和完整的地址写入缓冲器。对 CPU 而言, 本次“写”访问已完成, CPU 可以继续工作, 而写缓冲器将负责把该数据写入主存。在往写缓冲器写入地址和数据时, 如果缓冲器内存在被修改过的块, 就检查其地址, 看看本次写入数据的地址是否与缓冲器内的某个有效块的地址匹配。如果匹配, 就把新数据与该块合并。这叫做写合并(write merging)。如果没有这种优化措施, 按顺序地址连续“写”四次, 就可能会填满整个缓冲器; 而采用写合并, 就可以很容易地将这四个字放入缓冲器的同一块中。

图 5.7 说明了在有写合并和没有写合并的情况下, 写缓冲器的使用情况。图中每个缓冲器有 4 项, 每项能放 4 个字(32 字节)。各项的地址标在左边, 各项中的有效位 V 用于指出其后的 8 个字节是否已被占用。在没有写合并功能时, 写入地址为 200、208、216 和 224 的四次写(4 个字)占据了缓冲器的全部四项(图 5.7 上图); 而在有写合并功能时, 这四个字可以被合并为一项(图 5.7 下图)。当缓冲器已满, 并且没有地址相匹配的块时, Cache 和 CPU 就需要等待, 直到缓冲器有空闲项后才可继续运行。

以上为 Cache 命中的情况。当发生失效时,情况又会是怎样呢?

当发生读失效时,Cache 向 CPU 发出一个暂停信号,通知它等待,并从下一级存储器中读入 32 字节数据。21064 的 Cache 和它的下一级存储器之间的数据通路(21064 微处理器总线的数据通道)为 16 字节(128 位)。每次数据传送需 5 个时钟周期,传送全部 32 字节数据就要花 10 个时钟周期。因为 21064 的数据 Cache 是直接映象的,所以被替换块只有一个,别无选择。替换一个块意味着更新该块的数据、标识和有效位。

当发生写失效时,21064 采用不按写分配规则。也就是说,CPU 使数据“绕过”Cache,直接写入主存。

地址	V	V	V	V
200	1	0	0	0
208	1	0	0	0
216	1	0	0	0
224	1	0	0	0

地址	V	V	V	V
200	1	1	1	1
	0	0	0	0
	0	0	0	0
	0	0	0	0

图 5.7 没有写合并和有写合并

现在我们已经知道了数据 Cache 的工作原理,但仅有数据 Cache 还不够,处理器还要取指令。尽管可以使用单一的指令数据混合 Cache(称为统一 Cache 或混合 Cache)来同时提供数据和指令,但它有可能会成为瓶颈。例如,当按流水方式工作的处理器执行 Load 或 Store 指令时,可能会同时请求一个数据字和一个指令字。所以对于 Load 或 Store 操作,单一的 Cache 会出现结构冲突,导致 CPU 等待。解决这个问题的一个简单的方法是将单一的 Cache 分为两个 Cache;一个专门存放指令,另一个专门存放数据。大多数最近生产的处理器都采用了分离的 Cache。Alpha AXP 21064 就是如此,它有一个 8 KB 的指令 Cache,其结构和图 5.6 中的 8 KB 数据 Cache 几乎一样。

CPU 知道它发出的地址是指令地址还是数据地址,因此可以为它们设置不同的端口,这样就会加倍对存储系统和 CPU 之间数据通道带宽的要求。由于系统对指令和数据的操作特性不同,独立的指令 Cache 和数据 Cache 使我们能分

别对它们进行优化,它们各自采用不同的容量、块大小和相联度时的性能可能会更好。

表 5.4 列出了不同容量的指令 Cache、数据 Cache 以及混合 Cache 在相同条件下的失效率。这些数据是在块大小为 32 字节、映象方法为直接映象的条件下,针对 SPEC92 典型程序,在 DECstation 5000 上测出的平均值。对指令的访问约占所有访问的 75%。

表 5.4 指令 Cache、数据 Cache 和混合 Cache 失效率的比较

容 量	指令 Cache	数据 Cache	混合 Cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

从表 5.4 可以看出,指令 Cache 的失效率比数据 Cache 的低。分离的指令 Cache 和数据 Cache 消除了因 Cache 中的指令块和数据块互相冲突而引起的失效,但是这样一来也限定了分配给指令和数据的空间。究竟哪一方面对失效率的影响更大呢?若要公平地对分离 Cache 和混合 Cache 进行比较,就要求两种 Cache 的总容量相同。例如,分离的 1 KB 指令 Cache 和 1 KB 数据 Cache 就应该和容量为 2 KB 的混合 Cache 相比较。为了计算分离的指令 Cache 和数据 Cache 的平均失效率,首先必须知道对每种类型的 Cache 进行访问的百分比。从第二章可知,对指令 Cache 的访问占全部访问的 $100\% / (100\% + 26\% + 9\%)$,即大约 75%;而对数据 Cache 的访问占全部访问的 $(26\% + 9\%) / (100\% + 26\% + 9\%)$,即大约 25%。除改变失效率外,指令 Cache 和数据 Cache 相分离还会影响性能的其他方面,后面将详细介绍这一点。

5.2.6 Cache 性能分析

失效率与硬件速度无关,用它来评价存储系统的性能非常方便,所以我们经常使用它。但是,它也容易产生一些误导。一种更好的评测存储系统性能的指标是平均访存时间:

$$\text{平均访存时间} = \text{命中时间} + \text{失效率} \times \text{失效开销}$$

其中命中时间(hit-time)是指访问 Cache 命中时所用的时间,而另外两个术语我们在前面已经介绍过。平均访存时间的两个组成部分既可以用绝对时间(如命中时间为 2 ns),也可以用时钟周期数(如失效开销为 50 个时钟周期)来衡量。

平均访存时间仍然是衡量性能的一个间接指标,尽管它是一个比失效率更好的指标,但并不能代替程序执行时间。

我们可以用这个公式比较分离 Cache 和混合 Cache 的性能。

例 5.1 假设 Cache 的命中时间为 1 个时钟周期,失效开销为 50 个时钟周期,在混合 Cache 中一次 Load 或 Store 操作访问 Cache 的命中时间都要增加一个时钟周期(因为混合 Cache 只有一个端口,无法同时满足两个请求。按照前一章中有关流水线的术语,混合 Cache 会导致结构冲突),根据表 5.4 所列的失效率,试问指令 Cache 和数据 Cache 容量均为 16 KB 的分离 Cache 和容量为 32 KB 的混合 Cache 相比,哪种 Cache 的失效率更低?又假设采用写直达策略,且有一个写缓冲器,并且忽略写缓冲器引起的等待。请问上述两种情况下平均访存时间各是多少?

解 如前所述,约 75% 的访存为取指令。因此,分离 Cache 的总体失效率为

$$(75\% \times 0.64\%) + (25\% \times 6.47\%) = 2.10\%$$

根据表 5.4,容量为 32 KB 的混合 Cache 的失效率略低一些,只有 1.99%。

平均访存时间公式可以分为指令访问和数据访问两部分:

$$\text{平均访存时间} = \text{指令所占的百分比} \times (\text{指令命中时间} + \text{指令失效率} \times \text{失效开销}) +$$

$$\text{数据所占的百分比} \times (\text{数据命中时间} + \text{数据失效率} \times \text{失效开销})$$

所以,两种结构的平均访存时间分别为

$$\begin{aligned}\text{平均访存时间}_{\text{分离}} &= 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50) \\ &= (75\% \times 1.32) + (25\% \times 4.325) = 0.990 + 1.059 = 2.05\end{aligned}$$

$$\begin{aligned}\text{平均访存时间}_{\text{混合}} &= 75\% \times (1 + 1.99\% \times 50) + 25\% \times (1 + 1 + 1.99\% \times 50) \\ &= (75\% \times 1.995) + (25\% \times 2.995) = 1.496 + 0.749 = 2.24\end{aligned}$$

因此,尽管分离 Cache 的实际失效率比混合 Cache 的高,但其平均访存时间反而较低。分离 Cache 提供了两个端口,消除了结构相关。

执行一个程序所需的 CPU 时间与 Cache 的性能密切相关。考虑以下公式:

$$\text{CPU 时间} = (\text{CPU 执行周期数} + \text{存储器停顿周期数}) \times \text{时钟周期时间}$$

为了简化对各种 Cache 设计方案的评价,有时设计者们假设所有的访存停顿都是由 Cache 失效引起的,这是因为和其他原因引起的停顿(如 I/O 设备使用存储器引起的竞争)相比,失效引起的停顿占了绝大多数。这里也作这种假设。但是请注意,在计算最终的性能时,考虑所有存储器停顿是很重要的。

上面的 CPU 时间公式提出了一个问题,即 Cache 命中所用的时钟周期数应被看作是 CPU 执行时钟周期数的一部分,还是存储器停顿时钟周期数的一部分。尽管两种考虑都是合理的,但使用比较广泛的还是前一种。

可以用程序的访存总次数、失效开销(单位为时钟周期)以及“读”和“写”的失效率来计算存储器停顿的时钟周期数,即

$$\text{存储器停顿时钟周期数} = \text{“读”的次数} \times \text{读失效率} \times \text{读失效开销} + \\ \text{“写”的次数} \times \text{写失效率} \times \text{写失效开销}$$

一般通过将“读”的次数和“写”的次数合并,并求出“读”和“写”的平均失效率和平均失效开销,将上式简化为

$$\text{存储器停顿时钟周期数} = \text{访存次数} \times \text{失效率} \times \text{失效开销}$$

由于“读”和“写”的失效率和失效开销通常是不相等的,所以这只是一个近似公式。

从执行时间和存储停顿周期数中提取公因子“指令数”(IC),得

$$\text{CPU 时间} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{访存次数}}{\text{指令数}} \times \text{失效率} \times \text{失效开销} \right) \times \text{时钟周期时间}$$

其中“访存次数/指令数”为每条指令的平均访存次数。

有些设计者不喜欢用每次访存的平均失效次数,而更喜欢用每条指令的平均失效次数来衡量失效率,即

$$\frac{\text{失效次数}}{\text{指令数}} = \frac{\text{访存次数} \times \text{失效率}}{\text{指令数}}$$

这个指标的优点是与实现无关。例如,21064 的指令预取部件可能会重复地访问某一个字,如果使用每次访存的平均失效次数,而不用每条指令的平均失效次数来度量失效率,就可能会人为地降低失效率。该指标的不足之处是,每条指令的平均失效次数和体系结构是密切相关的。例如,80x86 和 DLX 每条指令的平均访存次数可能相差很大。因此,每条指令的平均失效次数在体系结构设计者研究同一系列中的计算机时,是最常使用的参数。这些设计者们常使用下述形式的 CPU 时间公式:

$$\text{CPU 时间} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{存储器停顿周期数}}{\text{指令数}} \right) \times \text{时钟周期时间}$$

下面来研究 Cache 对性能的影响。

例 5.2 我们用一个和 Alpha AXP 类似的机器作为第一个例子。假设 Cache 失效开销为 50 个时钟周期,当不考虑存储器停顿时,所有指令的执行时间都是 2.0 个时钟周期,Cache 的失效率为 2%,平均每条指令访存 1.33 次。试分析 Cache 对性能的影响。

$$\text{解 CPU 时间} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{存储器停顿周期数}}{\text{指令数}} \right) \times \text{时钟周期时间}$$

考虑 Cache 的失效后, 性能为

$$\begin{aligned} \text{CPU 时间}_{\text{有Cache}} &= IC \times (2.0 + 1.33 \times 2\% \times 50) \times \text{时钟周期时间} \\ &= IC \times 3.33 \times \text{时钟周期时间} \end{aligned}$$

因此, 当考虑了 Cache 的失效影响后, CPI 就会增大。本例中 CPI 从理想计算机的 2.0 增加到 3.33, 是原来的 1.67 倍。由于不管有没有 Cache, 时钟周期时间和指令数都保持不变, 所以 CPU 的时间也将增加到原来的 1.67 倍。然而, 若不采用 Cache, CPI 将增加为 $2.0 + 50 \times 1.33 = 68.5$, 即超过原来的 30 倍。

正如上面例子所说明的, Cache 的行为可能会对系统性能产生巨大的影响。而且, Cache 失效对于一个 CPI 较小而时钟频率较高的 CPU 来说, 影响是双重的:

1. CPI_{execution} 越低, 固定周期数的 Cache 失效开销的相对影响就越大。
2. 在计算 CPI 时, 失效开销的单位是时钟周期数。因此, 即使两台计算机的存储层次完全相同, 时钟频率较高的 CPU 的失效开销较大, 其 CPI 中存储器停顿这部分也就较大。

因此 Cache 对于低 CPI、高时钟频率的 CPU 来说更加重要, 而且在评价这类机器的性能时, 如果忽略 Cache 的行为, 就更容易出错。这又一次验证了 Amdahl 定律。

尽可能地减少平均访问时间是一个合理的目标, 而且在本章许多地方我们也是使用平均访问时间这个指标的, 但是请记住, 我们的最终目标是减少 CPU 的执行时间。下面的例子就说明了二者的区别。

例 5.3 考虑两种不同组织结构的 Cache: 直接映象 Cache 和两路组相联 Cache, 试问它们对 CPU 的性能有何影响? 先求平均访存时间, 然后再计算 CPU 性能。分析时请用以下假设:

- (1) 理想 Cache(命中率为 100%) 情况下的 CPI 为 2.0, 时钟周期为 2 ns, 平均每条指令访存 1.3 次。
- (2) 两种 Cache 容量均为 64 KB, 块大小都是 32 字节。
- (3) 图 5.8 说明, 在组相联 Cache 中, 必须增加一个多路选择器, 用于根据标识匹配结果从相应组的块中选择所需的数据。因为 CPU 的速度直接与 Cache 命中的速度紧密相关, 所以对于组相联 Cache, 由于多路选择器的存在而使 CPU 的时钟周期增加到原来的 1.10 倍。
- (4) 这两种结构 Cache 的失效开销都是 70 ns。(在实际应用中, 应取整为整数个时钟周期)
- (5) 命中时间为 1 个时钟周期, 64 KB 直接映象 Cache 的失效率为 1.4%, 相同容量的两路组相联 Cache 的失效率为 1.0%。

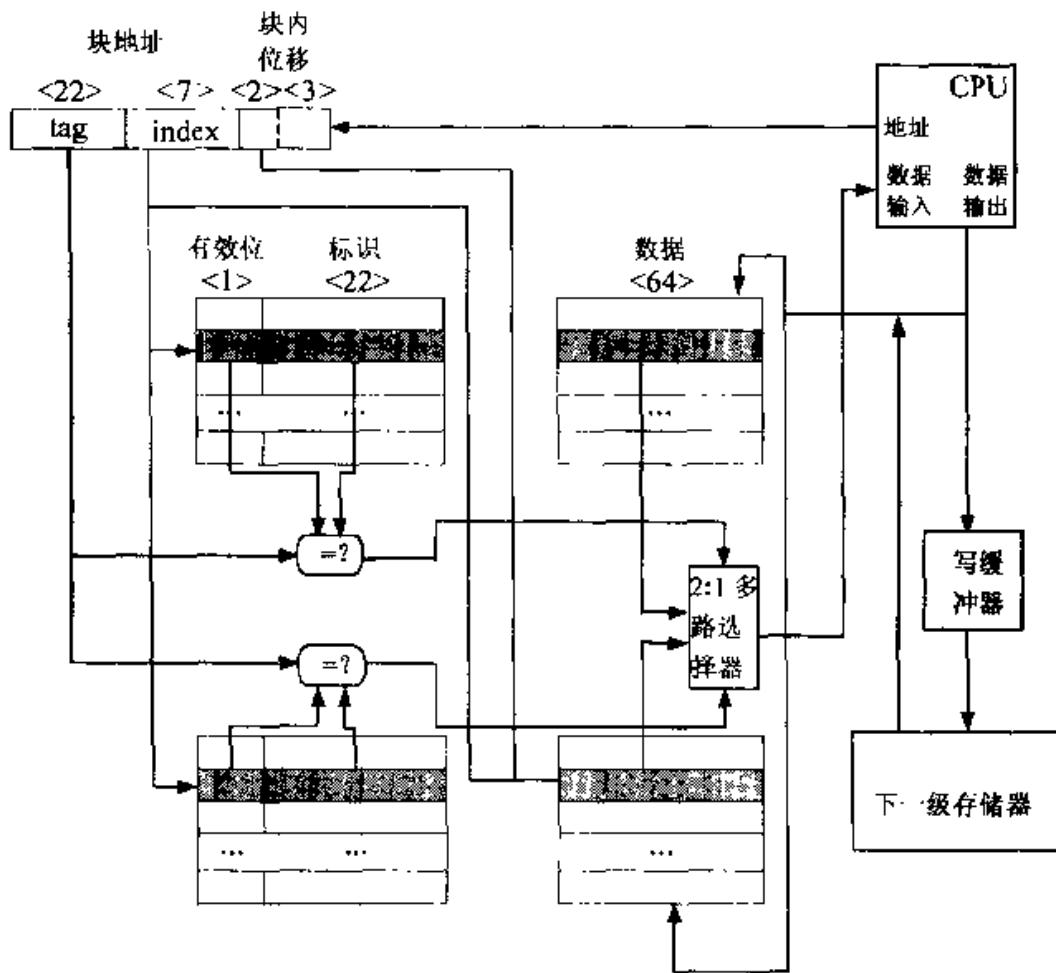


图 5.8 一个两路组相联 Cache(与图 5.6 对比, 注意本图中的多路选择器)

解 平均访存时间为

$$\text{平均访存时间} = \text{命中时间} + \text{失效率} \times \text{失效开销}$$

因此,两种结构的平均访存时间分别是

$$\text{平均访存时间}_{1\text{路}} = 2.0 + (0.014 \times 70) = 2.98 \text{ ns}$$

$$\text{平均访存时间}_{2\text{路}} = 2.0 \times 1.10 + (0.010 \times 70) = 2.90 \text{ ns}$$

两路组相联 Cache 的平均访存时间比较低。

CPU 性能为

$$\begin{aligned} \text{CPU 时间} &= IC \times (\text{CPI}_{\text{execution}} + \frac{\text{失效次数}}{\text{指令数}} \times \text{失效开销}) \times \text{时钟周期时间} \\ &= IC \times [(\text{CPI}_{\text{execution}} \times \text{时钟周期时间}) + (\frac{\text{访存次数}}{\text{指令数}} \\ &\quad \times \text{失效率} \times \text{失效开销} \times \text{时钟周期时间})] \end{aligned}$$

用 70 ns 代替(失效开销 × 时钟周期时间), 两种结构的性能分别为

$$\text{CPU 时间}_{1\text{路}} = IC \times (2.0 \times 2 + (1.3 \times 0.014 \times 70)) = 5.27 \times IC$$

$$\text{CPU 时间}_{2\text{路}} = IC \times (2.0 \times 2 \times 1.10 + (1.3 \times 0.010 \times 70)) = 5.31 \times IC$$

相对性能比为

$$\frac{\text{CPU时间}_{2\text{路}}}{\text{CPU时间}_{1\text{路}}} = \frac{5.31 \times IC}{5.27 \times IC} = \frac{5.31}{5.27} = 1.01$$

和平均访存时间的比较结果相反,直接映象 Cache 的平均性能稍好一些。这是因为在两路组相联的情况下,虽然失效次数减少了,但所有指令的时钟周期时间都增加了 10%。由于 CPU 时间是我们进行评价的基准,而且直接映象 Cache 的实现更简单,所以本例中直接映象 Cache 是较好的选择。

5.2.7 改进 Cache 性能

CPU 和主存之间在速度上越来越大的差距已引起了许多体系结构设计人员的关注。从 1989 到 1995 年,全世界共发表了 1 600 多篇有关 Cache 的研究论文。根据平均访存时间公式

$$\text{平均访存时间} = \text{命中时间} + \text{失效率} \times \text{失效开销}$$

可知,可以从以下三个方面改进 Cache 的性能:

- (1) 降低失效率
- (2) 减少失效开销
- (3) 减少 Cache 命中时间

下面将介绍 15 种 Cache 优化技术,其中 7 种用于降低失效率,5 种用于减少失效开销,3 种用于减少命中时间。(参见表 5.9)

5.3 降低 Cache 失效率的方法

许多有关 Cache 的研究都致力于降低 Cache 的失效率。本节就来讨论这个问题。

按照产生失效的原因不同,我们可以把失效分为以下三类(简称为“3C”):

- (1) 强制性失效(Compulsory miss)

当第一次访问一个块时,该块不在 Cache 中,需从下一级存储器中调入 Cache,这就是强制性失效。这种失效也称为冷启动失效或首次访问失效。

- (2) 容量失效(Capacity miss)

如果程序执行时所需的块不能全部调入 Cache 中,则当某些块被替换后,若又重新被访问,就会发生失效。这种失效称为容量失效。

- (3) 冲突失效(Conflict miss)

在组相联或直接映象 Cache 中,若太多的块映象到同一组(块)中,则会出现该组中某个块被别的块替换(即使别的组或块有空闲位置),然后又被重新访问的情况。这就是发生了冲突失效。这种失效也称为碰撞失效(collision)或干扰

失效(interference)。

表 5.5 针对 SPEC92 典型程序给出了上述三种失效所占的比例(这些数据是在 DECstation 5000 上测得的。假设 Cache 的块大小为 32 字节,并采用 LRU 算法)。为了说明高相联度的好处,表中列出了在直接映象、两路组相联、四路组相联和八路组相联的情况下冲突失效的值。可以看出:

- (1) 相联度越高,冲突失效就越少;
- (2) 强制性失效和容量失效不受相联度的影响;
- (3) 强制性失效不受 Cache 容量的影响,但容量失效却随着容量的增加而减少;
- (4) 表中的数据符合 2:1 的 Cache 经验规则,即大小为 N 的直接映象 Cache 的失效率约等于大小为 N/2 的两路组相联 Cache 的失效率。

表 5.5 在不同容量不同相联度的情况下,Cache 的总失效率以及“3C”所占的比例

Cache 容量	相联度	总失效率	失效率组成(相对百分比)				
			强制性失效	容量失效	冲突失效		
1 KB	1 路	0.133	0.002	1%	0.008	60%	0.052 39%
1 KB	2 路	0.105	0.002	2%	0.080	76%	0.023 22%
1 KB	4 路	0.095	0.002	2%	0.080	84%	0.013 14%
1 KB	8 路	0.087	0.002	2%	0.080	92%	0.005 6%
2 KB	1 路	0.098	0.002	2%	0.044	45%	0.052 53%
2 KB	2 路	0.076	0.002	2%	0.044	58%	0.030 39%
2 KB	4 路	0.064	0.002	3%	0.044	69%	0.018 28%
2 KB	8 路	0.054	0.002	4%	0.044	82%	0.008 14%
4 KB	1 路	0.072	0.002	3%	0.031	43%	0.039 54%
4 KB	2 路	0.057	0.002	3%	0.031	55%	0.024 42%
4 KB	4 路	0.049	0.002	4%	0.031	64%	0.016 32%
4 KB	8 路	0.039	0.002	5%	0.031	80%	0.006 15%
8 KB	1 路	0.046	0.002	4%	0.023	51%	0.021 45%
8 KB	2 路	0.038	0.002	5%	0.023	61%	0.013 34%
8 KB	4 路	0.035	0.002	5%	0.023	66%	0.010 28%
8 KB	8 路	0.029	0.002	6%	0.023	79%	0.004 15%
16 KB	1 路	0.029	0.002	7%	0.015	52%	0.012 42%
16 KB	2 路	0.022	0.002	9%	0.015	68%	0.005 23%

续表

Cache 容量	相联度	总失效率	失效效率组成(相对百分比)					
			强制性失效		容量失效		冲突失效	
16 KB	4 路	0.020	0.002	10%	0.015	74%	0.003	17%
16 KB	8 路	0.018	0.002	10%	0.015	80%	0.002	9%
32 KB	1 路	0.020	0.002	10%	0.010	52%	0.008	38%
32 KB	2 路	0.014	0.002	14%	0.010	74%	0.002	12%
32 KB	4 路	0.013	0.002	15%	0.010	79%	0.001	6%
32 KB	8 路	0.013	0.002	15%	0.010	81%	0.001	4%
64 KB	1 路	0.014	0.002	14%	0.007	50%	0.005	36%
64 KB	2 路	0.010	0.002	20%	0.007	70%	0.001	10%
64 KB	4 路	0.009	0.002	21%	0.007	75%	0.000	3%
64 KB	8 路	0.009	0.002	22%	0.007	78%	0.000	0%
128 KB	1 路	0.010	0.002	20%	0.004	40%	0.004	40%
128 KB	2 路	0.007	0.002	29%	0.004	58%	0.001	14%
128 KB	4 路	0.006	0.002	31%	0.004	61%	0.001	8%
128 KB	8 路	0.006	0.002	31%	0.004	62%	0.000	7%

图 5.9 是表 5.5 中数据的图示, 其中上图为绝对失效率, 下图为各种类型失效效率所占的百分比。请注意, 图中 4 路组相联的冲突失效所对应的区域为标有“4 路”和“8 路”两个区域的合并, 两路组相联的冲突失效所对应的区域为标有“2 路”、“4 路”和“8 路”3 个区域的合并。

从图中可以看出, SPEC92 程序的强制失效效率很小。其他许多运行时间较长的程序也是如此。在 3C 中, 冲突失效似乎是最容易减少的, 只要采用全相联, 就不会发生冲突失效。但是, 用硬件实现全相联是很昂贵的, 而且可能会降低处理器的时钟频率(见例 5.3), 从而导致整体性能的下降。至于容量失效, 除了增大 Cache 以外, 没有别的办法。在一个存储层次中, 如果高一级存储器的容量比程序所需的空间小得多, 就有可能出现抖动现象。这时大部分时间是花在两级存储器之间移动数据。出现抖动时, 由于大量进行替换, 机器的运行速度接近于只有第二级存储器的情况, 甚至更慢。

另一个减少 3C 的方法是增加块的大小, 以减少强制性失效。但在下面将看到, 块大小增加可能会增加其他类型的失效。

下面介绍 7 种降低失效效率的方法。需要强调的是, 许多降低失效效率的方法

会增加命中时间(hit-time)或失效开销(miss penalty)。因此,在具体使用时,要综合考虑,保证降低失效率确能使整个系统速度提高。

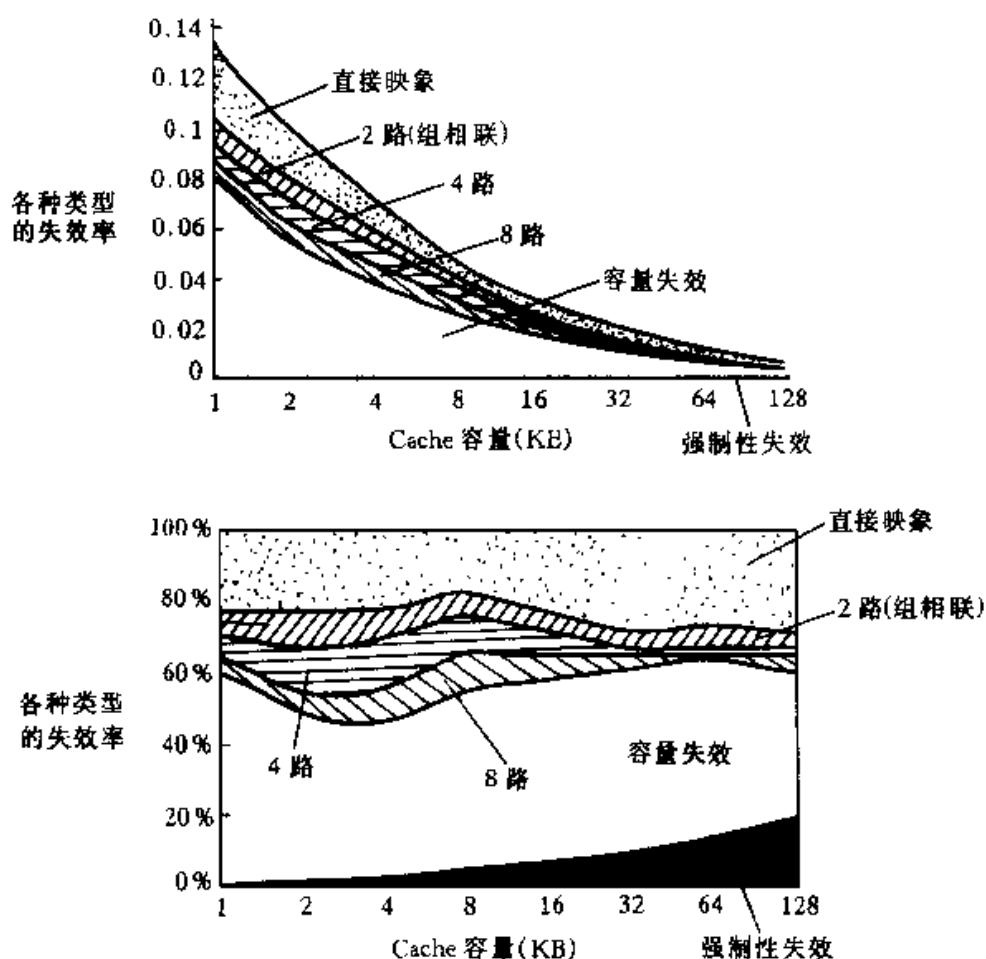


图 5.9 总失效率及“3C”的分布情况(根据表 5.5 的数据绘制)

5.3.1 增加 Cache 块大小

降低失效率最简单的方法是增加块大小。图 5.10 中对于一组不同的 Cache 容量,给出了失效率和块大小的关系(在与表 5.5 类似的情况下测得)。表 5.6 列出了图 5.10 的具体数据。从中可以看出:

(1) 对于给定的 Cache 容量,当块大小增加(从 16 字节开始)时,失效率开始是下降,后来反而上升了。

(2) Cache 容量越大,使失效率达到最低的块大小就越大。例如在本例中,对于大小分别为 1 KB、4 KB、16 KB、64 KB 和 256 KB 的 Cache,使失效率达到最低的块大小分别为 32 字节、64 字节、64 字节、128 字节、128 字节(或 256 字节)。

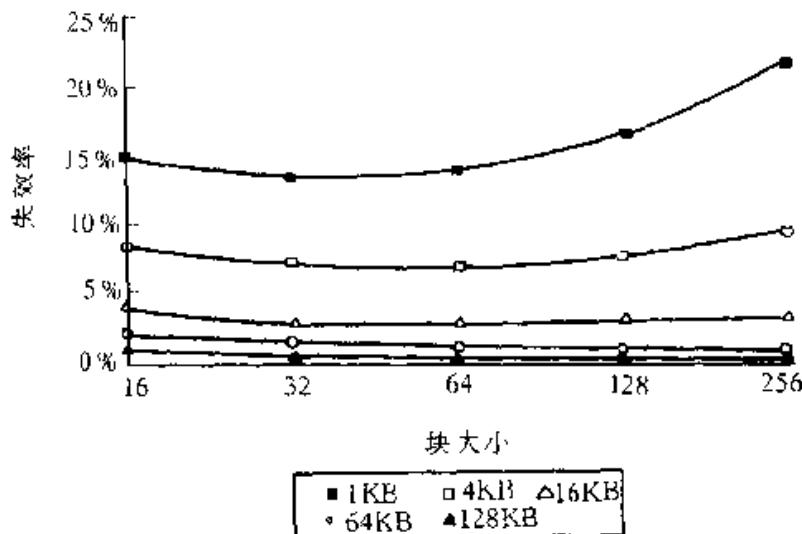


图 5.10 失效率随块大小变化的曲线

表 5.6 各种块大小情况下 Cache 的失效率

块大小 (字节)	Cache 容量				
	1 KB	4 KB	16 KB	64 KB	256 KB
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

导致上述失效率先下降后上升的原因，在于增加块大小会产生双重作用。一方面它减少了强制性失效，因为局部性原理有两方面的含义：时间局部性和空间局部性，增加块大小利用了空间局部性；另一方面，由于增加块大小会减少 Cache 中块的数目，所以有可能会增加冲突失效。在 Cache 容量较小时，甚至还会增加容量失效。刚开始增加块大小时，由于块大小还不是很大，上述的第一种作用超过第二种作用，从而使失效率下降。但等到块大小较大时，第二种作用超过第一种作用，使失效率上升。

显然，没有理由把块大小增加到使失效率反而上升的程度。此外，增加块大小同时也会增加失效开销，如果这个负面效应超过了失效率下降所带来的好处，就会使平均访存时间增加。这时，即使降低失效率也是无益的。

例 5.4 假定存储系统在延迟 40 个时钟周期后，每 2 个时钟周期能送出 16 个字节，即：经过 42 个时钟周期，它可提供 16 个字节；经过 44 个时钟周期，可提

供 32 个字节;依此类推。请问对于表 5.6 中列出的各种容量的 Cache,在块大小分别为多少时,平均访存时间最小?

解 平均访存时间为

$$\text{平均访存时间} = \text{命中时间} + \text{失效率} \times \text{失效开销}$$

假设命中时间与块大小无关,为 1 个时钟,那么对于一个块大小为 16 字节、容量为 1 KB 的 Cache 来说:

$$\text{平均访存时间} = 1 + (15.05\% \times 42) = 7.321 \text{ 个时钟周期}$$

而对于块大小为 256 字节、容量为 256 KB 的 Cache 来说,平均访存时间为

$$\text{平均访存时间} = 1 + (0.49\% \times 72) = 1.353 \text{ 个时钟周期}$$

表 5.7 列出了在这两种极端情况之间的各种块大小和各种 Cache 容量的平均访存时间。粗体字的数字为速度最快的情况:Cache 容量为 1 KB、4 KB、16 KB 的情况下块大小为 32 字节时速度最快;容量为 64 KB 和 256 KB 时,64 字节最快。实际上,这些块大小都是当今处理机 Cache 中最常见的。

表 5.7 各种块大小情况下 Cache 的平均访问时间

块大小 (字节)	失效开销 (时钟周期)	Cache 容量				
		1 KB	4 KB	16 KB	64 KB	256 KB
16	42	7.321	4.599	2.655	1.857	1.458
32	44	6.870	4.186	2.263	1.594	1.308
64	48	7.605	4.360	2.267	1.509	1.245
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

如前所述,Cache 设计者一直在努力同时减少失效率和失效开销。从失效开销的角度来讲,块大小的选择取决于下一级存储器的延迟和带宽两个方面。高延迟和高带宽时,宜采用较大的 Cache 块,因为这时每次失效时,稍微增加一点失效开销,就可以获得许多数据。与之相反,低延迟和低带宽时,宜采用较小的 Cache 块,因为采用大 Cache 块所能节省的时间不多。一个小 Cache 块失效开销的两倍与一个两倍于其大小的 Cache 块的失效开销差不多,而且采用小 Cache 块,块的数量多,就有可能减少冲突失效。

5.3.2 提高相联度

表 5.5 和图 5.9 已经说明了提高相联度会使失效率下降。从中我们可以得出两条经验规则。第一,对于表中所列出的 Cache 容量,从实际应用的角度来看,8 路组相联在降低失效率方面的作用已经和全相联一样有效。也就是说,采

用相联度超过 8 的方法实际意义不大, 第二条规则叫做 2:1 Cache 经验规则, 它是指容量为 N 的直接映象 Cache 的失效效率和容量为 $N/2$ 的两路组相联 Cache 的失效效率差不多相同。

许多例子都说明, 改进平均访存时间的某一方面是以损失另一方面为代价的。增加块大小的方法会在降低失效效率的同时增加失效开销, 而提高相联度则是以增加命中时间(hit-time)为代价。Hill 曾发现, 当分别采用直接映象和两路组相联时, 对于 TTL 或 ECL 板级 Cache, 命中时间相差 10%; 而对于定制的 CMOS Cache, 命中时间相差 2%。所以, 为了实现很高的处理器时钟频率, 就需要设计结构简单的 Cache; 但时钟频率越高, 失效开销就越大(所需的时钟周期数越多)。为减少失效开销, 又要求提高相联度。下面通过一个例子进一步说明。

例 5.5 假定提高相联度会按下列比例增大处理器时钟周期:

$$\text{时钟周期}_{2\text{路}} = 1.10 \times \text{时钟周期}_{1\text{路}}$$

$$\text{时钟周期}_{4\text{路}} = 1.12 \times \text{时钟周期}_{1\text{路}}$$

$$\text{时钟周期}_{8\text{路}} = 1.14 \times \text{时钟周期}_{1\text{路}}$$

假定命中时间为 1 个时钟, 直接映象情况下失效开销为 50 个时钟周期, 而且假设不必将失效开销取整。使用表 5.5 中的失效效率, 试问当 Cache 为多大时, 以下不等式成立?

$$\text{平均访存时间}_{8\text{路}} < \text{平均访存时间}_{4\text{路}}$$

$$\text{平均访存时间}_{4\text{路}} < \text{平均访存时间}_{2\text{路}}$$

$$\text{平均访存时间}_{2\text{路}} < \text{平均访存时间}_{1\text{路}}$$

解 在各种相联度的情况下, 平均访存时间分别为

$$\begin{aligned} \text{平均访存时间}_{8\text{路}} &= \text{命中时间}_{8\text{路}} + \text{失效效率}_{8\text{路}} \times \text{失效开销}_{8\text{路}} = 1.14 + \text{失} \\ &\quad \text{效率}_{8\text{路}} \times 50 \end{aligned}$$

$$\text{平均访存时间}_{4\text{路}} = 1.12 + \text{失效效率}_{4\text{路}} \times 50$$

$$\text{平均访存时间}_{2\text{路}} = 1.10 + \text{失效效率}_{2\text{路}} \times 50$$

$$\text{平均访存时间}_{1\text{路}} = 1.00 + \text{失效效率}_{1\text{路}} \times 50$$

在每种情况下的失效开销相同, 都是 50 个时钟周期。把相应的失效效率代入上式, 即可得平均访存时间。例如, 1 KB 的直接映象 Cache 的平均访存时间为

$$\text{平均访存时间}_{1\text{路}} = 1.00 + (0.133 \times 50) = 7.65$$

容量为 128 KB 的 8 路组相联 Cache 的平均访存时间为

$$\text{平均访存时间}_{8\text{路}} = 1.14 + (0.006 \times 50) = 1.44$$

利用这些公式和表 5.5 中给出的失效效率, 可得各种容量和相联度情况下 Cache 的平均访存时间, 如表 5.8 所示。表中的数据说明, 当 Cache 容量不超过 16 KB 时, 上述三个不等式成立。从 32 KB 开始, 对于平均访存时间有: 4 路组

相联的平均访存时间小于 2 路组相联的, 2 路组相联的小于直接映象的, 但 8 路组相联的却比 4 路组相联的大。

表 5.8 根据表 5.5 得到的平均访存时间

Cache 容量 (KB)	相联度(路)			
	1	2	4	8
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	1.79
64	1.70	1.60	1.57	1.59
128	1.50	1.45	1.42	1.44

请注意,本例中没有考虑时钟周期增大对程序其他部分的影响。

5.3.3 Victim Cache

增加 Cache 块大小和相联度是从 Cache 一出现就被体系结构设计者们用来降低失效率的两种经典方法。从本小节开始,我们来看一看近几年提出的几种方法,这些方法能在不影响时钟周期和失效开销的前提下降低 Cache 失效率。

一种能减少冲突失效次数而又不影响时钟频率的方法是:在 Cache 和它与下一级存储器的数据通路之间增设一个全相联的小 Cache, 称为 Victim Cache。图 5.11 为其结构框图。Victim Cache 中存放由于失效而被丢弃(替换)的那些块(即 victim)。每当发生失效时,在访问下一级存储器之前,先检查 Victim Cache 中是否含有所需的块。如果有,就将该块与 Cache 中某个块做交换。Jouppi 于 1990 年发现,含 1 到 5 项的 Victim Cache 对减少冲突失效很有效,尤其是对于那些小型的直接映象数据 Cache 更是如此。对于不同的程序,一个项数为 4 的 Victim Cache 能使一个 4 KB 直接映象数据 Cache 的冲突失效减少 20 % ~ 90 %。

从 Cache 的层次来看,Victim Cache 可以看成位于 Cache 和存储器之间的又一级 Cache,采用命中率较高的全相联策略,容量小,而且仅仅在替换时发生作用。

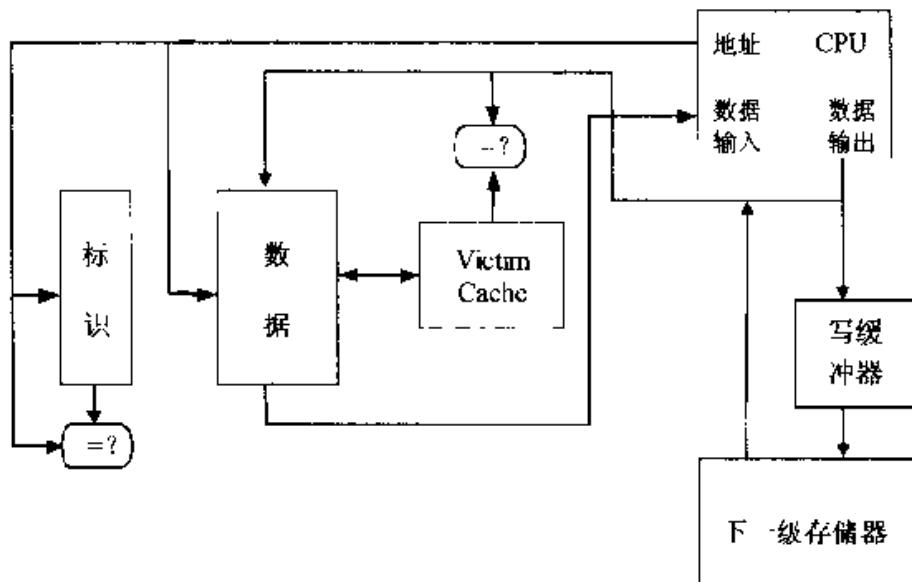


图 5.11 Victim Cache 在存储层次中的位置

5.3.4 伪相联 Cache

有一种方法既能获得多路组相联 Cache 的低失效率,又能保持直接映象 Cache 的命中速度,这种方法称为伪相联(pseudo-associate)或列相联(column associate)。采用这种方法时,在命中情况下,访问 Cache 的过程和直接映象 Cache 中的情况相同;而发生失效时,在访问下一级存储器之前,会先检查 Cache 另一个位置(块),看是否匹配。确定这个“另一块”的一种简单的方法是将索引字段的最高位取反,然后按照新索引去寻找“伪相联组”中的对应块。如果这一块的标识匹配,则称发生了“伪命中”。否则,就只好访问下一级存储器。

伪相联 Cache 具有一快一慢两种命中时间,它们分别对应于正常命中和伪命中的情况。图 5.12 中绘出了它们的相对关系。使用伪相联技术存在一定的危险:如果直接映象 Cache 里的许多快速命中在伪相联 Cache 中变成慢速命中,那么这种优化措施反而会降低整体性能。因此,要能够指出同一组中的两个块哪个为快速命中,哪个为慢速命中,这是很重要的。一种简单的解决方法就是交换两个块的内容。

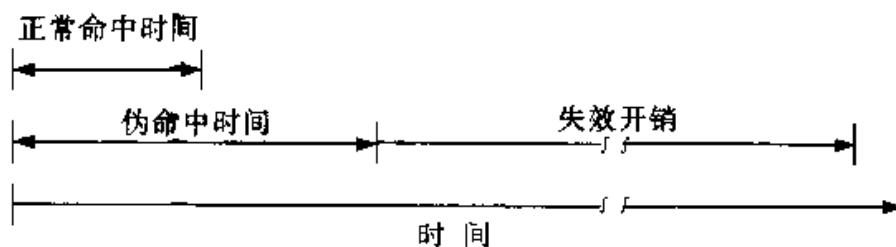


图 5.12 正常命中时间、伪命中时间和失效开销之间的关系

下面通过一个例子来说明伪相联带来的好处。

例 5.6 假设当在按直接映象找到的位置处没有发现匹配,而在另一个位置才找到数据(伪命中)时,需要 2 个额外的周期。仍用上个例子中的数据,问:当 Cache 容量分别为 2 KB 和 128 KB 时,直接映象、两路组相联和伪相联这三种组织结构中,哪一种速度最快?

解 首先考虑标准的平均访存时间公式:

$$\text{平均访存时间}_{\text{伪相联}} = \text{命中时间}_{\text{伪相联}} + \text{失效率}_{\text{伪相联}} \times \text{失效开销}_{\text{伪相联}}$$

我们从该公式的最后一部分着手。不管我们对命中的情况做了何种改进,失效开销总是相同的。为了确定失效率,需要知道什么时候会发生失效。只要我们总是通过把索引的最高位变反的方法来寻找另一块,在同一“伪相联”组中的两块就是用同一个索引选择得到的,这与在两路组相联 Cache 中所用的方法是一样的,因而它们的失效率相同,即

$$\text{失效率}_{\text{伪相联}} = \text{失效率}_{\text{2路}}$$

再看命中时间。伪相联 Cache 的命中时间等于直接映象 Cache 的命中时间加上在伪相联查找过程中命中(即伪命中)的百分比乘以该命中所需的额外时间开销,即

$$\text{命中时间}_{\text{伪相联}} = \text{命中时间}_{\text{1路}} + \text{伪命中率}_{\text{伪相联}} \times 2;$$

伪相联查找的命中率等于两路组相联 Cache 的命中率和直接映象 Cache 命中率之差:

$$\begin{aligned}\text{伪命中率}_{\text{伪相联}} &= \text{命中率}_{\text{2路}} - \text{命中率}_{\text{1路}} \\ &= (1 - \text{失效率}_{\text{2路}}) - (1 - \text{失效率}_{\text{1路}}) \\ &= \text{失效率}_{\text{1路}} - \text{失效率}_{\text{2路}}\end{aligned}$$

综合上述分析,有

$$\begin{aligned}\text{平均访存时间}_{\text{伪相联}} &= \text{命中时间}_{\text{1路}} + (\text{失效率}_{\text{1路}} - \text{失效率}_{\text{2路}}) \times 2 + \\ &\quad \text{失效率}_{\text{2路}} \times \text{失效开销}_{\text{1路}}\end{aligned}$$

将表 5.5 中的数据代入上面的公式,得

$$\text{平均访存时间}_{\text{伪相联}, 2\text{ KB}} = 1 + (0.098 - 0.076) \times 2 + (0.076 \times 50) = 4.844$$

$$\text{平均访存时间}_{\text{伪相联}, 128\text{ KB}} = 1 + (0.010 - 0.007) \times 2 + (0.007 \times 50) = 1.356$$

根据上一个例子中的表 5.8,对于 2 KB 的 Cache,可得

$$\text{平均访存时间}_{\text{1路}} = 5.90 \text{ 个时钟}$$

$$\text{平均访存时间}_{\text{2路}} = 4.90 \text{ 个时钟}$$

对于 128 KB 的 Cache,可得

$$\text{平均访存时间}_{\text{1路}} = 1.50 \text{ 个时钟}$$

$$\text{平均访存时间}_{\text{2路}} = 1.45 \text{ 个时钟}$$

可见,对于这两种 Cache 容量,伪相联 Cache 都是速度最快的。

尽管从理论上来说,伪相联是一种很有吸引力的方法,但它的多种命中时间会使 CPU 流水线的设计复杂化。因此伪相联技术往往应用在离处理器比较远的 Cache 上,如第二级 Cache。

5.3.5 硬件预取技术

Victim Cache 和伪相联 Cache 都能在不影响处理器时钟频率的前提下降低失效率,预取技术也能够实现这一点。指令和数据都可在处理器提出访问请求之前进行预取。预取内容可以直接放入 Cache,也可以放在一个访问速度比主存快的外部缓冲器中。

指令预取通常由 Cache 之外的硬件完成。例如,Alpha AXP 21064 微处理器在发生指令失效时取两个块:被请求指令块和顺序的下一指令块。被请求指令块返回时放入 Cache,而预取指令块则放在指令流缓冲器中;如果某次被请求的指令块正好在指令流缓冲器里,则取消对该块的访存请求,直接从指令流缓冲器中读出这一块,同时发出对下一指令块的预取访存请求。21064 的指令流缓冲器中只含一个 32 字节的块。Jouppi 的研究结果表明:对于块大小为 16 字节、容量为 4 KB 的直接映象指令 Cache,1 个块的指令流缓冲器就可以捕获 15%~25% 的失效,4 个块的指令流缓冲器可以捕获大约 50% 的失效,而 16 个块的指令流缓冲器则可以捕获 72% 的失效。

我们可以用相似的技术预取数据。经 Jouppi 统计,一个数据流缓冲器大约可以捕获 4 KB 直接映象 Cache 的 25% 的失效。对于数据 Cache,可以采用多个数据流缓冲器,分别从不同的地址预取数据。Jouppi 发现,用 4 个数据流缓冲器可以将命中率提高到 43%。

Palacharla 和 Kessler 于 1994 年针对一组科学计算程序,研究了既能预取指令又能预取数据的流缓冲器。他们发现,对于一个具有两个 64 KB 四路组相联 Cache(一个用于指令,一个用于数据)的处理器来说,8 个流缓冲器能够捕获其 50%~70% 的失效。

例 5.7 Alpha AXP 21064 采用指令预取技术,其实际失效率是多少?若不采用指令预取技术,Alpha AXP 21064 的指令 Cache 必须为多大才能保持平均访存时间不变?

解 假设当指令不在指令 Cache 里,而在预取缓冲器中找到时,需要多花一个时钟周期。下面是修改后的公式:

$$\text{平均访存时间}_{\text{预取}} = \text{命中时间} + \text{失效率} \times \text{预取命中率} \times 1 + \\ \text{失效率} \times (1 - \text{预取命中率}) \times \text{失效开销}$$

假设预取命中率为 25%,命中时间为 1 个时钟周期,失效开销为 50 个时钟周期。从表 5.4 可知,8 KB 指令 Cache 的失效率为 1.10%,则

$$\begin{aligned}\text{平均访存时间} &= 1 + (1.10\% \times 25\% \times 1) + [1.10\% \times (1 - 25\%) \times 50] \\ &= 1 + 0.00275 + 0.413 = 1.415\end{aligned}$$

为了得到相同性能下的实际失效率,由原始公式得

$$\text{平均访存时间} = \text{命中时间} + \text{失效率} \times \text{失效开销}$$

$$\begin{aligned}\text{失效率} &= (\text{平均访存时间} - \text{命中时间}) / \text{失效开销} = (1.415 - 1) / 50 \\ &= 0.83\%\end{aligned}$$

计算结果说明,采用预取功能后,8 KB 的 Cache 的实际失效率为 0.83 %,由表 5.4 可知,16 KB 指令 Cache 的失效率为 0.64 %,所以采用预取后,8 KB 的 Cache 的失效率介于普通 8 KB Cache 的失效率 1.10 % 和 16 KB Cache 的失效率 0.64 % 之间。

预取建立在利用存储器的空闲频带(若不采用预取,这些频带将浪费掉)的基础上。但是,如果它影响了对正常失效的处理,就可能会降低性能。利用编译器的支持,可以减少不必要的预取。

5.3.6 由编译器控制的预取

硬件预取的一种替代方法是在编译时加入预取指令,在数据被用到之前发出预取请求。预取有以下几种类型:

- 寄存器预取(register prefetch):把数据取到寄存器中。
- Cache 预取(Cache prefetch):只将数据取到 Cache 中,不放入寄存器。

这两种预取既可以是故障性的(faulting),也可以是非故障性的(nonfaulting)。故障性预取是指在预取时,若出现虚地址故障或违反保护权限,就会发生异常。而非故障性预取在遇到这种情况时则不会发生异常。按照这样的说法,一条正常的 Load 指令应该被认为是“故障性寄存器预取指令”。非故障性预取如导致异常,就转变为空操作。最有效的预取对程序是“语义上不可见的”:它既不会改变指令和数据之间的各种逻辑关系或存储单元的内容,也不会造成虚拟存储器故障。本节假定 Cache 预取都是非故障性的,也叫做非绑定(nonbinding)预取。

只有在预取数据的同时处理器还能继续执行的情况下,预取才是有意义的。这就要求 Cache 在等待预取数据返回的同时还能继续提供指令和数据。这种灵活的 Cache 称为非阻塞(nonblocking)Cache 或非锁定(lockup-free)Cache,后面将详细讨论。

和硬件控制的预取一样,编译器控制预取的目的也是要使执行指令和读取数据能重叠执行。循环是预取优化的主要目标,因为它们易于进行预取优化。如果失效开销较小,编译器只要简单地将循环体展开一次或两次,并调度好预取

和执行的重叠。如果失效开销较大,编译器就将循环体展开许多次,以便为后面较远的循环预取数据。

然而,发出预取指令需要花费一条指令的开销,因此,要注意保证这种开销不超过预取所带来的收益。编译器可以通过把重点放在那些可能会导致失效的访问,使程序避免不必要的预取,从而较大幅度地改善平均访存时间。

例 5.8 对于下面的程序,首先判断哪些访问可能会导致数据 Cache 失效。然后,加入预取指令以减少失效。最后,计算所执行的预取指令的条数以及通过预取避免的失效次数。假定:

(1) 我们用的是一个容量为 8 KB、块大小为 16 字节的直接映象 Cache,它采用写回法并且按写分配。

(2) a, b 分别为 3×100 (3 行 100 列)和 101×3 的双精度浮点数组,每个元素都是 8 个字节。当程序开始执行时,这些数据都不在 Cache 内。

```
for ( i = 0 ; i < 3 ; i = i + 1 )
    for ( j = 0 ; j < 100 ; j = j + 1 )
        a[ i ][ j ] = b[ j ][ 0 ] * b[ j+1 ][ 0 ];
```

解 编译器首先需要判断哪些访问可能造成 Cache 失效。否则就有可能会对本来就是命中的数据发出预取指令,浪费时间。

当没有预取功能时,对数组 a 中元素的写操作是按照它们在主存中的存放顺序进行的,因而对 a 的访问受益于空间局部性。由于每一块含两个元素,当 j 为偶数时是第一次访问一个块,为奇数时是第二次,因此,当 j 为偶数时失效,为奇数时命中。因为 a 有 3 行 100 列,所以对它的访问一共将导致命中和失效各 $3 \times 100/2 = 150$ 次。

数组 b 没有受益于空间局部性,但它两次受益于时间局部性:① i 的每一次循环都访问同样的元素;② 对于 j 的每一次循环,都使用一次和上一次循环相同的 b 的元素。不考虑冲突失效,则由于访问数组 b 而引起的全部失效为:当 $i=0$ 时,对所有 $b[j][0]$ 的访问以及当 $j=0$ 时第一次对 $b[j+1][0]$ 的访问。由于当 $i=0$ 时 j 从 0 递增到 99,因此对数组 b 的访问将引起 101 次失效。

所以,在没有预取功能时,这个循环一共将引起 $150 + 101 = 251$ 次数据 Cache 失效。

为了简化优化措施,我们将不考虑对循环刚开始时的一些访问的预取,也不必取消在循环结束时的预取。不过若它们是故障性预取,就不能这么简单地处理了。

根据上述分析,我们将循环分解,使得在第一次循环中不仅预取 a ,而且预取 b ,而在第二次循环中只预取 a ,因为此时 b 早已经被预取了。假设失效开销很大,预取必须至少提前 7 次循环进行。

```

for ( j = 0; j < 100; j = j + 1 )
    prefetch( b[ j + 7 ][ 0 ]); /* 预取 7 次循环后所需的 b( j , 0 ) */
    prefetch( a[ 0 ][ j + 7 ]); /* 预取 7 次循环后所需的 a( 0 , j ) */
    a[ 0 ][ j ] = b[ j ][ 0 ] * b[ j + 1 ][ 0 ]
}
for ( i = 1; i < 3; i = i + 1 ) {
    for ( j = 0; j < 100; j = j + 1 )
        prefetch( a[ i ][ j + 7 ]); /* 预取 7 次循环后所需的 a( i , j ) */
        a[ i ][ j ] = b[ j ][ 0 ] * b[ j + 1 ][ 0 ];
}

```

修改后的程序预取了以下数据:从 $a[i][7]$ 到 $a[i][99]$, 从 $b[7][0]$ 到 $b[100][0]$ 。失效情况为:第一个循环中访问 $a[0][0]$ 到 $a[0][6]$ 失效 $\lceil 7/2 \rceil = 4$ 次, 访问 $b[0][0]$ 到 $b[6][0]$ 失效 7 次; 第二个循环中访问 $a[i][0]$ 到 $a[i][6]$ 失效 $\lceil 7/2 \rceil \times 2 = 8$ 次。故总的失效次数减少为

$$4 + 7 + 8 = 19 \text{ 次}$$

避免了 232 次失效, 其代价是执行了 400 条预取指令, 这很可能是一个很好的折衷。

例 5.9 在以下条件下, 计算例 5.8 中所节约的时间:

- (1) 忽略指令 Cache 失效, 并假设数据 Cache 无冲突失效和容量失效。
- (2) 假设预取可以被重叠或与 Cache 失效重叠执行, 从而能以最大的存储带宽传送数据。
- (3) 不考虑 Cache 失效时, 修改前的循环每 7 个时钟周期循环一次。修改后的程序中, 第一个预取循环每 9 个时钟周期循环一次, 而第二个预取循环每 8 个时钟周期循环一次(包括外层 for 循环的开销)。
- (4) 一次失效需 50 个时钟周期。

解 修改前的双重循环一共执行 $3 \times 100 = 300$ 次循环。每次用 7 个时钟周期, 总时间就是 $300 \times 7 = 2100$ 个时钟周期再加上 Cache 失效的开销。失效开销一共是 $251 \times 50 = 12550$ 个时钟周期, 故总的时间为 14650 个时钟周期。

在修改后的程序中, 第一个预取循环共执行 100 次, 每次循环用 9 个时钟周期, 总的执行时间为 900 个时钟周期再加上 Cache 失效开销。失效开销一共是 $(4 + 7) \times 50 = 550$ 个时钟周期, 故总的时间为 1450 个时钟周期。第二个循环执行 200 次, 每次用 8 个时钟周期, 共 1600 个时钟周期, 再加上 $8 \times 50 = 400$ 个时钟周期的失效开销, 总的时间为 2000 个时钟周期。从前一个例子我们知道, 在执行这两个循环的 $1450 + 2000 = 3450$ 个时钟周期中, 一共执行了 400 条预取指令。如果假设预取操作与其他部分的执行完全重叠, 那么含预取指令的程序

比原程序快 $14\ 650 / 3\ 450 = 4.2$ 倍。

5.3.7 编译器优化

迄今为止,所介绍的技术(增加块大小,提高相联度,伪相联,硬件预取以及预取指令)都需要改变或者增加硬件,但下面介绍的方法无需对硬件做任何改动就可以降低失效率。

这种方法就是通过对软件的优化来降低失效率。这也许是硬件设计者最喜欢的解决方案。处理器和主存之间越来越大的性能差距促使编译器的设计者们去仔细研究存储层次行为,以期能通过编译时的优化来改进性能。这项研究同样也分为减少指令失效和减少数据失效两个方面。

我们能很容易地重新组织程序而不影响程序的正确性。例如,把一个程序中的几个过程重新排序,就可能会减少冲突失效,从而降低指令失效率。McFarling 研究了如何使用记录信息来判断指令组之间可能发生的冲突,并将指令重新排序以减少失效。他发现,这样可将容量为 2 KB、块大小为 4 字节的直接映象指令 Cache 的失效率降低 50%;对于容量为 8 KB 的 Cache,可将失效率降低 75%。他还发现,当能够使某些指令根本就不进入 Cache 时,可以得到最佳性能。但即使不这么做,优化后的程序在直接映象 Cache 中的失效率也低于未优化程序在同样大小的 8 路组相联 Cache 中的失效率。

数据对存储位置的限制比指令对存储位置的限制还要少,因此更便于调整顺序。我们对数据进行变换的目的是改善数据的空间局部性和时间局部性。例如,可以把对数组的运算改为对存放在同一 Cache 块中的所有数据进行操作,而不是按照程序员原来随意书写的顺序访问数组元素。

1. 数组合并(merging arrays)

这种技术通过提高空间局部性来减少失效次数。有些程序同时用相同的索引来访问若干个数组的同一维。这些访问可能会相互干扰,导致冲突失效。我们可以这样来消除这种危险:将这些相互独立的数组合并成为一个复合数组,使得一个 Cache 块中能包含全部所需的元素。

```
/* 修改前 */
int val [ SIZE ];
int key [ SIZE ];

/* 修改后 */
struct merge {
    int val ;
    int key ;
};
```

```
struct merge merged_array [ SIZE ];
```

这个例子有一个有趣的特点：如果程序员能正确地使用记录数组，他就能获得与本优化相同的益处。

2. 内外循环交换(loop interchange)

有些程序中含有嵌套循环，程序没有按照数据在存储器中存储的顺序进行访问。在这种情况下，只要简单地交换循环的嵌套关系就能使程序按数据在存储器中存储的顺序进行访问。和前一个例子一样，这种技术也是通过提高空间局部性来减少失效次数；重新排列访问顺序使得在一个 Cache 块被替换之前，能最大限度地利用块中的数据。

```
/* 修改前 */
for (j = 0 ; j < 100 ; j = j+1 )
    for (i = 0 ; i < 5000 ; i = i+1 )
        x [ i ][ j ] = 2 * x [ i ][ j ];
/* 修改后 */
for (i = 0 ; i < 5000 ; i = i+1 )
    for (j = 0 ; j < 100 ; j = j+1 )
        x [ i ][ j ] = 2 * x [ i ][ j ];
```

修改前的程序以 100 个字的跨距访问存储器，而修改后的程序顺序地访问一个 Cache 块中的元素，然后再访问下一块中的元素。和前一个例子不同，这种优化技术在不改变执行的指令数的前提下，提高了 Cache 的性能。

3. 循环融合(loop fusion)

有些程序含有几部分独立的程序段，它们用相同的循环访问同样的数组，对相同的数据作不同的运算。通过将它们融合为单一的一个循环，能使读入 Cache 的数据在被替换出去之前得到反复的使用。因此，和前面的两种技术不同，这种优化的目标是通过改进时间局部性来减少失效次数。

```
/* 修改前 */
for (i = 0 ; i < N ; i = i+1 )
    for (j = 0 ; j < N ; j = j+1 )
        a [ i ][ j ] = 1/b [ i ][ j ] + c [ i ][ j ];
for (i = 0 ; i < N ; i = i+1 )
    for (j = 0 ; j < N ; j = j+1 )
        d [ i ][ j ] = a [ i ][ j ] + c [ i ][ j ];
/* 修改后 */
for (i = 0 ; i < N ; i = i+1 )
    for (j = 0 ; j < N ; j = j+1 ) {
```

```

    a[i][j] = 1/b[i][j] * c[i][j];
    d[i][j] = a[i][j] + c[i][j];
}

```

修改前的程序在两个地方访问数组 a 和 c，一次在第一个循环里，另一次在第二个循环里。两次循环分隔较远，可能重复失效，即在第一个循环中访问某个元素失效之后，虽已将相应块调入 Cache，但在第二个循环中再次访问该元素时，还可能产生失效。而在修改后的程序中，第二条语句直接利用了第一条语句访问 Cache 的结果，无需做 Load 操作。

4. 分块

这种优化可能是 Cache 优化技术中最著名的一种，它也是通过改进时间局部性来减少失效。我们仍考虑对多个数组的访问，有些数组是按行访问，而有些则是按列访问。无论数组是按行优先还是按列优先存储，都不能解决问题，因为在每一次循环中既有按行访问的也有按列访问的。这种正交的访问意味着前面的变换方法（如内外循环交换）对此无能为力。

分块算法不是对数组的整行或整列进行访问，而是对子矩阵或块进行操作。其目的仍然是使一个 Cache 块在被替换之前，对它的访问次数达到最大。下面这个矩阵乘法程序会帮助我们理解为什么要采用这种优化技术。

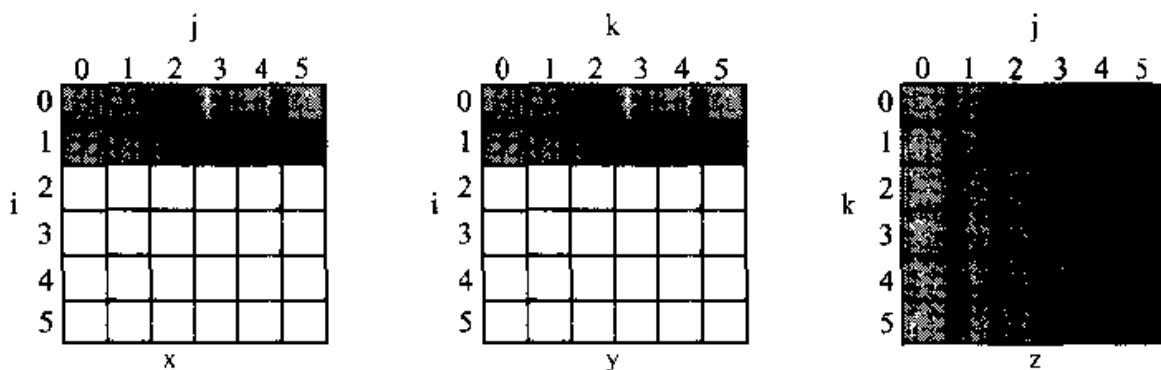
```

/* 修改前 */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1) {
        r = 0;
        for (k = 0; k < N; k = k + 1) {
            r = r + y[i][k] * z[k][j];
        }
        x[i][j] = r;
    }

```

两个内部循环读取了数组 z 的全部 $N \times N$ 个元素，并反复读取数组 y 的某一行中的 N 个元素，所产生的 N 个结果被写入数组 x 的某一行。图 5.13 给出了当 $i=1$ 时对三个数组的访问情况。其中黑色表示最近被访问过，灰色表示早些时候被访问过，而白色表示尚未被访问。

显然，容量失效次数的多少取决于 N 和 Cache 的容量。如果 Cache 只能放下一个 $N \times N$ 的数组和一行 N 个元素，那么至少数组 y 的第 i 行和数组 z 的全部元素能同时放在 Cache 里。如果 Cache 容量还要小的话，对 x 或 z 的访问都可能导致失效。在最坏的情况下， N^3 次操作会导致 $2N^3 + N^2$ 次失效。

图 5.13 当 $i=1$ 时对 x, y, z 三个数组的访问情况

为了保证正在访问的元素能在 Cache 中命中, 把原程序改为只对大小为 $B \times B$ 的子数组进行计算, 即只处理 B 个元素, 而不是像原来那样, 从 x 和 z 的第一个元素开始一直处理到最后一个。 B 称为分块因子(blocking factor)。

```
/* 修改后 */
for ( jj = 0; jj < N; jj = jj+B )
    for ( kk = 0; kk < N; kk = kk+B )
        for ( i = 0; i < N; i = i+1 )
            for ( j = jj; j < min(jj+B-1, N); j = j+1 ) {
                r = 0;
                for ( k = kk; k < min(kk+B-1, N); k = k+1 ) {
                    r = r + y[ i ][ k ] * z[ k ][ j ];
                }
                x[ i ][ j ] = x[ i ][ j ] + r;
            }
        }
```

图 5.14 说明了分块后对三个数组的访问情况。与图 5.13 相比, 所访问的元素个数减少了。只考虑容量失效, 访问存储器的总字数为 $2N^3/B + N^2$ 次, 大约降低到原来的 $1/B$ 。分块技术同时利用了空间局部性和时间局部性, 因为访问 y 时利用了空间局部性, 而访问 z 时利用了时间局部性。

虽然我们的目标一直是减少 Cache 失效, 分块技术还有助于进行寄存器分配。通过减小块大小, 使得寄存器能容纳下整个 Cache 块, 可以把程序中的 Load 和 Store 操作的次数减少到最小。

最后两小节重点讨论了针对 Cache 优化了的编译器和程序可能带来的好处。随着时间的发展, 处理器速度和存储器速度之间的差距越来越大, 这种好处的重要性只能是越来越大。上而已用了不少篇幅讨论降低 Cache 失效率的技术, 下面我们来看看减少平均访问时间的另一个组成部分——失效开销的方法。

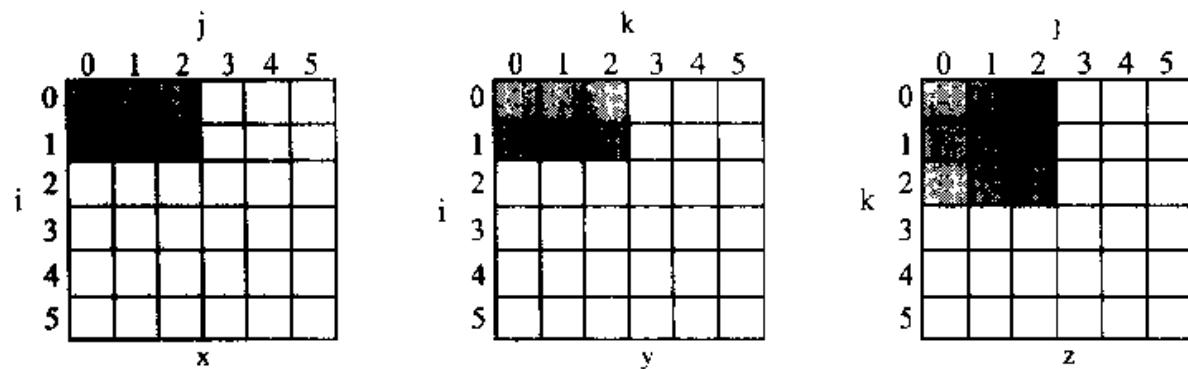


图 5.14 对数组 x,y,z 的访问时间

5.4 减少 Cache 失效开销

以往对 Cache 的研究一直把重点放在减少失效次数上。但是 Cache 性能公式却告诉我们,减少 Cache 失效开销可以跟降低 Cache 失效率一样带来性能上的提高。此外,图 5.2 说明,随着技术的发展,处理器速度的提高要快于 DRAM 速度的提高,这使得 Cache 失效开销的相对代价随时间不断增加。下面将给出解决这一问题的 5 种优化措施。其中最后一种是通过增加另二级 Cache 来减少失效开销,这也许是最令人感兴趣的方法。

5.4.1 让读失效优先于写

提高写直达 Cache 性能最重要的方法是使用一个大小适中的写缓冲器(见 5.2 节)。然而,写缓冲器却导致对存储器的访问复杂化,因为在读失效时,写缓冲器中可能保存有所读单元的最新值。

例 5.10 考虑以下指令序列:

SW	512(R0), R3	; M[512] ← R3	(Cache 索引为 0)
LW	R1, 1024(R0)	; R1 ← M[1024]	(Cache 索引为 0)
LW	R2, 512(R0)	; R2 ← M[512]	(Cache 索引为 0)

假设 Cache 采用写直达法和直接映象,并且地址 512 和 1024 映射到同一块,写缓冲器为 4 个字,试问寄存器 R2 的值总等于 R3 的值吗?

解 根据第三章的术语,这是一个存储器写后读数据相关。下面通过分析对 Cache 的访问来看看会发生什么错误。在执行 Store 指令之后,R3 中的数据被放入写缓冲器。接下来的第一条 Load 指令使用相同的 Cache 索引,因而产生一次失效。第二条 Load 指令欲把地址为 512 的存储单元的值读入寄存器 R2 中,这也会造成一次失效。如果此时写缓冲器还未将数据写入存储单元 512 中,

那么第二条 Load 指令将把错误的旧值读入 Cache 和寄存器 R2。如果不采取适当的预防措施, R2 的值就不会等于 R3 的值。

解决这一问题最简单的方法是推迟对读失效的处理, 直至写缓冲器清空。在写直达 Cache 中, 一个大小只有几个字的写缓冲器在发生读失效时几乎总有数据, 这就增加了读失效的开销。据 MIPS M/1000 的设计者估计, 等待一个大小为 4 个字的缓冲器清空, 会使读失效的平均开销增加 50%。另一种方法是在读失效时检查写缓冲器的内容, 如果没有冲突而且存储器可访问, 就可以继续处理读失效。

在写回法 Cache 中, 也能够减少处理器写回操作的实现开销。假定读失效将替换一个“脏”的存储块。我们可以不像往常那样先把“脏”块写回存储器, 然后再读存储器, 而是先把被替换的“脏”块拷入一个缓冲器, 然后读存储器, 最后再写存储器。这样 CPU 的读访问就能更快地完成。和上面的情况类似, 发生读失效时, 处理器既可以采用等待缓冲区清空的方法, 也可以采用检查缓冲区中各字的地址是否有冲突的方法。

5.4.2 子块放置技术

假如你正在设计一种要封装在 CPU 芯片中的 Cache, 你可能会发现标识太长, 在芯片上放不下, 或者访问速度太慢。解决这个问题的最简单的方法是增加块大小。它可以在不减少存放在 Cache 中的信息量的前提下减少标识所占的存储空间。此外, 它还可能会使失效率有所下降。但是, 失效开销的增大却可能会使之成为一种糟糕的选择。

我们的解决方法叫做子块放置技术(sub-block placement)。把一个 Cache 块划分为若干个小块, 称为子块(sub-blocks)。为每一个子块赋一位有效位, 用于说明该子块中的数据是否有效。因此, 标识匹配并不意味着这个字一定在 Cache 中, 只有当与该字对应的有效位也为“1”时才是。失效时只需从下一级存储器调入一个子块。这样, 一个 Cache 中就有可能有的子块有效, 有的子块无效。显然子块的失效开销小于完整 Cache 块的失效开销。子块可以被看作是地址标识之外的又一级寻址。

图 5.15 给出了一个例子。这里, 每一个块中含有 4 个子块, 每一个子块中含一个单元。在第一个 Cache 块(最上方)中, 所有子块的有效位都为 1, 这等价于普通 Cache 中一个块的有效位为 1 的情况。最后一个 Cache 块(最下方)的情况正好相反, 所有子块的有效位都是 0。在第二块中, 单元 300 和 301 所在的子块有效, 访问它们会命中; 而访问单元 302 和 303 则会发生失效。在第三块中, 访问单元 201 和 203 都将命中。如果不采用这种划分子块的组织方式, 当块大

小与这里的子块大小相同时,就有 16 个块,需 16 个地址标识,而不是 4 个。注意,对于采用子块放置方法的 Cache 来说,块不再是在 Cache 和存储器之间传送数据的最小单位。对于这样的 Cache 而言,块被定义为和地址标识相关的信息单位。

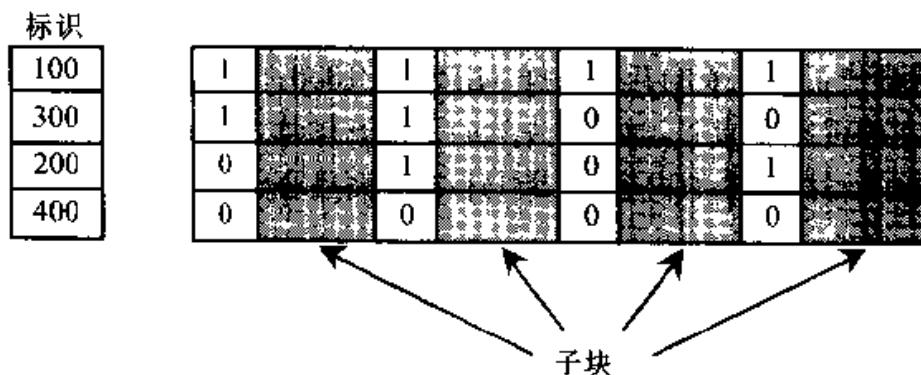


图 5.15 直接映象 Cache 中的子块

图 5.15 说明了使用子块可以减少标识所占的存储空间。如果图中的有效位都用完整的标识来替代,标识所占用的空间将大得多。这正是采用子块放置法的原因。

5.4.3 请求字处理技术

上面两种技术需要额外的硬件来减少失效开销,本小节介绍的技术则不必。当从存储器向 Cache 调入一块时,块中往往只有一个字是 CPU 立即需要的,这个字称为请求字(requested word)。

请求字处理技术正是着眼于这种每次调块时 CPU 只用到请求字的特性。当 CPU 所请求的字到达后,不等整个块都调入 Cache,就可把该字发送给 CPU 并重启 CPU。有两种具体的方案:

(1) 尽早重启(early restart): 在请求字没有到达时,CPU 处于等待状态。一旦请求字到达,就立即发送给 CPU,让等待的 CPU 尽早重启,继续执行。

(2) 请求字优先(requested word first): 调块时,首先向存储器请求 CPU 所要的请求字。请求字一旦到达,就立刻送往 CPU,让 CPU 继续执行,同时从存储器调入该块的其余部分。请求字优先也称为回绕读取(wrapped fetch)或关键字优先(critical word first)。

一般来说,这些技术仅当 Cache 块很大时才有效。因为当 Cache 块较小时,用不用这些技术,失效开销差别不大。此外,在采用这些方法时,若下一条指令正好访问 Cache 块的另一部分(以请求字为界,把 Cache 分为上下两部分。请求字属于其中一部分),则只能节省一个时钟周期。因为只有得到请求字的指令在流水线中可以前进,下一条指令必须停下来等待所需的数据。

5.4.4 非阻塞 Cache 技术

采用尽早重启动技术时,CPU 在继续执行之前仍需等待请求字到达。有些流水方式的机器采用记分牌或 Tomasulo 类(见第四章)控制方法,允许指令乱序执行(后面的指令可以跨越前面的指令先执行),CPU 无须在 Cache 失效时停顿。例如,失效发生后,CPU 在等待数据 Cache 给出数据的同时可以进行从指令 Cache 中取指令等工作。如果采用非阻塞(nonblocking)Cache 或非锁定(lock-up-free)Cache 技术,就可能把 CPU 的性能提高得更多,因为这种 Cache 在失效时仍允许 CPU 进行其他的命中访问。这种“失效下命中”(hit under miss)的优化措施在 Cache 失效时,不是完全拒绝 CPU 的访问,而是能处理部分访问,从而减少了实际失效开销。如果更进一步,让 Cache 允许多个失效重叠,即支持“多重失效下的命中”(hit under multiple miss)和“失效下失效”(miss under miss),则可进一步减少实际失效开销。不过,这种方法只有在存储器能处理多个失效的情况下才能带来好处。请注意,“失效下命中”措施大大增加了 Cache 控制器的复杂度,因为这时可能有多个访存同时进行。

可以同时处理的失效个数越多,所能带来的性能上的提高就越大。对于 SPEC92 典型程序,图 5.16 给出了对于不同的重叠失效个数,数据 Cache 的平均存储器等待时间(以周期为单位)与阻塞 Cache 平均存储器等待时间的比值。所考虑的 Cache 采用直接映象,容量为 8 KB,块大小为 32 字节,失效开销为 16 个时钟周期。测试程序为 18 个 SPEC92 程序。前 14 个测试程序为浮点程序,后 4 个为整数程序。在重叠失效个数为 1、2 和 64 的情况下,浮点程序的平均比值分别为 76%、51% 和 39%,而整数程序的平均比值则分别为 81%、78% 和 78%。从图中可以看出,对于浮点程序来说,重叠失效个数越多,性能提高就越多;但对于整数程序来说,重叠次数对性能提高影响不大,简单的“一次失效下命中”就几乎可以得到所有的好处。

例 5.11 对于图 5.16 所描述的 Cache,在两路组相联和“一次失效下命中”这两种措施中,哪一种对浮点程序更重要? 对整数程序的情况如何? 假设 8 KB 数据 Cache 的平均失效率为:对于浮点程序,直接映象 Cache 为 11.4%,两路组相联 Cache 为 10.7%;对于整数程序,直接映象 Cache 为 7.4%,两路组相联 Cache 为 6.0%。并且假设平均存储器等待时间是失效率和失效开销的积,失效开销为 16 个时钟周期。

解 对于浮点程序,平均存储器等待时间为

$$\text{失效率}_{\text{直接映象}} \times \text{失效开销} = 11.4\% \times 16 = 1.82$$

$$\text{失效率}_{\text{两路组相联}} \times \text{失效开销} = 10.7\% \times 16 = 1.71$$

$$1.71 / 1.82 = 0.94$$

即两路组相联 Cache 的平均存储器等待时间是直接映象 Cache 的 94%，而支持“一次失效下命中”技术的直接映象 Cache 的平均存储器等待时间是直接映象 Cache 的 76%，所以对于浮点程序来说，支持“一次失效下命中”的直接映象 Cache 比两路组相联 Cache 的性能更高。

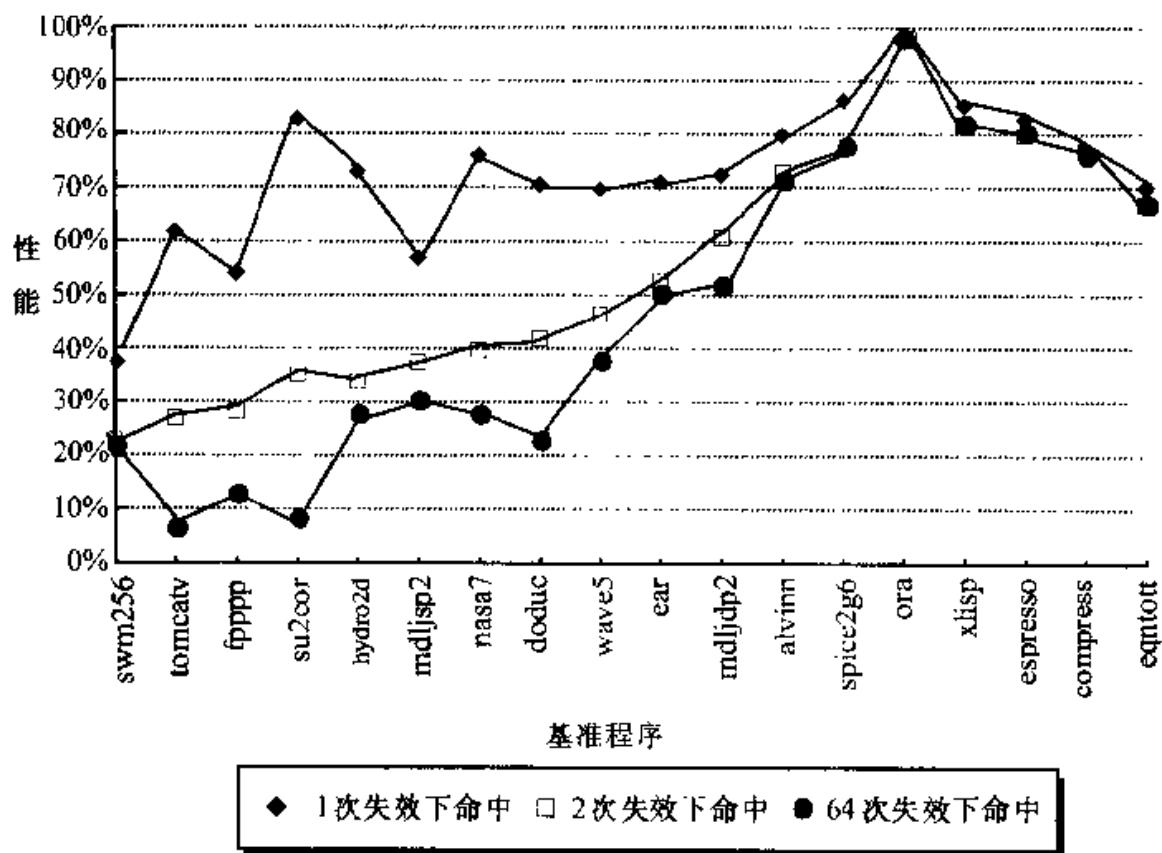


图 5.16 “多重失效下命中”Cache 的平均存储器等待时间与阻塞 Cache 的平均等待时间的比值

对于整数程序：

$$\text{失效率}_{\text{直接映象}} \times \text{失效开销} = 7.4\% \times 16 = 1.18$$

$$\text{失效率}_{\text{两路组相联}} \times \text{失效开销} = 6.0\% \times 16 = 0.96$$

$$0.96 / 1.18 = 0.81$$

即整数计算中，两路组相联 Cache 的平均存储器等待时间是直接映象 Cache 的 81%，而“一次失效下命中”技术把平均存储器等待时间降低到直接映象 Cache 的 81%。因此，对于整数程序来说，这两种技术的性能相同。

另外，“失效下命中”方法有一个潜在优点：它不会影响命中时间，而组相联却会。

5.4.5 采用两级 Cache

前面四种降低失效开销的技术都与 CPU 关系密切。这里讨论的第二级

Cache 技术则不管 CPU, 而把重点放在 Cache 和主存的接口上。

CPU 和主存之间的性能差距给体系结构设计者们提出了以下问题: 为了克服这个越来越大的性能差距, 使存储器和 CPU 的性能匹配, 是应该把 Cache 做得更快, 还是应该把 Cache 做得更大? 一种答案是: 二者兼顾。通过在原有 Cache 和存储器之间增加另一级 Cache, 构成两级 Cache。可以把第一级 Cache 做得足够小, 使其速度和快速 CPU 的时钟周期相匹配, 而把第二级 Cache 做得足够大, 使它能捕获更多本来需要到主存去的访问, 从而降低实际失效开销。

尽管增加一级存储层次在概念上直观、简单, 但性能分析却变得复杂了。有关第二级 Cache 的定义不太好理解。用下标 L1 和 L2 分别表示第一级和第二级 Cache, 原有的公式就变为

$$\text{平均访存时间} = \text{命中时间}_{L1} + \text{失效率}_{L1} \times \text{失效开销}_{L1}$$

$$\text{失效开销}_{L1} = \text{命中时间}_{L2} + \text{失效率}_{L2} \times \text{失效开销}_{L2}$$

所以,

$\text{平均访存时间} = \text{命中时间}_{L1} + \text{失效率}_{L1} \times (\text{命中时间}_{L2} + \text{失效率}_{L2} \times \text{失效开销}_{L2})$
在这个公式里, 第二级 Cache 的失效率是以在第一级 Cache 中不命中而到达第二级 Cache 的访存次数为分母来计算的。为避免二义性, 对于第二级 Cache 系统采用以下术语:

(1) 局部失效率

对于某一级 Cache 来说,

局部失效率 = 该级 Cache 的失效次数 / 到达该级 Cache 的访存次数

对于第二级 Cache 来说, 就是上面的失效率_{L2}。

(2) 全局失效率

对于某一级 Cache 来说,

全局失效率 = 该级 Cache 的失效次数 / CPU 发出的访存总次数

使用上面公式中的变量, 第二级 Cache 的全局失效率就是

$$\text{全局失效率}_{L2} = \text{失效率}_{L1} \times \text{失效率}_{L2}$$

因为访存都要经过第一级 Cache, 所以它的局部失效率比较大。对于两级 Cache, 全局失效率是一种比局部失效率更有用的衡量指标, 它指出了在 CPU 发出的访存中, 究竟有多大比例是穿过各级 Cache 最终到达存储器的。

例 5.12 假设在 1 000 次访存中, 第一级 Cache 失效 40 次, 第二级 Cache 失效 20 次。试问: 在这种情况下, 该 Cache 系统的局部失效率和全局失效率各是多少?

解 第一级 Cache 的失效率(全局和局部)是 40/1 000, 即 4%; 第二级 Cache 的局部失效率是 20/40, 即 50%, 第二级 Cache 的全局失效率是 20/1 000, 即 2%。

请注意,上述公式是针对读写操作混合而言的,而且假设第一级 Cache 采用写回法。当采用写直达法时,第一级 Cache 将不仅把失效,而且还把所有的写访问都送往第二级 Cache。另外还会使用一个写缓冲器。

对于第二级 Cache,我们有以下结论:

(1) 在第二级 Cache 比第一级 Cache 大得多的情况下,两级 Cache 的全局失效率和容量与第二级 Cache 相同的单级 Cache 的失效率非常接近。这时可以利用前面关于单级 Cache 的分析和知识。

(2) 局部失效率不是衡量第二级 Cache 的一个好指标,因为它是第一级 Cache 失效率的函数,可以通过改变第一级 Cache 而使之变化,而且不能全面反映两级 Cache 体系的性能。因此,在评价第二级 Cache 时,应用全局失效率这个指标。

下面考虑第二级 Cache 的参数。第一级 Cache 和第二级 Cache 之间的首要区别是第一级 Cache 的速度会影响 CPU 的时钟频率,而第二级 Cache 的速度只影响第一级 Cache 的失效开销。因此,对于第二级 Cache,在设计时可以有更多的考虑空间,许多不适合于第一级 Cache 的方案对于第二级 Cache 却可以使用。第二级 Cache 的设计只有两个问题需要权衡:它能否降低 CPI 中的平均访存时间部分?它的成本是多少?

首先,研究第二级 Cache 的容量。因为第一级 Cache 中的所有信息都会出现在第二级 Cache 中,第二级 Cache 的容量应比第一级的大许多。如果第二级 Cache 只是稍大一点,局部失效率将很高。因此,第二级 Cache 的容量一般很大,和过去计算机的主存一样大!大容量意味着第二级 Cache 可能实际上没有容量失效,只剩下一些强制性失效和冲突失效。现在的问题是:相联度(组相联中的路数 n)对于第二级 Cache 的作用是否会更大?

例 5.13 给出有关第二级 Cache 的以下数据:

- (1) 两路组相联命中时间增加 $10\% \times \text{CPU 时钟周期}$
- (2) 对于直接映象,命中时间 $L_2 = 10$ 个时钟周期
- (3) 对于直接映象,局部失效率 $L_2 = 25\%$
- (4) 对于两路组相联,局部失效率 $L_2 = 20\%$
- (5) 失效开销 $L_2 = 50$ 个时钟周期

试问第二级 Cache 的相联度对失效开销的影响如何?

解 对一个直接映象的第二级 Cache 来说,第一级 Cache 的失效开销为

$$\text{失效开销}_{\text{直接映象}, L_1} = 10 + 25\% \times 50 = 22.5 \text{ 个时钟周期}$$

对于两路组相联第二级 Cache 来说,命中时间增加了 $10\% (0.1)$ 个时钟周期,故第一级 Cache 的失效开销为

$$\text{失效开销}_{\text{两路组相联}, L_1} = 10.1 + 20\% \times 50 = 20.1 \text{ 个时钟周期}$$

在实际机器中,第二级 Cache 几乎总是和第一级 Cache 以及 CPU 同步操

作。相应地,第二级 Cache 的命中时间必须是时钟周期的整数倍。如果幸运的话,可以把该命中时间取整为 10 个时钟周期,否则就只好取整为 11 个时钟周期,但不管怎样,都比直接映象第二级 Cache 好:

$$\text{失效开销}_{\text{两路组相联}, 1:1} = 10 + 20\% \times 50 = 20.0 \text{ 个时钟周期}$$

$$\text{失效开销}_{\text{两路组相联}, 1:1} = 11 + 20\% \times 50 = 21.0 \text{ 个时钟周期}$$

可以利用 5.3 节介绍的技术来减少第二级 Cache 的失效率,从而达到减少失效开销的目的。提高相联度和伪相联方法都值得考虑,因为它们对第二级的命中时间影响很小,而且平均访存时间中很大一部分是由于第二级 Cache 失效而产生的。虽然较大容量的第二级 Cache 通过把数据分布到更多的 Cache 块中消除了一些冲突失效,但它同时也减少了容量失效,所以在直接映象的第二级 Cache 中,冲突失效所占的比例依然很大。

同样可以采用增加第二级 Cache 块大小的方法来减少失效。前面已经得出这样的结论:增加 Cache 块的大小也增加了小容量 Cache 的冲突失效次数(因为可能没有足够的位置来存放数据),导致失效率上升。但对于大容量的第二级 Cache 来说,这一点并不成为问题。而且由于访存时间相对来说较长,所以 64 字节、128 字节,甚至 256 字节的块大小都是第二级 Cache 经常使用的。图 5.17 给出了在存储器总线宽度为相对较窄的 32 位时,CPU 执行时间随第二级 Cache 块大小的变化而改变的情况。

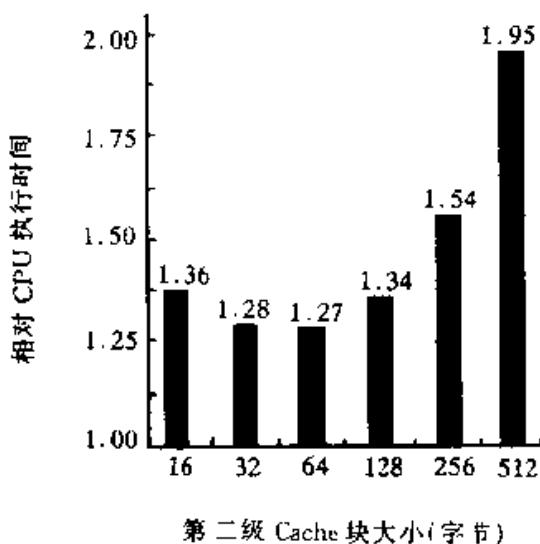


图 5.17 相对执行时间和第二级 Cache 块大小的关系

需要考虑的另一个问题是第一级 Cache 中的数据是否总是同时存在于第二级 Cache 中。如果是的话,就说第二级 Cache 具有多级包容性(multilevel inclusion property)。多级包容性是我们所希望的,因为它便于实现 I/O 和 Cache 之间内容一致性的检测。

为了减少平均访存时间,可以让容量较小的第一级 Cache 采用较小的块,而让容量较大的第二级 Cache 采用较大的块。在这种情况下,仍可实现包容性,但在处理第二级 Cache 失效时要做更多的工作:替换第二级 Cache 中的块时,必须作废所有映象到该块的第一级 Cache 中的块。这样不但会使第一级 Cache 失效率有所增加,而且会造成不必要的作废。如果结合使用其他一些性能优化技术(如非阻塞的第二级 Cache),包容性会进一步增加复杂度。

综合上述考虑,Cache 设计的本质是在快速命中和减少失效次数这两个方面进行权衡。大部分优化措施都是在改进一方的同时损害另一方。对于第二级 Cache 而言,由于它的命中次数比第一级 Cache 少得多,所以重点就转移到了减少失效次数上。这就导致了更大容量、更高相联度和块大小更大的 Cache 的出现。

5.5 减少命中时间

到目前为止,我们已经讨论了通过减少失效次数(5.3 节)和减少失效开销(5.4 节)来改进 Cache 性能的办法,现在我们来讨论减少命中时间的技术,它也是平均访存时间的三个组成部分之一。

命中时间的重要性在于它影响到处理器的时钟频率。在当今的许多机器中,往往是 Cache 的访问时间限制了处理器的时钟频率,即使在把访问 Cache 分为几个时钟周期来完成的机器中也是如此。因此减少命中时间不仅对减小平均访存时间很重要,而且对其他许多方面也是非常重要的。本节先讨论两种减少命中时间的通用技术,然后论述一个适用于写命中的优化措施。

5.5.1 容量小、结构简单的 Cache

采用容量小而且结构简单的 Cache,可以有效地提高 Cache 的访问速度。用地址的索引部分访问标识存储器,读出标识并与地址进行比较,是 Cache 命中过程中最耗时的部分。我们在第一章中就指出,硬件越简单,速度就越快。小容量 Cache 对减少命中时间当然有益。而且应使 Cache 足够小,以便可以与处理器做在同一芯片上,以避免因芯片外访问而增加时间开销。这一点是非常重要的。某些设计采用了一种折衷方案:把 Cache 的标识放在片内,而把 Cache 的数据存储器放在片外,这样既可以实现快速标识检测,又能利用独立的存储芯片来提供更大的容量。另一个建议是要保持 Cache 结构简单,例如采用直接映象 Cache。直接映象 Cache 的主要优点是可以让标识检测和数据传送重叠进行,这样可以有效地减少命中时间。所以,为了得到高速的时钟频率,第一级 Cache 应选用容量小且结构简单的设计方案。

5.5.2 虚拟 Cache

在采用虚拟存储器的机器中,每次访存都必须进行虚地址到实地址的变换,即将 CPU 发出的虚地址转换为物理地址。即便是容量小、结构简单的 Cache,也必须解决这个问题。

和失效相比,Cache 命中发生的频度高得多。按照“快速实现常见事件”的指导思想,应在 Cache 中使用虚拟地址。这样的 Cache 称为虚拟 Cache(virtual Cache),而物理 Cache(physical Cache)则是指那些使用物理地址的传统 Cache。直接用虚拟地址访问 Cache,在命中时消除了用于地址转换的时间。然而,人们在设计时并非都采用虚拟 Cache,为什么呢?其原因之一,是每当进行进程切换时,由于新进程的虚拟地址(有可能与原进程的相同)所指向的物理空间与原进程的不同,故需要清空 Cache。图 5.18 说明了这种清空对失效率的影响。解决这个问题的一种办法是在地址标识中增加一个进程标识符字段(PID),这样多个进程的数据可以混和存放于一个 Cache 中,由 PID 指出 Cache 中各块的数据是属于哪个程序的。为减少 PID 的位数,PID 经常是由操作系统指定。对于一个进程,操作系统从循环使用的几个数字中指定一个作为其 PID。进程切换时,仅当某个 PID 被重用(即该 PID 以前已被分配给了某个进程,现又把它分配给另一个进程)时,才需清空 Cache。

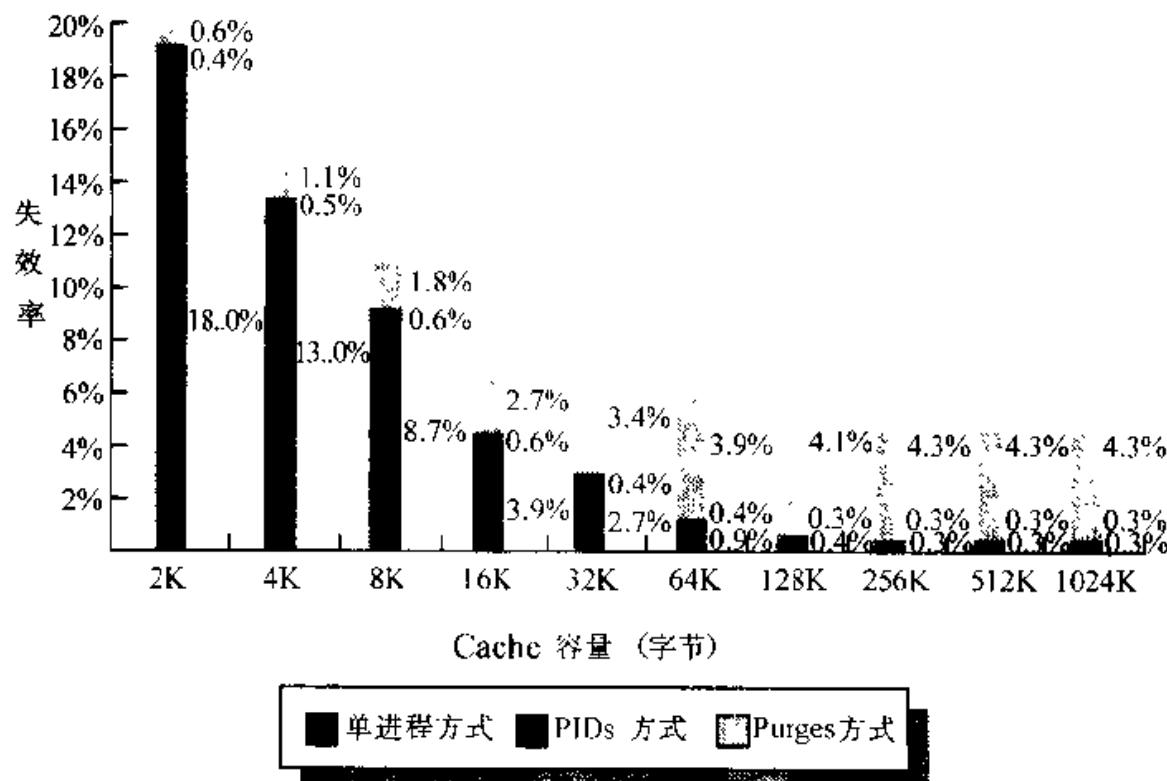


图 5.18 对于三种方式,虚地址 Cache 在不同容量下的失效率

图 5.18 说明了采用 PID 所带来的失效效率上的改进。图 5.18 给出了在以下三种情况下各种容量的虚拟 Cache 的失效效率：没有进程切换(单进程)，允许进程切换并使用进程标识符(PIDs)，允许进程切换但不使用进程标识符(purge)。从图中可看出：和单进程相比，PIDs 的绝对失效效率增加 0.3% ~ 0.6%；而和 purge 相比，PIDs 的绝对失效效率减少 0.6% ~ 4.3%，图中的数据是在 VAX 机上针对 Ultrix 操作系统统计的，并假设 Cache 采用直接映象，块大小为 16 字节。

虚拟 Cache 没有流行起来的一个原因是操作系统和用户程序对于同一个物理地址可能采用两种以上不同形式的虚拟地址来访问，这些地址称为同义(synonym)或别名(alias)。它们可能会导致同一个数据在虚拟 Cache 中存在两个副本。如果其中一个被修改，那么再使用另一个数据就是错误的。这种情况在物理 Cache 中是不会发生的，因为这些访问首先会把虚拟地址转换到同一物理地址，从而找到同一个物理 Cache 块。有一种用硬件解决这个问题的方法，叫做反别名法(anti-aliasing)，它保证每一个 Cache 块对应于唯一的一个物理地址。

如果强行要求别名的某些地址位相同，就可以用软件很容易地解决这一问题。例如，SUN 公司的 UNIX 要求所有使用别名的地址最后 18 位都相同，这种限制被称为页着色(page coloring)。这一限制使得容量不超过 2^{18} 字节(256 KB)的直接映象 Cache 中不可能出现一个 Cache 块有重复物理地址的情况。所有别名将被映象到同一个 Cache 块位置。

对于虚拟地址，最后还应考虑 I/O。I/O 通常使用物理地址，所以为了与虚拟 Cache 打交道，需要把物理地址映象为虚拟地址。

另一种实现快速命中的技术是把地址转换和访问 Cache 这两个过程分别安排到流水线的不同级中，从而使时钟周期加快，但这同时也增加了命中所需的时间。这种方法增加了访存的流水线级数，增加了分支预测错误时的开销，而且使得从 Load 指令流出到数据可用之间所需的时钟周期数增加。

还有一种方法，既能得到虚拟 Cache 的好处，又能得到物理 Cache 的优点。它直接用虚地址中的页内位移(页内位移在虚→实地址的变换中保持不变)作为访问 Cache 的索引，但标识却是物理地址。CPU 发出访存请求后，在进行虚→实地址变换的同时，可并行进行标识的读取。在完成地址变换之后，再把得到的物理地址与标识进行比较。

这种虚拟索引、物理标识方法的局限性，在于直接映象 Cache 的容量不能超过页的大小。Alpha AXP 21064 采用了这种方法，其 Cache 容量为 8 KB，最小页大小为 8 KB，所以可以直接从虚地址的页内位移部分中得到 8 位的索引(块大小为 32 字节)。

为了既能实现大容量的 Cache，又能使索引位数比较少，以便能直接从虚拟地址的页内位移部分得到，我们可以采用提高相联度的办法。这一点可以从下

面的公式中看出：

$$2^{\text{index}} = \frac{\text{Cache 容量}}{\text{块大小} \times \text{相联度}}$$

下面举一个极端的例子——IBM 3033 的 Cache。虽然研究结果已表明 8 路以上的组相联对减少失效率没多大好处，但 IBM 3033 的 Cache 仍采用了 16 路组相联，其主要好处是可以采用更大的 Cache。尽管 IBM 体系结构限制页的大小为 4 KB，但 16 路组相联却可以用物理索引对 64 KB(16×4 KB)的 Cache 进行寻址。图 5.19 给出了索引和页内位移的关系。页大小为 4 KB 意味着地址的最后 12 位不必进行转换，因此其中某些位可以用作访问 Cache 的索引。

我们也可以不采用提高相联度的方法，而由操作系统来实现页着色。操作系统通过使虚页地址和物理页地址的最后几位相同来实现这一点。采用这种方法时，索引的位数可以比先前的页内位移方法的索引位数多，并且仍然是对物理地址进行比较。

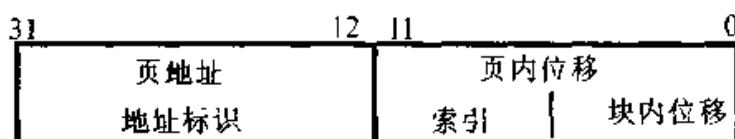


图 5.19 IBM 3033 的 Cache 中索引和页内位移的关系

另一种办法，是用一小块硬件来猜测虚页地址的后几位映象到什么样的物理地址。这块硬件可以是一个小表格，它对虚页地址进行散列变换。由此得到的猜测结果和地址的物理部分（页内位移）一起构成访问 Cache 的索引。用此索引读出相应的标识，并与经地址转换得到的物理地址进行比较，判断是否匹配。如果标识匹配，则发生了一次命中。如果不匹配，则要么就是数据不在 Cache 中，要么就是关于虚页地址最后几位的映象的猜测是错的。这时 Cache 一般会用正确的索引重新判断这次访存究竟是命中还是真的失效。

上述采用容量小且结构简单的 Cache 以及避免地址转换延迟的技术不仅能提高写命中的速度，而且能提高读命中的速度。下一节着重讨论仅能加快写命中的技术。

5.5.3 写操作流水化

写命中通常比读命中花费更多的时间，因为在写入数据之前必须先检测标识，否则就有可能将数据写到错误的单元中。我们可以通过把写操作流水化来提高写命中的速度。Alpha AXP 21064 和其他一些机器采用了这种技术。图 5.20 为说明流水化“写”的硬件结构框图。这里，标识和数据是分开存放的，因而能分别

独立地访问。每个写操作的工作被分为两个阶段来完成：第一阶段进行标识比较，并把标识和数据存入延迟写缓冲器中；第二阶段再进行数据的写入（若命中的话）。这两个阶段按流水方式工作。这样，当前“写”的标识比较就可以和上一个“写”的数据写入并行起来，实现每个时钟周期完成一个写操作（从 CPU 的角度）。该流水线与读操作无关。因为读操作的标识比较与数据读出本来就可以并行进行，所以无需专门的硬件支持。

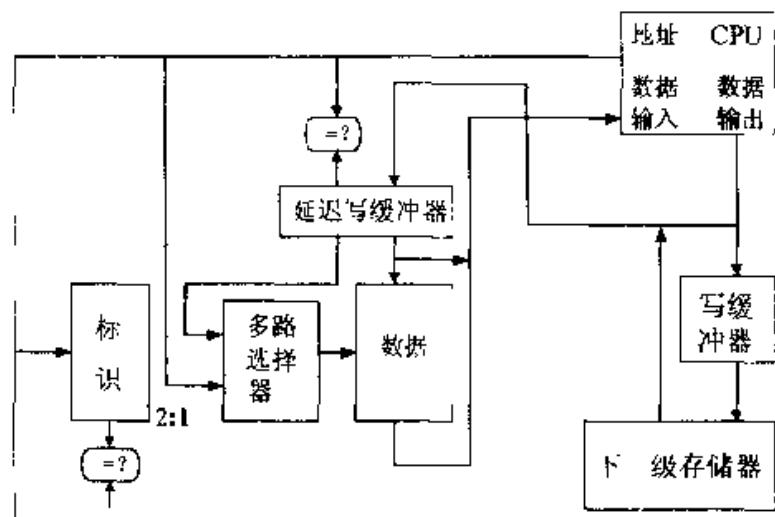


图 5.20 流水化写的硬件组织结构

5.5.4 Cache 优化技术小结

5.3 到 5.5 节中论述的减少失效率、失效开销和命中时间的技术通常会影响平均访存时间公式的其他组成部分，而且会影响存储层次的复杂性。表 5.9 对这些技术作了个小结，并估计了它们对复杂性的影响。表中“+”号表示这一技术改进了相应指标，“-”号表示它使该指标变差，而空格栏则表示它对该指标无影响。从表中可以看出，没有什么技术能同时改进两项或三项指标。表中关于复杂性的衡量是主观化的，0 表示最容易，3 表示最复杂。

表 5.9 Cache 优化技术小结

优化技术	失效率	失效开销	命中时间	硬件复杂度	评 价
增加块大小	+ -			0	实现容易；RS/6000 550 采用了 128 字节
提高相联度	+ -		-	1	MIPS R10000 为 4 路组相联
Victim Cache	+ -			2	HP7200 中采用了类似的技术

续表

优化技术	失效率	失效开销	命中时间	硬件复杂度	评 价
伪相联 Cache	+	-	-	2	已应用于 MIPS R10000 的第二级 Cache
硬件预取指令和数据	-	-	-	2	数据预取比较困难;仅被几台机器采用,如 Alpha 21064
编译器控制的预取	-	-	-	3	需采用非阻塞 Cache;有几种机器支持它
用编译技术减少 Cache 失效次数	-	-	-	0	向软件提出了新要求;有些机器提供了编译器选项
使读失效优先于写	-	+	-	1	在单处理机上实现容易,被广泛使用
子块放置	-	+	-	1	主要用于减少标识的数目
尽早重启机和关键字优先	-	+	-	2	已应用于 MIPS R10000 和 IBM 620
非阻塞 Cache	-	+	-	3	已应用于 Alpha 21064 和 R10000 中
第二级 Cache	-	+	-	2	硬件代价大;两级 Cache 的块大小不同时实现困难;被广泛采用
容量小且结构简单的 Cache	-	-	+	0	实现容易,被广泛使用
对 Cache 进行索引时不必进行地址变换	-	-	+	2	对于小容量 Cache 来说实现容易,已应用于 Alpha 21064
流水化写	-	-	+	1	已应用于 Alpha 21064

5.6 主存

主存是存储层次中紧接着 Cache 下面的一个层次。主存是数据输入的目的地，也是数据输出的发源地，它既被用来满足 Cache 的请求，也被用作 I/O 接口。主存的性能主要用延迟和带宽来衡量，以往，Cache 主要关心的是主存的延迟（它影响 Cache 的失效开销），而 I/O 则主要关心主存的带宽。随着第二级 Cache 的广泛使用，主存带宽对于 Cache 来说也变得重要了，这是因为第二级 Cache 的块大小较大的缘故。实际上，Cache 设计者可以通过增加 Cache 块的大小来利用高存储带宽。

5.6.1 存储器技术

传统上，人们一直用访问时间(access time)和存储周期(cycle time)这两项指标来衡量存储器延迟。访问时间是指从发出读请求到所需的数据到达为止所需的时间，而存储周期则是指两次相邻访存请求之间的最短时间间隔。存储周期比访问时间长，其原因之一是，在进行下一次访存之前，存储器需等待地址线进入稳定状态。

当早期 DRAM 的容量不断增加时，越来越多的地址线使芯片封装的成本越来越成为问题。解决这个问题的办法是进行地址线复用，这样可将芯片的地址引脚数减少一半。每次访存时，先发送一半的地址，称为行选通(Row Access Strobe 或 RAS)；接着发送另一半地址，称为列选通(Column Access Strobe 或 CAS)。这些名称是根据芯片的内部结构取定的，因为存储单元被组织成一个按行和按列寻址的矩形阵列。

DRAM 有一个特别的要求——刷新。这一要求来自 DRAM 的第一个字母“D”所代表的特性，即动态(dynamic)。DRAM 的优点之一是只用一个晶体管来存储一位信息。但读取这一位时，会破坏其中的信息。为防止信息丢失，每一位都必须定期地被刷新。幸运的是，通过读取某一行，就可以将该行中的所有位都刷新。因此，存储系统中每个 DRAM 在一定的时间窗口(例如 8 ms)内，都必须把它的每一行都访问一遍。存储控制器中包含定期刷新 DRAM 的硬件。

需要刷新意味着有时存储系统是不可访问的，因为这时它正在向每个芯片发送刷新信号。DRAM 刷新一次所需的时间一般是对 DRAM 的每一行进行一次完全访问(RAS 和 CAS)所需的时间。由于在概念上 DRAM 的存储矩阵是正方形，一次刷新所需的总步数通常就是 DRAM 容量的平方根。DRAM 设计者们努力把用于刷新的时间控制在总时间的 5% 以内。

与 DRAM 相反，SRAM 不需要刷新。SRAM 的第一个字母“S”代表静态

(static)。为防止信息在读出时被破坏, SRAM 中每位使用 4~6 个晶体管。DRAM 电路的动态特性要求数据在读后被写回, 因此其访问时间和存储周期不同, 而且 DRAM 需要定期刷新。SRAM 则不一样, 其访问时间和存储周期没有差别, 也不需要刷新。DRAM 设计的重点是大容量, 而 SRAM 设计既关心速度也关心容量(正是因为这样, SRAM 的地址线不能被复用)。当采用同一档次的实现技术时, DRAM 的容量大约是 SRAM 容量的 4~8 倍, SRAM 的存储周期比 DRAM 的快 8~16 倍, 但价格也要贵 8~16 倍。

几乎所有自 1975 年以来售出的计算机主存都是采用半导体 DRAM 做成的, 而几乎所有的 Cache 都是采用 SRAM。不过 Cray 系列巨型机是个例外, 例如 C-90 就采用 SRAM 作主存。

Amdahl 提出了一个经验规则: 为了保持系统平衡, 存储容量应随 CPU 速度的提高而线性增加。CPU 的设计者们指望依靠 DRAM 来满足这一需求。他们预期存储容量每 3 年提高 4 倍, 即每年提高 60%。然而不幸的是, DRAM 性能的提高却慢得多。从表 5.10 可以看出, DRAM 的行访问时间是每一代(3 年)约减少 22%, 即每年减少 7%。从表 5.10 中还可以看出, 随着 DRAM 从 NMOS 型变为 CMOS 型(1986 年), 列访问时间减少了一半。

表 5.10 各代 DRAM 的典型时间参数

推出年份	芯片容量	行选通(RAS)		列选通(CAS)	周期时间
		最慢的 DRAM	最快的 DRAM		
1980	64 Kb	180 ns	150 ns	75 ns	250 ns
1983	256 Kb	150 ns	120 ns	50 ns	220 ns
1986	1 Mb	120 ns	100 ns	25 ns	190 ns
1989	4 Mb	100 ns	80 ns	20 ns	165 ns
1992	16 Mb	80 ns	60 ns	15 ns	120 ns
1995	64 Mb	65 ns	50 ns	10 ns	90 ns

正如我们在图 5.2 中所看到的, CPU 和 DRAM 之间的性能差距在今天显然还是一个问题。Amdahl 定律告诫我们, 如果忽略计算的一个部分, 而去努力提高其余部分的速度, 其收效将是甚微的。前几节已介绍了如何通过改进 Cache 的结构来缩小这种性能差距, 但是仅仅增加 Cache 容量或增加 Cache 的级数并不一定是消除这种差距的经济有效的方法, 还需要采用新型的主存组织结构。下一小节将讨论几种能提高带宽的存储器组织技术。

5.6.2 提高主存性能的存储器组织结构

和减少延迟相比,采用新型的组织结构来提高存储带宽更容易。Cache 可以通过增加 Cache 块的大小来利用主存带宽的增加,因为在高带宽的情况下,块大小增大并不会使失效开销增加多少。

下面以处理 Cache 失效为例来说明各种存储器组织结构的好处。假设基本存储器结构的性能为:

- 送地址需 4 个时钟周期
- 每个字的访问时间为 24 个时钟周期
- 传送一个字的数据需 4 个时钟周期

如果 Cache 块大小为 4 个字,则失效开销为 $4 \times (4 + 24 + 4) = 128$ 个时钟周期,存储器的带宽为每个时钟周期 $1/8(16/128)$ 字节。

图 5.21 画出了几种提高存储系统速度的方案。图 5.21(a)是单字宽的存储器结构,这是一种最简单的方案。所有部件的宽度都是一个字;图 5.21(b)是多字宽存储器结构,它采用了宽度较大的存储器总线和 Cache;图 5.21(c)是多体交叉存储器结构,在这种结构中,总线和 Cache 的宽度都较窄,但存储器按交叉方式工作。下面先介绍对于一般存储器(包括 DRAM 和 SRAM)都适用的 4 种技术,然后在最后一小节介绍专用于 DRAM 的技术。

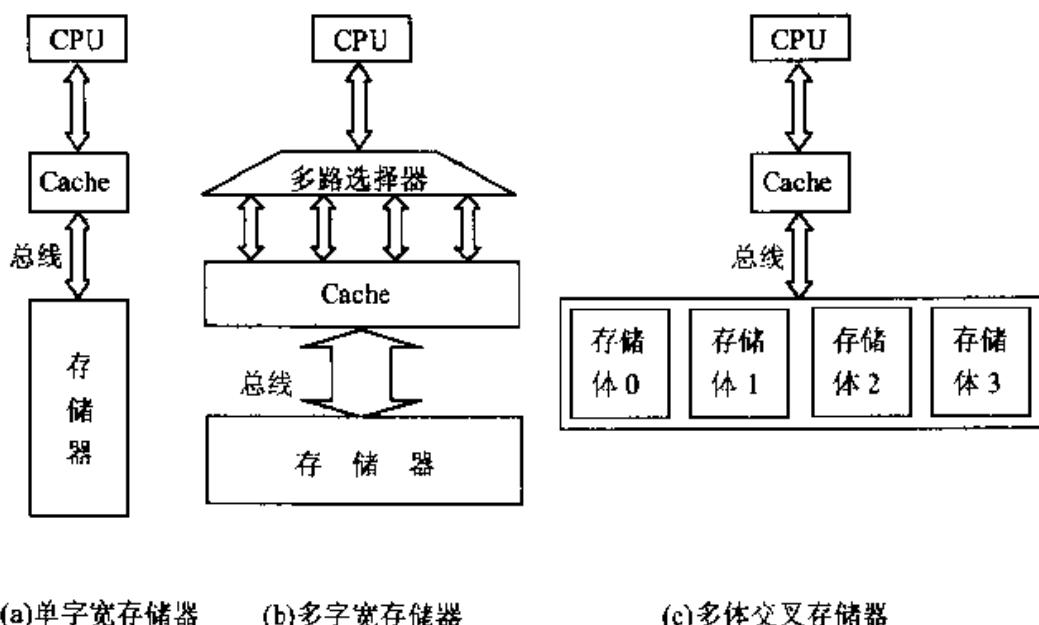


图 5.21 获得更高主存带宽的三种技术(举例)

1. 增加存储器的宽度

这是提高存储器带宽的最简单的方法。

由于 CPU 的大部分访存都是单字宽的,所以第一级 Cache 的宽度通常为一

个字。在不具有第二级 Cache 的计算机系统中,主存的宽度一般与 Cache 的宽度相同。因此,如果把 Cache 和主存的宽度增加为原来的 2 倍或 4 倍,则主存的频带也就相应地增加为原来的 2 倍或 4 倍。对于宽度为 2 个字的主存来说,上述例子中的失效开销就会从 4×32 个时钟周期降到 2×32 个周期,带宽变为每个时钟周期 $1/4$ 字节。当主存宽度为 4 个字时,失效开销就只剩下 1×32 个周期,带宽变为每个时钟周期 $1/2$ 字节。

但是,这种方法也存在一些不足之处。首先,它会增加 CPU 和存储器之间的连接通路(通常称为存储器总线)的宽度,使其实现代价提高。由于 CPU 访问 Cache 仍然是每次访问一个字,所以 CPU 和 Cache 之间需要有一个多路选择器,而且这个多路选择器可能会处在关键路径上。采用第二级 Cache 可以解决这个问题。这时可让第一级 Cache 的宽度为一个字,而在第一级 Cache 和第二级 Cache 之间放置一个多路选择器,这样它就不在关键路径上了。这种方法的第一个缺点,是当主存宽度增加后,用户扩充主存时的最小增量也增加了相应的倍数。

这种方法的最后一个缺点是,在具有纠错功能的存储器中实现对一行(一次可并行读出的数据)中部分数据的写入比较复杂。当进行这种写入时,相应行中其余的数据也必须读出,以便在写入新数据后,重新计算纠错码,并写回存储器中。如果纠错码是对于整行计算的,则增加存储器的宽度会增加这种“读出——修改——写回”操作的频度,因为宽度增加会使更多的写操作变成部分“写”。由于大多数的“写”为单字写,所以许多增加主存宽度的设计方案都是对子每 32 位数据就形成一个独立的纠错码。

Alpha AXP 21064 是采用宽主存的一个例子。其第二级 Cache、存储器总线以及存储器都是 256 位宽的。为了使用户在购买小容量存储器时也不损失宽度,DEC 不仅针对大容量存储器系统的情况出售当代的 DRAM(容量较大),而且也针对小容量存储器的情况出售过去几代的 DRAM(容量较小)。

2. 采用简单的多体交叉存储器

提高带宽的另一种方法,是在存储系统中采用多个 DRAM,并利用它们潜在的并行性。可以把存储芯片组织为多个体(bank),并让它们并行工作,从而能一次读或写多个字(而不是一个字)。一般来说,使用交叉存储器的目的是利用系统中所有 DRAM 的潜在带宽,而大部分存储系统(非交叉存储器)只启动包含所访问字的那个 DRAM。

存储体的宽度通常都是一个字,这样就无需改变总线的宽度和 Cache。但同时向几个体发送地址能使它们同时进行读访问。图 5.21(c)画出了这种结构。例如,若同时向 4 个体发送地址,则失效开销为 $4 + 24 + 4 \times 4 = 44$ 个时钟周期,带宽约为每个时钟周期 0.4 字节。多体结构对于写操作也是有用的。尽管通常在进行连续多个写时,后一个要等到前一个结束后才能进行,但如果采用多

体,而且这些写不是对同一个体进行操作,则能实现每拍完成一个写。这样的存储器组织结构对于采用写直达法的 Cache 尤为重要。

地址到存储体的映象方法影响着存储系统的行为。上面的例子假设四个存储体的地址是在字一级交叉的,即存储体 0 中每个字的地址对 4 取模都是 0,体 1 中每个字的地址对 4 取模都是 1,依此类推。图 5.22 说明了这种交叉方式。该图假设采用按字寻址,如要按字节寻址,且每个字为 4 个字节,则要把这些地址乘以 4。交叉存储器(interleaved memory)通常是指存储器的各个体是按字交叉的。这种交叉优化了顺序访问存储器的性能。字交叉存储器非常适合于处理 Cache 读失效,因为调块时块中的各个字是顺序读出的。在采用写回法的 Cache 中,不仅读出是顺序的,而且写也是顺序的,因而能从交叉存储器中获得更大的好处。

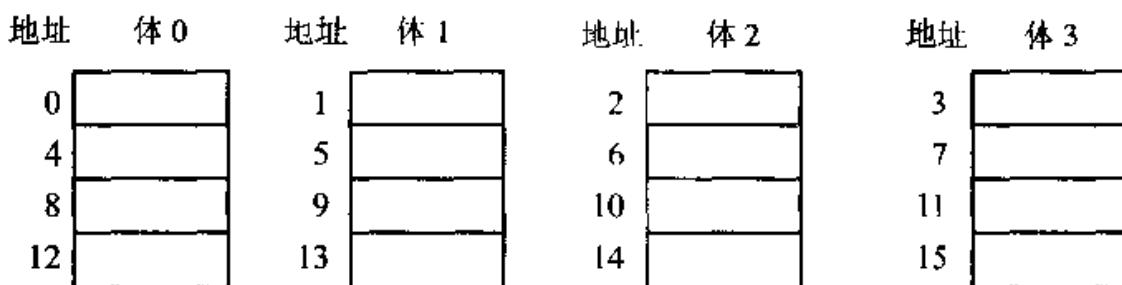


图 5.22 4 路多体交叉存储器

例 5.14 假设某台机器的特性及其 Cache 的性能为

- 块大小为 1 个字
- 存储器总线宽度为 1 个字
- Cache 失效率为 3 %
- 平均每条指令访存 1.2 次
- Cache 失效开销为 32 个时钟周期
- 平均 CPI(忽略 Cache 失效)为 2

试问多体交叉和增加存储器宽度对提高性能各有何作用?

如果当把 Cache 块大小变为 2 个字时,失效率降为 2%;块大小变为 4 个字时,失效率降为 1%。根据前面给出的访问时间,求在采用 2 路、4 路多体交叉存取以及将存储器和总线宽度增加一倍时,性能分别提高多少?

解 在改变前的机器中,Cache 块大小为一个字,其 CPI 为

$$2 + (1.2 \times 3\% \times 32) = 3.15$$

因为在本例中,时钟周期时间和指令数保持不变,所以可以仅通过比较 CPI 来说明性能的改进。

当将块大小增加为 2 个字时,在下面三种情况下的 CPI 分别为

32 位总线和存储器,不采用多体交叉: $2 + (1.2 \times 2\% \times 2 \times 32) = 3.54$

32 位总线和存储器,采用多体交叉: $2 + (1.2 \times 2\% \times (4 + 24 + 8)) = 2.86$

64 位总线和存储器,不采用多体交叉: $2 + (1.2 \times 2\% \times 1 \times 32) = 2.77$

可见,若只将 Cache 块的大小增加一倍,而其他条件不变,则会使机器的性能下降(CPI 由 3.15 增加为 3.54),而在另两种情况下,性能分别提高了 10% 和 14%。如果我们将块大小增加到 4 个字,可以得到以下数据:

32 位总线和存储器,不采用多体交叉: $2 + (1.2 \times 1\% \times 4 \times 32) = 3.54$

32 位总线和存储器,采用多体交叉: $2 + (1.2 \times 1\% \times (4 + 24 + 16)) = 2.53$

64 位总线和存储器,不采用多体交叉: $2 + (1.2 \times 1\% \times 2 \times 32) = 2.77$

和前面一样,只增加块的大小仍然会降低机器性能。采用多体交叉的 32 位宽的存储器的性能最高,提高了 25%,而 64 位的宽存储器和总线的性能只提高了 14%。

由以上可以看出,多体交叉存储器在逻辑上是一种宽存储器,只是为了共享内部资源(本例中为总线),把对各存储体的访问安排在不同的时间段进行。

那么存储器中应该含有多少个体呢?向量计算机采用以下衡量标准:

体的数目 \geq 访问体中一个字所需的时钟周期数

存储系统的设计目标是:对于顺序访问,每个时钟周期都能从一个不同的存储体中送出一个数据。为了便于理解上式为什么成立,假设在某个存储系统中,存储体数少于访问体中一个字所需的时钟周期数。例如体数为 8,而访问一个字需 10 个时钟周期。发出访问请求 10 个时钟周期后,CPU 将从存储体 0 得到一个字。随后,存储体 0 将开始读该存储体中的下一个字,而 CPU 则依次从其余 7 个存储体中得到后续的 7 个字。在第 18 个时钟周期,CPU 将需要由存储体 0 提供下一个字,但该字要到第 20 个时钟周期才被读出,CPU 只好等待。所以我们希望体数大于访问体中一个字所需的时钟周期数,以避免等待。

稍后将讨论对存储体进行非顺序访问时的冲突问题。这里,只需注意到,增加存储体的数目能够减少发生这种体冲突的机会。

然而,随着存储芯片容量的增加,构成一个相同容量的存储系统所需的芯片数越来越少,这使得多体存储器的价格变得昂贵得多。例如,一个 64 MB 的主存由 512 片 1 M \times 1 位的芯片组成,这些芯片很容易被分成 16 个体,每个体 32 块芯片。但若采用 64 M \times 1 位的芯片,则只需 8 块芯片,最多只能组成一个体。尽管 Amdahl/Case 平衡计算机系统经验规则指出:随着 CPU 性能的提高,应增加主存容量,但许多制造商却希望在基准型号的机器中提供小容量存储器的选择。这种 DRAM 芯片数量上的减少是交叉存储器最主要的不足。采用具有较

宽数据通路的 DRAM, 如 $16\text{ M}\times 4$ 位或 $8\text{ M}\times 8$ 位, 可以缓解这一问题。

交叉存储器的另一个不足是主存扩展较困难。为解决这一问题, 要么主存系统必须能够支持多代 DRAM, 就像 DEC 3000 model 800 那样, 要么主存扩展的最小增量就必须是主存容量的一倍。

3. 独立存储体

采用多体的最初目的是通过将顺序访存分配到不同的体来获得更高的存储器带宽。多体存储器的硬件复杂度没增加多少, 因为各存储体可以和存储控制器一起共享地址线, 各个体分时使用存储器的数据总线。交叉访存进一步推广, 就能同时进行多个独立的访存。这时应有多个存储控制器, 以允许多个体(或多组按字交叉的存储体)能独立操作。例如, 一台输入设备可能会使用某个存控, 访问某个存储体; Cache 读操作可能使用另一个存控, 访问另一个存储体; 而 Cache 写操作则可能使用第三个存控, 访问第三个存储体。在这个存储器中, 每个体需要有独立的地址线, 而且也许还需要有独立的数据总线。非阻塞 Cache 允许 CPU 在 Cache 失效时继续运行, 这就潜在地允许多个 Cache 失效被同时处理。这种设计仅在采用多体结构时有意义, 否则多个读操作只能通过一个存储端口进行, 所能得到的好处不大: 只能将访存操作和数据传送重叠进行。采用多体结构的另一个原因, 是共享公共存储器多处理机系统的需求。

独立存储体可以和上述按字交叉的多体结合起来使用, 即将存储器分为若干个独立的存储体, 而每个独立存储体内部又划分为若干个按字交叉方式工作的体, 如图 5.23 所示。有时称独立存储体为超体, 而称超体内按字交叉的部分为体。

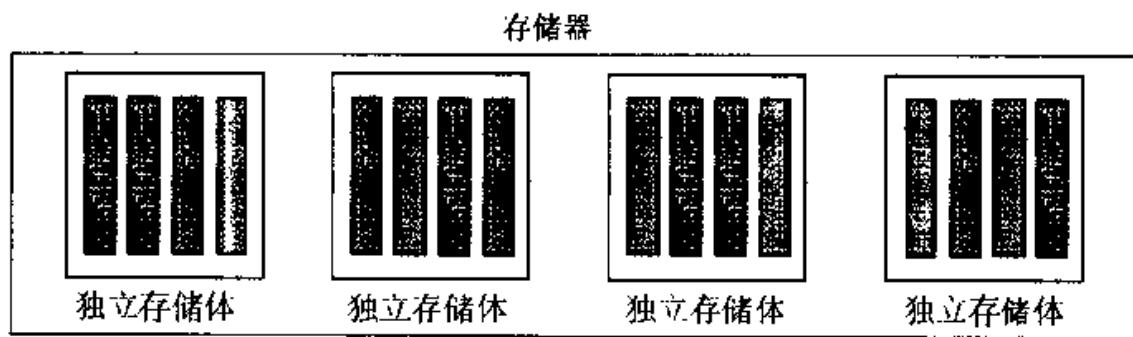


图 5.23 含独立存储体的存储器

4. 避免存储体冲突

在许多情况下, 都要求把存储系统设计成能支持多个独立的访存请求, 例如: 失效下的失效、能从多个不连续的地址读数据(收集)或向多个不连续的地址写数据(散播)的 DMA 方式 I/O、多处理机或向量计算机。这时存储器系统的性

能将取决于这些独立的访存请求发生体冲突的频度的高低。所谓体冲突,是指两个请求要访问同一个体。在传统的多体交叉结构中,顺序访问被处理得很好,不会发生体冲突。地址相差奇数值的访存也是如此。问题是当地址相差为偶数值时,冲突的频度就增加了。解决该问题的一种方法,是采用许多体去减少体冲突的次数。这种方法只有在较大规模的机器中才采用,例如,NEC SX/3 最多使用了 128 个体。

这种方法存在问题是:因为对存储器中数据的访问不是随机的,所以无论有多少个体,多个访问都有可能去访问同一个体。假如我们有 128 个存储体,按字交叉方式工作,并执行以下程序:

```
int x [ 256 ][ 512 ];
for ( j = 0 ; j < 512 ; j = j + 1 )
    for ( i = 0 ; i < 256 ; i = i + 1 )
        x [ i ][ j ] = 2 * x [ i ][ j ];
```

因为 512 是 128 的整数倍,同一列中的所有元素都在同一个体内,无论 CPU 或存储系统多么高级,该程序都会在数据 Cache 失效时暂停。

体冲突问题既可以用软件方法也可以用硬件方法解决。编译器可以通过循环交换优化来避免对同一个体的访问。更简单的一种方法是让程序员或编译器来扩展数组的大小,使之不是 2 的幂,从而强制使上述地址落在不同的体内。

在介绍硬件解决方法之前,我们先来回顾一下存储体是如何寻址的。对于给定的一个地址,它到存储体内其相应位置的映射关系可以用下面两个式子来表达:

$$\text{体号} = \text{地址 MOD 体数}$$

$$\text{体内地址} = \text{地址 / 体数}$$

传统上,为使上述计算最简便,存储系统的体数和每个体的容量都取为 2 的幂。

减少体冲突的一种硬件解决方案是使体数为素数。采用素数看起来似乎会需要更多的硬件来完成复杂的计算,如上述取模和除法运算。而且这些复杂的计算会延长每次访存的时间。

幸运的是,有几种硬件方法能快速地完成取模运算,尤其是当存储体数为素数,且为 2 的幂减 1 时,我们可以用下面的简单计算来代替除法运算:

$$\text{体内地址} = \text{地址 MOD 存储体中的字数}$$

因为一个存储体中包含的字数一般是 2 的幂,所以可以用位选择方法来替代上述除数为素数的除法。

这种简化的正确性可以用中国余数定理来证明,详略。

表 5.11 列出了三个存储模块的情况。每个模块为 8 个字。表中列出了用两种不同方法进行地址映象的结果:左边用的是传统的顺序交叉地址映象,右边

用的是修改后的新方法。当寻址一个字时，在前一种方法中需进行除法运算，而在后一种方法中只需对 2 的幂进行取模运算。

表 5.11 顺序交叉和取模交叉的地址映象举例

体内地址	存储体					
	顺序交叉			取模交叉		
	0	1	2	0	1	2
0	0	1	2	0	16	8
1	3	4	5	9	1	17
2	6	7	8	18	10	2
3	9	10	11	3	19	11
4	12	13	14	12	4	20
5	15	16	17	21	13	5
6	18	19	20	6	22	14
7	21	22	23	15	7	23

5. DRAM 专用交叉结构

到现在为止,已经讨论了四种增加存储器带宽的技术:更宽的存储器、交叉存储器、多体结构存储器以及避免体冲突。这些技术适用于以任何工艺制造的存储器,并且在 DRAM 出现之前就已被提出或使用。下面将介绍几种利用 DRAM 特性的技术。

前面讲过,对 DRAM 的访问分为行访问和列访问。DRAM 必须能将行访问所得到的一行信息暂存在其内部的缓冲器中,供列访问时用。一行的位数通常是 DRAM 容量的平方根,如:当 DRAM 容量为 64 Mb 时,一行是 8 Kb,当 DRAM 容量为 256 Mb 时,一行是 16 Kb,依此类推。为了改进性能,所有 DRAM 的时序信号都能做到允许反复访问缓冲器中的内容,而不用进行另外的行访问。这种优化有以下三种方式:

- Nibble 方式——每次进行行访问时,DRAM 除能够给出所需的位以外,还能给出其后的 3 位。
- Page 方式——缓冲器以 SRAM 的方式工作。通过改变列地址,可以随机地访问缓冲器内的任一位。这种访问可以反复进行,直到下一次行访问或刷新。
- Static column 方式——和 Page 方式类似,只是在列地址改变时,无需触发列访问选通线。

从 1 Mb 的 DRAM 开始,绝大多数 DRAM 能够完成上述三种方式中的任一种。

具体选择哪一种是在进行封装时确定的(通过选择接触点)。这些优化改变了 DRAM 存储周期的定义。表 5.12 列出了传统的存储周期以及优化后相邻两次访问之间(最快时)的时间间隔。不管采用上述哪一种优化方式,优化后的访问周期时间是一样的。

表 5.12 优化后 DRAM 的访问周期

芯片容量	行访问		列访问	周期时间	优化后的时间
	最慢的 DRAM	最快的 DRAM			
64 Kb	180 ns	150 ns	75 ns	250 ns	150 ns
256 Kb	150 ns	120 ns	50 ns	220 ns	100 ns
1 Mb	120 ns	100 ns	25 ns	190 ns	50 ns
4 Mb	100 ns	80 ns	20 ns	165 ns	40 ns
16 Mb	80 ns	60 ns	15 ns	120 ns	30 ns
64 Mb	65 ns	50 ns	10 ns	90 ns	25 ns

这些优化措施的好处在于它们是利用了 DRAM 本身就有的电路,系统的成本只增加一点就几乎能使带宽增加 3 倍。例如,Nibble 方式所利用的程序访存行为与交叉存储器的相同。Nibble 方式的芯片在内部一次能读出 4 位,并在 4 个优化后的存储周期内将这 4 位送出。除非总线传送时间比优化后的存储周期还要快,否则用这种方式实现 4 路交叉存储器的代价只是时序控制复杂了些。Page 方式和 Static column 方式也可用来实现更高程度的交叉存取,只需稍微增加控制复杂度。DRAM 的三态缓冲器带负载的能力一般较差,这意味着在传统的用多个存储芯片实现的多体交叉结构中,每个体都必须使用缓冲芯片。

还有一种新型的 DRAM,优化了 DRAM 和 CPU 之间的接口。这种新型 DRAM 的一个例子是 RAMBUS 公司生产的产品。该公司采用了传统的标准 DRAM 核,但为之提供一个新的接口,使得芯片更像一个存储系统而不是存储系统的一个组成部件。RAMBUS 还抛弃了 RAS/CAS,而用总线取而代之。该总线允许在发送地址和返回数据之间通过总线进行另一次访存。这种总线被称为包交换(packet-switched)总线或分离事务(split-transaction)总线。这种总线可以直接把存储芯片作为存储体。一块存储芯片能够为一次访存请求返回可变数量的数据,甚至还能完成自己的刷新。RAMBUS 提供了一个 1 字节宽的接口以及一个时钟信号,使芯片能和 CPU 时钟紧密同步。一旦地址流水线充满,一块芯片就能每 2 ns 传送 1 字节数据。

大多数主存系统采用 Page 方式等技术来减少 CPU 和 DRAM 之间性能的差距。和传统的多体交叉存储器不同,当 DRAM 容量增加时,采用这种技术也不会有什么缺点了。但是,新型的 DRAM,如 RAMBUS 则可能要用更多的成本来换取更高的带宽。

5.7 虚拟存储器

5.7.1 虚拟存储器基本原理

早在 1961 年,英国曼彻斯特大学的 Kilburn 等人就已提出了虚拟存储器的概念。经过 20 世纪 60 年代初到 70 年代初的发展完善,虚拟存储器已广泛应用于大中型计算机系统。目前几乎所有的计算机都采用了虚拟存储系统。

虚拟存储器是“主存~辅存”层次进一步发展的结果。它由价格较贵、速度较快、容量较小的主存储器 M_1 和一个价格低廉、速度较慢、容量很大的辅助存储器 M_2 (通常是硬盘)组成,在系统软件和辅助硬件的管理下,就像一个单一的、可直接访问的大容量主存储器。程序员可以用机器指令的地址码对整个程序统一编址,就如同程序员具有对应于这个地址码宽度的存储空间(称为程序空间)一样,而不必考虑实际主存空间的大小。虚拟存储器具有以下特点:

1. 多个进程可以共享主存空间

计算机系统中经常同时运行多个进程,每个进程有自己的地址空间。如果给每个进程都分配一个与其全地址空间大小相同的主存区域,显然代价太高。更何况许多进程只用到其地址空间中的一小部分。虚拟存储器把主存空间划分为较小的块(页或段),并以块为单位分配给各进程。这样,多个进程就可以共享一个较小的主存空间。

此外,大多数虚拟存储器还可以减少程序的启动时间,因为这时程序不需要像以往那样要等到全部程序和数据调入主存后才能开始执行。

2. 程序员不必做存储管理工作

在非虚拟存储器中,当一个程序的大小超过主存空间的大小时,程序员就必须把程序划分为若干个能装得进主存的部分,并识别哪些部分(称为覆盖)在装入主存时可以相互覆盖。程序员还要控制这些覆盖在程序执行过程中的装入与重新装入(从辅存)。这就是所谓的程序覆盖技术。在这种方法中,程序员要保证程序决不会访问主存空间大小以外的空间,而且要保证所需的覆盖会在适当的时机装入主存。显然,这种方法增加了编程的复杂度,降低了程序员的产出率。虚拟存储器解决了这个问题,它自动地对存储层次进行管理,免除了程序员的负担。这也正是当时发明虚拟存储器的初衷。

3. 采用动态再定位,简化了程序的装入

所谓程序定位,是指将程序空间中给出的逻辑地址映象到主存的物理地址。程序再定位有静态再定位和动态再定位之分。静态再定位是指在程序执行之前,在装入或再装入该程序的过程中,通过修改程序中的地址而完成地址空间的变换。而在程序的执行过程中,其在主存空间中的位置就不能再改变。动态再定位是指只有在程序的执行过程中,真正访问指令和数据时,才进行地址变换,产生物理地址。动态再定位使得同一程序可以很方便地装入主存中的任意一个位置执行。

虚拟存储器可以分为两类:页式和段式。页式虚拟存储器把空间划分为大小相同的块,称为页面。常用页面大小为4 KB~64 KB。而段式虚拟存储器则把空间划分为可变长的块,称为段。段的最小长度为1个字节,最大长度因机器而异,通常为 2^{16} 字节~ 2^{32} 字节。页面是对空间的机械划分,而段则往往是按程序的逻辑意义进行划分。

采用页式虚拟存储器还是段式虚拟存储器,对CPU有不同的影响。在页式虚拟存储器中,地址都是单一、固定长度的地址字,由页号和页内位移两部分组成。而在段式虚拟存储器中,地址需要用两个字表示,一个为段号,另一个为段内位移。这是因为段的长度是可变的。

页式和段式虚拟存储器各有优缺点,操作系统教材中有详细的论述。表5.13中列出了主要的几项比较。由于在段式中实现替换很复杂,现代计算机中几乎不采用纯段式。有些机器采用段式和页式的组合——段页式。段页式兼有两者的优点。在段页式中,每段被划分成一些页面,这样既保持了段作为逻辑单位的优点,又简化了替换的实现,而且段不必作为整体全部一次调入主存,可以以页面为单位部分调入。

表 5.13 页式和段式虚拟存储系统的比较

	页 式	段 式
地址大小(字)	1	2(段号和段内位移)
对程序员是否透明	对应用程序员透明	对应用程序员可能透明
替换一个块	容易(所有块大小相同)	困难(必须查找连续、大小可变的空闲主存块)
存储空间浪费	内部碎片(页内未用部分)	外部碎片(主存中的无用块)
磁盘传送效率	高(为平衡访问时间和传送时间,需调整页面大小)	不一定(比较小的段可能只需传送几个字节)

下面来看一下以下四个问题:

1. 映象规则

在处理虚存失效时,需要访问磁盘存储设备,因此虚存失效开销非常大。如果要在低失效率和简单的映象规则之间进行选择的话,操作系统设计者总是选择低失效率,因为失效开销实在是太大。因此,操作系统允许将块放在主存的任一位置,即采用全相联映象。

2. 查找算法

页式和段式管理都要使用一个由页号或段号作为索引的数据结构,这个数据结构分别称为页表和段表。这个数据结构中含有所要查找的块的物理地址。对于段式系统,段内位移加上段的物理地址就是最终的物理地址。而对于页式系统,只需简单地将页内位移拼接在相应页面的物理地址之后即可(见图 5.24)。

页表用虚页号作为索引,它所包含的项数与虚地址空间的总页面数相同。如果虚地址为 28 位,页大小为 4 KB,每个页表项为 4 字节,那么页表的大小就是 256 KB。为了减少页表的大小,有一些机器对虚地址进行散列变换,从而使页表的项数减少到与主存中的物理页面数目相等。物理页面数通常比虚拟页面数少很多。这种数据结构称为反向页表(inverted page table)。对于上面的例子,一个容量为 64 MB 的物理存储器只需大小为 64 KB($4 \times 64 \text{ MB} / 4 \text{ KB}$)的反向页表即可。

为减少地址变换时间,常设置一个专门用于地址变换的高速缓存。这种高速缓存称为 TLB(Translation Look-aside Buffer)或地址变换缓冲器(Translation Buffer)。稍后将详细讨论。

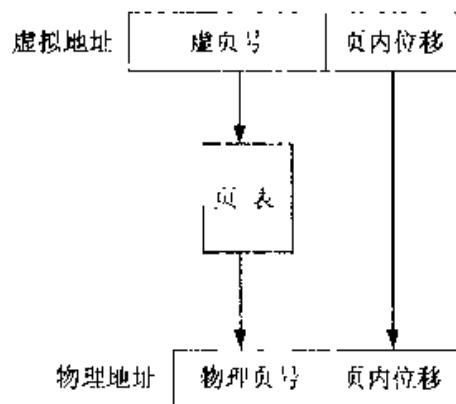


图 5.24 用页表实现虚拟地址到物理地址的映射

3. 替换算法

正如上面所提到的,操作系统的最主要指导思想是尽可能减少页故障(page fault)。和这个指导思想相一致,几乎所有的操作系统都设法替换最近最少使用(即 LRU)的页,因为这个页可能是最没有用的。为了帮助操作系统寻找 LRU 页,许多机器为主存中的每个页面设置了一个使用位(use bit),也称为访问位

(reference bit)。每当主存中的一个页面被访问时,其相应的使用位就被置“1”。操作系统定期地使所有使用位复位。这样,在每次复位之前,使用位的值就反映出了从上次复位到现在的这段时间中哪些页曾被访问过。通过用这种方法跟踪各个页被访问的情况,操作系统就能选择一个最近最少使用的页。

4. 写策略

主存的下一级是磁盘,磁盘的访问时间非常长,需要几十万至上百万个时钟周期。正是由于这两级存储器之间访问时间的巨大差距,目前还没有一个虚存操作系统能在 CPU 执行 Store 操作时将数据穿过主存直接写入磁盘,而且也不应该这么处理。因此,虚拟存储器总是采用写回策略。由于访问下一级存储器的开销非常大,虚拟存储器通常使用“脏”位(dirty bit)来保证只有被修改过了的块才被写回磁盘,这样就可避免对下一级存储器的不必要的访问。

5.7.2 快表(TLB)

页表一般都很大,是存放在主存中(有时页表本身也是按页存储的)。因此,每次访存都要引起对主存的两次访问:第一次是读取页表项,以获得所要访问数据的物理地址;第二次才是访问数据本身。显然,这在实际应用中是无法忍受的,性能受影响太大。

一般采用 TLB 来解决这个问题。TLB 是一个专用的高速缓冲器,用于存放近期经常使用的页表项,其内容是页表部分内容的一个副本。这样,进行地址变换时,一般直接查 TLB 就可以了。只有偶尔在 TLB 不命中时,才需要去访问内存中的页表。TLB 也利用了局部性原理:如果访存具有局部性,则这些访存中的地址变换也具有局部性,即所使用的页表项是相对簇聚的。TLB 也常称为快表或地址变换缓冲器。

TLB 中的项与 Cache 中的项类似,也是由两部分构成:标识和数据。标识中存放的是虚地址的一部分,而数据部分中则存放物理页帧号、有效位、存储保护信息以及其他一些辅助信息。为了使 TLB 中的内容与页表保持一致,当修改页表中的某一项时,操作系统必须保证 TLB 中没有该页表项的副本。

图 5.25 是 Alpha AXP 21064 数据 TLB 的结构。该 TLB 共包含 32 个项,除标识和物理地址外,每项还包含以下标志位:

- V——有效位。为“1”表示该 TLB 项有效。
- R——“读”允许位。为“1”表示允许对该页面进行“读”操作。
- W——“写”允许位。为“1”表示允许对该页面进行“写”操作。

图中的①、②、③、④等表示进行地址变换的步骤。TLB 采用全相联映象,所以当进行地址变换时,把虚拟地址送往各个标识,进行比较(图中的①和②)。显然,只有有效位为“1”的标识才有可能匹配。与此同时,根据 TLB 中的存储保

护信息对本次访存的类型进行检查,看是否越权(②)。

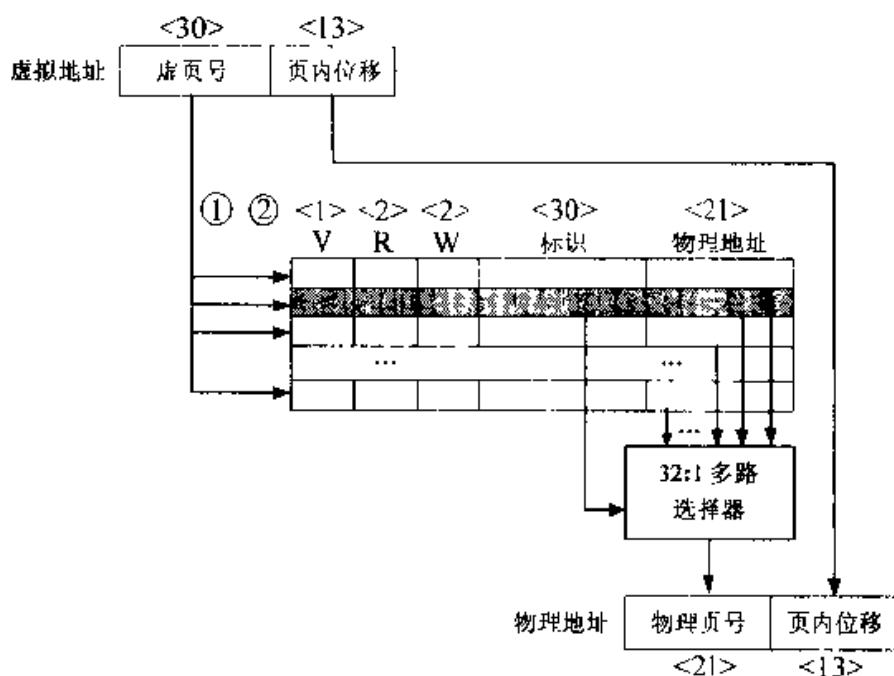


图 5.25 Alpha AXP 21064 的地址转换过程

若存在匹配的标识,则多路选择器把相应 TLB 项中的物理地址选出(③)。该地址与页内位移拼接成完整的 34 位物理地址(④)。

前面讲过,当把 Cache 和虚拟存储器相结合时,要解决如何使地址变换与访问 Cache 并行进行的问题。采用容量较小的 Cache,可以把访问 Cache 的索引限制在页内位移的范围之内(即可直接从页内位移中截取)。这样,虚地址一到,就可以立即索引 Cache。在读 Cache 标识的同时,把虚页号送给 TLB 进行地址变换。在这两个操作完成之后,把从 TLB 得到的物理地址与 Cache 标识进行比较。在这种方案中,Cache 索引用的是虚地址,而标识用的则是物理地址。

地址变换很容易处在决定处理器时钟周期的关键路径上,因为即使是采用最简单的 Cache,也需要读取 TLB 中的值并对其进行比较。所以,一般 TLB 比 Cache 的标识存储器更小,而且更快,这样才能保证 TLB 的读出操作不会使 Cache 的命中时间延长。例如,在 Alpha AXP 21064 中,数据 Cache 有 256 块,而数据 TLB 则只有 32 块(项)。由于 TLB 的速度至关重要,所以有时 TLB 的访问按流水方式实现。

5.7.3 页面大小的选择

页面大小是虚拟存储器的主要参数之一。大页面和小页面各有优点,所以选择页面大小是一个如何进行权衡的问题。以下几点说明了较大页面的好处:

- (1) 页表的大小与页面大小成反比。较大的页面可以节省实现地址映象所

需的存储空间及其他资源；

- (2) 较大的页面可以使快速 Cache 命中的实现更简单；
- (3) 在主存和辅存之间(也许还通过网络)传送较大的页面比传送较小的页面更有效；
- (4) TLB 的项数有限，对于给定数目的项数，较大的页面意味着可以高效地实现更多存储空间的地址变换，从而减小 TLB 失效的次数。

正是由于最后这个原因，近期的微处理器对多种页面大小提供了支持。对于有些程序来说，TLB 失效对 CPI 的影响可能会和 Cache 失效的影响一样大。

采用较小页面的主要动机是为了节省存储空间。在页面较小的情况下，当一片连续的虚存空间不是页面大小的整数倍时，最后一页内空间的浪费较少。这种页面内部的未用存储空间称为内部碎片。假设每个进程有三个主要的段，即代码、堆和堆栈，则平均每个进程浪费的空间为 1.5 个页面(3×0.5 页)。对于内存为几兆字节，页面大小为 4 KB 或 8 KB 的机器来说，这种浪费是可以忽略不计的。但是，当页面很大(超过 32 KB)时，就会浪费许多空间(包括主存和辅存)和 I/O 频带。

最后一点是进程的启动时间。许多进程都比较小，所以采用较大的页面会增加调用进程的时间。

5.8 进程保护和虚存实例

在多道程序(multiprogramming)的情况下，计算机可以被多道同时运行的程序所共享。多道程序的出现，对程序间的保护和共享提出了新的要求。在当今的计算机中，这是与虚拟存储器紧密联系在一起的。

多道程序引出了进程(process)的概念。用比喻的说法，进程就是程序呼吸的空气和生存的空间；也就是说，一个正在运行的程序加上它继续执行所需的所有状态就是进程。分时共享是多道程序的一种变形，它是指几个交互用户同时共享 CPU 和存储器，它使每个用户都以为有一台供自己使用的计算机。因此，在任何时刻，分时系统都必须能从一个进程切换到另外一个进程。这称为进程切换(process switch)或关联切换(context switch)。

不管进程是不间断地从开始一直执行到结束，还是在执行过程中不断地被中断并与其他进程切换，它都必须执行正确。保证进程正确行为既是计算机设计者也是操作系统设计者的责任。计算机设计者必须保证进程状态中有关 CPU 的部分能够被保存和恢复，操作系统设计者则必须保证进程之间的计算互不干扰。保护一个进程的状态不被其他进程干扰的最安全方法，是将进程的当前状态保存在磁盘上。但这样一来，进程切换就需要几秒钟才能完成，这对于分

时系统来说实在是太长了。所以,这个问题一般由操作系统这样来解决:将主存空间分为几个区域,使得主存中可以同时存放多个不同进程的状态。这意味着操作系统设计者需要计算机设计者的帮助,由计算机设计者提供一定的保护机制,使进程之间不能互相修改。除了保护机制以外,计算机还提供了多个进程共享程序和数据的能力,以允许进程之间进行通信和节省存储空间(通过减少存储器中相同信息的拷贝份数)。

5.8.1 进程保护

最简单的保护机制是用一对寄存器来检查每一个地址,以确保地址在两个界限之间。传统上这两个界限分别叫做基地址(base)和上界地址(bound)。一个地址若满足

$$\text{基地址} \leq \text{地址} \leq \text{上界地址}$$

则该地址是有效的。在有些系统中,地址被看作是无符号的整数,这时,地址界限检测条件就变为

$$(\text{基地址} + \text{地址}) \leq \text{上界地址}$$

如果用户进程有权改变基地址寄存器和上界地址寄存器的内容,那么就不能保证用户进程不被其他进程所破坏,所以用户进程不能改变它们。然而操作系统必须能改变这两个寄存器,这样它才能够进行进程切换。因此,计算机设计者还要完成另外三项工作,以帮助操作系统设计者保护进程不被其他进程破坏。

1. 提供至少两种模式,用于区分正在运行的进程是用户进程还是操作系统进程。有时称后者为内核(kernel)进程、超级用户(supervisor)进程或管理(executive)进程。

2. 使 CPU 状态的一部分成为用户进程可读但不可写的。这包括基地址/上界地址寄存器、用户/管理模式位和异常许可/禁止位。用户进程无权修改这些状态,因为如果用户进程能改变地址界限检查、赋给自己管理特权或禁止异常出现,操作系统就无法控制它们了。

3. 提供一种机制,使得 CPU 能从用户模式进入管理模式和从管理模式进入用户模式。前一种模式变换一般是通过系统调用(system call)来完成。系统调用由一条特殊指令实现,该指令将控制权传送到管理程序空间中一个特定位置。系统调用点处的 PC 值会被保存起来,CPU 状态将被置为管理模式。调用结束后返回用户模式很像从子程序返回,它将恢复原先的用户/管理模式。

基地址和上界地址构成了最小的保护系统,而虚拟存储器则提供了一种与这种简单模式不同且更细微的方法。我们已经知道,CPU 地址必须经过从虚拟地址到物理地址的变换。这种映象为硬件提供了进一步检测程序中的错误或保护进程的机会。最简单的做法就是给每个页或段增加许可标志。例如,因为现

在几乎没有程序有意改变它们自己的程序,所以通过给页面提供只读保护,操作系统就能检测到对程序的意外修改。这种页面级保护可以通过增加用户/内核保护进一步扩展,以防止用户程序访问属于内核的页。只要CPU提供读/写信号和用户/内核信号,地址变换硬件就能很容易地在非法的访存操作造成破坏之前检测到它们。这些非法访存操作只会中断CPU并引发操作系统调用。

进程之间的保护可以通过为每个进程分配一个专用的页表,并使每个页表指向存储器中的不同页面来实现。显然,必须禁止用户程序修改它们自己的页表,否则保护就不起作用了。

根据需要还可以把保护进一步升级。例如,在CPU保护结构中加入环,可将访存保护由原来的两级(用户级和核心级)扩展到更多级。在军事系统中密级可分为绝密、秘密、机密和非机密等四个级别。与此类似,在同心圆环结构的级别中,最可靠的程序可以访问所有信息,第二可靠的程序可以访问除最高级以外的任何信息,依次类推。最后是“平民”级程序,它最不可靠,可访问信息的范围也就最小。此外,对哪些存储区能存放程序(即执行保护),甚至对各级别之间的入口点,也有一定的限制。Intel的Pentium采用了环形保护结构。但至今我们还不清楚环型结构在实际应用中是否是对简单的“用户/核心模式”系统的改进。

仅有上述这种简单的环型结构密级还不够。例如,当要对给定密级中的一个程序进行限制时,就要采用一种新的分级系统,这种系统类似于现实生活中的加锁和解锁(而不是上述军事密级模型)。一个程序只有在得到钥匙后,才能解锁并访问数据。为了使这些钥匙(或称功能)起作用,硬件和操作系统必须能显式地将它们从一个程序传到另一个程序,以防止程序进行伪造。为了能实现钥匙的快速检测,需要很多的硬件支持。

5.8.2 页式虚存举例: Alpha AXP 的存储管理和 21064 的 TLB

Alpha AXP 体系结构采用段页相结合的方式,既提供了存储保护,又将页表大小减少到最小。Alpha 根据 64 位地址的最高两位将地址空间分为三个段: seg0 (最高位为 0), kseg (最高位为 1, 次高位为 0) 和 seg1 (最高两位都是 1)。其中 kseg 段留给操作系统内核使用,该段整个空间具有相同的保护权限,而且不需要存储管理。用户进程使用 seg0 和 seg1 段,这两个段所映射到的各页面具有不同的保护权限。图 5.26 中给出了 seg0 和 seg1 的布局: seg0 从地址 0 开始向上生长,而 seg1 则从最高端地址开始向下生长。现在许多系统都采用了这种存储空间预分段与页式管理相结合的某种方法。这种方法有不少优点,如: 分段将地址空间分为几个部分,预留出页表空间,而页式管理则实现虚存、重定位和保护。

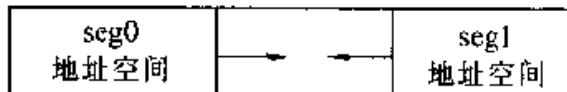


图 5.26 Alpha 机器中 seg0 段和 seg1 段的结构

即使按照这种方法将 64 位地址空间分为三个段，页表仍然会占用很大的空间。为了使页表的大小合理，Alpha 使用三级页表来实现对地址空间的映射。虚地址中有三个域： l_1 、 l_2 、 l_3 ，分别用于查找这三级页表，如图 5.27 所示。图 5.27 说明了 Alpha AXP 的地址变换过程，具体步骤如下：

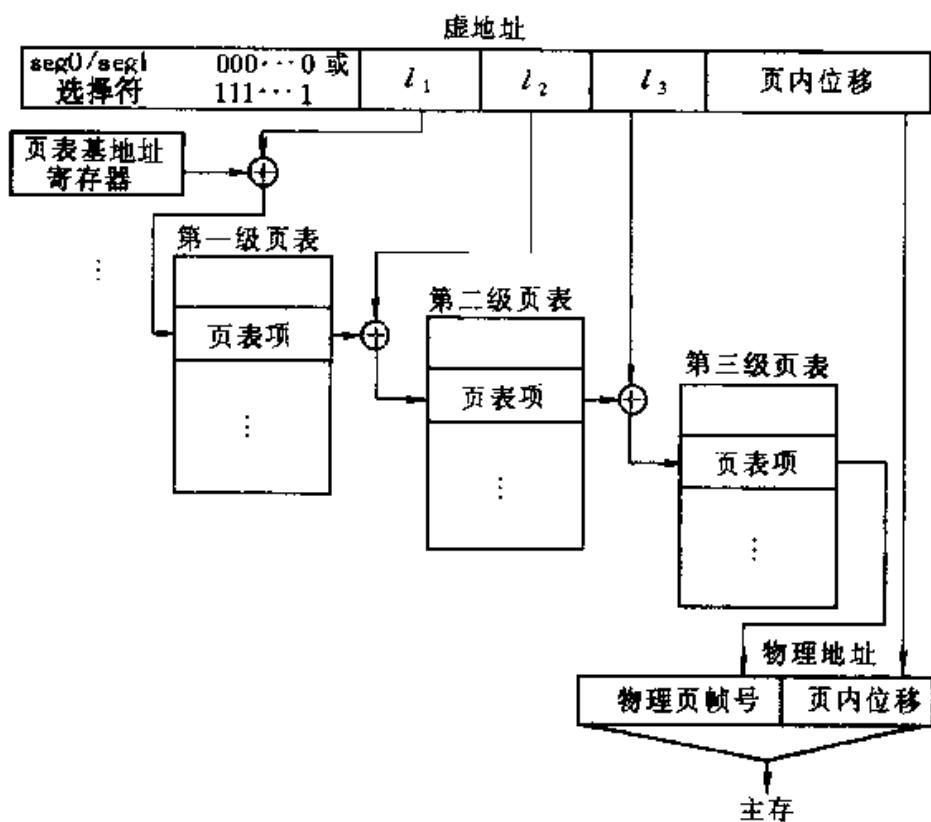


图 5.27 Alpha 机器中虚地址的变换过程

- (1) 将虚地址 l_1 域的值和页表基地址寄存器中的值相加，然后用相加结果作为地址访问存储器，得到第二级页表的基地址；
- (2) 将虚地址中 l_2 域的值和第二级页表基地址相加，然后根据相加结果访存，得到第三级页表的基地址；
- (3) 将虚地址中 l_3 域的值和第三级页表基地址相加，然后根据相加结果访存，最后得到所访问页面的物理地址；
- (4) 将上一步所得到的地址与页内位移拼接，便得到完整的物理地址。

在 Alpha AXP 体系结构中，每个页表的大小都被限制在一个页面以内，所以所有的页表基地址都是物理地址，无需进一步变换。

Alpha 中所有页表的页表项(Page Table Entry, 即 PTE)都是 64 位的。PTE 的前 32 位为物理页帧号, 而后 32 位则包含以下 5 个保护域:

- 有效域——为“1”表示该页帧号有效, 可被硬件用于地址变换
- 用户读许可域——为“1”表示允许用户程序读该页内的数据
- 内核读许可域——为“1”表示允许内核进程读该页内的数据
- 用户写许可域——为“1”表示允许用户程序将数据写入该页
- 内核写许可域——为“1”表示允许内核进程将数据写入该页

此外, PTE 中还有一些域是留给操作系统软件使用的。由于每次 TLB 失效时, Alpha 将依次访问三级页表中的每一级, 因此可以在三个地方进行保护权限检查。但 Alpha 只在访问第三级 PTE 时才进行保护权限检查, 而在访问前两级页表时只检查它们的有效位是否为“1”。由于在 Alpha 中, PTE 的长度为 8 个字节, 而且页表大小正好为一个页面, 而 Alpha AXP 21064 的页面大小为 8 KB, 所以每个页表有 1 024 个 PTE。在 64 位虚地址中, l_1 、 l_2 、 l_3 这三个域都是 10 位, 页内位移为 13 位, 所以还有 $64 - (3 \times 10 + 13) = 21$ 位没有定义。seg0 地址的最高位为“0”; 而 seg1 地址的最高位和次高位都是“1”。Alpha 要求 l_1 以左的所有位的值都相同, 所以 seg0 地址的高 21 位都是“0”, 而 seg1 地址的高 21 位都是“1”。这意味着 21064 的虚地址实际上只有 43 位有用, 而不是 64 位。物理地址本应为 $32 + 13 = 45$ 位, 但 Alpha AXP 体系结构要求物理地址位数比虚拟地址少。21064 把物理地址限制为 34 位, 从而节省了芯片空间。

最大虚地址和最大物理地址与页面大小紧密相关。Alpha 体系结构文档中允许 Alpha 机器的最小页面大小为 8 KB~64 KB。在页面大小为 64KB 时, 虚地址增加为 $3 \times 13 + 16 = 55$ 位(这时 l_1 、 l_2 、 l_3 都是 13 位, 页内位移为 16 位), 而最大物理地址则增加为 $32 + 16 = 48$ 位。

页表由操作系统管理, 用户程序不能进行写入, 这样就能防止进行非法地址变换, 从而防止出现错误。用户可以试图访问任何虚拟地址, 但到底是访问实存的哪个位置, 是由操作系统通过控制页表项的内容来确定的。这种管理方法也便于实现进程之间的存储器共享。只要使各进程地址空间中要共享的页面的页表项指向同一个实存页面即可。

上述存储管理结构首先在 Alpha AXP 21064 中得到了实现。为减少地址变换时间, 21064 采用了两个 TLB, 一个用于指令访问, 另一个用于数据访问。表 5.14 给出了这两个 TLB 的关键参数。Alpha 允许由操作系统告诉 TLB 将连续的多个页面作为同一个页面来处理。这种合成页面的大小有三种选择: 最小页面大小的 8 倍、64 倍或 512 倍。由于页面大小可变, 所以在查 TLB 时, 必须检查相应的 PTE 所映象的页面大小是否也匹配。这增加了 PTE 匹配的复杂性。

表 5.14 Alpha AXP 21064 TLB 的存储层次参数

参 数	描 述
块 大 小	1 PTE(8 字节)
命 中 时 间	1 个时钟周期
平均失效开销	20 个时钟周期
TLB 容 量	指令 TLB:8 个 PTE 用于大小为 8 KB 的页,4 个 PTE 用于大小为 4 MB 的页(共 96 个字节) 数据 TLB:32 个 PTE 用于大小分别为 8 KB、64 KB、512 KB 和 4 MB 的页(共 256 个字节)
块替換策略	随机法(但不替换刚用过的)
写 策 略	不适用
块映象策略	全相联

在当今的计算机中,Alpha AXP 21064 的存储管理是非常典型的。它采用页面级地址变换,并依赖于操作系统的正确操作来为多个进程共享计算机提供安全保证。与其他存储管理相比,Alpha 的主要不同在于其虚地址超过了 32 位。

5.9 Alpha AXP 21064 存储层次

前面已经分别介绍了 Alpha AXP 21064 存储层次的组成部分,本节将对其整体结构和工作过程作一概述。图 5.28 是 21064 存储层次的整体结构图。

下面从 Alpha 开机开始讨论。开机后,芯片上的硬件用一个外部 PROM 中的指令加载指令 Cache。这样初始化可以使容量为 8 KB 的指令 Cache 省去有效位,因为这时 Cache 里的指令都是有效的,只不过可能不是用户程序感兴趣的罢了。同时,硬件还把数据 Cache 中的所有有效位都清“0”,并将程序计数器 PC 的值量为 kseg 段中的地址,这样指令地址就不必进行变换,因而也就暂时不需要 TLB 了。

另外,还要用进程的有效页表项(PTE)来更新指令 TLB。对于要映射的每个页面,内核程序都用其页表项的内容来更新 TLB。指令 TLB 共含 12 个 PTE,其中 8 个用于 8 KB 的页面,4 个用于 4 MB 的页面(只有一些大程序,如操作系统、数据库等,才使用大小为 4 MB 的页面。这些程序在执行时很可能访问到其大部分代码)。TLB 失效时将调用特权结构库(Privileged Architecture Library,简称 PAL)软件,由它对 TLB 进行更新。PAL 程序实际上就是一些特殊的机器语言例程,其中包含了一些与具体机器相关的扩充,能访问诸如 TLB

之类的低级硬件。PAL 程序执行时,禁止异常发生,并且在取指令时不进行存储访问权限检查,从而能完成对 TLB 的填写工作。

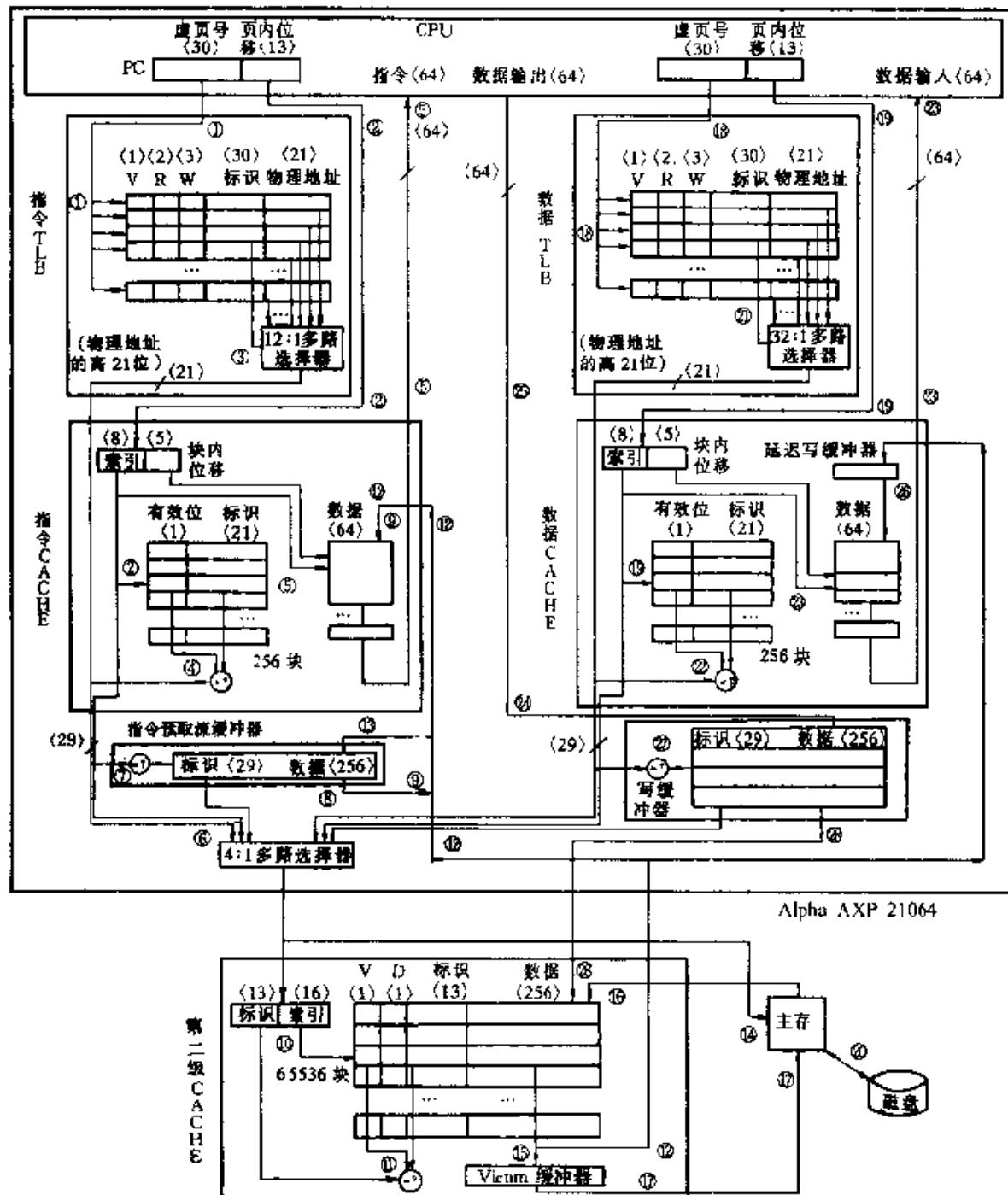


图 5.28 Alpha AXP 21064 存储层次整体结构图

一旦操作系统已准备好运行一个用户进程,它就将 PC 置为 seg0 段中的相应地址。

图 5.28 标出了一次读指令并执行指令的所有步骤,下面按照图中的步骤进行介绍。在取指令时,首先将地址的页号部分(高 30 位)送给指令 TLB(图 5.28

中的①),同时把页内位移中的高 8 位作为索引送给容量为 8 KB 的直接映象指令 Cache(含 256 个大小为 32 字节的块)②。全相联的 TLB 同时搜索其中的全部 12 项,以判断是否有某个有效的 PTE 和指令地址匹配③。除进行地址变换外,TLB 还检查 PTE 中的保护信息,以判断本次访问是否会导致异常。当本次访问违反了所访问页面的保护权限或所访问的页面不在主存中时,将引起异常。如果没有发生异常,并且地址变换后产生的物理地址与指令 Cache 中相应标识相匹配④,则本次访问命中。这时将根据页内位移的低 5 位从相应的块(32 字节)中读出 8 个字节送往 CPU⑤。

如果访问指令 Cache 失效,则将访问第二级 Cache,并同时检查预取指令流缓冲器⑥。如果在指令流缓冲器中找到了所需要的指令⑦,则把相应的 8 个字节送给 CPU,将指令缓冲器中相应的块写入指令 Cache⑧,并取消对第二级 Cache 的访问。图中第 6 步⑥至第 9 步⑨在一个时钟周期内便可完成。

如果在指令流缓冲器中没有找到所需要的指令,就由第二级 Cache 继续寻找相应的块。21064 微处理器能支持容量为 128 KB 到 8 MB 的直接映象第二级 Cache,失效开销为 3 个时钟周期到 16 个时钟周期。本节我们以 DEC 3000 model 800 Alpha AXP 的存储系统为例。该机器的第二级 Cache 的容量为 2 MB(含有 65 536 个大小为 32 字节的块)。因此在第二级 Cache 中,29 位的块地址划分为两部分:13 位的标识和 16 位的索引⑩。第二级 Cache 根据该索引读出标识,并和地址中的标识进行比较⑪。如果匹配,则第二级 Cache 按以下方式在 10 个时钟周期内将相应的块传送给第一级 Cache:在头 5 个时钟周期中先送出 16 个关键字节(含访问的指令),然后在接下来的 5 个时钟周期中再送出剩下的 16 个字节⑫。(第一级和第二级 Cache 之间数据通路的宽度为 128 位,即 16 字节)。与此同时,发出对顺序的下一个数据块的请求,然后在下一个 10 个时钟周期中,将该块装入指令预取流缓冲器⑬。

进行上述预取时,不再通过 TLB 进行地址转换,而是通过将引起本次失效的物理地址加上 32 字节得到新地址,并检查新地址是否和原地址在同一个页面内。如果新地址跨越了页边界,就不进行预取。

如果在第二级 Cache 也没有找到所需指令,就把物理地址送往存储器⑭。DEC 3000 model 800 将其存储器划分为 4 块存储主板(memory mother board),每块板上装有 2~8 个 SIMM。每个 SIMM 含 10 块 DRAM,其中 8 块用于存储信息,另 2 块用于每边的检错和纠错。SIMM 可以是单面的也可以是双面的。DRAM 的容量可以为 1 Mb、4 Mb 或 16 Mb,因此 model 800 的存储器容量在 8 MB($4 \times 2 \times 8 \times 1 \times 1/8$)到 1 024 MB($4 \times 8 \times 8 \times 16 \times 2/8$)之间,并且其宽度总是 256 位。当处理器发出访存请求后,从存储器读出 32 个字节并将其送入第二级 Cache 所需的平均时间为 36 个时钟周期 第二级 Cache 每次只能装入 16 字节。

因为第二级 Cache 采用写回法,任何一次失效都可能导致把旧块(被替换出的块)写回存储器。为了不影响新数据的装入,21064 把被替换出的块放入 Victim 缓冲器(⑯)。新数据一到达就立即被装入 Cache(⑰),然后才把旧数据从 Victim 缓冲器中写回存储器(⑱)。在 Victim 缓冲器中只有一个块,所以如果在前一次失效被处理完之前又发生失效,那就只好暂停,直到 Victim 缓冲器清空。

假定所取出的指令是 Load 指令。它将把数据地址中的页号部分送给数据 TLB(⑲),同时把页内位移中的高 8 位作为索引送给数据 Cache(⑳)。数据 TLB 是一个含有 32 个 PTE 的全相联 Cache,每个 PTE 所对应的页面大小为 8 KB 至 4 MB。TLB 失效将导致自陷,调用 PAL 程序装入相应页面的 PTE。在最坏的情况下,所访问的页面不在主存中,这时操作系统要从磁盘将该页面调入主存(㉑)。由于处理页故障所需的时间较长,其间可以执行上百万条的指令,所以操作系统将从等待的进程中选一个调入执行。

假设数据 TLB 中相应的 PTE 有效(㉒),则把 Cache 中的标识和物理页帧号进行比较(㉓)。若比较结果为匹配,则从 32 字节的数据块中读出所需的 8 个字节送往 CPU(㉔);若为失效,则访问第二级 Cache,具体步骤和访问指令失效时的相同。

当指令为 Store 时,应如何处理呢?这时同样要把地址中的页号和索引分别送给数据 TLB 和数据 Cache(㉕)和(㉖),进行地址变换和保护权限检查。然后将经地址变换所得到的物理地址送往数据 Cache(㉗)和(㉘)。由于数据 Cache 采用写直达法,要写的数据同时被送往写缓冲器(㉙)和数据 Cache(㉚)。5.5 节曾经讲过,21064 按流水方式处理写命中,所以,在当前这一拍,只对本次的写入地址进行检测,看是否命中。与此同时,前一个写命中的数据被写入 Cache。若本次“写”为命中,则把要写入的数据放入写流水线缓冲器。若为失效,则只把数据送给写缓冲器,这是因为数据 Cache 不按写分配(即写不命中时不调块)。

现在来看一看写缓冲器。写缓冲器中共有 4 项,每项存放一个完整的 Cache 块。如果写缓冲器已满,那么 CPU 就必须等待,直到某个块被写入第二级 Cache。如果写缓冲器未满,则 CPU 继续执行,并且将本次“写”的地址送给写缓冲器(㉛),判断是否与其中的某个块匹配。这是为了把对连续地址的多个写合并到同一个块中。

所有的写入最终都将到达第二级 Cache。如果对第二级 Cache 的写访问为命中,则把数据写入该 Cache(㉜)。由于第二级 Cache 采用写回法,所以它不能按流水方式处理写操作。它写一个 32 字节的数据块需 15 个时钟周期:5 个用于检查地址,10 个用于写数据;当写入 16 字节或更少的数据时,它需用 10 个时钟周期:5 个用于检查地址,5 个用于写数据。在这两种情况下,Cache 都将该块标记为“脏”。

如果对第二级 Cache 的访问为失效, 就检查该 Cache 中要替换的块是否为“脏”。如果为“脏”, 则应跟前面一样, 将该块放入 Victim 缓冲器(⑯)。如果要写入的数据是一个整块, 只需简单地把数据写入 Cache, 并将该块标记为“脏”即可; 如果要写入的数据不足 32 字节, 那还要进行一次访存, 因为第二级 Cache 的写策略是按写分配。

5.10 小结

由于主存和 CPU 是用不同的材料制成的, 所以难以设计和生产出在速度上和快速 CPU 相匹配的存储器。我们是依靠局部性原理来解决这个问题的。局部性原理的正确性和实用性在现代计算机存储层次中的各层(包括从磁盘到 TLB)都已得到了验证, 表 5.15 对本章所介绍的各存储层次的属性作了一个总结。

表 5.15 本章存储层次实例小结

	TLB	第一级 Cache	第二级 Cache	虚拟存储器
块大小 (字节)	4~8 (1 个 PTE)	4~32	32~256	4 096~16 384
命中时间 (时钟周期)	1	1~2	6~15	10~100
失效开销 (时钟周期)	10~30	8~66	30~200	700 000~6 000 000
失效率(局部)	0.1%~2%	0.5%~20%	15%~30%	0.000 01%~0.001%
容量	32~8 192 字节 (8~1 024 个 PTE)	1~128 KB	256 KB~16 MB	16~8 192 MB
下一级存储器	第一级 Cache	第二级 Cache	页模式 DRAM	磁盘
映象规则	全相联/组相联	直接映象	直接映象/组相联	全相联
查找方法	标识/块	标识/块	标识/块	表
替换算法	随机	无(直接映象)	随机	近似 LRU
写策略	写页表时清空	写直达/写回法	写回法	写回法

然而, 各级存储层次的设计决策是相互影响的, 而且体系结构设计者必须从整个系统的角度出发, 才能作出明智的决策。对存储层次设计者来说, 最主要和最困难的问题是如何选择各层的参数, 使它们能很好地相互配合, 达到良好的整

体性能,而不是去发明什么新技术。CPU 的速度在不断提高,但它们花在等待存储器上的时间也越来越多了。为了解决这一问题,人们提出了一些新的设计方案,提供了更多的选择。例如可变页大小、伪相联 Cache 以及能面向 Cache 优化的编译器等。幸运的是,在平衡价格、性能和复杂度这三项指标方面经常存在一个技术上的“优化点”。在设计时如果不能找到该优化点,就会损失性能和浪费硬件、设计时间及调试时间。体系结构设计者们通过仔细的量化分析可以找到该优化点。

习 题 五

5.1 解释以下术语

存储层次	全相联映象	直接映象	组相联映象
替换算法	LRU	写直达法	写回法
按写分配法	不按写分配法	写合并	命中时间
失效率	失效开销	强制性失效	容量失效
冲突失效	2:1 Cache 经验规则	相联度	Victim Cache
伪相联 Cache	故障性预取	非故障性预取	非阻塞 Cache
子块放置技术	尽早重启	请求字优先	多级包容性
虚拟 Cache	多体交叉存储器	存储体冲突	TLB

5.2 简述“Cache - 主存”层次与“主存 - 辅存”层次的区别。

5.3 降低 Cache 失效率有哪几种方法? 简述其基本思想。

5.4 简述减少 Cache 失效开销的几种方法。

5.5 通过用编译器对程序进行优化来改进 Cache 性能的方法有哪几种? 简述其基本思想。

5.6 在“Cache - 主存”层次中,主存的更新算法有哪两种? 它们各有什么特点?

5.7 组相联 Cache 的失效率比相同容量直接映象 Cache 的失效率低。由此能否得出结论:采用组相联一定能带来性能上的提高? 为什么?

5.8 写出三级 Cache 的平均访问时间的公式。

5.9 给定以下的假设,试计算直接映象 Cache 和两路组相联 Cache 的平均访问时间以及 CPU 的性能。由计算结果能得出什么结论?

假设:

- (1) 理想 Cache 情况下的 CPI 为 2.0, 时钟周期为 2 ns, 平均每条指令访存 1.2 次;
- (2) 两种 Cache 容量均为 64 KB, 块大小都是 32 字节;
- (3) 组相联 Cache 中的多路选择器使 CPU 的时钟周期增加了 10%;
- (4) 这两种 Cache 的失效开销都是 80 ns;
- (5) 命中时间为 1 个时钟周期;
- (6) 64 KB 直接映象 Cache 的失效率为 1.4%, 64 KB 两路组相联 Cache 的失效率为 1.0%。

5.10 假设有一台计算机具有以下特性:

- (1) 95% 的访存在 Cache 中命中；
- (2) 块大小为两个字，且失效时整块被调入；
- (3) CPU 发出访存请求的速率为 10^9 字/s；
- (4) 25% 的访存为写访问；
- (5) 存储器的最大流量为 10^9 字/s(包括读和写)；
- (6) 主存每次只能读或写一个字；
- (7) 在任何时候，Cache 中有 30% 的块被修改过；
- (8) 写失效时，Cache 采用按写分配法。

现欲给该计算机增添一台外设，为此首先想知道主存的频带已用了多少。试对于以下两种情况计算主存频带的平均使用比例。

- (1) 写直达 Cache；
- (2) 写回法 Cache。

5.11 在例 5.6 有关伪相联 Cache 的计算中，假设当发生慢速命中时，硬件会把慢速块的内容与相应快速块的内容互换，以使得以后对该地址的访问都变成快速命中。现在假设：不进行内容互换，而且在这种情况下慢速命中仅比快速命中多一个时钟周期，试求：

- (1) 使用与例 5.6 中类似的方法，推导出平均访存时间的公式。
- (2) 利用(1)中得到的公式，对于例 5.6 中的两种情况(2 KB Cache 和 128 KB Cache)，重新计算伪相联 Cache 的平均访存时间。请问哪一种伪相联更快？

5.12 假设当采用理想存储系统时的基本 CPI 是 1.5，试利用表 5.5，分别对于下述三种 Cache 计算 CPI：

- (1) 16 KB 直接映象统一 Cache，采用写回法；
- (2) 16 KB 两路组相联统一 Cache，采用写回法；
- (3) 32 KB 直接映象统一 Cache，采用写回法。

第六章 输入输出系统

6.1 概述

输入/输出系统简称 I/O 系统,它包括 I/O 设备和 I/O 设备与处理机的连接。在计算机系统中,I/O 系统完成与外部系统的信息交换,是 von Neumann 结构计算机系统的四大组成部分之一。但长期以来,在计算机体系结构中对 I/O 部分重视较少,说起计算机系统的性能,往往只提 CPU 的性能,许多人认为 CPU 的速度就是计算机的速度,加之 I/O 设备又称为“外围设备”,因此往往受到忽视。

其实,一台没有 I/O 的计算机如同一台没有轮子的汽车一样,只能是一堆废铁。就算一台计算机的 CPU 速度再高,但如果用户在键入命令后,好半天才得到响应,那么用户也不会喜欢这台计算机。首先看一下几个与 I/O 有关的问题。

(1) 对系统性能的影响:Amdahl 定律告诉我们,计算机的性能主要由系统中最慢的部分(称为系统瓶颈)决定。计算机的 CPU 性能提高得很快,按目前状况,每 18 个月提高一倍。要是 I/O 性能不随之改进的话,即使 CPU 再快也没有多大意义,整机性能的提高也会受到严重限制。

(2) 不同系统中 I/O 的差异:微机、工作站、大型机和巨型机之间在应用领域上差异很大,因此其 I/O 类型和数量也有很大差异。目前各类系统中 CPU 的速度差异正在减小,计算机系统的差距主要是 I/O 系统的差距。

(3) I/O 系统的设计问题与 CPU 设计遇到的问题不同:设计 I/O 的三个标准是成本、性能和容量。其中容量包含:

- 计算机能和哪些不同类型的 I/O 设备相兼容;
- 每一台计算机能带多少个 I/O 设备。

例 6.1 假设一台计算机的 I/O 处理占 10%,当其 CPU 性能改进,而 I/O 性能保持不变时,系统总体性能会出现什么变化?

- 如果 CPU 的性能提高 10 倍
- 如果 CPU 的性能提高 100 倍

解 假设原来的程序执行时间为 1 个单位时间。如果 CPU 的性能提高 10

倍,则程序的计算(包含 I/O 处理)时间为

$$(1 - 10\%)/10 + 10\% = 0.19$$

即整机性能只能提高约 5 倍,差不多有 50% 的 CPU 性能浪费在 I/O 上。

如果 CPU 性能提高 100 倍,程序的计算时间为

$$(1 - 10\%)/100 + 10\% = 0.109$$

而整机性能只能提高约 10 倍,表示有 90% 的性能浪费在没有改进的 I/O 上了。

对于现代计算机,往往认为通过操作系统的多进程技术可以回避 I/O 处理时间的问题。当一个进程在等待 I/O 处理的时候,另外的一些进程可以在 CPU 上运行。

这里从几个角度来看这个问题。首先,多进程技术增加的是系统吞吐率,而不能够减少响应时间。对于所有用户来说,他们最关心的是响应时间,也就是它们的计算任务需要多少时间可以完成。这一点在交互式系统,如工作站、PC 等环境下更为突出。实际的多进程系统中,一旦因为进程等待 I/O 而切换进程,现场交换往往也需要 I/O 操作,从而产生新的等待。第二点,很多台式计算机,尤其是 PC 类计算机,往往是一个人使用,根本没有很多的进程,因此也无从进行进程切换,用户只能等待 I/O 完成。

实际上,I/O 性能无论对系统的性能还是对系统的效率都有很大的影响。

6.2 存储设备

I/O 系统所涉及的设备数量、种类、类型都比较繁杂,对计算机系统整体性能的影响也大不一样。本节将重点讨论存储设备,它们对计算机系统的整体性能影响最大。

6.2.1 磁盘设备

目前常用的存储设备主要有磁盘、磁带、光盘等。这里主要讨论这些设备的一些知识以及它们对计算机系统设计的影响。

1. 磁盘

从 1965 年以来,尽管许多人在努力想用其他存储介质来取代磁盘,但磁盘始终占据着后备存储器的主宰地位。原因有二:第一,磁盘一直是虚拟存储器技术的物质基础,执行程序时,磁盘用作为交换缓冲区。第二,关机时,磁盘作为操作系统和所有应用程序的非易失性的驻留介质。这里不讨论软盘,而着重于硬盘。

磁盘由一组绕轴旋转的盘片组成,盘片的数量为 1~20 片,目前磁盘系统的

转速一般在每分钟 3 600 转到 12 000 转之间，即 $3\text{ 600 r/min} \sim 12\text{ 000 r/min}$ (r/min 又记为 rpm)。

盘片直径从 1.3 in(相当于 3.3 cm)到 14 in(35.6 cm)。每个盘片的表面分成以中心为圆心的磁道，每条磁道存放信息，典型情况下，每个盘片有 500~2 500 条磁道。每条磁道又分成若干扇区，扇区是磁盘进行存储分配的物理基本单元，每条磁道一般分为几十到几百个扇区。扇区之间留有空隙。参见图 6.1。一般情况下，所有磁道具有相同数目的扇区。由于靠外面的磁道较长，所以其记录密度比内部的磁道低。对应的还有一种让外道有更多扇区的方式，叫做“等位密度”，随着智能接口标准(如 SCSI)的推广和普及，这种方式也比较常见了。而 IBM 大型机甚至允许用户选择扇区的容量。

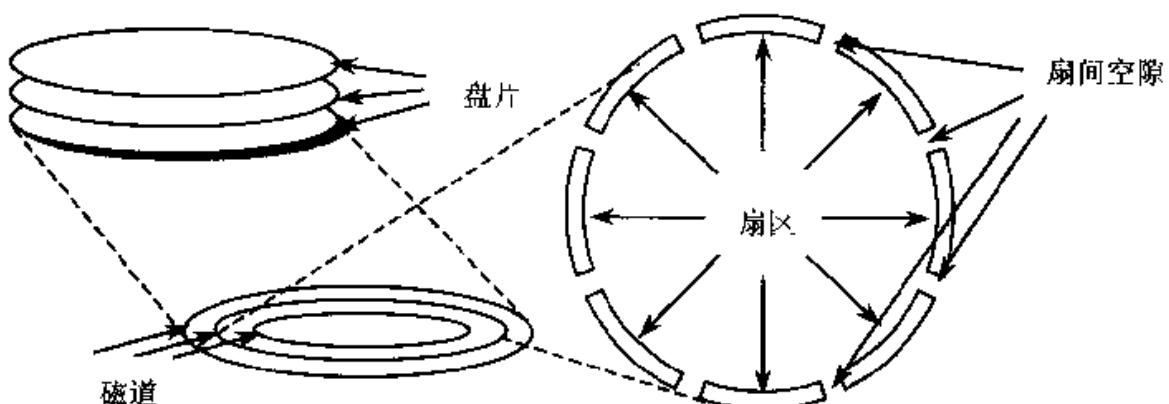


图 6.1 磁盘的盘片、磁道和扇区

一般每个盘片配一个磁头进行读写，磁头位于磁头臂上，通过磁头臂移动把磁头放到确定的道上。磁盘上的数据采用某种游程长度受限码进行编码，以提高磁盘表面的数据记录密度和提高数据读写的可靠性。磁盘的磁头都连在一起，同时运动。我们将同时处于一组磁头下的所有磁道称为一个柱面，以方便计算和分析。可将磁盘性能的公式定义为

$$\text{访问磁盘时间} = \text{寻道时间} + \text{旋转时间} + \text{传输时间} + \text{控制器开销}$$

下面逐个讨论这些参数。

(1) 寻道时间(seek time)

若要读写扇区，磁盘控制器首先发出命令将磁头移动到包含有所需数据的磁道上，这个过程称为“寻道”，所需要的时间叫做“寻道时间”。磁盘厂家在产品手册上要列出磁盘的最小寻道时间、最大寻道时间和平均寻道时间。最小寻道时间和最大寻道时间可以通过实际测量获得，而平均寻道时间的处理方法就多种多样了。工业界使用的平均寻道时间是统计结果，是磁盘上所有磁道寻道时间的和除以寻道次数(或者磁道总数)。常见的平均寻道时间的公布值约为 $6\text{ ms} \sim 20\text{ ms}$ ，但是由于磁盘访问的局部性(程序局部性所致)，实际应用当

中的平均寻道时间为公布值的 25%~33%，

(2) 旋转时间(rotation latency 或 rotation delay)

所需扇区转到磁头之下所需要的时间称为旋转时间，大部分磁盘的转速在 3 600 r/min 到 10 000 r/min，平均延迟是磁盘转半圈的时间，所以对大部分磁盘的平均旋转时间 T_{AR} 在

$$T_{AR} = \frac{0.5 \text{ r}}{3600 \text{ r/min}} = \frac{0.5 \text{ r}}{3600 / 60 \text{ r/s}} = 0.0083 \text{ s} = 8.3 \text{ ms}$$

和

$$T_{AR} = \frac{0.5 \text{ r}}{10000 \text{ r/min}} = \frac{0.5 \text{ r}}{10000 / 60 \text{ r/s}} = 0.003 \text{ s} = 3.0 \text{ ms}$$

之间。

注意，磁盘访问包含两个机械部分：磁头移动到期望的磁道上和其后期望的扇区旋转到磁头下，这两部分工作所需时间为毫秒量级。

(3) 传输时间(transfer time)

传输时间是指在磁头下传输一个数据块（通常是一个扇区）所花的时间。传输时间由块的大小、旋转速度、磁道记录密度和连接磁盘电子器件的速度确定，与传输时间对应的重要 I/O 参数是数据传输率。

现在磁盘驱动器上都有用半导体存储器组成的数据缓冲存储器。容量为 64 KB~1 MB。从盘面上读出的数据先送到缓冲存储器，再从缓冲存储器经过接口送到主机。因此数据传输率有两个：一是从盘面到缓冲存储器的内部传输率；一是从缓冲存储器到主机的外部传输率。内部传输率等于记录的位密度乘以盘面旋转的线速度。因此提高盘转速有助于提高传输率。现在磁盘驱动器中盘转速从 3 600 r/min 提高到 10 000 r/min，最高的如希捷(Seagate)公司的 ST19101FC 磁盘的转速已达到 10 000 r/min，它的内部传输率达 177 Mb/s。外部传输率则与接口有关。随着内部传输率的提高，磁盘接口技术也在不断改进。现在已有 IDE、EIDE、Ultra-EIDE、SCSI、Fast and Wide-SCSI、FC-AL 等接口得到大量应用，其中 Ultra-EIDE 接口的传输率为 33 MB/s，许多驱动器都已接近或超过这一指标。SCSI 接口也有多种标准出现，Fast and Wide-SCSI 已达到 80MB/s；更高的则是 FC-AL 接口，达到 200 MB/s。网络服务器和多媒体技术需要这样高的传输率。现在已有采用 FC-AL 接口的磁盘驱动器问世。

控制磁盘及磁盘与主存之间的数据传输，需要一系列的控制器和通道来完成。其复杂度因计算机成本而异。例如，因为传输时间只占磁盘整个访问的一小部分，所以希望在磁盘访问的时候断开与存储器的连接，而让其他部分向主存传输数据。另外期望通过一次访问较多内容（如多个扇区）来缓冲这种访问延迟，这也叫预读（Read Ahead），这样，附近的请求就能访问刚刚取来的扇

区。因此，磁盘访问时间还要加上一部分，即控制器时间，这个时间是控制器在执行 I/O 访问时的额外开销。另外，除了以上四部分时间外，还要加上等待磁盘空闲的时间（即排队延迟）。

例 6.2 对于目前一般的磁盘而言，读或写一个 512 字节的扇区的平均时间是多少？假设此时磁盘空闲，这样没有排队延迟；公布的平均寻道时间是 9 ms，传输速度是 4 MB/s，转速是 7 200 r/min，控制器的开销是 1 ms。

解 平均磁盘访问时间

$$\begin{aligned}
 &= \text{平均寻道时间} + \text{平均旋转延迟} + \text{传输时间} + \text{控制器开销} \\
 &= 9 \text{ ms} + \frac{0.5}{7200 \text{ r/min}} + \frac{0.5 \text{ KB}}{4.0 \text{ MB/s}} + 1 \text{ ms} = 9 \text{ ms} + 4.2 \text{ ms} + 0.125 \text{ ms} + 1 \text{ ms} \\
 &= 14.3 \text{ ms}
 \end{aligned}$$

假设实际测得的寻道时间是公布值的 33%，则答案是：

$$3 \text{ ms} + 4.2 \text{ ms} + 0.1 \text{ ms} + 1 \text{ ms} = 8.3 \text{ ms}$$

表 6.1 给出了希捷公司三种磁盘的特性，这三种盘表示了到 1997 年为止磁盘的最大密度，这些磁盘均采用 SCSI 或快速 ATA 标准总线接口。

表 6.1 希捷公司三种磁盘的特性参数

特 性	Seagate ST423451	Seagate ST19171	Seagate ST92255
磁盘直径(in)	5.25	3.50	2.50
格式化后容量(MB)	23 200	9 100	2 250
盘面数	28	20	10
MTF(b)	500 000	1 000 000	300 000
转速(r/min)	5 400	7 200	4 500
内部传输速度(Mb/s)	86~124	80~124	可达 60.8
外部接口	快速 SCSI-2 (8~16 位)	快速 SCSI-2 (8~16 位)	快速 ATA
外部传输率(MB/s)	20~40	20~40	可达 16.6
最小寻道时间(磁道间)(ms)	0.9	0.6	4
平均寻道时间+旋转时间(ms)	11	9	14
电源/机箱(W)	26	13	2.6
MB/W	892	700	865
体积(in ³)	322	37	8
MB/in ³	72	246	273

(4) 磁盘成本

磁盘的成本取决于许多因素：磁头、磁盘片、磁盘的机械部分、控制电路、电源及电缆。驱动能力较小的磁盘机在功耗和体积上占很大优势。目前，每兆字节的成本与直径已无太大关系，但与可靠性有关。

增大单片或者单盘的容量是降低磁盘每兆字节成本最有效的方法。因为机械电子部件的成本可以平均分配到单位容量中去，这也是制造大容量磁盘的原因之一。另外一个降低成本的方法是提高产量，它可以将设计和生产设备的费用平均分配到每个成品中去，较小容量的磁盘正是以这种方法来降低和抵消制造成本的。所以具有每兆字节最佳价格的是中等容量的磁盘，这种磁盘有足够的容量来抵消机械电子部件的开销，并且有足够的产量来保持价格优势。

磁盘在容量、体积和功耗上还会不断改进。旋转式的磁盘工作方式不会有太大变化，磁盘的转速会不断加快，但由于是机械动作，所以增幅不会太大。过快的速度，会对生产工艺带来较大的技术复杂性，比如高速运转会导致盘片变形，从而使磁头定位会出现问题等。

磁盘记录数据的密度一般用“磁表面记录密度”来表示，也就是每平方英寸上的位数，即

$$\text{磁表面记录密度(每平方英寸)} = \text{表面道密度(每英寸)} \times \text{道上位密度(每英寸)}$$

在 1988 年之前，磁表面记录密度每年增长约 29%，即每 3 年翻一番；1988 年以后，磁表面记录密度每年增长提高到 66%，即每 3 年翻两番，与传统的 DRAM 增长达到同步。到 1996 年，商品化产品的最高密度是 2.6 Gb/in^2 ；而 1998 年实验室中已达 500 Gb/in^2 。

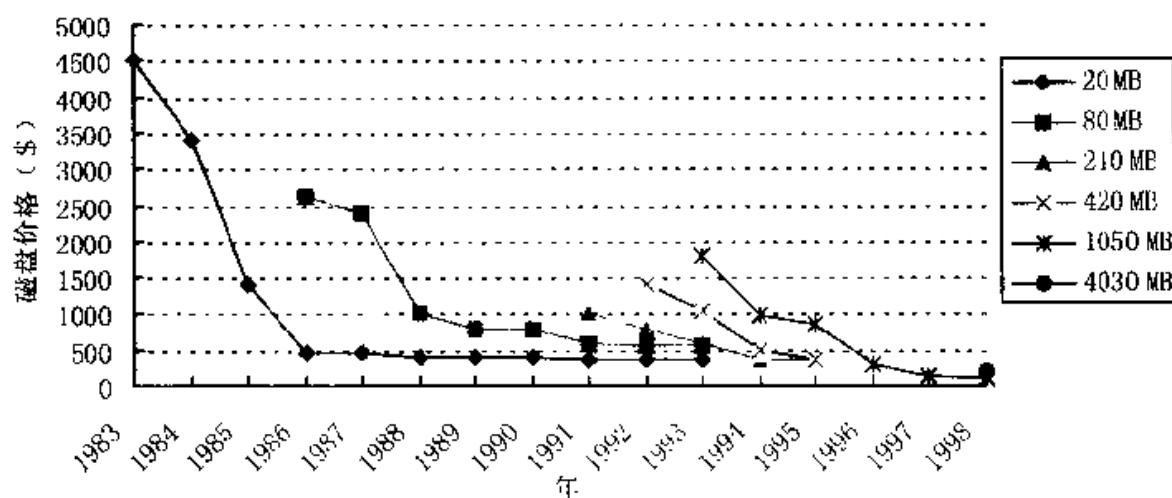


图 6.2 磁盘价格的变化

每兆字节的价格以和磁表面密度提高同样的速率下降，在这方面，小型驱动器扮演了重要的角色。图 6.2 列出了在 1983 年到 1998 年之间每个 PC 机磁盘

的价格,同时表明了价格的迅速下降和容量的增加,图 6.3 将这些开销转换成每兆字节的价格,表明在十几年中其降低了 100 多倍。实际上,在 1992 年和 1998 年之间,PC 机磁盘的每兆字节价格降低速率大约是每年 2 倍。

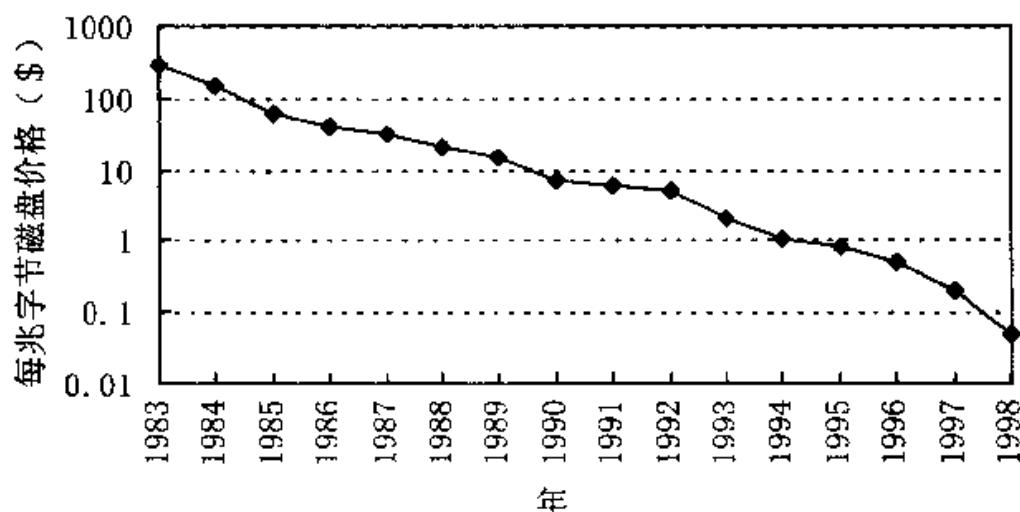


图 6.3 每兆字节磁盘价格的变化

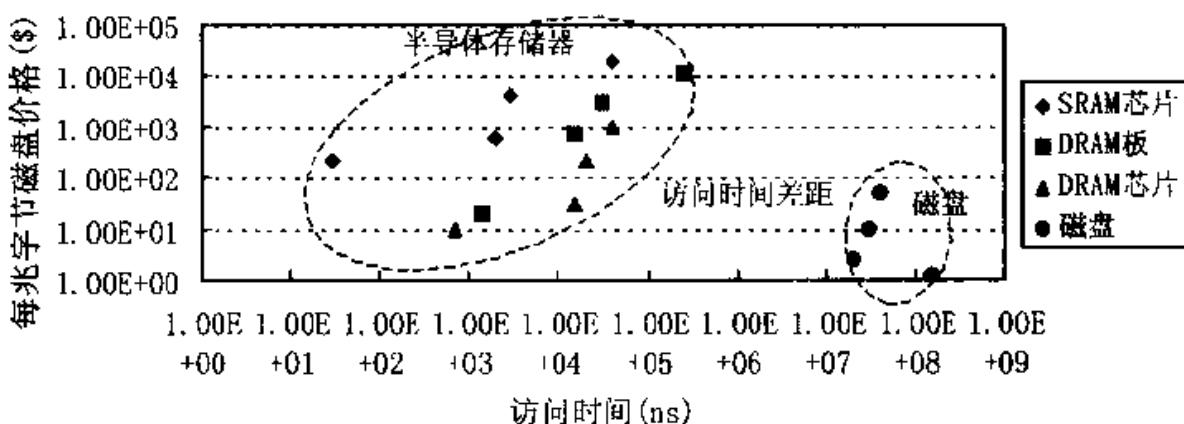


图 6.4 磁盘和半导体存储器之间的访问时间差距

因为物体小容易旋转,所以若容量相同,小磁盘会节省电能。同样,小驱动器磁道少,所以寻道距离短。因此,提高密度(在一个磁道上每英寸的位数)既可提高传输速度,又能改善寻道时间。目前,3.5 in 和 2.5 in 驱动器技术已经普及,转速已经从 20 世纪 80 年代标准的 3 600 r/min 提高到当前的 7 200 r/min。

磁盘在后备存储器中的地位曾受到过多次考验,主要原因就是所谓“访问时间差距”问题(如图 6.4 所示),意指磁盘的性能价格比高于主存,但访问速度却要低得多,换句话说,性能价格比与速度要求差距太大。1995 年磁盘每兆字节的价格比系统中 DRAM 每兆字节的价格便宜 100 倍,但是 DRAM 大约要快 100 000 倍,所以许多人都致力于发明能够填补这种访问时间差距的技术,但是至今都失败了。

到目前为止,与磁盘竞争的产品都未能把握正确的时机出现于市场,每当这些新产品问世时,磁盘通过改进性能和降低价格,仍牢牢地占据着市场。

2. 半导体盘

半导体盘是按磁盘访问方式工作的 DRAM 阵列,在按块访问的存储设备领域中,它是磁盘的有力竞争者。半导体盘只有固态盘(SSDs)和扩充存储器(ES)两种。固态盘是由 DRAM 阵列加上一个保持系统存储数据的电池构成;扩充存储器是一个只支持块传送的大容量存储器中的一个区域,如 DOS 操作系统环境下的 EMS 区域。

ES 好像是由软件控制的 Cache(在块传送的时候 CPU 暂停),而 SSD 在操作系统看来就是一个磁盘。SSD 和 ES 的优点是永久性、速度快、高传送速率和高可靠性。它们不仅仅是一个大存储器,而且是可以独立地、用特殊的命令来访问,因此较少产生读写错误。SSD 和 ES 的块访问方式可以将纠错码扩展到更多的字上,可以用较小的开销支持较高的错误恢复率。例如,IBM 的 ES 使用较高的错误恢复率使其可用性较高,因此可以用不太可靠(但较便宜)的 DRAM 生产 ES。SSD 的不同之处在于 SSD 作为单独设备可以由多个 CPU 共享。将半导体盘放在 I/O 设备中,可以克服现有 32 位计算机中的地址空间限制。SSD 和 ES 的最大缺点是成本太高,每兆字节的价格大约是磁盘价格的 50 倍。

3. 盘阵列(RAID)

盘阵列(RAID),即 Redundant Array of Inexpensive Disks,即廉价磁盘冗余阵列,简称盘阵列技术。1988 年,美国加州大学 Berkeley 分校的 Patterson 教授等人提出了廉价磁盘冗余阵列(RAID)的概念。以此作为盘阵列技术发展的起点,在不到 10 年的时间里,盘阵列技术已经完成了从高速发展到逐步成熟的过程。它是目前解决计算机 I/O 瓶颈的有效方法之一,有着广阔的发展前景。计算机及其相关技术的飞速发展和计算机应用的日益普及,对计算机存储技术提出了越来越高的要求。因此,盘阵列这种容量大、速度快、可靠性高、造价低廉的存储技术越来越被人们所重视。下面简单介绍各级 RAID 的结构特点(表 6.2)。

(1) RAID0

亦称数据分块,即把数据分布在多个盘上,实际上是非冗余阵列,无冗余信息。严格地说,它不属于 RAID 系列。

(2) RAID1

亦称镜像盘,使用双备份磁盘。每当数据写入一个磁盘时,将该数据也写到另一个冗余盘,这样形成信息的两份制品。如果一个磁盘失效,系统可以到镜像盘中获得所需要的信息。镜像是最昂贵的解决方法。特点是系统可靠性很高,但效率很低。

(3) RAID2

位交叉式海明编码阵列。原理上比较优越,但冗余信息的开销太大,因此未被广泛应用。

表 6.2 RAID 盘阵列分级

RAID 级	数据磁盘数	可正常工作的最多失效盘数	检测磁盘数
0 非冗余	8	0	0
1 镜像	8	1	8
2 存储器式 ECC	8	1	4
3 位交叉奇偶校验	8	1	1
4 块交叉奇偶校验	8	1	1
5 块交叉分布奇偶校验	8	1	1
6 P+Q 冗余	8	2	2
7 Cache + 异步	8	2	2

(4) RAID3

位交叉奇偶校验盘阵列,是单盘容错并行传输的阵列。即数据以位或字节交叉的方式存于各盘,冗余的奇偶校验信息存储在一台专用盘上。在这种组织中,不是将全部数据备份,而是存储足够的冗余信息,能够在磁盘出故障时使数据得到恢复。

在 RAID3 中,将磁盘分组,读写要访问组中所有盘,每组中有一个盘作为校验盘。校验盘一般采用奇偶校验法,当一个磁盘出故障时,可以通过奇偶校验磁盘中的校验和来恢复出错数据。这种结构可以简单地理解为:先将分布在各个数据盘上的一组数据加起来,将和存放在冗余盘上。一旦某一个盘出错,只要将冗余盘上的和减去所有正确盘上的数据,得到的差就是出错盘上的数据。RAID3 奇偶校验通常是模 2 和。这种方法的缺点是恢复时间较长,但由于磁盘出错很少,因此人们可以接受。镜像方法其实是 RAID3 的一种特殊情况,即一个组中只有两个盘,一个做数据盘,一个做奇偶校验磁盘,并且奇偶校验是通过复制数据来实现的。

(5) RAID4

专用奇偶校验独立存取盘阵列。即数据以块(块大小可变)交叉的方式存于各盘,冗余的奇偶校验信息存在一台专用盘上。

(6) RAID5

块交叉分布式奇偶校验盘阵列,是旋转奇偶校验独立存取的阵列。即数据以块交叉的方式存于各盘,但无专用的校验盘,而是把冗余的奇偶校验信息均匀地分布在所有磁盘上。图 6.5 实际上还包括 RAID3,它的数据分布与 RAID4 相

同。在 RAID3 中,每次都要访问所有盘(按条访问),用于计算校验和。而在 RAID4 中,允许读/写一个数据盘和校验盘(按块访问),多个对磁盘的访问可以并行执行,这样就可以提高访问速度。

在图 6.5 中,RAID3 一次写操作将需要读其他三个磁盘的数据,与需要写入的信息一起计算校验和。然后将新的数据写入数据盘,将新的奇偶校验和写入奇偶校验盘。

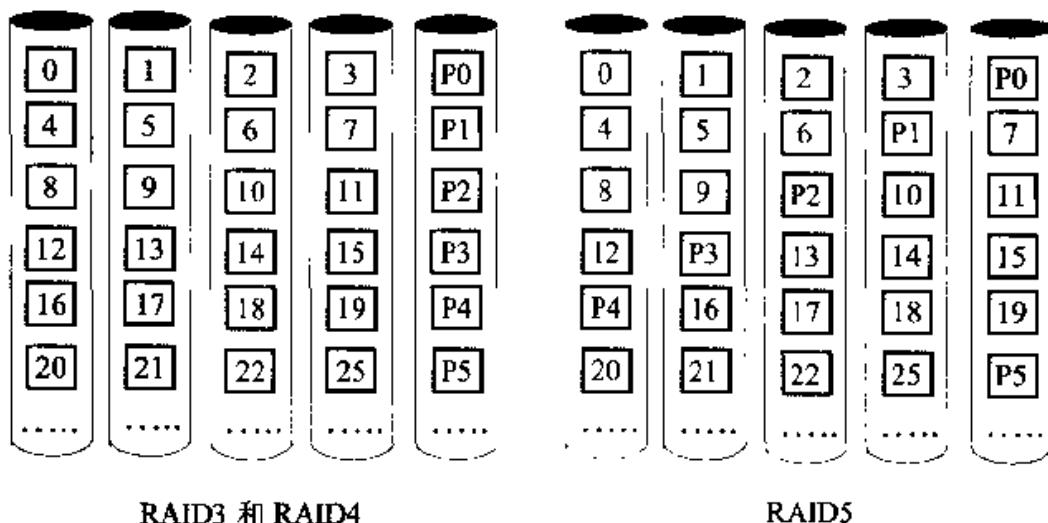


图 6.5 RAID 3、RAID 4 和 RAID5 中的校验

要降低这种额外开销,关键是奇偶校验和计算要简单。在写新数据的时候,只检查已改变的位,并修改奇偶校验盘中相应的位。在 RAID4 中,先根据新数据的位置读出旧数据及旧的奇偶校验和,将旧数据和新数据比较找出改变位,再改变旧的奇偶校验和的相应位,然后将新数据和新奇偶校验和写入。所以,在 RAID4 中,一次写其实是对两个磁盘的四次访问。这样便代替 RAID3 中对所有磁盘的访问。

RAID4 这种方法可以支持多种类型的读写访问,缺点是奇偶校验磁盘必须在写时被修改,从而奇偶校验磁盘形成顺序写的瓶颈。为了解决奇偶校验写的瓶颈,奇偶校验信息可以分布在所有磁盘中,这样就不会形成写操作的单个瓶颈,图 6.5 表明了数据在这种磁盘阵列组织中的分布情况,这就是 RAID5。

对于 RAID5,数据块每一行的相联奇偶校验不再限制在单一磁盘上,只要块单元不位于同一个磁盘内,这种组织方法就可以支持多个写同时执行。例如,写块 8 时必须同样访问奇偶校验块 P2,所以要占用第一个和第三个盘,写块 5 时,同样要访问奇偶校验块 P1,要访问第二和第四个磁盘,这样,两次写可以同时执行。

(7) RAID6

双维奇偶校验独立存取盘阵列。即数据以块(块大小可变)交叉的方式存于

各盘,冗余的检、纠错信息均匀地分布在所有磁盘上。并且每次写入数据都要访问一个数据盘和两个校验盘,可容忍双盘出错。

(8) RAID7

RAID7 是采用 Cache 和异步技术的 RAID6,使响应速度和传输速率有了较大提高。

RAID7 可以达到每秒几十兆字节的吞吐率,同时又能从故障中恢复数据,具有很高的可用性,所以非常受欢迎,RAID7 在外存储系统中所起的作用将不断增强,并会获得广泛应用。盘阵列技术发展如此之快的原因主要有以下几点:

(1) 计算机技术的高速发展使之与外存储器之间的 I/O 瓶颈问题日益突出,从而使盘阵列技术的应用有了迫切的需求背景。

(2) 在以前的磁盘应用中,一些单项技术(如交叉存取、数据分块、缓冲存储等)实际上已为盘阵列技术做了很好的前期准备。

(3) 在近年来 PC 机发展的推动下,小盘驱动器的性能与大型磁盘机相比有了更高速的发展,从而使盘阵列技术的发展在硬件资源上有了前提条件。

(4) 微电子技术的发展,使能支持盘阵列技术的大规模芯片成为可行。反之,盘阵列技术的发展也为本身开拓了更加广阔的应用前景。

随着盘阵列技术的逐步成熟,目前世界上已有几十家主要的盘阵列厂商在生产和销售各种级别的 RAID 产品。其中,处于领导地位的主要有 HP、DEC 和 DG 等。目前实现盘阵列的方式主要有三种:

(1) 软件方式,即阵列管理软件由主机来实现。其优点是成本低,缺点是要过多地占用主机时间,并且带宽指标上不去。

(2) 阵列卡方式,即把 RAID 管理软件固化在 I/O 控制卡上,从而可不占用主机时间,一般用于工作站和 PC 机。

(3) 子系统方式,这是一种基于通用接口总线的开放式平台,可用于各种主机平台和两络系统。对以上三种方式,可根据具体应用对象进行选择。

目前,盘阵列技术研究的主要热点问题有以下几个方面:

(1) 新型阵列体系结构;

(2) RAID 结构设计与其所记录文件特性的关系;

(3) 在 RAID 冗余设计中,综合平衡性能、可靠性和开销的问题;

(4) 超大型盘阵列在物理上如何构造和连结的问题等。

随着这些问题的解决,盘阵列的性能将会大幅度提高。像其他计算机外存一样,盘阵列也将向大容量、高传输率、高可靠性以及体积更小、重量更轻、功耗更小的方向发展。

6.2.2 磁带设备

1. 磁带

磁带与磁盘几乎是同时成为计算机系统外部存储设备的,因为它采用的磁记录技术与磁盘类似,所以具有相同档次的记录密度,磁盘和磁带性能价格比的差异主要取决于它们的机械构成:

(1) 磁盘盘片具有有限的存储面积,并且存储介质被封装在每个读部件内,提供毫秒级的随机访问;

(2) 磁带绕在可转动轴上,一个读部件可以使用多盘磁带(没有长度限制),但磁带需要顺序访问,每次访问都可能需要较长的反绕、退出和加载时间,等待时间较长(数秒)。

磁带的最大优点是容量极大、技术成熟和单位价格低廉,最大的缺点是访问时间较长。这种差异恰好使得磁带成为磁盘的备份技术。

磁带技术的主要受限因素是其线速度不定。为解决该问题,提出了“螺旋扫描磁带(Helical Scan Tapes)”,使磁带保持同样的线速度,这种技术以20~50的倍数增加记录密度,螺旋扫描磁带目前已被普遍使用在视频录像设备中,大大降低了磁带和读部件的开销。

磁带的另外一个缺点最易磨损,螺旋磁带只能使用几百遍,传统的高质量磁带则可以使用几百万次。螺旋扫描磁头同样容易磨损,通常额定指标为连续使用2 000小时。

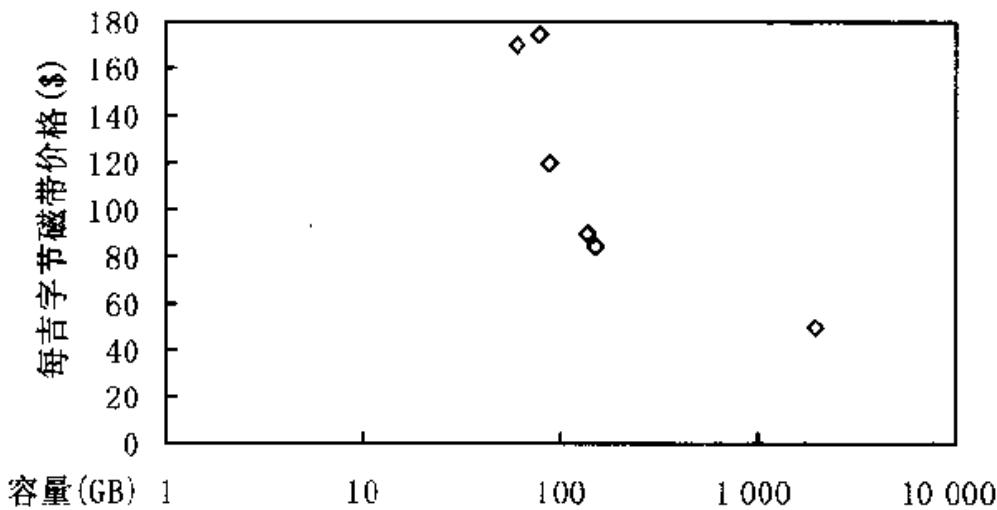


图 6.6 磁带容量与吉字节(GB)磁带价格的关系

2. 自动磁带库

虽然磁带机可以读写“无限长”的磁带,但需要操作员来更换磁带,为了解除操作中的负担,同时也为了加速换带速度,便产生了自动磁带库。自动磁带库通

过机械手自动地安装和更换磁带,相当于又提供了一个新的存储器层次,这种自动化的磁带库可在无人工干预的情况下,十几秒内访问几太字节(TB)的信息。STC 的 PowderHorn 可以处理 6 000 个磁带,提供的总容量达 60 TB。

自动磁带库可以通过加大规模,以达到降低成本(单位价格)的目的。图 6.6 表明从小系统(1995 年小于 100 GB)到大系统(大于 1 000 GB)每吉字节价格的变化情况。在大系统中,这种海量存储器的主要缺点就是带宽受限。

6.2.3 光盘设备

另一种竞争存储设备市场的是光盘,光盘是通过激光来读写的,主要有只读类(如 CD-ROM、DVD-ROM 等)和可写类(如 CD-R、MO 等)。另外成组的光盘设备也可以构成高性能的阵列设备。

1. 光盘

最常用的只读类光盘的全称为光学至密盘(Optical Compact Disk),简称 CD-ROM,它的特点是容量大(640 MB)、成本低、便于保管、存储寿命长(数据可稳定保存 20 年)、便于携带、读出设备价格便宜等,最大问题是不能够写入。因此 CD-ROM 适于作为软件和资料的载体,目前基本上替代了几年前广泛使用的软盘。

可写类光盘包括两类,一次性写和多次写。一次性写的光盘通常称为可记录光盘 CD-R(CD-Recordable),又称为 WORM(Write Once Read Many)盘,出厂时是空白的,用户通过写入设备将数据写入 CD-R 中。写入以后的 CD-R 和 CD-ROM 一样使用。CD-R 盘片和读写设备的成本都略高于 CD-ROM,其他各方面的特性与 CD-ROM 相当,并且可以通过普通的 CD-ROM 读设备读出,因此特别适合于作为数据备案的存储介质。

多次写光盘又称为 WMRM(Write Many Read Many)盘,主要采用磁光(MO)存储技术。MO 产品自 20 世纪 80 年代末、90 年代初开始进入市场,随后不断更新换代。目前,市场上的 MO 驱动器主要有 3.5 in 和 5.25 in 两种。此外,也有少数公司生产 8 in(如富士通)和 12 in(如尼康)的产品,但性能稍差。MO 光盘的容量更大(有 600 MB、1.2 GB、2.4 GB 等规格),保存携带和使用都很方便。MO 最大的问题是目前盘片和读写设备的价格昂贵,且各厂家的标准不统一,因此不够普及。这种 WMRM 盘作为大型软件编制、多媒体软件产品研制过程中的备案介质是非常合适的。

2. 盘库、光盘塔和光盘阵列

随着 CD-ROM 计算机光盘技术的发展,光盘的存储量大;价格低,受到人们的欢迎。同时 CD-Recorder 作为一种崭新的存储方式更加拓宽了 CD-ROM 的应用领域,使人们能够轻松地将自己的数据、文档等信息资源存于 CD 光盘中,

因此实现光盘资源共享就越来越受到人们的关注。随着计算机网络技术的发展,光盘网络的优势与重要性已日益显现出来,它有效地实现了光盘资源在网络中共享,极大地提高了光盘的利用率,成为未来人们不可抑制的趋势。

将多台光盘机组合在一起有三种结构,分别是光盘库(也叫自动换盘机,即 Jukebox)、光盘塔(CD-ROM Tower)和光盘阵列(CD-ROM Array)。

(1) 光盘库

光盘库是一种能自动把机框中存放的许多片光盘选出并装入光盘机进行读写的设备。它利用光盘可换的特点,可形成一个巨大的联机资料库,通常采用普通的光盘机和光盘片。最常见的是采用 CD-ROM 盘的自动换盘机。自动换盘机是利用机械手来装卸光盘。因此机械结构比较复杂,装卸光盘也需要较长时间。整机可放置的光盘片数量一般从几十片到上百片。这种联机容量巨大的设备对于图书馆之类的机构是很有用的,大量不常用的数据一旦需要就能尽快找到。各种光盘机如 CD-ROM 机,磁光盘机、相变盘机,各种规格如 360 mm(14 in)、305 mm(12 in)、130 mm(5.25 in)、90 mm(3.5 in)都可以组成光盘库。光盘库有如下几个优点:

① 费用低:CD 是一种费用极低的数据存储及检索媒介,其硬件成本只是其他光电存储设备成本的 30%,处理 1 MB 数据的成本少于 1 美分。目前一张可存储 650 MB 数据的 CD-R 只要人民币 20 多元,并且还在不断降价。

② 可兼容性及低风险:由于产品的标准化,CD 技术兼容于各种硬件及软件平台,用户可以很方便地将自己的 CD-R 数据盘片放入任何一台计算机的光驱(甚至以后的 DVD 驱动器)中,其通用性明显优于 MO,标准化降低了 CD 为适应未来技术变化所承担的风险。

③ 随机存取:磁带存储由于数据按顺序存储,而不允许随机读取。CD 则不同,它能以随机的方式更快地检索。

④ 耐用性:CD-ROM 可存放 100 年,CD-R 可存放 25 年,这都比其他存储介质的存储寿命长得多,而且保管容易,占用空间少。

⑤ 多媒体功能:CD 盘片支持连续、同步、数字化的音频和视频数据,而 CD 盘片的下一代产品 DVD 将支持更高质量的多媒体数据。装有 200 张 DVD 盘片的 DVD 库届时将可存储高达 1 TB~4 TB 的数据。

惠普公司的 HP SureStore Optical 330fx/600fx 就是一种光盘库。它最多可装配 12 台 130 mm(5.25 in)磁光盘机,可放置 238 片光盘,每片 2.6 GB,总容量是 618 GB。

德国 NSM 公司的 Satellite 系列光盘库,可以装下 135 张光盘片。在它正面 24 cm×40.9 cm 的面积内装着 5 个 135 cm 半高标准盒状光盘夹(pack)或者传送器(mail slot)。每个盘夹可装下 15 张 CD 盘片,并附带识别器,以便在更换

“离线(off-line)”的光盘夹时,光盘库不用重新扫描就能马上识别盘夹内的内容;而传送器则是用来传送单片 CD 的。

(2) 光盘塔

光盘塔实际上是多个 CD-ROM 放在一起,再加上相应的控制器和网络连接设备,构成一个网络存储设备。许多光盘机通过标准接口(如 SCSI)电缆连接起来,一根典型的 SCSI 接口电缆可以连接 7 台光盘机,用软件控制读写其中某一台。这种结构寿命比自动换盘机长,结构简单造价也低,读取光盘速度也快,但容量较小。光盘塔中光盘机数量受到 SCSI 设备地址数的限制。

图 6.7 为美国 CD Dimensions 公司的 AMS-CD07327-7T 光盘塔。北京金盘公司的 GD-7-4X 光盘塔就是这种产品。它采用 SCSI-2 接口,有 7 台 CD-ROM 光盘机,可读 CD-ROM、CD-DA、CD-I、Photo CD 等格式。



图 6.7 光盘塔

光盘塔还可以通过网络连接,从而打破了 CD-ROM 驱动器个数的限制,使 CD-ROM 光盘塔在网络上的使用轻松、方便、快捷和随意,就像用户在使用自己的服务器一样。作为独立的服务器,它不需要使用网上的文件服务器或工作站资源。通过它将 CD-ROM 数据放在局域网上,有助于最大限度地减少网络阻塞。直接上网的光盘塔,体现了网络提供 I/O 功能的主要特点。

① 安装简便。将光盘塔直接挂接在网线上或集线器上,无须中断关闭网络,即安装即用。

② 易于管理。光盘塔上网后,一个光盘塔对应一个盘符;光盘塔的光盘驱动器分别对应这一盘符下的子目录。在网络中光盘塔可以同时被多个用户访问,也允许多用户同时访问光盘塔中的某一片光盘。

③ 使用方便。在 NetWare 和 Windows NT 网上, 用户将光盘塔作为 NetWare 或 NT 的卷来使用;对于 NFS 用户来说光盘塔被认作是一个 UNIX 远程服务器, 只需给它一个 IP 地址, 用户即可通过 IP 地址访问; 在广域网中光盘塔被看成是一个独立的 Web 服务器。

④ 资源共享 光盘塔中的数据资源能同时为网络中的所有用户共享, 网上的用户在不同地点可通过网络终端同时访问光盘塔中的每一片光盘。

⑤ 远程访问 用户可以将计算机通过电话线连通网络, 根据需要访问网络中的所有光盘。

(3) 光盘阵列

从阵列技术的基本原理来说, 光盘阵列与磁盘阵列有一定的相似性。但光盘具有盘片可换、每道(柱面)只有一个读写头、寻道时间较长等特点, 因此光盘阵列技术又有其特殊性。

首先要考虑盘片可换的问题。一方面, 光盘机及其所装的盘片可能有对应关系问题;另一方面, 在线工作盘片和离线备用盘片在内容上可能有逻辑联系问题。因此, 在光盘阵列中必须考虑可换光盘的有序管理问题。如果采用自动换盘机构, 则阵列管理软件要对光盘库中的所有盘片进行编址, 并能控制自动换盘机构按址换盘; 如果采用人工换盘, 则管理软件要能标记和识别光盘片的序号, 并要给出换盘顺序的提示等。

又由于光盘的寻道和旋转等待时间比磁盘长, 而且每道只有一个读写头, 因此在大量数据连续读写时, 寻道的次数要比磁盘多。因此, 在设计光盘阵列时, 必须考虑尽量合理地分配数据存储位置, 以减少寻道次数和距离。通过合理调度, 尽量实现顺序操作, 以消除长距寻道及旋转等待。对光盘阵列而言, 较大的缓存和优化的调度策略显得特别重要, 可以说, 它是能否实现光盘阵列快速响应的关键技术之一。

6.3 总线

在计算机系统中, 必须将许多子系统连接起来, 例如, 存储器、CPU、I/O 设备等等, 这就需要一种统一的接口, 总线就是这样一种接口。

总线作为各子系统之间共享的通信链路, 两个主要优点就是低成本和多样性。通过定义一个统一的互连方法, 各种新型设备可以很容易连接起来; 对于连接到不同计算机系统中的外设, 由于采用一组标准的线路, 并且多设备共享, 所以开销较低。

总线的主要缺点是它必须独占使用, 造成了设备信息交换的瓶颈, 从而限制了系统中总的 I/O 垃吐量。当系统所有 I/O 操作必须通过一条中心总线时, 它

的带宽限制就和存储器带宽的问题一样了,有时可能还很严重。例如在巨型计算机中由于CPU速度非常高,所以I/O的速度也要很高,设计满足高性能处理器要求的总线系统是非常具有挑战性的工作。

总线设计存在很多技术难点,一个重要原因就是总线上信息传送速度极大地受限于各种物理因素,如总线的长度、设备的数目、信号的强度(因此要考虑总线的负载)等,这些物理因素限制了总线性能的提高。另外,对高速率(低延迟)的要求和对高吞吐量的要求也可能造成设计需求上的冲突。

6.3.1 总线分类

1. 按用途分类

总线可以分为“CPU-存储器总线”和“I/O总线”。由于要连接许多不同类型、不同带宽的设备,因此I/O总线比较长,并且还应遵循总线标准。而CPU-存储器总线则比较短,通常具有较高的速度,并且要和存储器系统的速度匹配来优化带宽。在设计阶段,CPU-存储器总线的设计人员应当了解所有相连设备,而I/O总线的设计人员不可能了解所有设备,但应了解设备的不同类型和不同能力(延迟和带宽)。为了降低成本,有些计算机对存储器和I/O设备使用同一条总线。

2. 按设备定时方式分类

按设备定时方式总线可分为同步总线和异步总线两大类。

同步总线上所有设备通过统一的总线系统时钟进行同步。同步总线成本低,因为它不需要设备之间互相确定时序的逻辑。但是同步总线也有缺点,总线操作必须以相同的速度运行。由于各种设备都要精确地以公共时钟为定时参考,因此在时钟频率很高时容易产生时钟相位漂移错误。

异步总线上的设备之间没有统一的系统时钟,设备自己内部定时。设备之间的信息传递用总线发送器和接收器控制。异步总线容易适应更广泛的设备类型,扩充总线时不用担心时钟时序和时钟同步问题。但在传输时,异步总线需要额外的同步开销。

6.3.2 总线基本工作原理

在深入细致研究总线之前,先描述一下总线的操作过程和定义一下总线的物理性质。下面介绍三个常用的参数。

(1) T_p : 总线信号传输延迟。即在总线上的每个设备都取到和识别一个信号需要的最大时间。

(2) T_{sk} : 响应其他设备的最大时间,这个参数在同步总线中是一个重要的参数。

(3) T_{op} : 设备的操作时间。

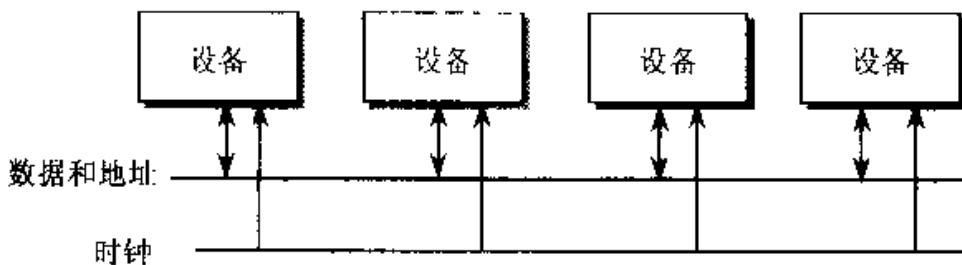


图 6.8 简单同步总线示意图

最简单的同步总线如图 6.8 所示,总线的周期必须大 T_p 和 $2T_{sk}$ 的和,单同步总线时序如图 6.9 所示,同步总线的实例有 PCI、NUBUS 和 Multibus 等。图 6.10 表示了异步总线的时序。异步总线有 FutureBus、VMEBUS 等。

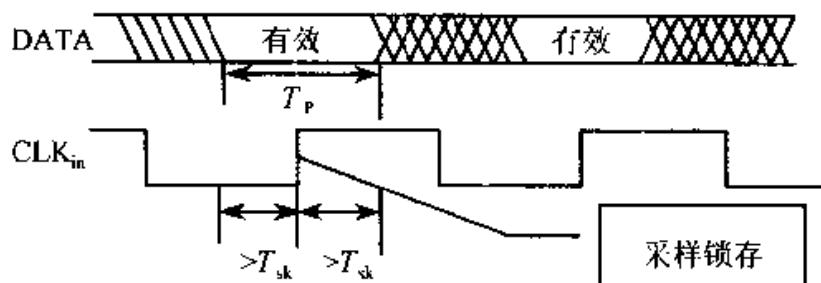


图 6.9 同步总线时序

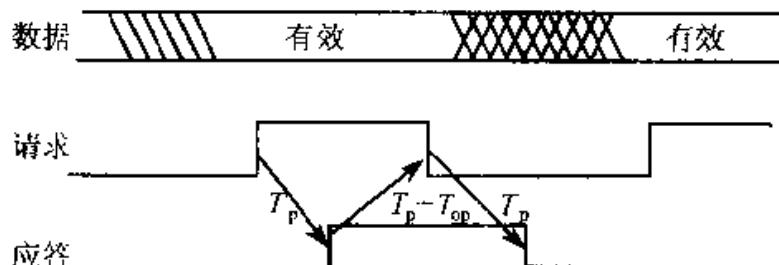


图 6.10 异步总线时序

所有总线必须有仲裁器,仲裁总线上设备对总线使用的请求。通过仲裁,分配总线使用权。只有获得总线使用权的设备才能够在总线上传送信息。

6.3.3 总线使用

总线的任务就是进行信息交换,主要包括读和写操作。对于 CPU—存储器总线,“读”是将存储器的数据送到 CPU 或外围设备,“写”是将数据写入存储器。一次总线传输分为两个部分:地址操作和数据传送。

下面讨论同步总线的数据传送过程。在读操作中,首先将访问单元地址和一些表明读的控制信号放到总线上。在图 6.11 中,读信号为低电平时表示有效,存储器则在接收到地址和读控制信号后,从总线上返回数据和适当的应答信

号来作为响应。等待信号为高电平时表示需要等待,存储器在送出数据以后,降低等待信号表示一次传送过程结束。一次写操作需要 CPU 或者 I/O 设备将地址和控制信号同时发出,数据信号可以同时发出,也可以稍后发出。

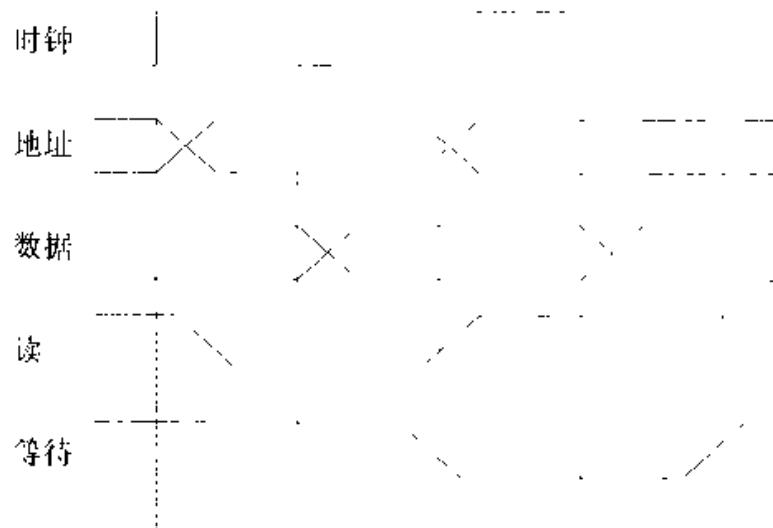


图 6.11 典型的总线读操作

总线的设计取决于要达到的性能和价格,表 6.3 给出了设计总线时需要考虑的一些问题。是否采用独立的地址和数据线、数据总线的位数和传输块大小这三点对总线的性能和价格的影响最为明显。高性能总线同时带来高成本。

表 6.3 总线的主要可选特性

选择	高性能	低价格
总线宽度	独立的地址和数据总线	分时复用数据和地址总线
数据总线宽度	越宽越快(64/128 位)	越窄越便宜
传输块大小	块越大总线开销越小	每次传送单字
总线主设备	多个(需要仲裁)	单个(无需仲裁)
分离处理	采用	不用
定时方式	同步	异步

若想使总线的成本低,可以采用地址和数据分时复用的总线;若要追求总线的高性能,就要采用独立的地址和数据总线。增加总线位数是直接改进性能最容易的方法,减少总线位数是降低价格最容易的方法。

总线主设备是能够初始化总线操作的设备,例如 CPU 总是总线主设备,当

有多个 CPU 或有多个能够初始化读写操作的 I/O 设备时, 总线就具有多个主设备。如果有多个主设备, 就必须要有总线仲裁机制来处理多个主设备使用总线时的竞争, 以选择由哪个主设备来获得总线使用权。

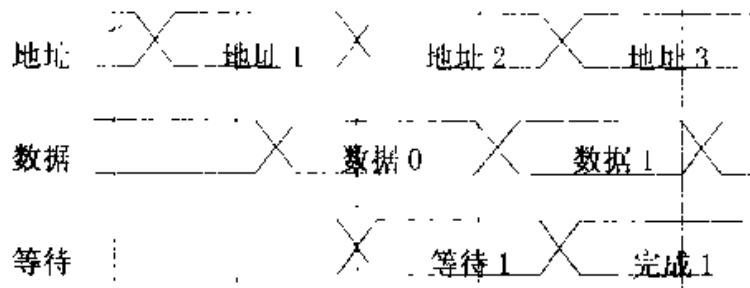


图 6.12 分离处理总线

在有多个主设备时, 总线就可以通过数据打包来提高总线带宽, 这样就不必在整个传输过程中都占有总线, 这种技术叫做“分离处理(split transaction)”或者“流水处理”或者“包开关总线”等, 图 6.12 为分离处理总线工作过程示意图。读操作分为两部分: 一个是包含地址的读请求和一个包含数据的存储器应答, 每个操作都必须标记清楚, 以便 CPU 和存储器可以识别它们。当从所请求的存储器地址处读一个字时, 分离处理允许其他总线主设备使用总线, 因此, CPU 必须能够识别总线上发来的数据, 存储器必须能识别总线上返回的数据。分离处理总线有较高的带宽, 但是它的数据传送延迟比独占总线方法要大。

表 6.3 的最后一项是“定时方式”, 选择总线是同步还是异步。同步总线的控制线中包含一个供总线上所有设备使用的时钟。对于地址和数据, 相对于时钟具有固定的时间要求(协议)。同步总线很少甚至不需要附加逻辑电路来决定下一步的动作, 所以这种总线既快又便宜。同步总线有两个主要缺点: 一是在总线上所有操作都要以同样的时钟速率进行; 二是由于时钟通过长距离传输后相位会漂移, 因而同步总线距离很短, 对于高速同步总线这个问题尤其严重。目前的 CPU-存储器总线通常是同步总线。

异步总线上没有统一的参考时钟, 每个设备自带时钟。总线上的发送设备和接收设备采用握手协议。图 6.13 是一个主设备在异步总线上执行一次写的过程。异步处理可以满足大量不同设备的连接, 传输距离较长, I/O 总线很多都是采用异步总线。

选择同步总线还是异步总线, 主要应考虑实际传输距离和可连接的设备数, 即不但要考虑数据带宽, 还要注意 I/O 系统的能力。同步总线通常比异步总线快, 因为它避免了传输时握手协议的额外开销。异步总线能够适应多种类型的不同设备。

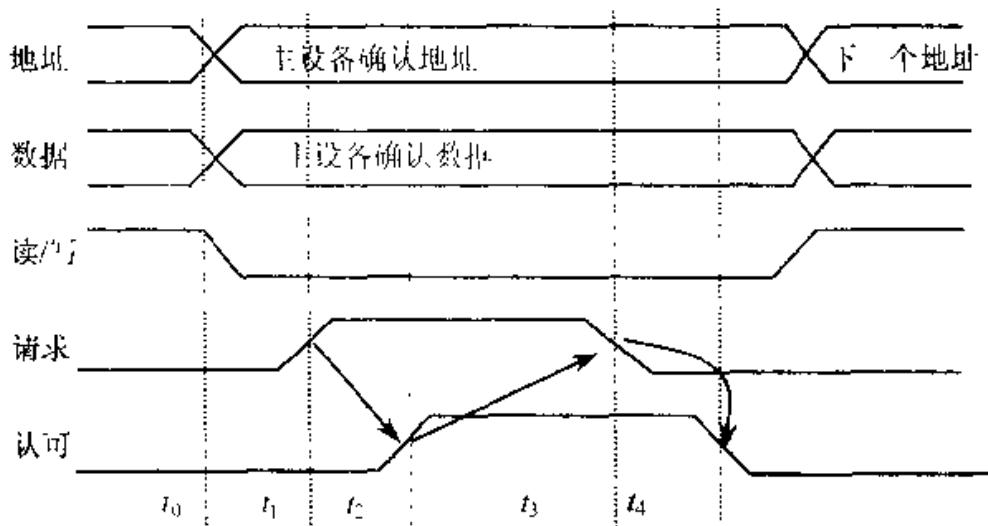


图 6.13 异步总线上设备执行写操作

6.3.4 总线标准和实例

在计算机系统中，I/O 设备的数目和性能差异很大，应该允许客户根据需要对计算机中的 I/O 设备进行增减。设备是通过接口连接的，所以增加设备可以看作是对 I/O 总线的扩展。通过制定标准，使计算机和 I/O 设备能够相互独立地进行设计，所以选择总线是系统设计中一项重要的工作。只要计算机和 I/O 设备的设计都满足相应的标准，那么 I/O 设备和计算机可以任意连接。实际上，I/O 总线标准就是定义设备连接的文件。

广泛使用的 I/O 总线可能成为事实上的标准，例如 PDP-11 的 UniBus 和 IBM PC-AT 的 ISA 总线就是如此。一旦某种 I/O 设备被某种大批量的机器采用，其他计算机中也会设计与这种设备的 I/O 接口，以便使用这种 I/O 设备。有时标准也来自于某些 I/O 设备制造商，“智能外设接口”(IPI)和 Ethernet 都是由制造商合作形成的标准。如果标准十分成功，最后将被 ANSI、ISO 或 IEEE 等标准化组织所接受，成为推荐标准。有些总线标准直接由标准委员会制定，例如 PCI 总线。

(1) PCI

PCI(Peripheral Component Interconnect, 即外围器件互连)是一种为 CPU 和外设之间提供高性能数据通道的总线，它由以 Intel 公司为首的一个 PCI 特别兴趣小组(Special Interest Group)制订并维护。PCI 总线是一种 32 位总线，可进行 32 位地址空间的寻址，也可成组传送数据。PCI 总线有一个特别的地址空间——配置空间(configuration space)，用于协助实现总线设备的即插即用功能。PCI 总线主要有以下一些特点：

- ① 数据线和地址线采用多路复用结构，减少了引脚数；

- ② PCI 总线定义了两种电信号标准环境:5 V 和 3.3 V;
- ③ 总线信号与处理器无关,可以支持多系列的处理器;
- ④ 透明的 32/64 位总线,允许 32 位和 64 位总线设备相互操作;
- ⑤ PCI 支持总线扩展和设备的自动配置。

(2) USB

USB(通用串行总线)端口试图取代串口、并口和 PS/2 的连接。提供这种端口的目的是为了提供设备快速的热插拔和即插即用的能力,其设备包括扫描仪、数码相机、打印机、键盘、鼠标、调制解调器、游戏杆、显示器、音箱乃至 UPS 联网等。现今 PC 机上的高质量串口最大速率可达 115.2 kb/s,而 USB 的传输速率则高达 12 Mb/s。USB 可以消除 Wintel PC 对有限的串口和并口连接的争用。理论上一台 PC 运用 USB 集线器可以支持 127 台外部设备。此外,USB 设备不需要使用诸如网卡之类的接口,从而可以节省宝贵的扩展槽,并且只要使用方式正确,还可以降低系统的功耗。

(3) FireWire

USB 适用于键盘和调制解调器之类的低速外设,FireWire 适用于磁盘和视频图像系统等高速设备。FireWire 串行总线标准最早是 Apple 公司在 Macintosh 上使用的,目前已经被 IEEE 组织采用,并定为 IEEE 1394 标准。FireWire 可以实现即插即用,具有更高的数据传输速率(高达 100 Mb/s~200 Mb/s,USB 为 12 Mb/s)。这样高的速率使它具备了足够的带宽支持 30 帧/s 的视频和音频双信号信道的传输。FireWire 不需要配备专用计算机来控制设备之间的连接。

它可以随意连接机顶盒、HDTV、打印机、DVD 等设备,可以使 DVD 播放机能在 PC 与 TV 之间切换,也可以打印从录像机上抓下来的彩色图片。

目前 FireWire 设备是比 USB 设备更出色的产品。FireWire 这项技术将主要用于高端领域,如专业性视频系统等。现在有若干家制造商销售 PCI FireWire 接口卡,但它们连接运行的大多数设备的价格相当于甚至略高于一台计算机,如 Sony 公司的数字视频设备。与 USB 一样,支持 FireWire 的系统软件目前还比较少。

即将推出的 400 Mb/s 的 FireWire 标准,其速度是 100Base-T 快速以太网的 4 倍。从长远来看,速度极高的 FireWire 线路将会在某些领域中取代以太网。FireWire 已经开始受到使用 Wintel 视频设备的广大用户的欢迎,但是要等到它与 PC 融合后才会真正赢得大批用户,同时 FireWire 设备的大量销售也将有助于价格的下落。

表 6.4 总结了 4 种常见的 I/O 总线(包括接口)的一些典型特征。表 6.5 总结了在服务器中可以找到的三种 CPU - 存储器总线的一些特征。通过比较,我们可以看出 I/O 标准总线和 CPU - 存储器总线之间的性能上的差距。

表 6.4 几种常用总线

	S总线	PCI	IPI	SCSI-2
数据宽度	32位	32或64位	16位	8~16位
时钟频率	16~25MHz	33MHz	异步	10MHz或异步
总线的主设备数	多个	多个	一个	多个
读32位的带宽	33MB/s	33MB/s	25MB/s	20MB/s或6MB/s
峰值带宽	89MB/s	111MB/s	25MB/s	20MB/s或6MB/s
标准	无	2.0	ANSI X3.129	ANSI X3.131

表 6.5 几种 CPU-存储器总线

	HP Summit	SGI Challenge	SUN XDBus
数据宽度	128位	256位	144位
时钟频率	60MHz	48MHz	66MHz
总线的主设备数	多个	多个	多个
峰值带宽	960MB/s	1200MB/s	1056MB/s
标准	无	无	无

6.3.5 设备的连接

设备的连接和工作方式分为直接传送、程序查询、中断、DMA、I/O 处理机等。总线是连接 CPU 和 I/O 设备的基本途径。我们首先看看 I/O 总线的物理连接。I/O 总线的连接一般有两种选择，连接到存储器上，或者连接到 Cache 上。图 6.14 是一种典型的系统和总线的组织。在一些低成本的系统中，往往 I/O 总线就是存储器总线，这种结构下总线上的 I/O 命令将影响 CPU 的访存。

确定了物理接口后，就要确定 CPU 对于 I/O 设备的寻址方式。最常用的方法之一是“存储器映射 I/O”，在这种方法中，将一部分存储器地址空间分配给 I/O 设备，对这些地址进行读写将引起 I/O 设备的数据传输。也可以将一部分 I/O 空间用作为 I/O 控制，这样，访问这一部分存储器地址空间，就是对设备发控制命令或者查询设备状态。

另一种常用的方法是在 CPU 中有专用的 I/O 指令。这种情况下，I/O 指令使 CPU 发出一个 I/O 操作标志信号，在 Intel 80x86 和 IBM 370 等计算机中就

使用这种方法。

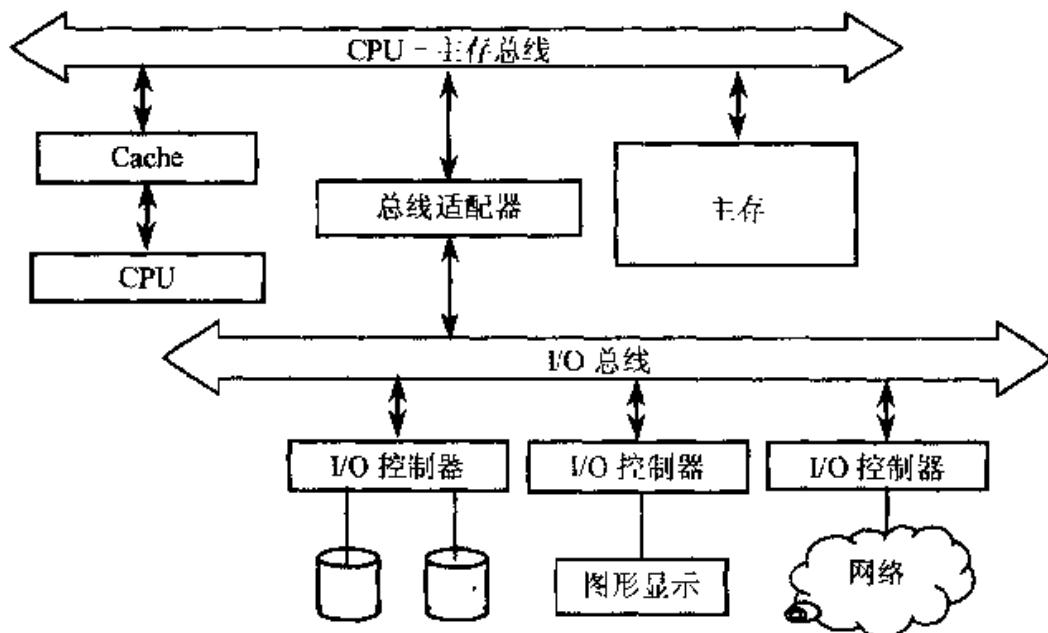


图 6.14 典型的总线连接

无论选择哪一种寻址方法,每个 I/O 设备都有提供状态和控制信息的寄存器,CPU 都得通过按规定写控制寄存器来完成对 I/O 设备的操作。

大多数 I/O 是多周期操作,例如 DEC LP11 行式打印机有两个设备寄存器:一个存放状态信息,一个存放要打印的数据。状态寄存器包含打印机在打印完一个字符后设置的“完成位”和一个表明打印机塞纸或缺纸的“出错位”,待打印数据的每个字节存放在数据寄存器中,CPU 一直要等到打印机设置“完成位”后,才能够向数据寄存器中放入另一个字符。

这种简单接口叫做“轮询”,CPU 需要不断监测状态位以确定是否该做下一个 I/O 操作。由于 CPU 比 I/O 设备快得多,所以轮询就要浪费大量的 CPU 时间,从而发明了“中断”方式。中断方式只有在 I/O 设备需要访问时再通过中断启动 CPU。

大部分系统中的设备采用“中断驱动 I/O”,这种方法允许 CPU 在等待 I/O 设备操作时运行其他进程。例如,LP11 有一种模式,允许完成位或出错位为“1”时向 CPU 发中断信号。作为常规应用,中断驱动 I/O 是实现多任务操作系统和提高响应速度的关键。

中断的缺点是操作系统处理中断事件的开销,在每秒具有上百个 I/O 事件的实时应用系统中,这种开销会影响应用的实时性。对实时系统的解决方法是利用时钟来定期中断 CPU,然后 CPU 查询所有的 I/O 设备,收集 I/O 事件。

6.3.6 CPU 与 I/O 处理的匹配

中断驱动 I/O 将 CPU 从等待 I/O 事件中解放出来,但是 CPU 仍然花费许多周期在数据传输上,例如传输有 2 048 个字的磁盘块将至少需要 2 048 次取和 2 048 次存,再加上中断所需要的开销。因为 I/O 事件通常涉及到块的传输,所以许多计算机系统中加入了“直接存储器访问(DMA)”硬件,允许在没有 CPU 干预的情况下传输多个字。

DMA 是一种当 CPU 执行任务时,在存储器和 I/O 设备之间传输数据的特殊处理器,它处在 CPU 外部,称为 DMA 控制器。CPU 开始只要设置包含存储器地址和需要传输的字节数的 DMA 寄存器即可。一旦 DMA 传输完成,DMA 控制器向 CPU 发中断。在一个计算机系统中可能有多个 DMA 设备,DMA 控制器是 I/O 设备控制器的一部分。

为进一步减轻 CPU 的负担,可以采用比 DMA 更强的 I/O 处理机(或称 I/O 控制器、通道控制器等)。I/O 处理机根据固定的程序或从操作系统装入程序来操作,操作系统通常设置一个包含数据源地址、目的地址和数据大小等信息的“I/O 控制块”队列,然后 I/O 处理器从队列中取出控制块,完成 I/O 处理,并且当完成由 I/O 控制块指定的任务后发出一个中断信号到 CPU。LP11 在打印一页(每行 80 个字符、每页 60 行)时需要 4 800 个中断,而使用一个 I/O 处理机只需要一次中断。

I/O 处理机和多处理机类似,它们使计算机系统中有多个程序可以同时执行。I/O 处理机是一种特殊装置,有特定的任务,所以并行性非常有限,而且 I/O 处理机通常情况下只是传输信息,或者对信息进行简单的加工,如格式转换等。

6.4 通道处理机

在大型计算机系统中,外围设备的台数一般比较多,设备的种类、工作方式和工作速度的差别也比较大。为了把对外围设备的管理工作从 CPU 中分离出来,从 IBM 360 系列机开始,普遍采用通道处理机技术。

6.4.1 通道的作用和功能

在大型计算机系统中,如果仅仅采用前面介绍过的程序控制、中断和 DMA 这三种基本的 I/O 方式来管理外围设备,会引起如下两个问题:

1. 所有外围设备的 I/O 工作全部都要由 CPU 来承担,CPU 的 I/O 负担很重,不能专心于用户程序的计算。

低速外围设备每传送一个字符都要由 CPU 执行一段程序来完成,而高速外围设备虽然使用 DMA 方式减少了 CPU 的干预,但初始化工作仍然需要 CPU 用程序来完成。在大型计算机系统中,这种 I/O 工作对 CPU 的时间占用实际上是一种浪费。

避免这种浪费的方法之一就是设置专门的 I/O 处理机来分担全部或大部分的 I/O 工作。例如,管理所有低速外围设备的 I/O 工作,对 DMA 接口的初始化工作,控制 DMA 的数据传送、数据格式的变换、设备状态的检测等。这样就能进一步提高整个计算机系统功能分散化的程度,充分发挥 CPU 的计算潜力。

2. 大型计算机系统中的外围设备台数虽然很多,但是一般并不同时工作。如果为每一台设备都配置一个接口,必然是一种浪费。特别是 DMA 接口,它的硬件代价很高。连接 DMA 接口的磁盘或磁带存储器等一般并不同时工作。

采用 DMA 方式传送数据,提高了输入输出数据的速度,节省了 CPU 的时间,但这是以为每一台快速外围设备都配备一个专用的 DMA 通道作为代价的。在微型和小型计算机系统中,由于快速外围设备的台数很少,所用 DMA 控制器的数量比较少。而在大型计算机系统中,快速外围设备的数量显著增加,就存在一个如何让 DMA 控制器能被多台设备共享的问题,以提高附加硬件的利用率。

为了使 CPU 摆脱繁重的 I/O 负担和共享 I/O 接口,在大型计算机系统中采用通道处理机是一种比较好的选择。

通道处理机能够负担外围设备的大部分 I/O 工作,包括管理所有按字节传输方式工作的低速和中速外围设备以及按数据块传输方式工作的高速外围设备,对 DMA 接口的初始化,设备故障的检测和处理等。通道处理机虽然不是一台具有完整指令系统的处理机,但是可以把它看作是一台能够执行有限 I/O 指令,并且能够被多台外围设备共享的小型 DMA 专用处理机。

在一台大型计算机系统中可以有多个通道,一个通道可以连接多个设备控制器,而一个设备控制器又可以管理一台或多台外围设备,这样就形成了一个非常典型的 I/O 系统的四级层次结构。

一般说来,通道的功能应该包括以下几个方面:

1. 接受 CPU 发来的 I/O 指令,根据指令要求选择一台指定的外围设备与通道相连接。
2. 执行 CPU 为通道组织的通道程序,从主存中取出通道指令,对通道指令进行译码,并根据需要向被选中的设备控制器发出各种操作命令。
3. 给出外围设备的有关地址,即进行读/写操作的数据所在的位置。如磁盘存储器的柱面号、磁头号、扇区号等。
4. 给出主存缓冲区的首地址,这个缓冲区用来暂时存放从外围设备上输入的数据,或者暂时存放将要输出到外围设备中去的数据。

5. 控制外围设备与主存缓冲区之间数据交换的个数,对交换的数据个数进行计数,并判断数据传送工作是否结束;

6. 指定传送工作结束时要进行的操作。例如,将外围设备的中断请求及通道的中断请求送往 CPU 等。

7. 检查外围设备的工作状态是正常还是故障。根据需要将设备的状态信息送往主存指定单元保存。

8. 在数据传输过程中完成必要的格式变换,例如,把字拆卸为字节,或者把字节装配成字等。

为此,通道应该能够执行一组通道指令,而且还要具有完成上述功能的硬件。通道的主要硬件包括寄存器部分和控制部分,寄存器部分有:数据缓冲寄存器、主存地址计数器、传输字节数计数器、通道命令字寄存器、通道状态字寄存器。控制部分有:分时控制、地址分配、数据传送、数据装配和拆卸等控制逻辑。

通道对外围设备的控制通过 I/O 接口和设备控制器进行,对于各种不同的外围设备,设备控制器的结构和功能也各不相同。然而,通道与设备控制器之间一般采用标准的 I/O 接口来连接。指令通过标准接口送到设备控制器,设备控制器解释并执行这些通道命令,完成命令指定的操作,并且将各种外围设备产生的不同信号转换成标准接口和通道能够识别的信号。另外,设备控制器还能够记录外围设备的状态,并把状态信息送往通道和 CPU。

6.4.2 通道的工作过程

在一般用户程序中,通过调用通道程序来完成一次数据输入输出的过程如图 6.15 所示,CPU 执行用户程序和管理程序,通道处理机执行通道程序的时间关系如图 6.16 所示。主要过程分为以下三步:

1. 在用户程序中使用访管指令进入管理程序,由 CPU 通过管理程序组织一个通道程序,并启动通道。

在多任务或多用户系统中,I/O 指令属于特权指令,一般用户程序不允许使用这些指令。如果在用户程序中要进行 I/O 操作,必须通过一条请求 I/O 的广义指令进入操作系统,通过调用操作系统的管理程序来使用外围设备。

广义指令由一条访管指令和若干个参数组成,如图 6.15 所示。访管指令的地址码部分实际上就是这条访管指令要调用的管理程序入口地址。当用户程序执行到要求进行 I/O 操作的访管指令时,产生自愿访管中断请求。CPU 响应这个中断请求后,转向管理程序入口。

管理程序根据广义指令提供的参数——如设备号、交换长度和主存起始地址等信息来编制通道程序。通道程序编制好后,放在主存储器中与这个通道相对应的通道程序缓冲区中。另外,在管理程序中还要把通道程序的入口地址置

入主存储器的通道地址单元。在管理程序的最后,用一条启动 I/O 设备指令来启动通道开始工作。

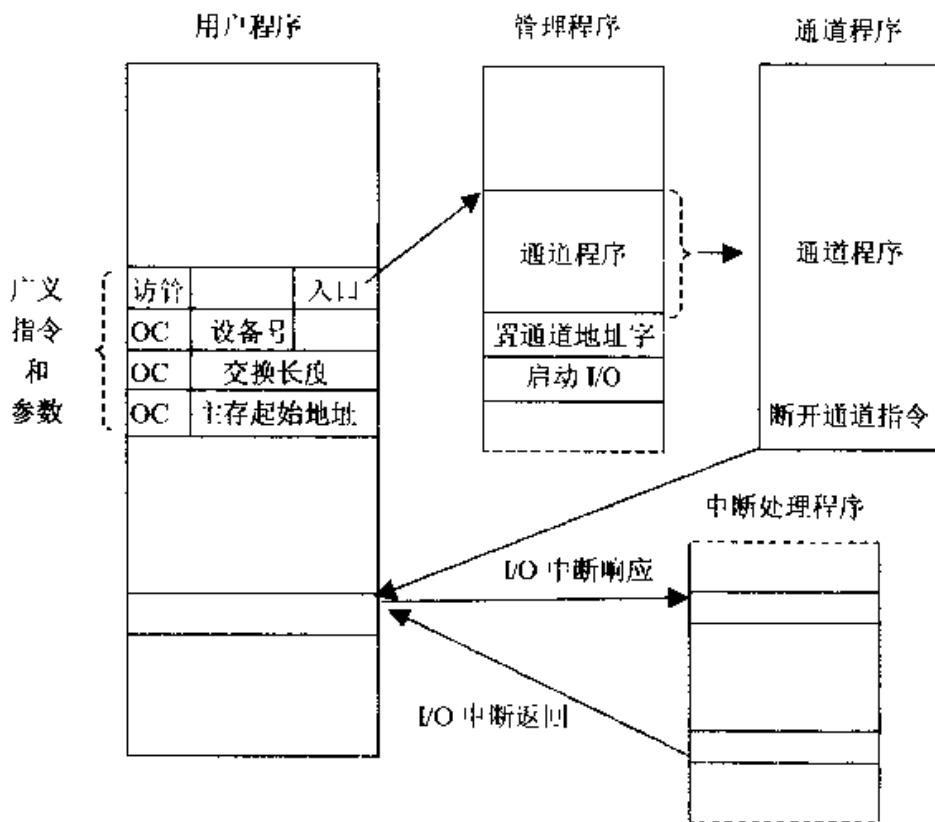


图 6.15 通道完成一次数据传输的主要过程

CPU 执行用户程序, 用户程序调用管理程序及通道处理机执行通道程序的时间关系如图 6.16 所示。

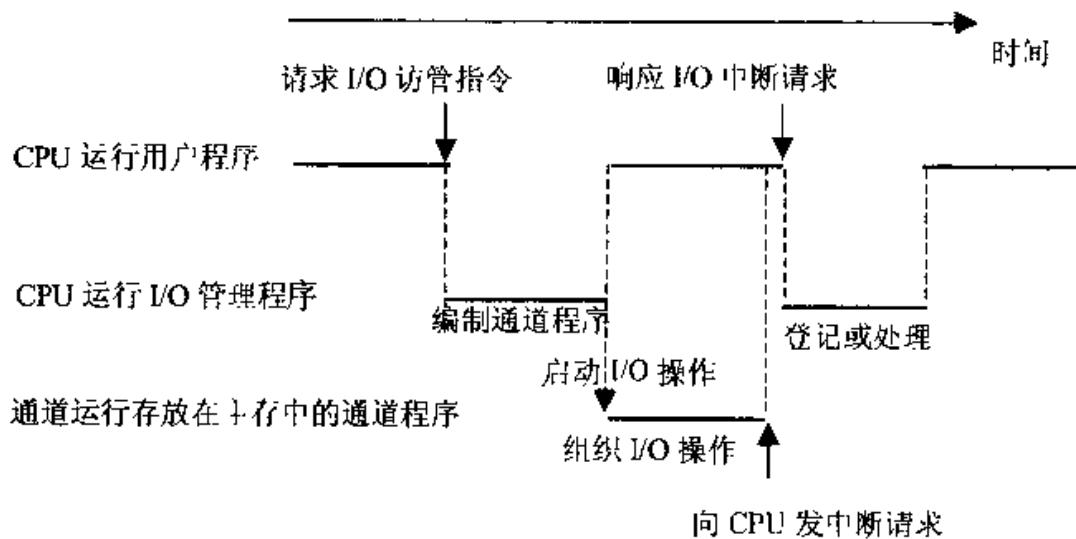


图 6.16 通道程序、管理程序和用户程序的执行时间关系

启动 I/O 设备指令是一条主要的 I/O 指令, 它属于特权指令, 在被访管指

令调用的系统管理程序的最后一条执行。启动 I/O 指令的工作流程为：先选择指定的通道和子通道，如果它们是在线而且是空闲的，就从主存中取出通道地址字按照通道地址字给出的通道程序起始地址从主存的通道缓冲区中取出第一条通道指令，通道指令经过校验，如果指令格式是正确的，再选择指定的设备控制器和设备。如果被选择的设备是在线的，就向它发启动命令。设备被启动后，将向通道发回答信号，如果设备的回答是一个全“0”字节，则表示这台设备已经接受并执行了启动命令，设备的启动过程也就全部完成了。在上述过程中，如果任何一个地方不正确，则表示启动 I/O 设备没有成功，形成相应的条件码并结束启动过程。通道通过检测这些条件码，能够知道设备为什么没有启动成功。

2. 通道处理机执行 CPU 为它组织的通道程序，完成指定的数据 I/O 工作。从图 6.16 中给出的时间关系可以看出，通道处理机执行通道程序是与 CPU 执行用户程序并行的。

通道被启动后，CPU 就可以退出操作系统的管理程序，返回到用户程序中继续执行原来的程序，而通道开始与设备之间的数据传送。当通道处理机执行完通道程序的最后一条通道指令“断开通道”时，通道的数据传输工作就全部结束了。

3. 通道程序结束后向 CPU 发中断请求。CPU 响应这个中断请求后，第一次进入操作系统，调用管理程序对 I/O 中断请求进行处理。

如果是正常结束，则管理程序进行必要的登记等工作；如果是故障、错误等异常情况，则进行异常处理。然后，CPU 返回到用户程序继续执行。

这样，每完成一次 I/O 工作，CPU 只需要两次调用管理程序，大大减少了对用户程序的打扰。当系统中有多个通道同时工作时，CPU 与多种不同类型、不同工作速度的外围设备可以充分并行工作。

在通道与设备之间的数据传送过程中，如果在同一个通道中有多台设备同时工作则要反复重新选择设备，即找出当前要传送数据的是哪一台设备。对于低速设备，每传送完一字节就要重新选择设备，而对于高速设备，通常每传送完一个数据块后需要重新选择设备。当然，如果一个通道只管理一台高速设备，那么，完成一次数据传送过程只需要做一次设备选择工作。

6.4.3 通道种类

根据多台外围设备共享通道的不同情况，可将通道分为三种类型：字节多路通道、选择通道和数组多路通道，这三种类型的通道与 CPU、设备控制器和外围设备的连接关系如图 6.17 所示。

1. 字节多路通道

字节多路通道(byte multiplexor channel)是一种简单的共享通道，主要为多

台低速或中速的外围设备服务。字节多路通道采用分时方式工作,依靠它与CPU之间的高速数据通路分时为多台设备服务。

字节多路通道可以有不同的工作方式。如果连接在通道上的各个设备轮流占用一个很短的时间片(通常小于100 μs)传输一个字节,或者说,不同的设备在它所分得的时间片内与通道在逻辑上建立不同的传输连接,则称为字节交叉方式(byte-interleave mode)。如果允许一个设备一次占用通道比较长的时间传输一组数据,或者说,设备与通道的连接可以根据需要维持到一组数据全部传送完成,则称为成组方式(block mode)。两种工作方式之间的转换可以是自动进行的,它通过一个超时机制进行控制。如果在超时机制预置的时间内,数据仍没有传送完毕,则自动转入成组方式工作,否则,继续采用字节交叉方式工作。

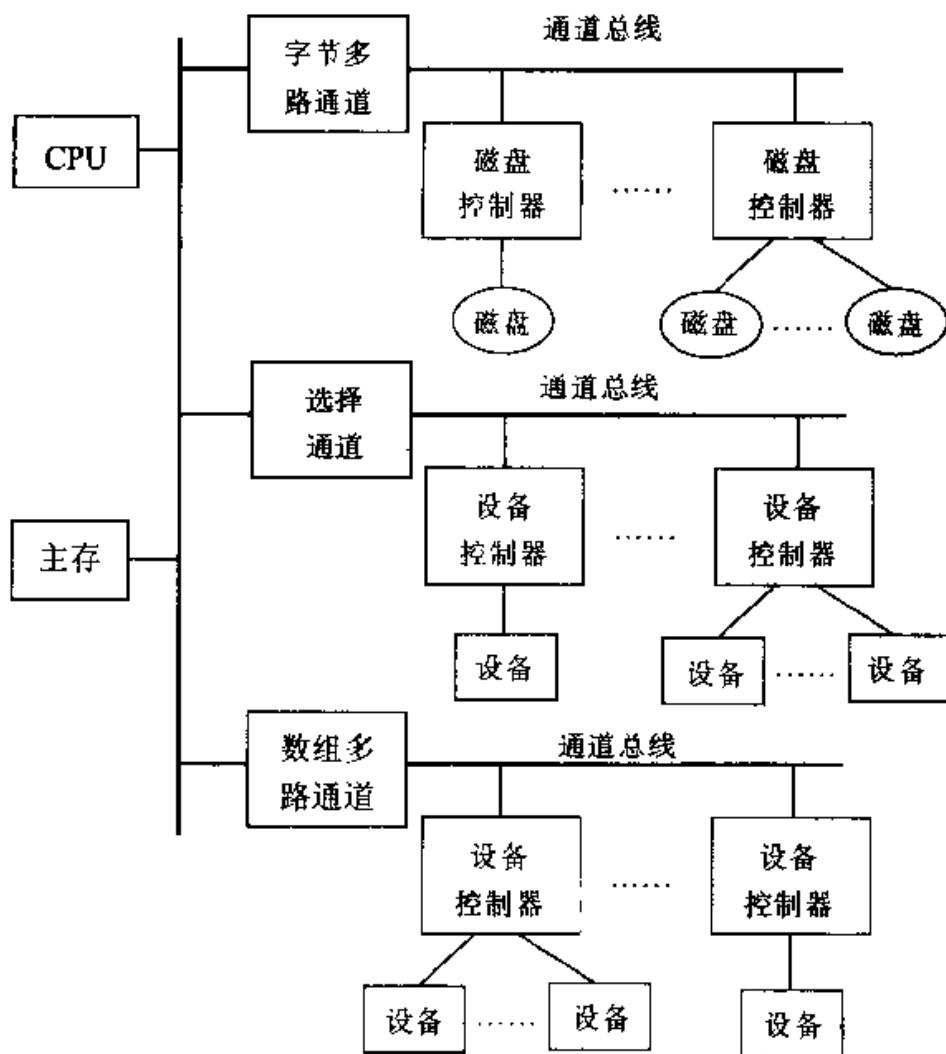


图 6.17 三种类型的通道与 CPU、设备控制器和外围设备的连接关系

字节多路通道包含有多个子通道,每个子通道连接一个设备控制器,最少需要有一个字节缓冲寄存器,一个状态/控制寄存器以及指明固定地址的少量硬件。与各个子通道有关的参数,如主存数据缓冲区地址、交换字节个数等都存放

在主存固定单元中。当通道在逻辑上与某台设备连接时,就从主存相应的单元中把有关参数取出来,根据主存数据缓冲区地址访问主存储器,读出或写入一个字节,并将交换字节个数减1,将主存数据缓冲区地址增至下一个数据的地址。在这些工作都完成之后,就将通道与该设备在逻辑上断开。

2. 数组多路通道

数组多路通道适于为高速设备服务。这些设备传输速率很高,但寻址等辅助操作时间很长。为了充分利用通道,尽量使各台高速设备重叠操作。它每次选择一个高速设备后传送一个数据块(对于磁盘和磁带等磁表面存储器,数据块大小通常为512个字节),并轮流为多台外围设备服务。

数组多路通道之所以能够并行地为多个高速外围设备服务,是因为这些高速外围设备并不能在整个数据输入输出时间内单独利用通道的全部传输能力。例如,从磁盘存储器读出一个文件的过程,可以分为以下三步:

第一步是定位。把读写磁头移动到记录该文件的磁道上,这要依靠机械动作来完成,称为定位时间或寻道时间,一般需要几十毫秒。

第二步是找扇区。等待读写磁头转动到记录该文件的起始扇区位置,称为等待时间。等待时间的长短主要与两个因素有关,一是磁盘的转速,二是磁头定位到所需要的磁道时,磁头所处位置与记录该文件起始扇区位置的相对距离。因此,等待时间的长短是随机的,最长为磁盘转一周所需的时间,最短为零。取平均值,通常称为平均等待时间。因此,磁盘存储器的平均等待时间一般小于10 ms。

第三步是读出数据。目前,高速磁盘存储器的数据传输率已经达到33 MB/s以上,因此,读出一个扇区(512个字节)只需要十几微秒时间。

通常把前两部分时间加起来称为磁盘存储器的寻址时间,第三部分时间称为数据传输时间。从上面的分析可以看出,磁盘存储器的寻址时间一般要比数据传输时间长两个数量级以上。因此,像选择通道那样,一个高速通道始终只为一台高速外围设备服务存在很大的浪费,并没有能够充分发挥高速通道的数据传输潜力,数组多路通道正是为了解决这一问题而提出来的。

数组多路通道在向一台高速设备发出定位命令后就立即从逻辑上与该设备断开,直到定位完成时再进行连接,发出找扇区命令后再一次断开,直到开始数据传送。因此,数组多路通道的实际工作方式是:通道在为一台高速设备传送数据时,有多台高速设备可以在定位或者在找扇区。

3. 选择通道

选择通道也是为多高速外围设备(如磁盘存储器等)服务的。在传送数据期间,该通道只能为一台高速外围设备服务,但在不同的时间内可以选择不同的设备。

一旦选中某一设备，通道就进入“忙”状态，直到该设备的数据传输工作全部结束为止，这就是选择通道(selector channel)。选择通道可以认为是只有一个以成组方式工作的子通道，只有一套完整的硬件，它逐个为物理上连接的几台高速外围设备服务。

因为外围设备与通道控制器之间通常是以字节为单位传送数据的，而通道与主存储器之间要以字为单位传送数据，一个字的长度一般为 32 位或 64 位，数据格式变换部件完成字到字节的拆卸及字节到字的装配。

6.4.4 通道中的数据传送过程

一个字节多路通道是分时为多台低速和中速外围设备服务的，在有 P 台设备同时连接到一个字节多路通道上时，它的数据传送过程如图 6.18 所示。

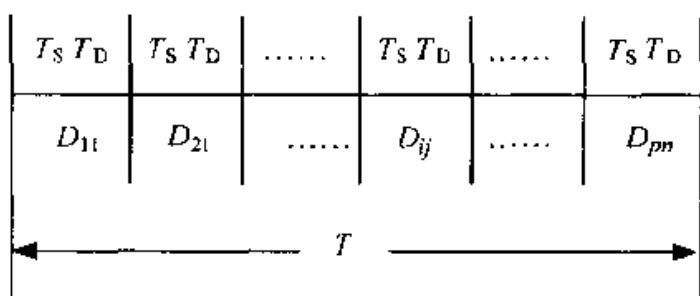


图 6.18 字节多路通道的数据传送过程

在图 6.18 中，每一个参数的含义如下，这些参数同样也适用于图 6.19 所示的数组多路通道和图 6.20 所示的选择通道。

T_S : 设备选择时间。从通道响应设备发出数据传送请求开始，到通道实际为这台设备传送数据所需要的时间。

T_D : 传送一个字节所用的时间，实际上就是通道执行一条通道指令“数据传送”所用的时间。

P : 在一个通道上连接的设备台数，且这些设备同时都在工作。

n : 每一个设备传送的字节个数，这里，假设每一台设备传送的字节数都相同，都是 n 个字节。

D_{ij} : 连接在通道上的第 i 台设备传送的第 j 个数据，其中： $i = 1, 2, \dots, p$ ； $j = 1, 2, \dots, n$ 。

T : 通道完成全部数据传送工作所需要的时间。

在字节多路通道中，通道每连接一个外围设备，只传送一个字节，然后又与另一台设备连接，并传送一个字节。因此，在图 6.19 中，设备寻址时间 T_S 和数据传送时间 T_D 是间隔进行的。

当一个字节多路通道上连接有 P 台外围设备，每一台外围设备都传送 n 个

字节时,总共所需要的时间 T 计算如下:

$$T_{\text{BYTE}} = (T_S + T_D)pn \quad (6.1)$$

数组多路通道在一段时间内只能为一台高速设备传送数据,但同时可以有多台高速设备在寻址,包括定位和找扇区。

数组多路通道的数据传送过程如图 6.19 所示,图中所用的参数与上面的字节多路通道相同,另外还有如下几个参数:

T_{D_i} :通道传送第 i 个数据所用的时间,其中: $i = 1, 2, \dots, n$ 。

D_i :通道正在为第 i 台设备服务,其中: $i = 1, 2, \dots, p$ 。

k :一个数据块中的字节个数。在一般情况下, $k < n$ 。对于磁盘、磁带等磁表面存储器, $k = 512$ 。

数组多路通道每连接一台高速设备,一般传送一个数据块,传送完成后,又与另一台高速设备连接,再传送一个数据块,因此,在图 6.19 中,在一个设备寻址时间 T_S 之后,有连续 k 个数据传送时间 T_D 。

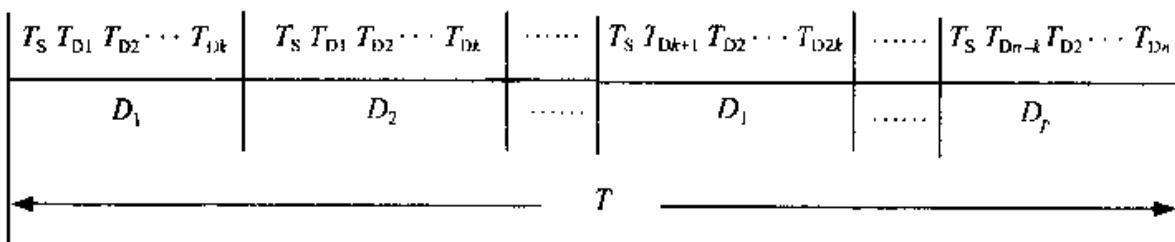


图 6.19 数组多路通道的数据传送过程

当一个通道连接有 P 台外围设备,每一台外围设备都传送 n 个字节时,总共所需要的时间 T 计算如下:

$$T_{\text{BLOCK}} = \left(\frac{T_S}{k} + T_D \right) pn \quad (6.2)$$

选择通道在一段时间内只能单独为一台高速外围设备服务,当这台设备的数据传送工作全部完成后,通道才能为另一台设备服务,因此,选择通道实际上是逐个为物理上连接的几台高速外围设备服务的。

选择通道的工作过程如图 6.20 所示,图中所用的参数与上面的字节多路通道和数组多路通道相同。

在选择通道中,通道每连接一个外围设备,就把这个设备的 n 个字节全部传送完成,然后再与另一台设备相连接,因此,在图 6.20 中,在一个设备寻址时间 T_S 之后,有连续 n 个数据传送时间 T_D 。

当一个选择通道上连接有 p 台外围设备,每一台外围设备都传送 n 个字节时,总共所需要的时间 T 计算如下:

$$T_{\text{SELECT}} = \left(\frac{T_S}{n} + T_D \right) pn \quad (6.3)$$

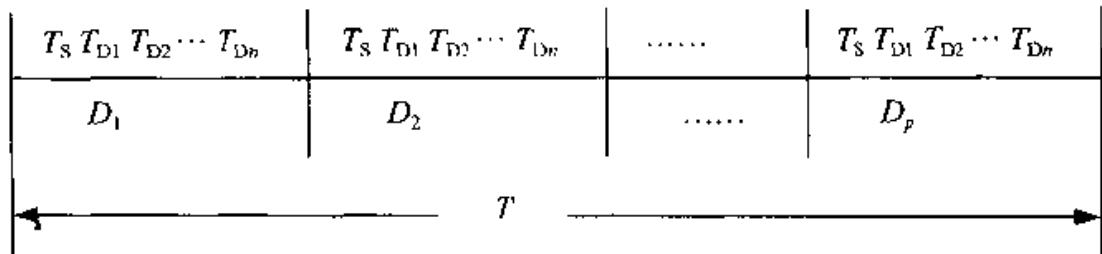


图 6.20 选择通道的数据传送过程

6.4.5 通道的流量分析

通道流量又称为通道吞吐率、通道数据传输率等,它是指一个通道在数据传送期间,单位时间内能够传送的最大数据量,一般用字节个数来表示。一个通道在满负荷工作状态下的流量称为通道最大流量。通道最大流量主要与通道的工作方式(指字节多路通道、选择通道和数组多路通道)、在数据传送期内通道选择一次设备所用的时间 T_S 和传送一个字节所用的时间 T_D 等因素有关。

根据通道流量的定义和上一节中的(6.1)、(6.2)和(6.3)式,可以得到三种通道的最大流量计算公式如下:

$$f_{\text{MAX-BYTE}} = \frac{pn}{(T_S + T_D)pn} = \frac{1}{T_S + T_D} \quad (6.4)$$

$$f_{\text{MAX-BLOCK}} = \frac{pn}{\left(\frac{T_S}{k} + T_D\right)pn} = \frac{1}{\frac{T_S}{k} + T_D} \quad (6.5)$$

$$f_{\text{MAX-SELECT}} = \frac{pn}{\left(\frac{T_S}{n} + T_D\right)pn} = \frac{1}{\frac{T_S}{n} + T_D} \quad (6.6)$$

根据字节多路通道的工作原理可知,它的实际流量是连接在这个通道上的所有设备的数据传输率之和,即

$$f_{\text{BYTE}} = \sum_{i=1}^p f_i$$

对于数组多路通道和选择通道,在一段时间内一个通道只能为一台设备传送数据,而且,这时的通道流量就等于这台设备的数据传输率。因此,这两种通道的实际流量就是连接在这个通道上的所有设备中数据流量最大的那个,即

$$f_{\text{BLOCK}} \leq \max_{i=1}^p f_i$$

$$f_{\text{SELECT}} \leq \max_{i=1}^p f_i$$

为了保证通道能够正常工作,即不丢失数据,各种通道实际流量应该不大于通道最大流量,应该满足下列不等式关系:

$$I_{\text{BYTE}} \leq I_{\text{MAX-BYTE}}$$

$$I_{\text{BLOCK}} \leq I_{\text{MAX-BLOCK}}$$

$$I_{\text{SELECT}} \leq I_{\text{MAX-SELECT}}$$

两边的差值越小,通道的利用率就越高。当两边相等时,通道处于满负荷工作状态。在实际设计最大通道流量时,应留有一定的余量。例如,对于字节多路通道,通道的最大流量应略大于所有连接在这个通道上的设备的流量之和。如果一个字节多路通道的最大流量正好等于连接在这个通道上的所有设备的流量之和,当所有设备的数据传送请求集中出现时,有可能要丢失数据。

6.5 I/O 与操作系统

采用什么硬件技术进行 I/O 处理是由操作系统来控制的。例如,早期的 UNIX 系统中使用许多 16 位微处理器作为 I/O 控制器,为了避免 16 位寻址空间被全部用完的问题,UNIX 限制每次 I/O 传输的最大块为 63 KB (15 位地址),无论文件有多大,I/O 控制器也无法一次传输大于 63 KB 的数据。在计算机系统中,I/O 系统硬件的功能如何使用,往往完全是由操作系统来决定的。

6.5.1 I/O 和 Cache 数据一致性

使用 Cache 增加了操作系统的负担。Cache 会使一个数据出现两个副本,一个在 Cache 中、一个在主存中;在虚拟存储器系统中,可能导致三个副本:Cache、主存和磁盘上各有一份,从而可能造成数据不一致的问题。操作系统和硬件必须确保在提供 Cache 和虚拟存储器时,CPU 读到的、I/O 输出的都是最新数据。

I/O 与计算机的连接方式是引起数据不一致问题的原因之一。如果它连接到 CPU 的 Cache 上,如图 6.21 所示,那就不会产生数据不一致的问题,所有 I/O 设备和 CPU 都能在 Cache 中看到最新的数据,而且存储器中的机制能保证数据的其他副本得到及时更新。这种连接的不足之一是损失了性能:I/O 要传输的数据往往不是 CPU 当前需要访问的数据,也就是说,所有 I/O 数据都必须经过 Cache,但是大部分都不会立即被 CPU 使用。这样,作为 CPU 常用数据的缓冲,Cache 的作用被大大减弱了。另外,在这种连接方式下,CPU 和 I/O 要竞争访问 Cache,因此需要仲裁。

如果 I/O 直接连到存储器上,由于 CPU 有 Cache,I/O 与存储器之间的数据交换就不会干扰 CPU。但这样就会产生数据不一致问题,这种数据不一致问

题有两个方面：

- (1) 存储器中可能不是 CPU 产生的最新数据, 所以 I/O 系统从存储器中取出来使用的是陈旧数据;

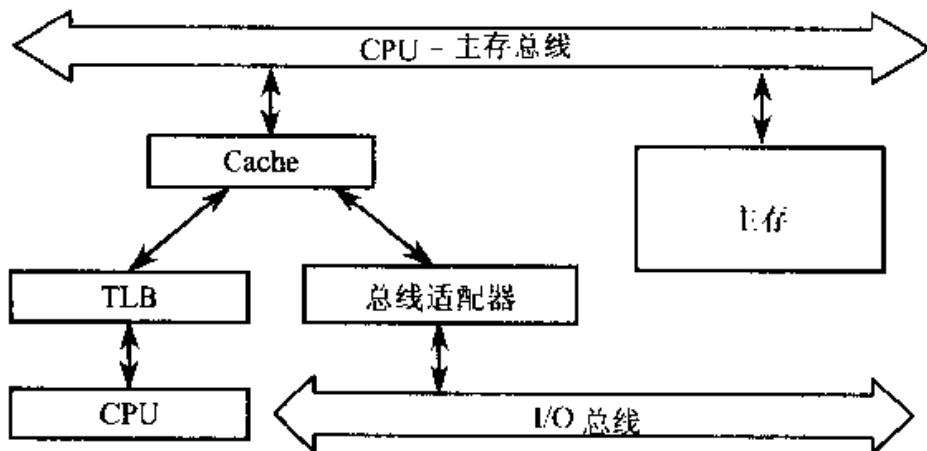


图 6.21 I/O 与 Cache 的直接连接

- (2) I/O 与存储器交换数据之后, 在 Cache 中被 CPU 使用的可能就是陈旧数据。

第一个问题是, 如果系统中存在 Cache, 而 I/O 连接到存储器上, 如何确保 I/O 使用到正确的数据? 写直达 Cache 可以保证存储器和 Cache 有相同的数据; 写回 Cache 则需要操作系统帮助进行数据检查: 根据 I/O 使用的存储器地址来清除 Cache 相应的块, 确保 I/O 使用的数据不在 Cache 中。但这个检查的动作要花很长时间, 因为必须将 I/O 使用的存储器地址与 Cache 中的标志(tag)逐个进行比较。这个地址检查过程也可以使用硬件完成, 这样, 当 I/O 使用的数据块在 Cache 中时, 就对 Cache 块进行写回。

第二个问题是确保 Cache 中的数据在 I/O 操作之后能够及时更新, 操作系统可以保证 I/O 操作的数据不在 Cache 中。如果不能确认这一点, 操作系统要根据 I/O 操作的存储器地址清除 Cache 中的相应块, 以确保这些数据不在 Cache 中。同样, 无论 I/O 操作的数据是否在 Cache 中, 都要花时间。这个过程也可以使用硬件完成, 通过对 I/O 使用的存储器地址和 Cache 的标志(tag)进行比较, 清除 Cache 中陈旧的数据。

这些问题实质上与在第七章中将要讨论的多处理机中的 Cache 一致性问题是相同的, I/O 设备可以看作是多处理机中的特殊处理机。

6.5.2 DMA 和虚拟存储器

如果使用虚拟存储器, 那么就存在 DMA 是使用虚拟地址还是物理地址来传输数据的问题。使用物理地址进行 DMA, 存在以下两个问题:

(1) 对于超过一页的数据缓冲区,由于缓冲区使用的页面在物理存储器中不一定是连续的,所以传输将会发生问题。

(2) 假设, DMA 正在存储器和帧缓冲器之间传输数据时, 操作系统从存储器中移出一些页面(或重新分配), DMA 将会在存储器中错误的页面上传输数据。

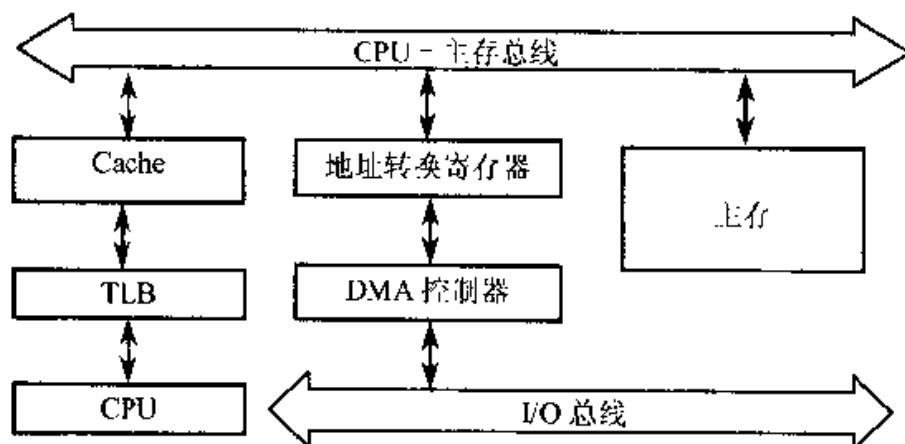


图 6.22 通过 DMA 连接的 I/O

解决这个问题的方法之一是采用“虚拟 DMA”技术, 它允许 DMA 设备直接使用虚存地址, 在 DMA 期间由硬件将虚拟地址映射到物理地址。I/O 使用的缓冲区在虚拟存储器中要求是连续的, 但物理页面可以分散在物理存储器中。如果使用虚拟 DMA 的进程在内存中被移动, 操作系统能够及时地修改相应的 DMA 地址表, 或者在 DMA 期间锁定存储器中相应的页面。图 6.22 中给出了用于 DMA 设备的地址转换寄存器。

6.6 I/O 系统设计

要设计一个好的 I/O 系统, 需要满足性能价格比的目标, 避免 I/O 中的瓶颈, 在存储器和 I/O 设备之间带宽要平衡, 因为在从 I/O 设备、存储器到 CPU 的信息通道上, 性能由最差的设备决定。

设计 I/O 系统要对系统中连接的各种 I/O 设备、各种 I/O 连接方式进行分析, 包括它们的性能、成本、容量等。下面是设计 I/O 系统的六个步骤, 每一步的结果, 通常是由市场需求或者性能价格比决定的。

- (1) 列出计算机需要连接的 I/O 设备的类型, 或者列出机器将要支持的标准总线。
- (2) 列出每种 I/O 设备的指标, 如容量、功耗、接口、总线槽、扩充箱等。
- (3) 列出每种 I/O 设备的价格, 包括这种设备所需要的专用控制器的价格。

(4) 记录每个 I/O 设备对系统资源的要求,包括:

- 初始化、设备操作(如处理中断)和完成 I/O 等需要的指令时钟周期数;
- 由于等待 I/O 完成、使用存储器、总线或 Cache 等,导致 CPU 空转的时钟周期数;
- 回到原来环境(如清除 Cache 等)所需的 CPU 时钟周期数。

(5) 列出每种 I/O 设备所需要的存储器和 I/O 总线资源,即使在 CPU 不使用存储器的情况下,主存与 I/O 总线之间的带宽也是受限的。

(6) 最后一步是要分析和评价这些 I/O 设备不同连接方法的性能。性能分析可以使用数学的排队论方法,也可以使用程序模拟方法。

通过以上分析,可以根据市场需求和性能、价格等指标,选择最好(优化)的 I/O 系统设计方案。

性能价格比对 I/O 的组成与实现有直接的影响。性能标准可以根据应用需要选择,如每秒兆字节(MB/s)的传输率、每秒 I/O 操作次数等。对于高性能 I/O 系统设计,受限因素可能是设备的速度、设备的数量、存储器的速度或 CPU 的速度等。对于低成本的设计,需要考虑的因素往往是 I/O 设备本身以及它们与 CPU 的通信方式。而面向性能价格比的设计就必须考虑两方面的因素。为了更清楚地说明这些思想,让我们来看几个例子。

例 6.3 首先看直接将磁盘页读入 Cache 对 CPU 带来的影响。假设:

- 每页为 16 KB,Cache 块大小为 64 字节;
- 对应新页的地址不在 Cache 中;
- CPU 不访问新页中的任何数据;
- Cache 中 95% 的被替换块将再次被读取,并引起一次失效;
- Cache 使用写回方法,平均 50% 的块被修改过;
- I/O 系统缓冲能够存储一个 Cache 完整的块(这称为速度匹配缓冲区,使存储器与 I/O 的速度得到匹配);
- 访问或失效在所有 Cache 块中均匀分布;
- 在 CPU 和 I/O 之间,没有其他访问 Cache 的干扰;
- 无 I/O 时,每 100 万个时钟周期中,有 15 000 次失效;
- 失效开销是 30 个时钟周期。如果替换块被修改过,则再加上 30 个周期用于写回主存。
- 假设计算机平均每 100 万周期处理 1 页。

分析 I/O 对于性能的影响有多大?

解 每个主存页有 $16 \text{ K} / 64 = 256$ 块。

因为是按块传输,所以 I/O 传输本身并不引起 Cache 失效。但是,它可能要替换 Cache 中的有效块。如果这些被替换块中有一半是被修改过的块,将需要

$(256/2) \times 30 = 3840$ 个时钟周期将这些被修改过的块写回主存。

这些被替换出去的块中,有 95% 的后续需要访问,从而又产生 $95\% \times 256 = 244$ 次 Cache 失效,将再次发生替换。由于这次被替换的 244 块中数据是从 I/O 直接写入 Cache 的,因此所有块都为被修改块,需要写回主存(因为 CPU 不会直接访问从 I/O 来的新页中的数据,所以它们不会再次立即从主存中调入 Cache),需要时间为 $244 \times (30 + 30) = 14640$ 个时钟周期。

没有 I/O 时,每一页平均使用 100 万个时钟周期,Cache 失效 15 000 次,其中 50% 被修改过,所需的处理时间为

$$(15000 \times 50\%) \times 30 + (15000 \times 50\%) \times (30 + 30) = 675000 \text{ (时钟周期)}$$

使用 I/O 造成的额外性能损失比例为

$$(3840 + 14640) \div (1000000 + 675000) = 18480 \div 1675000 = 0.0011 = 1.1\%$$

即大约产生 1.1% 的性能损失。

例 6.4 基于以下的性能和价格信息:

- 一个 500 MIPS 的 CPU,价格为 \$30 000;
- 一个宽度 16 字节、访问周期为 100 ns 的存储器;
- I/O 总线传输速度为 200 MB/s,可接 20 个 SCSI-2 总线及控制器;
- 每条 SCSI-2 总线传输率为 20 MB/s,最多连接 15 个磁盘;
- 每个 SCSI-2 控制器价格 \$1 500,每次磁盘 I/O,需要增加 1 ms 的控制器开销;
- 每次磁盘 I/O,操作系统要使用 10 000 条 CPU 指令;
- 对于 8 GB 的大磁盘或 2 GB 的小磁盘,价格均为 \$0.25 /MB;
- 磁盘转速为 7 200 r/min,平均寻道时间为 8 ms,传输速率为 6 MB/s;
- 要求外存容量为 200 GB;
- 平均每次 I/O 传输 16 KB。

假设每次磁盘访问都需要平均寻道时间和平均旋转时间的开销;所有设备的空间利用率为 100%;工作负载在所有磁盘上均匀分布。计算使用 2 GB 小磁盘或 8 GB 大磁盘时,每次 IOPS(一般用 IOPS 表示每秒 I/O 数)的价格。

解 由于性能受限于通路中性能最差的环节,所以需要估算每一种结构中各个 I/O 部件的最大性能,从而确定具有最佳性能的结构。

首先分别计算 CPU、存储器和 I/O 总线所能承受的最大 IOPS。

CPU 完成的 I/O 最大性能取决于 CPU 的速度和进行磁盘 I/O 所需的指令数:

$$\text{IOPS}_{\text{CPU}} = 500 \text{ MIPS} \div 10000 \text{ 指令} = 50000 \text{ IOPS}$$

即 CPU 每秒可执行 50 000 次 I/O;

存储系统的最大性能可由存储器周期、存储器宽度和 I/O 传输的块大小来

计算：

$$\text{IOPS}_{\text{MM}} = 16 \text{ 字节} \div 100 \text{ ns} \div 16 \text{ KB} = 10,000 \text{ IOPS}$$

即存储器允许每秒执行 10 000 次 I/O；

I/O 总线的最大性能受限于总线带宽和 I/O 传输的块大小：

$$\text{IOPS}_{\text{I/O}} = 200 \text{ MB/s} \div 16 \text{ KB} = 12,500 \text{ IOPS}$$

即 I/O 总线允许每秒执行 12 500 次 I/O；因此，无论选择哪个磁盘，受主存限制的最大性能都不会超过 10 000 个 IOPS。

现在分析 SCSI-2 控制器的性能。在 SCSI-2 总线上传输 16 KB 的时间是

$$T_{\text{SCSI-2}} = 16 \text{ KB} \div 20 \text{ MB/s} = 0.8 \text{ ms}$$

由于 SCSI-2 控制器要增加 1 ms 的开销，所以每次 I/O 需要 1.8 ms。因此，SCSI-2 的性能为

$$\text{IOPS}_{\text{SCSI-2}} = 1 / 1.8 \text{ ms} \approx 556 \text{ IOPS}$$

即 SCSI-2 允许每秒执行约 556 次 I/O。由于 I/O 一般要用到多个控制器（设备重叠），所以 556 IOPS 还不一定能限制整个系统的性能指标。

最后一部分通路是磁盘本身。平均磁盘 I/O 的时间是

$$\begin{aligned} T_{\text{DISK}} &= 8 \text{ ms} + 0.5 \div 7,200 \text{ r/min} + 16 \text{ KB} \div 6 \text{ MB/s} \approx 8 \text{ ms} + 4.2 \text{ ms} + 2.7 \text{ ms} \\ &= 14.9 \text{ ms} \end{aligned}$$

所以磁盘的最大性能是

$$\text{IOPS}_{\text{DISK}} = 1 / 14.9 \text{ ms} \approx 67 \text{ IOPS}$$

磁盘个数取决于盘的容量。200 GB 要求 25 个 8 GB 盘，或 100 个 2 GB 盘。所有磁盘的最大性能为

$$\text{IOPS}_{\text{8GBDISK}} = 67 \text{ IOPS} \times 25 = 1,675 \text{ IOPS}$$

$$\text{IOPS}_{\text{2GBDISK}} = 67 \text{ IOPS} \times 100 = 6,700 \text{ IOPS}$$

因此，只要有足够的 SCSI-2 接口，每个磁盘可以使用一个，则 8 GB 盘磁盘的性能可以达到 1 675 IOPS，2 GB 磁盘的性能可以达到 6 700 IOPS。

虽然已经确定了 I/O 各种通路的性能，但我们仍然要确定需要几个 SCSI-2 接口和控制器，每个控制器需要连接多少磁盘，因为这将进一步影响 I/O 性能。I/O 总线受限于 20 个 SCSI-2 控制器，每个 SCSI-2 接口最多连接 15 个磁盘（标准限定）。对于 8 GB 的盘，最少需要 2 个 SCSI-2 控制器 ($\lceil 25 \div 15 \rceil = 2$)，对于 2 GB 的盘，最少需要 7 个 SCSI-2 控制器 ($\lceil 100 \div 15 \rceil = 7$)。

若使用 8 GB 盘，这样配置的最大 IOPS 为

$$2 \times 1,675 = 3,350 \text{ IOPS}$$

若使用 2 GB 盘，最大 IOPS 为

$$7 \times 6,700 = 46,900 \text{ IOPS}$$

连接 15 个磁盘时, SCSI-2 的最大性能比磁盘总 I/O 吞吐率低, 所以我们可以增加 SCSI-2 总线数量, 减少每条 SCSI-2 上连接的磁盘数量, 使 SCSI-2 不再成为 I/O 瓶颈。首先求每个 SCSI-2 接口支持的磁盘数:

$$\lfloor 556 \div 67 \rfloor = 8 \text{ 个}$$

再求所需要的 SCSI-2 控制器数。对于 8 GB 盘, SCSI-2 控制器数为

$$\lceil 25 \div 8 \rceil = 4 \text{ 个}$$

对于 2 GB 盘, SCSI-2 控制器数为

$$\lceil 100 \div 8 \rceil = 13 \text{ 个}$$

这求出了四种结构的性能: 25 个 8 GB 盘用 2 个或 4 个 SCSI-2 控制器; 100 个 2 GB 盘用 7 个或 13 个 SCSI-2 控制器。由于最大性能受限于瓶颈, 所以公式为

$$\text{Min}(\text{CPU 性能, 主存性能, I/O 总线性能, 磁盘性能, 控制器性能})$$

对于 8 GB 盘, 2 个 SCSI-2 控制器(控制器为瓶颈), 最大性能为

$$\text{Min}(50\,000, 10\,000, 12\,500, 1\,675, 1\,112) = 1\,112 \text{ IOPS}$$

对于 8 GB 盘, 4 个 SCSI-2 控制器(磁盘为瓶颈), 最大性能为

$$\text{Min}(50\,000, 10\,000, 12\,500, 1\,675, 2\,224) = 1\,675 \text{ IOPS}$$

对于 2 GB 盘, 7 个 SCSI-2 控制器(控制器为瓶颈), 最大性能为

$$\text{Min}(50\,000, 10\,000, 12\,500, 6\,700, 3\,892) = 3\,892 \text{ IOPS}$$

对于 2 GB 盘, 13 个 SCSI-2 控制器(磁盘为瓶颈), 最大性能为

$$\text{Min}(50\,000, 10\,000, 12\,500, 6\,700, 7\,228) = 6\,700 \text{ IOPS}$$

四种结构的价格计算如下。对于 8 GB 盘, 2 个 SCSI-2 控制器, 有

$$\$30\,000 + \$1\,500 \times 2 + 25 \times (8 \text{ GB} \times \$0.25/\text{KB}) = \$84\,200$$

对于 8 GB 盘, 4 个 SCSI-2 控制器, 有

$$\$30\,000 + \$1\,500 \times 4 + 25 \times (8 \text{ GB} \times \$0.25/\text{KB}) = \$87\,200$$

对于 2 GB 盘, 7 个 SCSI-2 控制器, 有

$$\$30\,000 + \$1\,500 \times 7 + 100 \times (2 \text{ GB} \times \$0.25/\text{KB}) = \$91\,700$$

对于 2 GB 盘, 13 个 SCSI-2 控制器, 有

$$\$30\,000 + \$1\,500 \times 13 + 100 \times (2 \text{ GB} \times \$0.25/\text{KB}) = \$100\,700$$

最后, 对于四种结构, 每 IOPS 的价格分别是 \$76、\$52、\$24 和 \$15。考虑到每秒最大平均 I/O 次数, 关键资源的利用率为 100%, 最佳性能价格比是小容量盘和多个控制器的组合。小盘配置的性能价格比是大盘配置的 3.5 倍。小盘配置的缺点是, 由于磁盘数量大, 系统的可用性较低, 当然可以考虑采用 RAID 的方法。

6.7 小结

按照 Amdahl 定律,不重视 I/O 将导致系统资源、特别是 CPU 资源的浪费,这要引起计算机系统设计人员、特别是系统集成人员的注意。

外存储器由各种大容量存储设备构成,如磁盘存储器、光盘存储器等。大容量存储设备按照存取方式又可分成两大类,即随机存取设备和顺序存取设备。磁盘存储器和光盘存储器属于随机存取设备,它们采用地址指示保存的数据,如磁盘的磁头号、磁道号、扇区号等。磁带存储器属于顺序存取设备,它的数据保存在一个个块里,要找数据必须从头一个块一个块地读出查找。磁带存储存取花费时间多,但容量大,单位容量的成本很低,因此常作为备份存储设备,用以脱机保存数据。

磁盘性能每年提高 4%~6%,而 CPU 性能的提高速度快得多,约每 18 个月翻一番。这种性能间的差距使人们不断发明新的结构和设备来填充这个地带,如使用文件 Cache 改进延迟,使用 RAID 盘阵列改进吞吐率等。将来对于 I/O 的需要将会包括更好的算法、更好的结构和更大的 Cache。然而,随着磁盘、磁带的容量和每兆位价格的不断改进,磁盘和磁带(库)仍将继续长期占据着 I/O、特别是外存储产品的广大市场,其应用范围也进一步拓展。

光盘存储产品以其极高的存储密度、有限的可写性、介质可换、可靠性高、抗损伤、抗灰尘能力强等优势,广泛用于图形、图像及数据归档管理等领域。随着技术的不断进步,其应用领域迅速拓宽,目前正融入电视、音响、图像存储和数据处理等实用系统中。其巨大的存储容量和快速检索能力将使其成为快速发展的 HDTV、多媒体技术和大型金融、商贸、通信等管理网络所不可缺少的组成部分。

总线作为各子系统之间共享的通信链路,具有低成本和多样性的优点。通过定义统一的互连方法,各种新型设备可以很容易连接起来;通过多设备共享一组标准的线路,可以降低开销。

在大型计算机系统中,外围设备的台数一般比较多,设备的种类、工作方式和工作速度的差别也比较大。为了把对外围设备的管理工作从 CPU 中分离出来,普遍采用通道处理机技术。目前,在大型主机系统中仍都采用通道处理机技术。

要设计一个好的 I/O 系统,要对系统中连接的各种 I/O 设备、各种 I/O 连接方式进行分析,包括性能、成本、容量等;要避免 I/O 中的瓶颈,在存储器和 I/O 设备之间带宽要平衡。同时,要通过操作系统处理好 I/O、Cache 和主存数据的一致性问题。

最后需要强调一点,I/O 性能的好坏直接影响到系统的整体性能,I/O 系统在计算机系统中非常重要,特别是在实际应用当中更为突出。绝不应当由于可

将 I/O 设备称为“外围设备”,就忽视 I/O 系统的设计

习 题 六

- 6.1** 假设一台计算机的 I/O 处理占 10%, 当其 CPU 性能改进到原来的 100 倍, 而 I/O 性能仅改进为原来 2 倍时, 系统总体性能会有什么改进?
- 6.2** 假设磁盘空闲, 这样没有排队延迟; 公布的平均寻道时间是 9 ms, 传输速度是 4 MB/s, 转速是 5 400 r/min, 控制器的开销是 1 ms。问读或写一个 512 字节的扇区的平均时间是多少?
- 6.3** 按表 6.2 中的数据, 问用哪和磁盘最适合于建立 100 GB 的镜像盘阵列(RAID1)存储子系统? 又问哪种磁盘最适合于建立 100 GB 的镜像盘阵列(RAID5)存储子系统?
- 6.4** 盘阵列有哪些分级? 各有什么特点?
- 6.5** 磁带库作为海量档案存储设备, 平均每个磁带的读者很少。假设一个系统中, 磁带读取速率为 9 MB/s, 更换一个磁带需要 30 s, 请计算 10 个读者全部读完 6 000 个磁带, 需要多长时间?
- 6.6** 同步总线和异步总线各有什么优缺点? 总线的主要参数有哪些? 各是什么含义?
- 6.7** 一台计算机由一个 CPU、一台 I/O 设备 D 以及通过一个字宽度的共享总线连接的主存储器 M 组成。CPU 每秒最多能执行 10^8 条指令, 平均每条指令要花 5 个指令周期; 假定 3 个周期要用存储总线, 存储器读出或存入操作各占用一个机器周期。另假定 CPU 在 95% 的运行时间内不执行任何 I/O 操作, 现在 I/O 设备 D 要把一个非常长的数据块送入主存。
 - (1) 如果采用程序控制 I/O, 并且每传送一个字需要执行两条指令, 试估计通过 D 的最大 I/O 数据传送速率 f_{max} 。
 - (2) 如果采用 DMA 传输, 也请估计一下 f_{max} 。
- 6.8** 一个字节多路通道, 由 I/O 设备提出请求来启动数据传输, 只有当通道还有足够多余的能力为提出的请求服务时, 它才接受请求。试设计一个便于通道实现的算法, 以决定是否能接受从一台 I/O 设备来的服务请求?
 - (1) 某字节多路通道连接 6 台外设, 其数据传输速率依次为 50 KB/s, 40 KB/s, 20 KB/s, 100 KB/s, 10 KB/s, 25 KB/s。
 - (2) 设计此通道的工作周期。
 - (3) 安排响应各外设请求的优先次序, 从 6 台设备同时发生请求开始, 画出此通道工作示意图。
 - (4) 写出各台设备第一次请求处理完的时刻。
- 6.9** 在有 Cache 的计算机系统中, 进行 I/O 操作时, 会产生哪些数据不一致问题? 如何克服?
- 6.10** 假设在一个计算机系统中:
 - (1) 每页为 32 KB, Cache 块大小为 128 字节;
 - (2) 对应新页的地址不在 Cache 中, CPU 不访问新页中的任何数据;
 - (3) Cache 中 95% 的被替换块将再次被读取, 并引起一次失效;

- (4) Cache 使用写回方法, 平均 60% 的块被修改过;
- (5) I/O 系统缓冲能够存储一个完整的 Cache 块 (这称为速度匹配缓冲区, 使存储器与 I/O 的速度得到匹配);
- (6) 访问或失效在所有 Cache 块中均匀分布;
- (7) 在 CPU 和 I/O 之间, 没有其他访问 Cache 的干扰;
- (8) 无 I/O 时, 每 100 万个时钟周期中, 有 18 000 次失效;
- (9) 失效开销是 40 个时钟周期。如果替换块被修改过, 则再加上 30 个周期用于写回主存。
- (10) 假设计算机平均每 200 万周期处理一页。

试分析 I/O 对于性能的影响有多大?

第七章 多处理机

7.1 引言

长期以来,人们一直在说单处理机的发展正在走向尽头。然而,在近十年中,随着微处理器的发展,单机性能增长达到了自晶体管计算机出现以来的最高速度。在另一方面,我们也确信并行计算机在未来将会发挥更大的作用。这个观点是基于以下三个事实:第一,要获得超过单处理器的性能,最直接的方法就是把多个处理器连在一起;第二,自1985年以来,体系结构的改进使性能迅速提高,这种改进的速度能否持续下去还不清楚,但通过复杂度和硅技术的提高而得到的性能的提高正在减小;第三,并行计算机应用软件已有缓慢但稳定的发展。

本章将重点放在多处理机设计的主流上,即中小规模的机器(处理器的个数小于100)。这种机器无论在现存数量上还是在价值总值上都在目前占主导地位。对于大规模处理机(处理器的个数大于100),我们只做简要介绍,因为这种机器的体系结构现在还很不确定,市场的前景也难以预测。在过去,高端科学计算领域主要由向量计算机占据,近期已成为由小规模的并行计算机(通常由4~16个处理器构成)占主导。

7.1.1 并行计算机体系结构的分类

按照 Flynn 分类法,根据计算机中指令和数据的并行状况可把计算机分成单指令流单数据流(SISD)、单指令流多数据流 SIMD、多指令流单数据流 MISD 和多指令流多数据流 MIMD 四类。许多早期的多处理机是 SIMD 机器,但在最近几年,MIMD 显然已成为通用多处理机体系结构的选择,这是由下列两个因素引起的:

1. MIMD 具有灵活性。通过适当的软硬件支持,MIMD 可以用作单用户机器,针对一个应用程序发挥出其高性能;也可以用作多道程序机器,同时运行许多任务;还可以是这两种功能的某种组合。

2. MIMD 可以充分利用商品化微处理器在性能价格比方面的优势。实际上,现有的多处理机几乎都采用与工作站和单处理机服务器相同的微处理器。

根据多处理机系统中处理器个数的多少,可把现有的 MIMD 机器可分为两

类。每一类代表了一种存储器的结构和互连策略。由于多处理机的规模大小这个概念的含义是随时间而变化的,所以我们用存储器结构来区分机器。第一类机器称为集中式共享存储器结构(Centralized Shared-Memory Architecture)。这类多处理机在目前至多有几十个处理器。由于处理器数目较小,可通过大容量的 Cache 和总线互连使各处理器共享一个单独的集中式存储器。因为只有一个单独的主存,而且从各处理器访问该存储器的时间是相同的,所以这类机器有时被称为 UMA(Uniform Memory Access)机器。这类集中式共享存储器结构是目前最流行的结构。图 7.1 为此类机器结构的示意图。

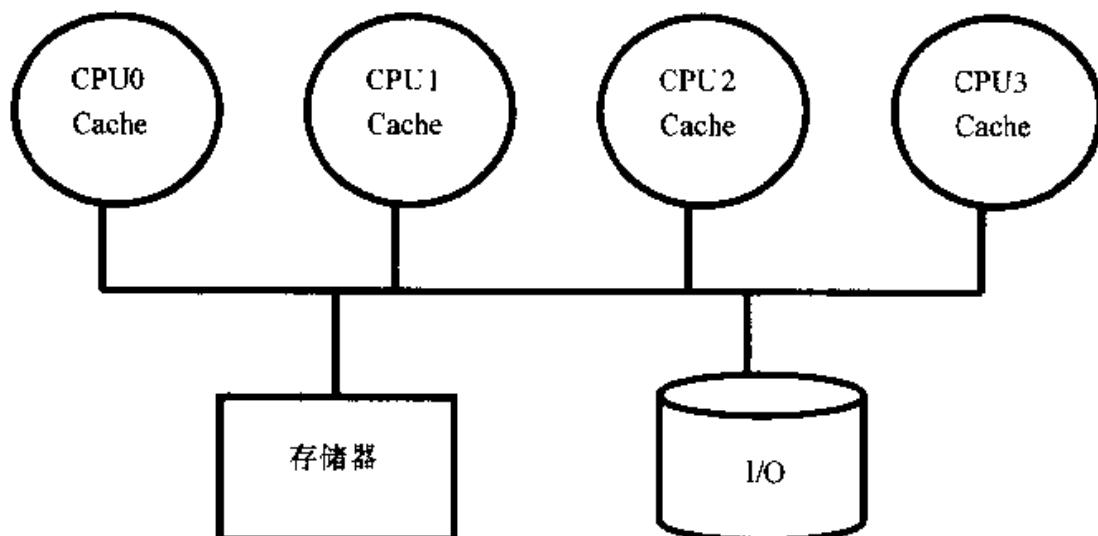


图 7.1 集中式共享存储器多处理机的基本结构

第二类机器具有分布的物理存储器。为支持较大多数的处理器,存储器必须分布到各个处理器上,而非采用集中式,否则存储器系统将不能满足处理器带宽的要求。系统中每个结点包含了处理器、存储器、I/O 以及互连网络接口。随着处理器性能的迅速提高和处理器对存储器带宽要求的不断增加,甚至在较小规模的多处理机系统中,采用分布式存储器结构也优于采用集中式共享存储器结构(这也是不按规模大小进行分类的一个原因)。当然,分布式存储器结构需要高带宽的互连。图 7.2 给出了此类机器的结构。

将存储器分布到各结点有两个好处:第一,如果大多数的访问是针对本结点的局部存储器,则可降低对存储器和互连网络的带宽要求;第二,对局部存储器的访问延迟低。分布式存储器体系结构最主要的缺点是处理器之间的通信较为复杂,且各处理器之间访问延迟较大。

通常情况下,I/O 和存储器一样也分布于多处理机的各结点当中。每个结点内还可能包含较小数目(2~8)的处理器,这些处理器之间可采用另一种技术(例如通过总线)互连形成簇(cluster),这样形成的结点叫做超结点。采用超结

点对机器的基本运行原理没有影响。由于采用分布式存储器结构的机器之间的主要差别在于通信方法和分布式存储器的逻辑结构方面,所以本章将集中讨论每个结点只有一个处理器的机器。

在 7.2 节中将对这两类机器展开详细讨论。

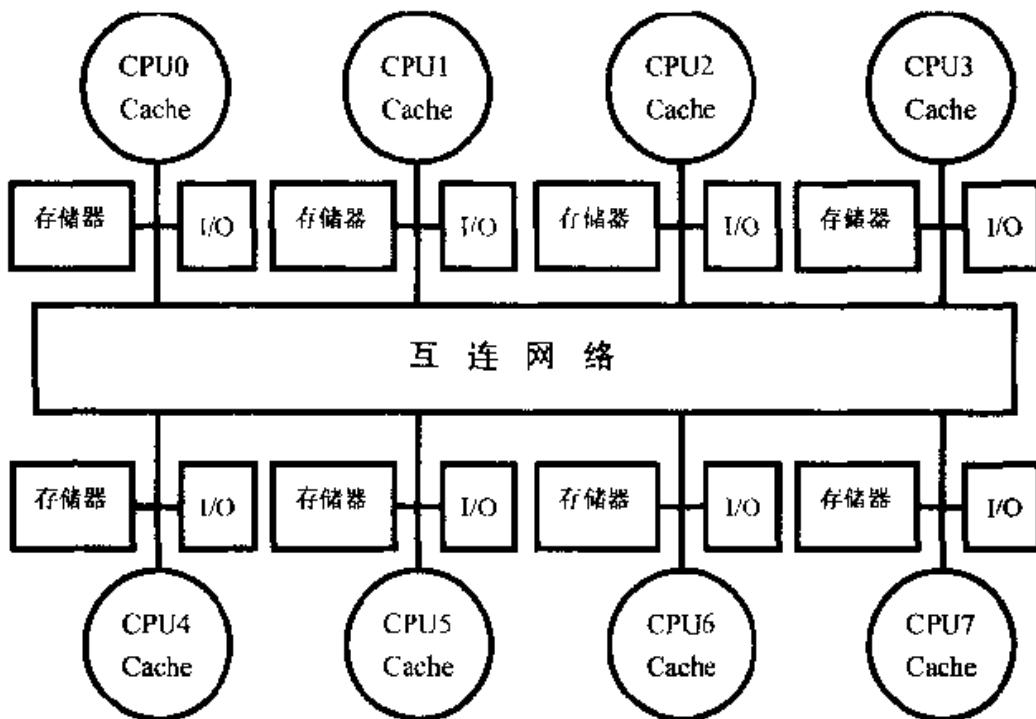


图 7.2 分布式存储器结构的机器基本结构

7.1.2 通信模型和存储器的结构模型

如上所述,大规模的多处理机必须采用多个存储器,它们在物理上分布于各个处理结点中。在各个处理器之间通信的实现上,目前有两种可供选择的方案。第一种方案是物理上分离的多个存储器可作为一个逻辑上共享的存储空间进行编址,这样一个处理器如果具有访问权,就可以访问任何一个其他的局部存储器,这类机器的结构被称为分布式共享存储器(DSM, Distributed Shared-Memory)或可缩放共享存储器(SSM, Scalable Shared-Memory)体系结构。此处共享指的是地址空间的共享,即两个处理器相同的物理地址指向分布存储器中的同一个单元。与集中式存储器机器相反,DSM 机器被称为 NUMA(Non-Uniform Memory Access)机器,这是因为其访问时间依赖于数据在存储器中的存放位置。

另一种方案是整个地址空间由多个独立的地址空间构成,它们在逻辑上也是独立的,远程的处理器不能对其直接寻址。在这种机器的不同处理器中,相同的物理地址指向不同存储器的不同单元,每一个处理器—存储器模块实际上是一个单独的计算机,因而这种机器也称为多计算机(multicomputers)。它完

完全可以由多个独立的计算机通过局域网相连构成。如果应用要求的通信较少或无需通信，那么采用这种方案只需将机器连结成簇来支持应用，这是一种很经济的途径。

对应于上述两种地址空间的组织方案，分别有相应的通信机制。对于共享地址空间的机器，可利用 Load 和 Store 指令中的地址隐含地进行数据通信，因而可称为共享存储器机器。对于多个地址空间的机器，数据通信要通过处理器间显式地传递消息完成，因而这种机器常称为消息传递机器。

消息传递机器根据简单的网络协议，通过传递消息来请求某些服务或传输数据，从而完成通信。例如，一个处理器要对远程存储器上的数据进行访问或操作，它就发送消息，请求传递数据或对数据进行操作，在这种情况下，消息可以看成是一个远程进程调用(RPC, Remote Process Call)。目的处理器接收到消息以后，执行相应的操作或代替远程处理器进行访问，并发送一个应答消息将结果返回。如果请求处理器在发送一个请求后一直要等到应答结果才继续运行，则这种消息传递称为同步的。现有许多机器的软件系统已为发送和接收消息的各种操作提供了支持，包括传送复杂的参数和返回结果。程序员可以利用 RPC 功能完成消息传递。

如果数据发送方知道别的处理器需要数据，通信也可以从数据发送方而不是数据接受方的角度开始，数据能够不先经请求就直接送往数据接受方。这种消息传递方法往往是异步进行的，使发送方可立即继续原来程序的运行。通常接收方所需的消息如果还未到达，则它需要进行封锁等待。此外，如果接收方还未处理完前一个消息，发送方也会因要发送消息而进行封锁等待。

7.1.3 通信机制的性能

在通信机制中有下面三个关键的性能指标：

1. 通信带宽——理想状态下的通信带宽受限于处理器、存储器和互连网络的带宽。进行通信时，结点内与通信相关的资源被占用，这种占用限制了通信速度。

2. 通信延迟——理想状态下通信延迟应尽可能地小。通信延迟的构成为

$$\text{通信延迟} = \text{发送开销} + \text{跨越时间} + \text{传输延迟} + \text{接收开销}$$

跨越时间是指一个数字信号从发送方的线路端传送到接收方的线路端所经过的时间，而传输延迟是全部的消息量除以线路带宽。发送和接收的开销一般都很大，数量上取决于通信机制及其实现。通信延迟是很关键的指标，它对多处理器机的性能和程序设计有很大的影响。通信机制的主要特点直接影响通信延迟和资源占用。例如，如果每次通信必须由操作系统处理，将会加大开销和通信资源的占用，进而引起带宽的降低和整个通信延迟的增加。

3. 通信延迟的隐藏——如何才能较好地将通信和计算或多次通信之间重叠起来,以实现通信延迟的隐藏,这是一个很复杂而且困难的问题。通信延迟隐藏是一种提高性能的有效途径,但它对操作系统和编程者来讲增加了额外的负担。通常的原则是:只要可能就隐藏延迟。

上面这些性能指标均受到通信机制和应用特点的影响。应用中的通信数据量是最主要的影响因素。通常情况下,一个好的通信机制应该无论对于通信数据量较小还是较大,通信模式规整还是不规整,以及不同类型的应用都表现出灵活性和高效性。当然,考虑任何一种通信机制时,设计者必须兼顾性能和开销两个方面。

7.1.4 不同通信机制的优点

每种通信机制各有优点,共享存储器通信主要有以下优点:

- (1) 与常用的集中式多处理机使用的通信机制兼容。
- (2) 当处理器通信方式复杂或程序执行动态变化时易于编程,同时在简化编译器设计方面也占有优势。
- (3) 当通信数据较小时,通信开销较低,带宽利用较好。
- (4) 通过硬件控制的 Cache 减少了远程通信的频度,减少了通信延迟以及对共享数据的访问冲突。

消息传递通信机制的主要优点包括:

- (1) 硬件较简单。
- (2) 通信是显式的,从而引起编程者和编译程序的注意,着重处理开销大的通信。

当然,可在支持上面任何一种通信机制的硬件模型上建立所需的通信模式平台。在共享存储器上支持消息传递相对简单,因为发送一条消息可通过将一部分地址空间的内容复制到另一部分地址空间来实现。但相反时,在消息传递的硬件上支持共享存储器就困难得多,所有对共享存储器的访问均要求操作系统提供地址转换和存储保护功能,即将存储器访问转换为消息的发送和接收。因而实现通信时软件的巨大开销严重地限制了可支持的应用程序范围。

最初的分布式存储器机器均采用消息传递机制,因为它显然较为简单。但近些年来,特别是 20 世纪 90 年代后半期以来设计的机器几乎都采用共享存储器通信。至于超大规模机器(MPP, Massively Parallel Processors, 一般超过 100 个处理器)将会支持哪种通信机制尚不太清楚,共享式存储器、消息传递以及综合的方法都有可能。

尽管现在通过总线连接的集中式存储器机器在市场上仍占主导地位,但从长远来看,在技术上的趋势是朝着中等规模的分布式共享存储器机器方向发展。

7.1.5 并行处理面临的挑战

并行处理面临着两个重要的挑战,它们均可通过 Amdahl 定律得以解释。第一个是程序中有限的并行性,第二个是相对较高的通信开销。有限的并行性使机器要达到好的加速比十分困难,如下例所示。

例 7.1 如果想用 100 个处理器达到 80 的加速比,求原计算程序中串行部分所占比例。

解 Amdahl 定律为

$$\text{加速比} = \frac{1}{\frac{\text{可加速部分比例}}{\text{理论加速比}} + (1 - \text{可加速部分比例})}$$

为了简单,假设程序只在两种模式下运行,即使用所有处理器的并行模式和只用一个处理器的串行模式。假设并行模式下的理论加速比即为处理器的个数,加速部分的比例即并行部分所占的比例。代入上式:

$$80 = \frac{1}{\frac{\text{并行比例}}{100} + (1 - \text{并行比例})}$$

得出:并行比例=0.9975。

可以看出要用 100 个处理器达到 80 的加速比,串行计算的部分只能占 0.25%。当然,要使加速比与处理器个数的增长成线性,整个程序必须全部并行。实际的程序并非在完全并行或串行模式下运行,而是常常在低于全部处理器个数的状态下运行。

面临的第二个挑战主要是指多处理机中远程访问的较大延迟。在现有的机器中,处理器之间的数据通信大约需要 50~10 000 个时钟周期,这主要取决于通信机制、互连网络的种类和机器的规模。表 7.1 表明了几种不同的并行机中远程访问一个字的延迟。其中的访问时间对于共享存储器机器来说,是指远程 Load 的时间;对于消息传递机器来说,则是指一次发送并回答的时间。

表 7.1 远程访问一个字的延迟时间

机 器	通信机制	互连网络	处理机数量	典型远程存储器访问时间
SPARC Center	共享存储器	总线	≤20	1 μs
SGI Challenge	共享存储器	总线	≤36	1 μs
Cray T3D	共享存储器	3 维环网	32~2 048	1 μs
Convex Exemplar	共享存储器	交叉开关+环	8~64	2 μs

续表

机 器	通信机制	互连网络	处理机数量	典型远程存储器访问时间
KSR-1	共享存储器	多层次环	32~256	2~6 μ s
CM-5	消息传递	胖树	32~1 024	10 μ s
Intel Paragon	消息传递	2维网格	32~2 048	10~30 μ s
IBM SP-2	消息传递	多级开关	2~512	30~100 μ s

以上数据充分说明了通信延迟的重要影响。下面再来看一个简单的例子

例 7.2 一台 32 个处理器的计算机, 对远程存储器访问时间为 2 000 ns。除了通信以外, 假设计算中的访问均命中局部存储器。当发出一个远程请求时, 本处理器挂起。处理器时钟时间为 10 ns, 如果指令基本的 CPI 为 1.0(设所有访存均命中 Cache), 求在没有远程访问的状态下与有 0.5% 的指令需要远程访问的状态下, 前者比后者快多少?

解 有 0.5% 远程访问的机器的实际 CPI 为

$$\begin{aligned} \text{CPI} &= \text{基本 CPI} + \text{远程访问率} \times \text{远程访问开销} \\ &= 1.0 + 0.5\% \times \text{远程访问开销} \end{aligned}$$

$$\text{远程访问开销} = \text{远程访问时间} / \text{时钟时间} = 2 000 \text{ ns} / 10 \text{ ns} = 200 \text{ 个时钟}$$

$$\text{因此, CPI} = 1.0 + 0.5\% \times 200 = 2.0$$

它为只有局部访问的机器的 $2.0 / 1.0 = 2$ 倍, 因此在没有远程访问的状态下的机器速度是有 0.5% 远程访问的机器速度的 2 倍。

实际中的性能分析会复杂得多, 因为一些非通信的访存操作可能不命中局部存储器, 而且远程访问开销也并非是一个常量。例如, 因为多个远程访问而引起的全局互连网络冲突会使延迟加大, 从而使远程访问性能更差一些。

并行程序和远程通信延迟是使用多处理机面临的两个最大的挑战。应用中并行性不足的问题主要通过采用并行性更好的算法来解决, 而远程访问延迟的降低主要靠体系结构支持和编程技术。例如可通过用硬件 Cache 保存共享数据, 或数据重组等增加局部访问来减少远程访问。也可通过预取等技术来减少延迟。

本章在以下两节中进一步讨论有关减少远程通信延迟时间的技术。其中 7.2 节讨论怎样采用 Cache 共享数据以减少远程访问频率; 7.4 节讨论同步问题, 因为同步本身包含了处理器间的通信, 是一个潜在的瓶颈。

在并行处理中, 负载平衡、同步和存储器访问延迟等影响性能的因素常依赖

于高层应用特点,如应用程序中数据的分配,并行算法的结构以及数据在空间和时间上的访问模式等,依据应用特点我们可把多机工作负载大致分成两类:单个程序在多处理机上的并行工作负载和多个程序在多处理机上的并行工作负载。

7.1.6 并行程序的计算/通信比率

反映并行程序性能的一个重要的度量是计算与通信的比率。如果比值较高,就意味着应用程序中相对于每次数据通信都要进行较多的计算。如前所述,通信在并行计算中是开销很大的,因而较高的计算/通信比率十分有益。在一个并行处理环境下,当要增加处理器的数目,或增大所求解问题的规模,或者两者同时都增大时,都要对计算/通信比率的变化加以分析。例如,在增加处理器数目同时知道这个比率的变化,会对应用能获得的加速比有清楚的了解。又如了解程序处理的数据集合大小的变化对这个比率的影响也是至关重要的。当处理器增多时,每个处理器计算量减小而通信量增大。当问题规模增大时,通信量的变化会更加复杂,这与算法的细节有关。

通常状况下,计算/通信比率随着处理数据规模的增大而增加,随着处理器数目的增加而降低。这说明用更多的处理器来求解一个固定大小的问题会导致不利因素的增加,因为处理器之间通信量加大了。同时也说明增加处理器时应该调整数据的规模,从而使通信的时间保持不变。

7.2 多处理机的存储器体系结构

7.2.1 集中式共享存储器体系结构

第五章已讨论过,多级 Cache 可以降低处理器对存储器带宽的要求。如果每个处理器对存储器带宽的要求都降低了,那么多个处理器就可以共享一个存储器。自 20 世纪 80 年代以来,随着微处理器逐渐成为主流,人们设计出了许多通过总线共享一个单独物理存储器的小规模多处理机。由于大容量 Cache 在很大程度上降低了对总线带宽的要求,当处理器规模较小时,这种机器十分经济。以往的这种机器一般是将 CPU 和 Cache 做在一块板上,然后插入底板总线。在目前的这种机器中,每块板上处理器数目高达 4 个,而 21 世纪初的某个时候,则可能会在一个单独的芯片上做出多个处理器,构成一个多处理机。图 7.1 是这种机器的一个简单的示意图。

这种体系结构支持对共享数据和私有数据的 Cache 缓存。私有数据供一个单独的处理器使用,而共享数据则供多个处理器使用。共享数据主要是用来供处理器之间通过读写它们进行通信。私有数据缓冲在 Cache 中降低了平均访存

时间和对存储器带宽的要求,使程序的行为类似于单机。共享数据可能会在多个 Cache 中被复制,这样做除了可降低访存时间和对存储器带宽的要求外,还可减少多个处理器同时读共享数据所产生的冲突。但共享数据进入 Cache 也产生了一个新的问题,即 Cache 的一致性问题。

1. 多处理器的一致性(Coherence)

在第五章已讨论过 Cache 的引进对 I/O 操作产生了一致性问题,因为 Cache 中的内容可能与由 I/O 子系统输入输出形成的存储器对应部分的内容不同。这个问题在多处理器上仍旧存在。此外对共享数据,不同处理器的 Cache 都保存有对应存储器单元的内容,因而在操作中就可能产生数据的不一致。表 7.2 通过两个处理器 Cache 对应同一存储器单元产生出不同值的例子说明了这个问题,这通常称为 Cache 一致性问题。

假设初始条件下各个 Cache 无 X 值,X 单元值为 1,并且假设是写直达方式的 Cache,X 单元被 A 写完后,A 的 Cache 和存储器中均为新值,但 B Cache 中不是,如果 B 处理器读 Cache,它将得到 1。

表 7.2 由两个处理器(A 和 B)读写引起的 Cache 一致性问题

时间	事件	CPU A Cache 内容	CPU B Cache 内容	X 单元存储器内容
0				1
1	CPU A 读 X	1		1
2	CPU B 读 X		1	1
3	CPU A 将 0 存入 X	0	1	0

可以非正式地定义:如果对某个数据项的任何读操作均可得到其最新写入的值,则认为这个存储系统是一致的。这个定义尽管很直观,但很模糊和简单,现实中的情况要复杂得多。这个简单的定义包括了存储系统行为的两个不同方面:第一个方面是指返回给读操作的是什么值(what);第二个方面是指什么时候才能将已写入的值返回给读操作(when)。

若一个存储器满足以下三点,则称该存储器是一致的:

(1) 处理器 P 对 X 进行一次写之后又对 X 进行读,读和写之间没有其他处理器对 X 进行写,则读的返回值总是写进的值。

(2) 一个处理器对 X 进行写之后,另一处理器对 X 进行读,读和写之间无其他写,则读 X 的返回值应为写进的值

(3) 对同一单元的写是顺序化的,即任意两个处理器对同一单元的两次写,从所有处理器看来顺序都应是相同的。例如,对同一地址先写 1,再写 2,任何处

理器均不会先读到 2, 然后又读到 1。

第一条属性保持了程序的顺序, 即使在单机中也要求如此。第二条属性给出了存储器一致性的概念。如果一个处理器不断读取旧的数据, 则可以肯定认为这个存储器是不一致的。

写操作的顺序化要求更严格。处理器 P1 对 X 单元进行一次写, 接着处理器 P2 对 X 也进行一次写, 如果不保证写操作顺序化, 就可能出现这种情况: 某个处理器先看到 P2 写的值而后看到 P1 写的值。解决这个问题最简单的方法就是写操作顺序化, 使同一地址所有写的顺序对任何处理器看来是相同的。这种属性称为写顺序化(write serialization)。

尽管上面三条已充分地保证了一致性, 但什么时候才能获得写进去的值仍是一个重要的问题。我们不可能要求在一个处理器对 X 写后立刻就能在另外的处理器上读出这个值, 这在实现上较为复杂。例如, 一个处理器对 X 进行写后, 很短时间内另一处理器对 X 进行读, 不可能确保读到的数据就是写入的值, 因为此时写入的值有可能在那一时刻还没离开进行写的处理器。为了简化讨论, 我们这里假设: 直到所有的处理器均看到了写的结果, 一次写操作才算完成; 处理器任何访存均不能影响写的顺序, 这就允许处理器无序读, 但必须以程序规定的顺序进行写。

2. 实现一致性的基本方案

多处理机和 I/O 的一致性问题尽管本质上相似, 但有着不同的特点, 从而解决的方法也不同。在 I/O 中, 多个数据拷贝是很少见的, 也是应尽力避免的。与之不同, 多处理机上的程序可能使几个 Cache 中保存相同的数据拷贝。在一致的多处理机中, Cache 提供了共享数据的迁移和复制(migration and replication)功能。共享数据的迁移是把远程的共享数据项拷贝放在本处理器局部的 Cache 中使用, 从而降低了对远程共享数据的访问延迟。共享数据的复制是把多个处理器需要同时读取的共享数据项的拷贝放在各自局部 Cache 中使用, 复制不仅降低了访存的延迟, 也减少了访问共享数据所产生的冲突。一般情况下小规模多处理机不是采用软件而是采用硬件技术实现 Cache 一致性。

对多个处理器维护一致性的协议称为 Cache 一致性协议(Cache-coherent protocol)。实现 Cache 一致性协议的关键是跟踪共享数据块的状态。目前有两类协议, 它们采用了不同的共享数据状态跟踪技术:

(1) 目录(directory)——物理存储器中共享数据块的状态及相关信息均被保存在一个称为目录的地方, 我们将在 7.2.2 节中详细论述。

(2) 监听(snooping)——每个 Cache 除了包含物理存储器中块的数据拷贝之外, 也保存着各个块的共享状态信息。Cache 通常连在共享存储器的总线上, 各个 Cache 控制器通过监听总线来判断它们是否有总线上请求的数据块。本节

中将着重讨论这种方法。

在使用多个处理器,每个 Cache 都与单个共享存储器相连组成的多处理器中,一般都采用监听协议,因为这种协议可利用已有的物理连接(总线到存储器)来修改 Cache 中的状态信息。

3. 两种协议

可以用两种方法来维持上面所讲的一致性要求。一种方法是在一个处理器写某个数据项之前保证它对该数据项有唯一的访问权,对应这种方法的协议称为写作废(write invalidate)协议。因为写入新的内容将产生共享数据拷贝的不一致,因此应该使别的拷贝失效。它是目前应用最普遍的协议。无论是在监听还是在目录的模式下,唯一的访问权保证了在进行写操作后不存在其他的可读或可写的拷贝:别的所有的拷贝全部作废了。可以通过一次写操作后另一处理器接着读的过程来看一下如何保持数据一致性:既然写要求唯一的访问权,那么其他处理器上的拷贝已作废,因此,当其他处理器再次进行读操作时,就会产生读失效,从而取出新的数据拷贝。要保证进行写的处理器具有唯一的访问权,必须禁止别的处理器和它同时写。如果两个处理器同时写数据,那么竞争胜者就会将另一个处理器的拷贝作废,当胜者完成写后另一个处理器要进行写时,它必须先取一个新的数据拷贝,即已被更新的数据。因此,这种协议保证了顺序写。表 7.3 给出了一个在监听方案和写回 Cache(write-back Cache)的条件下写作废协议模式的一个例子。

表 7.3 在写回 Cache 的条件下,监听总线中写作废协议的实现

处理器行为	总线行为	CPU A Cache 内容	CPU B Cache 内容	主存 X 单元 内容
				0
CPU A 读 X	Cache 失效	0		0
CPU B 读 X	Cache 失效	0	0	0
CPU A 将 X 单元写 1	作废 X 单元	1		0
CPU B 读 X	Cache 失效	1	1	1

另外一种协议是写更新协议(write update)。它是当一个处理器写某数据项时,通过广播使其他 Cache 中所有对应的该数据项拷贝进行更新。为减少协议所需的带宽,应知道 Cache 中该数据项是否为共享状态,也就是别的 Cache 中是否存在该数据项拷贝。如果不是共享的,则写时无需进行广播。表 7.4 表示的是一个写更新协议的操作过程。近十年来,这两种协议均得到了发展,但在目

前的应用中,写作废协议占了大多数。

表 7.4 在写回 Cache 的条件下,监听总线中写更新协议的实现

处理器行为	总线行为	CPU A Cache 内容	CPU B Cache 内容	主存 X 单元内容
				0
CPU A 读 X	Cache 失效	0		0
CPU B 读 X	Cache 失效	0	0	0
CPU A 将 X 单元写 1	广播写 X 单元	1	1	1
CPU B 读 X		1	1	1

写更新和写作废协议性能上的差别来自三个方面:

- (1) 对同一数据的多个写而中间无读操作的情况,写更新协议需进行多次写广播操作,而在写作废协议下只需一次作废操作。
- (2) 对同一块中多个字进行写,写更新协议对每个字的写均要进行一次广播,而在写作废协议下仅在对本块第一次写时进行作废操作即可。写作废是针对 Cache 块进行操作,而写更新则是针对字(或字节)进行操作。
- (3) 从一个处理器写到另一个处理器读之间的延迟通常在写更新模式中较低,因为它写数据时马上更新了相应的其他 Cache 中的内容(假设读的处理器 Cache 中有此数据)。而在写作废协议中,需要读一个新的拷贝。

在基于总线的多处理机中,总线和存储器带宽是最紧缺的资源,因此写作废协议成为绝大多数系统设计的选择。当然,在设计小数目处理器(2~4)的多处理机时,各个处理器紧密地相连,更新所要求的带宽还可以接受。尽管如此,考虑到存储器性能和相关带宽要求的增长,更新模式很少采用,因此在本章的剩余部分我们只关注写作废协议。

4. 监听协议的基本实现技术

小规模多处理机中实现写作废协议的关键是利用总线进行作废操作。当某个处理器进行写数据时,必须先获得总线的控制权,然后将要作废的数据块的地址放在总线上。其他处理器一直监听总线,它们检测该地址所对应的数据是否在它们的 Cache 中。若在,则作废相应的数据块。获取总线控制权的顺序性保证了写的顺序性,因为当两个处理器要同时写一个单元时,其中一个处理器必然先获得总线控制权,之后它使另一处理器上对应的拷贝作废,从而保证了写的严格顺序性。

当写 Cache 未命中时,除了将其他处理器上相应的 Cache 数据块作废外,还

要从存储器取出该数据块。对于写直达(write-through)Cache,因为所有写的数据同时被写回主存,则从主存中总可以取到最新的数据值。

对于写回 Cache,得到数据的最新值会困难一些,因为最新值可能在某个 Cache 中,也可能在主存中。在 Cache 失效时可使用相同的监听机制。每个处理器都监听放在总线上的地址,如果某个处理器发现它含有被请求数据块的一个已修改过的拷贝,就将这个数据块直接送给发出读请求的处理器,并停止其对主存的访问请求。因为写回 Cache 所需的存储器带宽较低,因此尽管在实现的复杂度上有所增加,但一般在多处理机实现上仍很受欢迎。下面将着重研究在写回 Cache 条件下的实现技术。

Cache 中块的标志位可用于实现监听过程。每个块的有效位(valid)使作废机制的实现较为容易。由写作废或其他事件引起的失效处理很简单,只需将该位设置为无效即可。而对于写,则希望知道是否别的 Cache 中也有此数据的共享拷贝。因为如果没有别的 Cache 拷贝,则无需将写地址放在写回 Cache 的总线上,这样可减少所用的时间并降低所需的带宽。

为了分辨某个数据块是否共享,需要给每个 Cache 块加一个特殊的状态位来说明它是否为共享。拥有唯一的 Cache 块拷贝的处理器通常称为这个 Cache 块的拥有者(owner),处理器的写操作使自己成为对应 Cache 块的拥有者。当对共享块进行写时,本 Cache 将写作废的请求放在总线上,Cache 块状态由共享(shared)变为非共享(unshared)或者专有(exclusive)。如果以后另一处理器要求访问该块,则状态会再转化为共享。

基于总线一致性协议的实现通常采用在每个结点内嵌入一个 Cache 状态控制器,该控制器根据来自处理器或总线的请求,改变所选择的数据块的状态。

为了简单起见,在协议实现时可不区分对一个共享数据块的写命中和写失效,在两种状态下,我们均作为写失效看待。当总线上出现写失效时,任何具有该 Cache 数据块的处理器结点进行作废处理。如果是写回 Cache 且该块是唯一的,则那个 Cache 要进行写回操作。将写命中作为写失效处理可减少总线事务的数目,简化控制器。

在上述协议中假设操作具有原子性(atomic),即操作进行过程中不能被打断,例如将写失效的检测、申请总线和接收响应作为一个单独的原子操作。但在实现中的情况会比这复杂得多。

因为每次总线任务均要检查 Cache 的地址位,这可能与 CPU 对 Cache 的访问冲突。可通过下列两种技术之一降低冲突:复制标志位或采用多级包含 Cache。

复制标志位可使 CPU 对 Cache 访问和监听并行进行。当然,Cache 失效时,处理器要对两套 Cache 标志位进行操作。类似地,如果监听到了一个相匹配的

地址,也需对两套 Cache 的标志位进行操作。处理器与监听同时操作标志位发生冲突时,非抢先者将被挂起或被延迟。

多级包含 Cache 中 Cache 由多级构成,例如通常由二级构成,靠近 CPU 的第一级 Cache 是较远的第二级 Cache 的一个子集。许多设计中都采用多级 Cache 来减少处理器的带宽要求。在第一级 Cache 中的每个数据块均包含在第二级 Cache 中,因而监听可针对第二级 Cache 进行,而处理器的大多数访问针对第一级 Cache。如果监听命中第二级 Cache,它就必须垄断对各级 Cache 的访问,更新状态并可能回写数据,这通常需要挂起处理器对 Cache 的访问。如果将第二级 Cache 中的标志位复制会更有效地减少 CPU 和监听之间的冲突。

在基于总线的采用写作废协议的多处理机中,几个不同的方面共同决定了其性能,尤其是 Cache 的整体性能依赖于单处理器 Cache 的失效开销和通信开销两方面。其中通信开销包括由其引起的作废和相应的 Cache 失效的开销。处理器数目、Cache 大小以及 Cache 块大小的改变以不同的方式影响着这两个方面,这些方面结合起来产生对整体性能的影响。

7.2.2 分布式共享存储器体系结构

支持共享存储器的可缩放机器既可以支持也可以不支持 Cache 一致性。从硬件简单角度出发,可以不支持 Cache 一致性,而着重于存储器系统的可缩放性上。已有几家公司生产了这种机器,Cray T3D 是一个很好的例子。在这类机器中,存储器分布于各结点中,所有的结点通过网络互连。访问可以是本地的,也可是远程的,由结点内的控制器根据地址决定数据是驻留在本地存储器还是驻留在远程存储器,如果是后者,则发送消息给远程存储器的控制器来访问数据。这些系统都有 Cache,为了解决一致性问题,规定共享数据不进入 Cache,仅有私有数据才能保存在 Cache 中。当然也可以通过软件显式地控制一致性。这种机制的优点是所需的硬件支持很少,因为远程访问存取量仅是一个字(或双字)而不是一个 Cache 块。

这种方法有几个主要的缺点:

(1) 实现透明的软件 Cache 一致性的编译机制能力有限。基于编译的软件一致性目前还未实现,最基本的困难是基于软件的一致性算法不能足够准确地预测出实际共享数据块,把可能共享的每个数据块均看作是共享数据块,这就导致了多余的一致性开销。

(2) 没有 Cache 一致性,机器就不能利用取出同一块中的多个字的开销接近于取一个字的开销这个优点,这是因为共享数据是以 Cache 块为单位进行管理的。当每次访问要从远程存储器取一个字时,不能有效利用共享数据的空间局部性。

(3) 诸如预取等延迟隐藏技术对于多个字的存取更为有效,比如针对一个 Cache 块的预取。

对远程存储器访问的巨大延迟与对本地 Cache 访问的短延迟相比,突出地反映出了这些缺点。例如,Cray T3D 本地访问延迟为两个时钟周期,并且可以被流水化,而一次远程访问则需约 150 个时钟周期。

将 Cache 一致性共享存储器模式拓展到大规模多处理机面临着新的挑战。尽管可以用缩放性更好的网络来取代总线,将存储器分布开来,从而使存储器的带宽具有缩放性,但是监听一致性协议机制本身的可缩放性较差。在监听协议中,每当 Cache 失效时,要与所有的 Cache 进行通信,包括对于可能共享的数据的写。没有一个集中的记录 Cache 状态的数据结构是基于监听机制的基本优点,因为它的代价较低。但当系统的规模变大时,它又是致命的弱点。此外,监听的访问量与处理器个数的平方(N^2)成正比,即使总线的带宽随系统规模线性增长(N),而实际的性能还是下降到 $1/N$ 。

在支持 Cache 一致性的可缩放的共享存储器体系结构中,关键是寻找替代监听协议的一致性协议。一种可供选择的协议是目录协议,目录是用一种专用的存储器所记录的数据结构,它记录着可以进入 Cache 的每个数据块的访问状态、该块在各个处理器的共享状态以及是否修改过等信息。目前目录协议的实现是对每个存储器块分配一个目录项,目录的信息量与存储器块的数量和处理器数量的乘积成比例。对于少于 100 个处理器的机器这不成问题,因为由目录操作引起的延迟开销可以忍受。而对于较大规模的机器,由于目录内容较多,就需要寻找更有效的目录结构。例如只保存少量的块(比如只保存 Cache 中的块而非全部的存储器块)的信息或者每个目录项保存较少的状态位等。

为防止目录成为瓶颈,目录一般与存储器一起分布在系统中,从而对于不同目录内容的访问可以在不同的结点进行,正如对不同存储器的访问一样。分布式目录中将数据块的共享状况保存在一个固定的单元中,从而使一致性协议避免了广播操作。图 7.3 表示的是对每个结点增加目录表后的分布式存储器的系统结构。

1. 基于目录的 Cache 一致性

和监听协议一样,目录协议也必须完成两个主要的操作:处理读失效和处理对共享、干净(clean)块的写(对一个共享块的写失效处理是这两个操作的简单结合)。为实现这两种操作,目录必须跟踪每个 Cache 块的状态。Cache 块状态有三种:

- 共享(shared)——在一个或多个处理器上具有这个块的拷贝,且主存中的值是最新值(所有 Cache 均相同),
- 未缓冲(uncached)——所有处理器的 Cache 都没有此块的拷贝。

- 专有(exclusive)——仅有一个处理器上有此块的拷贝,且已对此块进行了写操作,而主存的拷贝仍是旧的。这个处理器称为此块的拥有者(owner)。

除了要跟踪每个 Cache 块的状态之外,由于写作废操作的需要,还必须记录共享此块的处理器信息。最简单的方法是对每个主存块设置一个位向量,当此块被共享时,每个位指出与之对应的处理器是否有此块的拷贝。当此块为专有时,可根据位向量来寻找此块的拥有者。

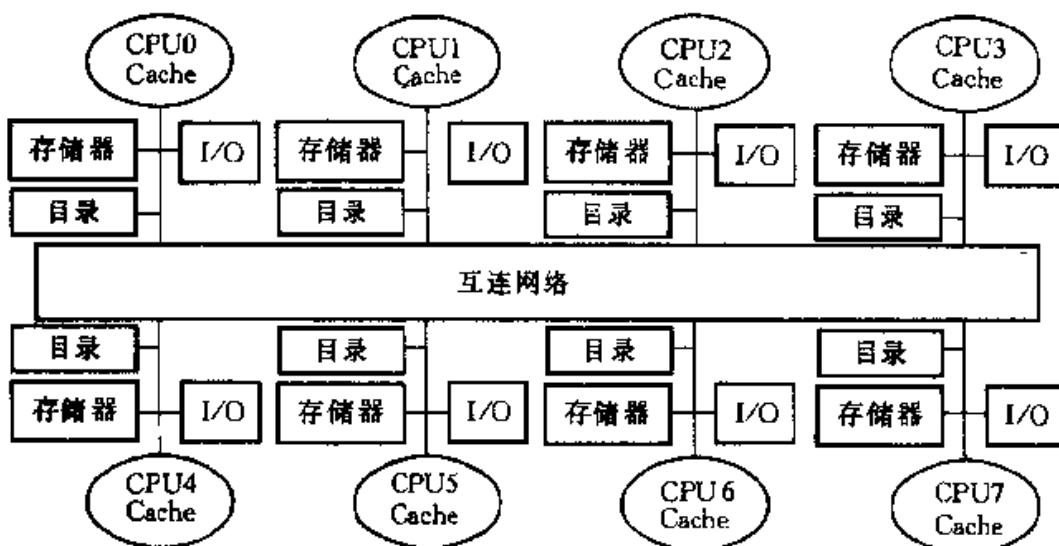


图 7.3 对每个结点增加目录后的分布式存储器结构

每个块的 Cache 状态转换与前面监听 Cache 的情况相同。这里仍旧采用在监听 Cache 中所做的简单假设:对共享数据进行写总会产生一次写失效;处理器封锁该数据直到写操作完毕。因为处理器间不是总线连接,又要避免广播,从而引出两个要考虑的问题:第一,我们不能用连接机制来仲裁,仲裁是监听机制下总线的一项功能;第二,因为连接是面向消息的(总线是面向事务的),所有的消息必须有明确的响应返回。

宿主(home)结点指存放有存储器块和对应地址目录项的结点。因为物理地址空间是静态分布的,对于某一给定的物理地址,含有其存储单元及目录项的结点是确定的。单元地址的高位指出结点号,面低位表示在结点存储器内的偏移量。发出请求的结点可能也是宿主结点,这时消息成为结点内处理的事务。

对数据请求响应时,要将宿主结点的值返回给请求结点。数据写回在两种情况下发生:Cache 中某个块替换时必须要写回宿主存储器;响应宿主结点发出的取数和取数/作废消息时也要写回。只要数据块由专有状态变成共享状态就要写回,因为任何修改过的块必须是唯一的,且任何共享块在宿主存储器中必定是有效的。

本节中,为减少消息的类型和协议的复杂性,假设消息接收和处理的顺序与

消息发送顺序相同。但实际情况并不一定如此,从而会产生其他的复杂性。

下面再来描述一下目录协议的基本点:在每个结点增加了目录存储器用于存放目录,存储器的每一块在目录中对应有一项,每一个目录项主要有“状态”和“位向量”两种成分。“状态”描述该目录所对应存储块的当前情况;“位向量”共有 N 位,其每一位对应于一个处理器的局部 Cache,用于指出该 Cache 中有无该存储块的拷贝。当处理器对某一块进行写操作时,只要根据位向量通知具有相应拷贝的处理器进行作废操作。而这些处理器的数量 n 一般比系统的规模小得多($n \ll N$),从而大大减少了访问量。由于访问量只与 n 有关,而与系统规模大小无关,这就支持了系统的可扩展性。

2. 目录协议的基本实现技术

基于目录的协议中 Cache 的基本状态与监听协议中的相同,目录项的状态也类似于前面所讲的。一个 Cache 块状态转换的操作本质上与监听模式相同。在监听模式中,写失效操作要在总线上进行广播或传送,现在则由从宿主结点取数和目录控制器有选择地发出的写作废操作代替。与监听协议相同,当对 Cache 块进行写时,其必须是专有状态。任何一个共享状态的块在存储器中有其最新拷贝。

在基于目录的协议中,目录承担了一致性协议操作的主要功能。发往一个目录的消息会产生两种不同类型的动作:更新目录状态和发送消息满足请求服务。前面所述的目录项的三个状态(共享、未缓冲和专有)是针对此块的所有拷贝而不是其中某一个拷贝。存储块可能在任何结点中均无拷贝,也可能在多个结点中同时存在拷贝且可共享读,或者仅在一个结点中有唯一拷贝且可专有写。除了每个块的状态外,目录项还用位向量记录拥有此块拷贝的处理器,表示出共享集合。对目录表的请求处理需更新共享集合,作废操作时也要读取这个集合。

目录项可能接收到三种不同的请求:读失效、写失效或数据写回。假设这些操作是原子的。为了进一步理解对目录项的操作,下面来看看在各个状态下所接收到的请求和相应的操作:

(1) 当一个块处于未缓冲状态时,对此块发出的请求及处理操作为:

读失效——将存储器数据送往请求方处理器,且本处理器成为此块的唯一共享结点,本块的状态转换为共享。

写失效——将存储器数据送往请求方处理器,此块成为专有,表示仅存于此块的唯一有效拷贝,共享集合仅有该处理器,这标识出了拥有者。

(2) 当一个块是共享状态时,存储器中的数据是其当前最新值,对此块发出的请求及处理操作为:

读失效——将存储器数据送往请求方处理器,并将其加入共享集合。

写失效——将数据送往请求方处理器,对共享集合中所有的处理器发送写

作废消息,且将共享集合置为仅含有此处理器,本块的状态变为专有。

(3) 当某块处于专有状态时,本块的最新值保存在共享集合指出的拥有者处理器中,从而有三种可能的目录请求:

读失效——将“取数据”的消息发往拥有者处理器,使该块的状态转变为共享,并将数据送回目录结点写入存储器,进而把该数据返送请求方处理器,将请求方处理器加入共享集合。此时共享集合中仍保留原拥有者处理器(因为它仍有一个可读的拷贝)。

写失效——本块将有一个新的拥有者。给旧拥有者处理器发送消息,要求其将数据块送回目录结点,从而再送到请求方处理器,使之成为新的拥有者,并设置新拥有者的标志。此块的状态仍旧是专有。

数据写回——拥有者处理器的 Cache 要替换此块时必须将其写回,从而使存储器中有最新拷贝(宿主结点实际上成为拥有者),此块成为非共享,共享集合为空。

实际机器中采用的目录协议要做一些优化。比如对某个专有块发出读或写失效时,此块将先被送往宿主结点存入存储器,再将其送往原始请求结点,而实际中的机器很多都是将数据从拥有者结点直接送入请求结点(与写回宿主结点的同时)。

基于目录的 Cache 一致性协议采取了“以空间换时间”的策略,减少了访问次数但增加了目录存储器,它的大小与系统规模 N 的平方成正比。为此,人们对基于目录的 Cache 一致性进行了多种改进,提出了有限映射(limited-map)目录和链式结构(chained)目录。有限映射目录假定同一数据在不同 Cache 中的拷贝数总小于一个常数 m ($m \ll N$), m 即为目录中位向量的长度,因而大大减小了目录存储器的规模。有限映射目录的缺点是:当同一数据的拷贝数大于 m 时,必须作特殊处理。链式结构目录不但目录存储器规模小,而且不存在有限映射目录关于 m 的限制,但一致性协议比较复杂。相对于有限映射目录和链式结构目录,前而介绍的基于目录的一致性协议称为全映射(full-map)。

基于目录的 Cache 一致性协议是完全由硬件实现的。此外,还可以用软硬结合的办法实现,即将一个可编程协议处理器嵌到一致性控制器中,这样既减少了成本,又缩短了开发周期。因为可编程协议处理器可以根据实际应用需要很快开发出来,而一致性协议处理中的异常情况可完全交给软件执行。这种软硬结合实现 Cache 一致性的代价是损失了一部分效率。

到目前为止所讨论的一致性协议都作了一些简化的假设,实际中的协议必须处理以下两个实际问题:操作的非原子性和有限的缓存。操作的非原子性产生实现的复杂性;有限的缓存可能导致死锁问题。设计者面临的一个问题是通过非原子的操作和有限的缓存设计出一种正确的且无死锁的协议,这些因素是

所有并行机面临的基本问题。

7.3 互连网络

互连网络是将集中式系统或分布式系统中的结点连接起来所构成的网络，这些结点可能是处理器、存储模块或者其他设备，它们通过互连网络进行信息交换。在拓扑上，互连网络为输入和输出两组结点之间提供一组互连或映象(mapping)。

本节介绍构造多处理机的互连网络。首先讨论互连网络的通信特性和拓扑结构，然后再来分析并行结构的可扩展性。我们希望得到的是数据传送速率高、延迟低、通信频带宽的网络。

7.3.1 互连网络的性能参数

互连网络的拓扑可以采用静态或动态的结构。静态网络由点和点直接相连而成，这种连接方式在程序执行过程中不会改变。动态网络是用开关通道实现的，它可动态地改变结构，使其与用户程序中的通信要求匹配。静态网络常用来实现一个系统中子系统或计算结点之间的固定连接。动态网络常用于集中式共享存储器多处理系统中。

在分析各种网络的拓扑结构之前，我们先来定义几个常用于估算网络复杂性、通信效率和价格的参数。一般说来，网络用图来表示。这种图由用有向边或无向边连接的有限个结点构成。其结点数称为网络规模(network size)。

结点度 与结点相连接的边的数目称为结点度(node degree)。这里的边表示链路或通道。链路或通道是指网络中连接两个结点并传送数字信号的通路。在单向通道的情况下，进入结点的通道数叫做入度(in degree)，而从结点出来的通道数则称为出度(out degree)，结点度是这两者之和。结点度应尽可能地小并保持恒定。

网络直径 网络中任意两个结点间最短路径长度的最大值称为网络直径。网络直径应当尽可能地小。

等分宽度 在将某一网络切成相等两半的各种切法中，沿切口的最小通道边数称为通道等分宽度(channel bisection width)。等分宽度是能很好地说明将网络等分的交界处最大通信带宽的一个参数。

另一个量化参数是结点间的线长(或通道长度)。它会影响信号的延迟、时钟扭斜和对功率的需要。

对于一个网络，如果从其中的任何一个结点看，拓扑结构都是一样的话，则称此网络为对称网络。对称网络较易实现，编制程序也较容易。

路由(routing) 在网络通信中对路径的选择与指定。互连网络中路由功能较强将有利于减少数据交换所需的时间,因而能显著地改善系统的性能。通常见到的处理单元之间的数据路由功能有移数、循环、置换(一对一)、广播(一对全体)、选播(多对多)、个人通信(一对多)、混洗、交换等。这些路由功能可在环形、网络形、超立方体以及多级网络上实现。

为了反映不同互连网络的连接特性,每一种互连网络可用一组互连函数来定义。如果把互连网络的 N 个入端和 N 个出端各自用整数 $0, 1, \dots, N-1$ 代表,则互连函数表示互连的出端号和入端号的一一对应关系。令互连函数为 f ,则它的作用是:对于所有的 $0 \leq j \leq N-1$, 同时存在入端 j 连至出端 $f(j)$ 的对应关系。下面介绍几种数据路由功能:

1. 循环(rotation)

若把互连函数 $f(x)$ 表示为

$$(x_0, x_1, x_2, \dots, x_j)$$

则代表对应关系为

$$f(x_0) = x_1, f(x_1) = x_2, \dots, f(x_j) = x_0$$

$j+1$ 称为该循环的周期。

2. 置换(permutation)

指对象的重新排序。对于 n 个对象来说,有 $n!$ 种置换。 n 个对象可照此重新排序,全部的置换形成一个与复合运算有关的置换集合。例如,置换 $\pi = (a, b, c)(d, e)$ 表示了置换映射: $f(a) = b, f(b) = c, f(c) = a, f(d) = e$ 和 $f(e) = d$ 。这里循环 (a, b, c) 周期为 3, 循环 (d, e) 周期为 2。

可以用交叉开关来实现置换,也可以用一次或多次通过多级网络来实现某些置换,还可用移数或广播操作实现置换。

3. 均匀混洗(shuffle)

Harold Stone 于 1971 年为并行处理应用提出了一种特殊转换功能,它所对应的映射如图 7.4(a)所示,图 7.4(b)为其逆过程。

一般说来,为了对 $n = 2^k$ 个对象均匀混洗,可以用 k 位二进制数 $x = (x_{k-1}, \dots, x_1, x_0)$ 来表示定义域中的每个对象。均匀混洗将 x 映射到 $f(x)$,得到 $f(x) = (x_{k-2}, \dots, x_1, x_0, x_{k-1})$ 。这是将 x 循环左移 1 位得到。

4. 超立方体路由功能

图 7.5 表示的是一个三维二进制立方体网络。它有三种路由功能,可根据结点的二进制地址 $(C_2 C_1 C_0)$ 中的某一位来确定。例如,可以根据最低位 C_0 寻址,即在最低位 C_0 不同的相邻结点之间交换数据,如图 7.5(b)所示。同样,分别根据中间位 C_1 (图 7.5(c)) 和最高位 C_2 (图 7.5(d)) 可得其他两种路由模式。一般情况下,一个 n 维超立方体共有 n 种路由功能,分别由 n 位地址中

的每一位求反位值来确定。将 $x = (x_{n-1}, \dots, x_k, \dots, x_1, x_0)$ 映射到 $f(x)$, 得到 $f(x) = (\bar{x}_{n-1}, \dots, \bar{x}_k, \dots, x_1, x_0)$ 。

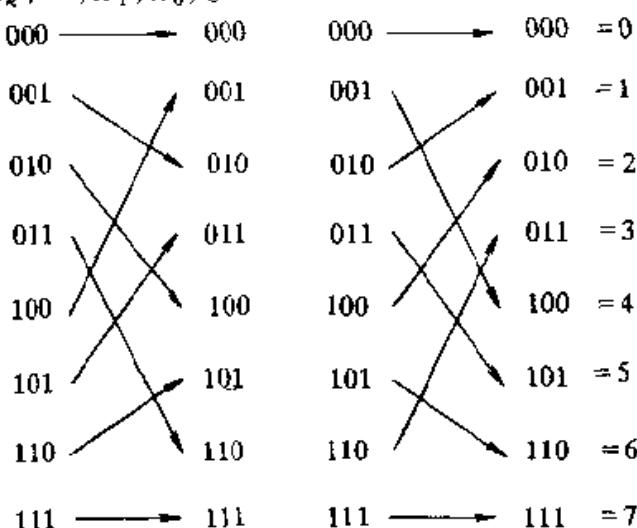
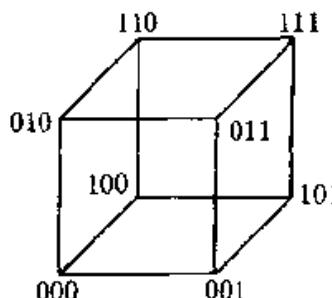


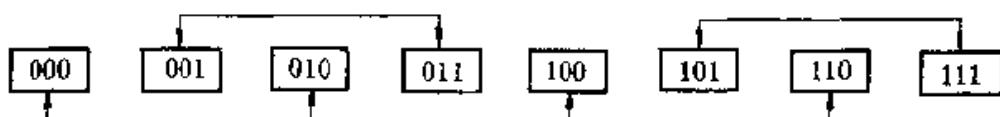
图 7.4 对象为 8 的均匀混洗及其逆映射



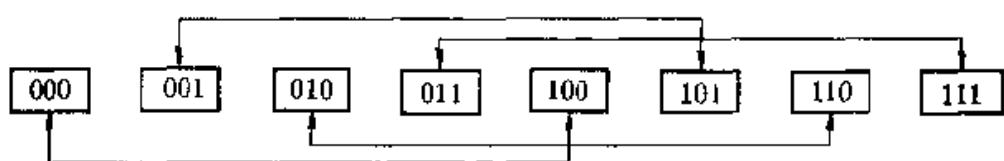
(a) 3-立方体 (结点地址用二进制表示)



(b) 根据最低位 C_0 路由



(c) 根据中间位 C_1 路由



(d) 根据最高位 C_2 路由

图 7.5 由二进制 3-立方体确定的三种路由功能

5. 广播和选播

广播是一种一对全体的映射,选播是一个子集到另一子集(多对多)的映射。消息传递型多处理机一般有广播信息机构,广播常常作为多处理机中的全局操作来处理。

通过上面的讨论,可概括出影响互连网络性能的因素为:

- (1) 功能特性——网络如何支持路由、中断处理、同步、请求/消息组合和一致性。
- (2) 网络时延——单位消息通过网络传送时最坏情况下的时间延迟。
- (3) 带宽——通过网络的最大数据传输率,用 MB/s 表示。
- (4) 硬件复杂性——诸如导线、开关、连接器、仲裁和接口逻辑等的造价。
- (5) 可扩展性——在增加机器资源使性能可扩展的情况下,网络具备模块化可扩展的能力。

7.3.2 静态连接网络

静态网络使用直接链路,它一旦构成后就固定不变。这种网络比较适合于构造通信模式可预测或可用静态连接实现的计算机系统。下面介绍几种静态网络的拓扑结构、网络参数及其可扩展性。

1. 线性阵列(linear array) 这是一种一维的线性网络,其中 N 个结点用 $N-1$ 个链路连成一行(图 7.6(a))。内部结点度为 2,端结点度为 1。直径为 $N-1$, N 较大时,直径就比较长。等分宽度为 1。线性阵列是连接最简单的拓扑结构。这种结构不对称,当 N 很大时,通信效率很低。

在 N 很小的情况下,使用线性阵列是相当经济和合理的。由于直径随 N 线性增大,因此当 N 比较大时,就不应使用这种方案了。应当指出,线性阵列与总线的区别很大,总线是通过切换与其连接的许多结点来实现时分特性的,而线性阵列则允许不同的源结点和目的结点对并行地使用其不同的部分(通道)。

2. 环和带弦环(chordal ring) 环是用一条附加链路将线性阵列的两个端点连接起来而构成的(图 7.6(b))。环可以单向工作,也可以双向工作。它是对称的,结点度是常数 2。双向环的直径为 $N/2$,单向环的直径是 N 。

如果将结点度由 2 提高至 3 或 4,即可得到如图 7.6(c) 和 7.6(d) 所示的两种带弦环。增加的链路愈多,结点度愈高,网络直径就愈小。16 个结点的环(图 7.6(b))与两个带弦环(图 7.6(c) 和 7.6(d))相比,网络直径分别由 8 减至 5 和 3。在极端情况下,图 7.6(f) 的全连接网络(completely connected network)的结点度为 15,直径最短,为 1。

3. 循环移数网络(barrel shifter) 图 7.6(e) 所示的是一个循环移数网络,其结点数 $N=16$,它是通过在环上每个结点到所有与其距离为 2 的整数幂的结点之间都增加一条附加链而构成的。这就是说,如果 $|j-i|=2^r, r=0,1,2,\dots$,

$n=1$, 网络规模 $N=2^n$, 则结点 i 与结点 j 连接。这种循环移数网络的结点度为 $d=2n-1$, 直径 $D=n/2$ 。

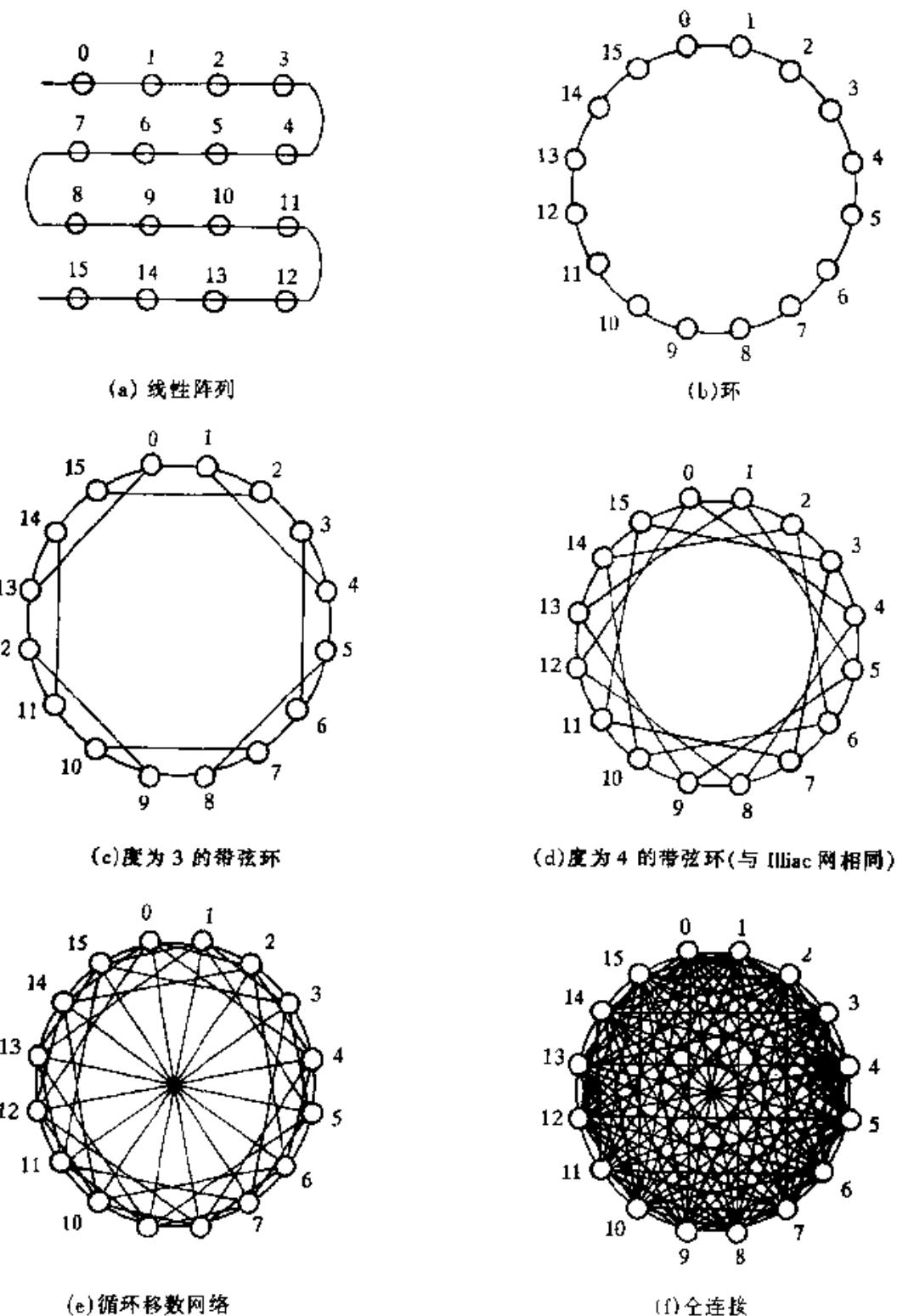


图 7.6 几种线性网络

显然, 循环移数网络的连接特性与结点度较低的任何带弦环相比都有了改进。对 $N=16$ 的情况, 循环移数网络的结点度为 7, 直径为 2。它的复杂性比全

连接网络(图 7.6(f))低得多。

4. 树形和星形(tree and star) 一棵 5 层 31 个结点的二叉树如图 7.7(a)所示。一般说来,一棵 k 层完全平衡的二叉树有 $N = 2^{k-1}$ 个结点。最大结点度是 3, 直径是 $2(k-1)$ 。由于结点度是常数,因此二叉树是一种可扩展的结构,但其直径较长。哥伦比亚大学于 1987 年研制成的 DADO 多处理机即采用 10 层二叉树形式,有 1 023 个结点。

星形是一种 2 层树,结点度较高,为 $d = N - 1$ (图 7.7(b))。直径较小,是一常数 2。星形结构一般用于有集中监督结点的系统中。

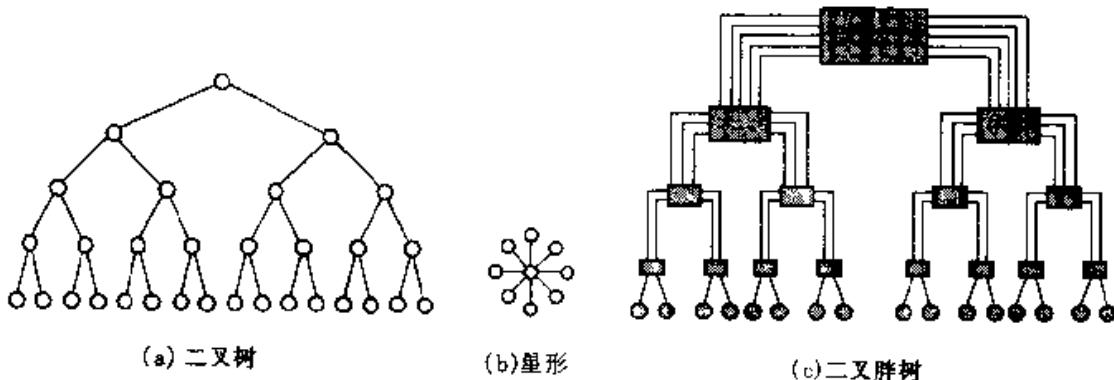


图 7.7 树形、星形、胖树形网络

5. 胖树形 1985 年 Leiserson 提出将计算机科学中所用的一般树结构修改为胖树形(fat tree)。二叉胖树结构如图 7.7(c)所示,胖树的通道宽度从叶结点往根结点上行方向逐渐增宽,它更像真实的树,愈靠近树根的枝叉愈粗。

使用传统二叉树的主要问题之一就是通向根结点的瓶颈问题,这是因为根部的交通最忙。胖树的提出使该问题得到了缓解。

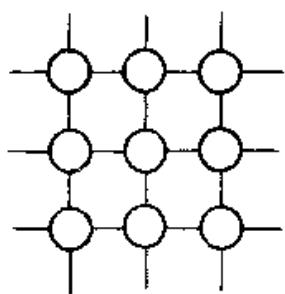
6. 网格形和环形网 图 7.8(a)为一个 3×3 网格形网络。这是一种比较流行的结构,它已经以各种变体形式在 CM-2 和 Intel Paragon 等机器中得到实现。

一般说来, $N = n^k$ 个结点的 k 维网络的内部结点度为 $2k$, 网络直径为 $k(n-1)$ 。必须指出,图 7.8(a)所示的纯网格形不是对称的。边结点和角结点的结点度分别为 3 或 2。

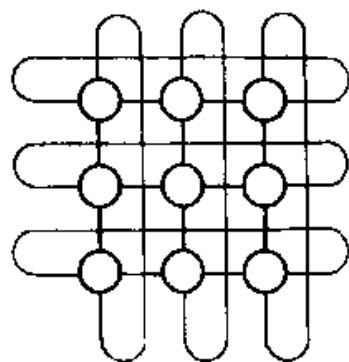
图 7.8(b)所示的环形网可以看做是直径更短的另一种网格。这种拓扑结构将环形和网格组合在一起,并能向高维扩展。环形网沿阵列每行和每列都有环形连接。一般说来,一个 $n \times n$ 二元环网的结点度为 4, 直径为 $2 \times \lfloor n/2 \rfloor$

环网是一种对称的拓扑结构,所有附加的回绕连接可使其直径比网格结构减少二分之一。

7. 超立方体 这是一种二元 n -立方体结构,它已在 nCUBE 和 CM-2 等系统中得到了实现。一般说来,一个 n -立方体由 $N = 2^n$ 个结点组成,它们分布在 n 维上,每维有两个结点。8 个结点的 3-立方体如图 7.9(a)所示。



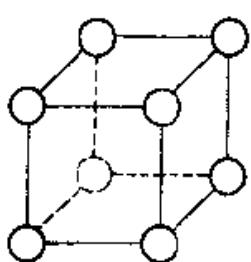
(a) 网格形



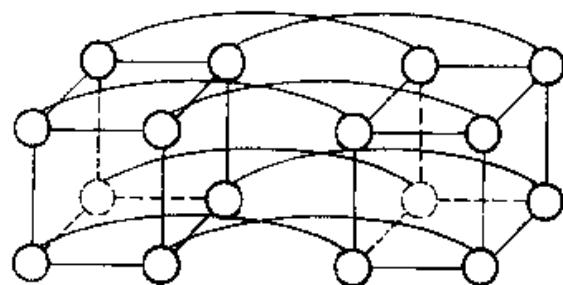
(b) 环形网

图 7.8 一维网格和环形网

4-立方体可通过将两个 3-立方体的相应结点互连组成,如图 7.9(b)所示。一个 n -立方体的结点度等于 n ,也就是网络的直径。实际上,结点度随维数线性地增加,所以很难设想超立方体是一种可扩展结构。



(a) 3-立方体



(b) 由 2 个 3-立方体组成的 4-立方体

图 7.9 超立方体

8. k 元 n -立方体网络 环形、网格形、环网形、二元 n -立方体(超立方体)等网络都是 k 元 n -立方体网络系统的拓扑同构体。图 7.10 所示就是一种 4 元 3-立方体网络。

参数 n 是立方体的维数, k 是基数或者说是沿每个方向的结点数(多重性)。这两个数与网络中结点数 N 的关系为

$$N = k^n, \quad (n = \log_k N)$$

k 元 n -立方体的结点可用基数为 k 的 n 位地址 $A = a_0a_1a_2\cdots a_n$ 来表示, 其中 a_i 代表第 i 维结点的位置。为简单起见, 所有链路都认为是双向的。网络中每条线代表两个通信通道, 每个方向一个。图 7.10 中各结点之间的连线都是双向链路。

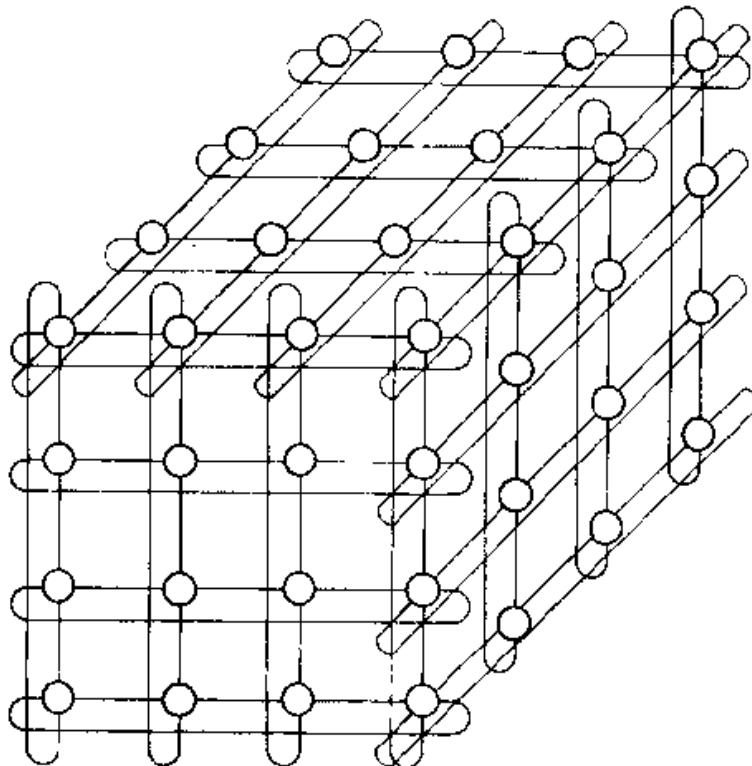


图 7.10 $k=4$ 和 $n=3$ 的 k 元 n -立方体网络; 未画出隐藏的结点和连接

按照惯例, 低维 k 元 n -立方体称为环网, 而高维二元 n -立方体则称为超立方体。

低维网络在负载不均匀情况下运行较好, 因为它们有较多的资源共享。在高维网络中, 连线常分配给指定的维, 各维之间不能共享。例如, 在二元 n -立方体中, 可能有的线已达到饱和, 而物理上分配给其他维的相邻的连线却都还空闲。

两络直径的变化范围很大。但随着硬件路由技术的不断革新(如虫孔方式), 路由已不是一个严重问题, 因为任意两结点间的通信延迟在高度流水线操作下几乎是固定不变的。链路数会影响网络价格, 等分宽度将影响网络的带宽, 对称性会影响可扩展性和路由效率。

7.3.3 动态连接网络

为了达到多用或通用的目的, 需要采用动态连接网络, 它能根据程序要求实

现所需的通信模式。它不用固定连接,而是沿着连接通路使用开关或仲裁器以提供动态连接特性。按照价格和性能增加的顺序,动态连接网络的排队次序为总线系统、多级互连网络(MIN)和交叉开关网络。

采用动态网络的多处理机的互连是在程序控制下实现的。定时、开关和控制是动态互连网络的三个主要操作特征。定时可以用同步方式,也可以用异步方式进行。同步网络由一个全局时钟来控制,用它来同步网络的全部动作。异步网络利用握手机制来协调需要使用的同一网络内各种设备。

根据级间连结方式,单级网络(single-stage network)也称循环网络(recirculating network),因为数据项在到达最后目的地之前可能在单级网络中循环多次。单级网络的成本比较低,但在建立某种连接时可能需要多次通过网络。交叉开关和多端口存储器结构都属于单级网络。

多级网络由一级以上的开关元件构成。这类网络可以把任一输入与任一输出相连。级间连接模式的选择取决于网络连接特性。不同级的连接模式可能相同也可能不相同,这与所设计的网络的类型有关。Omega网、Flip网和Baseline网都是多级网络。

如果同时连接多个输入输出对时,可能会引起开关和通信链路使用上的冲突,这种多级网络称为阻塞网络(blocking network)。阻塞网络的实例有Omega网(Lawrie, 1975)、Baseline网(Wu 和 Feng, 1980)、Banyan网(Goke 和 Lipovski, 1973)和Delta网(Patel, 1979)。经过图形转换后,可以证明一些阻塞网络是等价的。实际上,大多数多级网络都是阻塞网络。在阻塞网络中,为了建立某些输入输出之间的连接,可能需要多次通过网络。

如果多级网络通过重新安排连接方式可以建立所有可能的输入输出之间的连接,则称之为非阻塞网络(nonblocking network)。这类网络中,任何输入输出对之间总可以建立连接通路。Benes网络(Benes, 1965)具有这种功能,但是它的级数比一般阻塞网络增加了一倍才实现非阻塞连接。如果增加级数或者限制连接模式,某些阻塞网络也可以成为非阻塞网络。

下面根据级数和阻塞或非阻塞来讨论几类主要的开关网络。首先介绍总线、交叉网络和多端口存储器结构,然后讨论多级网络。

1. 总线系统

总线系统实际上是一组导线和插座,用于处理与总线相连的处理器、存储模块和外围设备间的数据业务。总线只用于源(主部件)和目的(从部件)之间处理业务。在多个请求情况下,总线仲裁逻辑必须每次能将总线服务分配或重新分配给一个请求。

基于这一原因,数字总线已被称为多个功能模块间的争用总线(contention bus)或时分总线(time-sharing bus)。总线系统价格较低,带宽较窄。它有很多

可用的工业和 IEEE 总线标准。

图 7.11 所示的是一种总线连接的多处理机系统。系统总线在处理机、I/O 子系统、存储模块或辅助存储设备(磁盘、磁带机等)之间提供了一条公用通信通路。系统总线通常设置在印刷电路板底板上。其他的处理器板、存储器板或设备接口板都通过插座或电缆插入底板。

主动设备或主设备(处理机或 I/O 子系统)产生访问存储器的请求,被动设备或从设备(存储器或外围设备)则响应请求。公用总线是在分时基础上工作的。总线研制中的重要问题有总线仲裁、中断处理、一致性协议和总线事务的处理等。

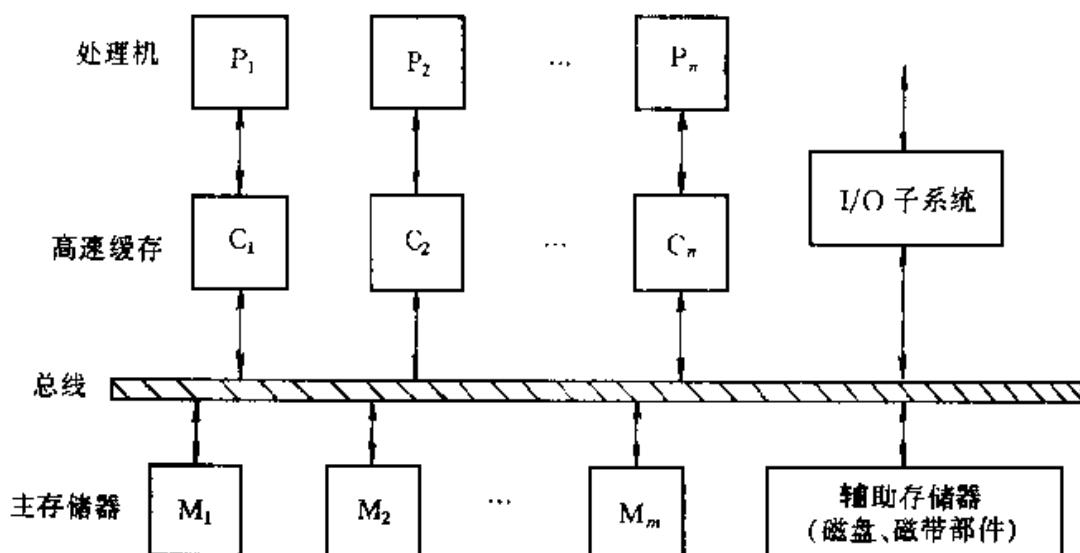


图 7.11 一种总线连接的多处理机系统

2. 交叉开关网络

在交叉开关网络中,每个输入端通过一个交叉点开关可以无阻塞地与一个空闲输出端相连。交叉开关网络是单级网络,它由交叉点上的一元开关构成。交叉网络主要用于中小型系统。

从存储器读出的数据一旦可用时,该数据通过同一交叉开关回送给请求的处理器。通常,这类交叉开关网络需要使用 $n \times m$ 个交叉点开关。正方形交叉开关网络($n = m$)可以无阻塞地实现 $n!$ 种置换。

交叉开关网络是单级无阻塞置换网络。交叉开关网络中每个交叉点是一个可以打开或关闭的一元开关,提供源(处理器)和目的(存储器)之间点对点的连接通路。对一个 $n \times n$ 的交叉开关网络来说,需要 n^2 套交叉点开关以及大量的连线。当 n 很大时,交叉开关网络所需要的硬件数量非常巨大。因此,到目前为止只有 $n \leq 16$ 的小型交叉开关网络用在商品化的机器中。

在交叉开关网络的每一行中可以同时接通多个交叉点开关,所以交叉点开

关网络中 n 对处理器可以同时传送数据。

交叉开关网络的带宽和互连特性最好。图 7.12 所示的是一种交叉开关网络。

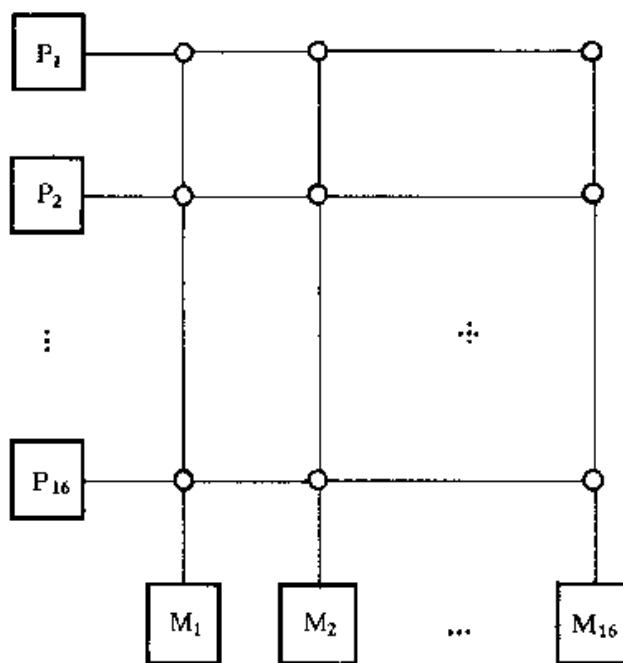


图 7.12 一种交叉开关网络

交叉开关网络每个周期可以实现 n 个数据传输,与每个总线周期只传一个数据相比,它的频宽最高。交叉开关网络对小型多处理机系统来说性能价格比较高。但单级交叉开关网络一旦构成后将不能扩充。

3. 多端口存储器

由于大型系统使用交叉开关网络的成本无法承受,所以许多大型的多处理机系统都采用多端口存储器结构。其主要思想是将所有交叉点仲裁逻辑和跟每个存储器模块有关的开关功能移到存储器控制器中。

图 7.13 所示为典型的多端口存储器结构。由于增加了访问端口和相应的逻辑线路,存储器模块的成本就变得较为昂贵。从图中可以看到,每个存储器模块的 n 个输入端口与 n 个开关相连,一次只能接收 n 台处理器中的一个请求。

多端口存储器结构是一个折衷方案,它介于低成本低性能的总线系统和高成本高带宽的交叉开关系统之间。总线被所有处理器和与之相连的设备模块分时地共享。多端口存储器则负责分解各台处理器的请求冲突。

当 m 和 n 值很大时,这种多端口存储器结构将变得十分昂贵。典型的多处理机应用配置是四台处理器和 16 个存储器模块。多端口存储器结构的多处理机系统也不能扩展,因为端口数目一旦固定后,如果不重新设计存储控制器就无

法再增加处理器了。还有一个缺点是当系统配置很大时,需要大量的互连电缆和连接器。

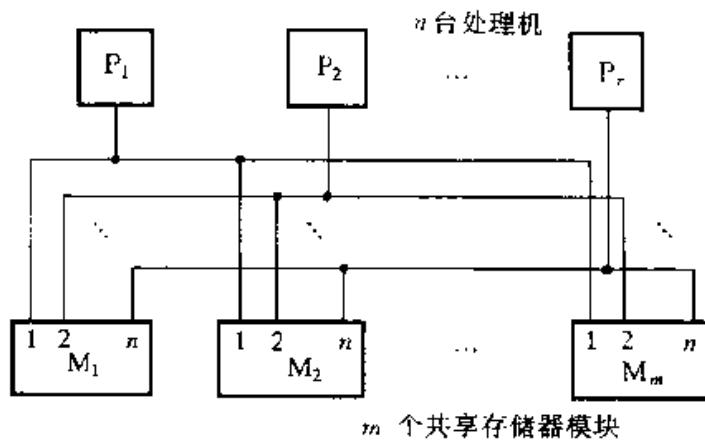


图 7.13 用于多处理机系统的多端口存储器结构

4. 多级网络

多级网络可用于构造大型多处理机系统。一种通用多级网络如图 7.14 所示,其中每一级都用了多个 $a \times b$ 开关,相邻级开关之间都有固定的级间连接 (ISC)。为了在输入和输出之间建立所需的连接,可用动态设置开关的状态来实现。

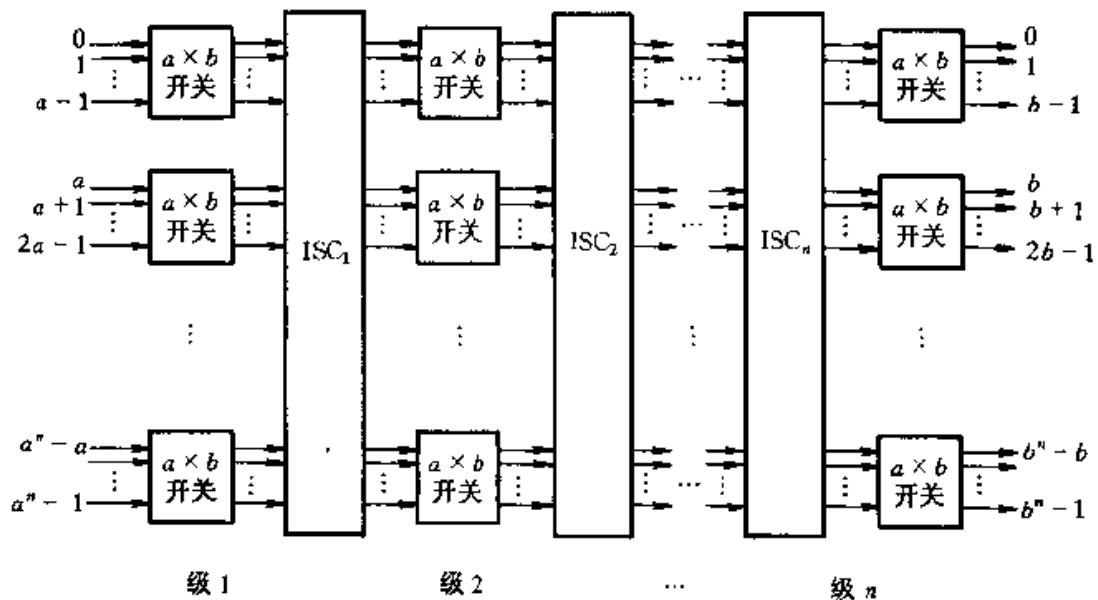


图 7.14 由 $a \times b$ 开关模块和级间构成的通用多级互连网络结构

各种多级网络的区别就在于所用开关模块和级间连接模式的不同。一个 $a \times b$ 开关模块有 a 个输入和 b 个输出。一个二元开关与 $a=b=2$ 的 2×2 开关模块相对应。在理论上 a 与 b 不一定要相等,但实际上 a 和 b 经常选为 2 的整

数幂,即 $a = b = 2^k, k \geq 1$ 。最简单的开关模块是 2×2 开关。常用的级间连接模式包括混洗、交叉、立方体连接等。这里只介绍 Omega 网络,它已经应用于现在的机器。

图 7.15(a)至 7.15(d)画出了用于构造 Omega 网络的 2×2 开关四种可能的连接方式。图 7.15(e)所示的是一个 16×16 Omega 网络,共需 4 级 2×2 开关。网络左侧有 16 个输入,右侧有 16 个输出。形成对 16 个对象的均匀混洗模式。

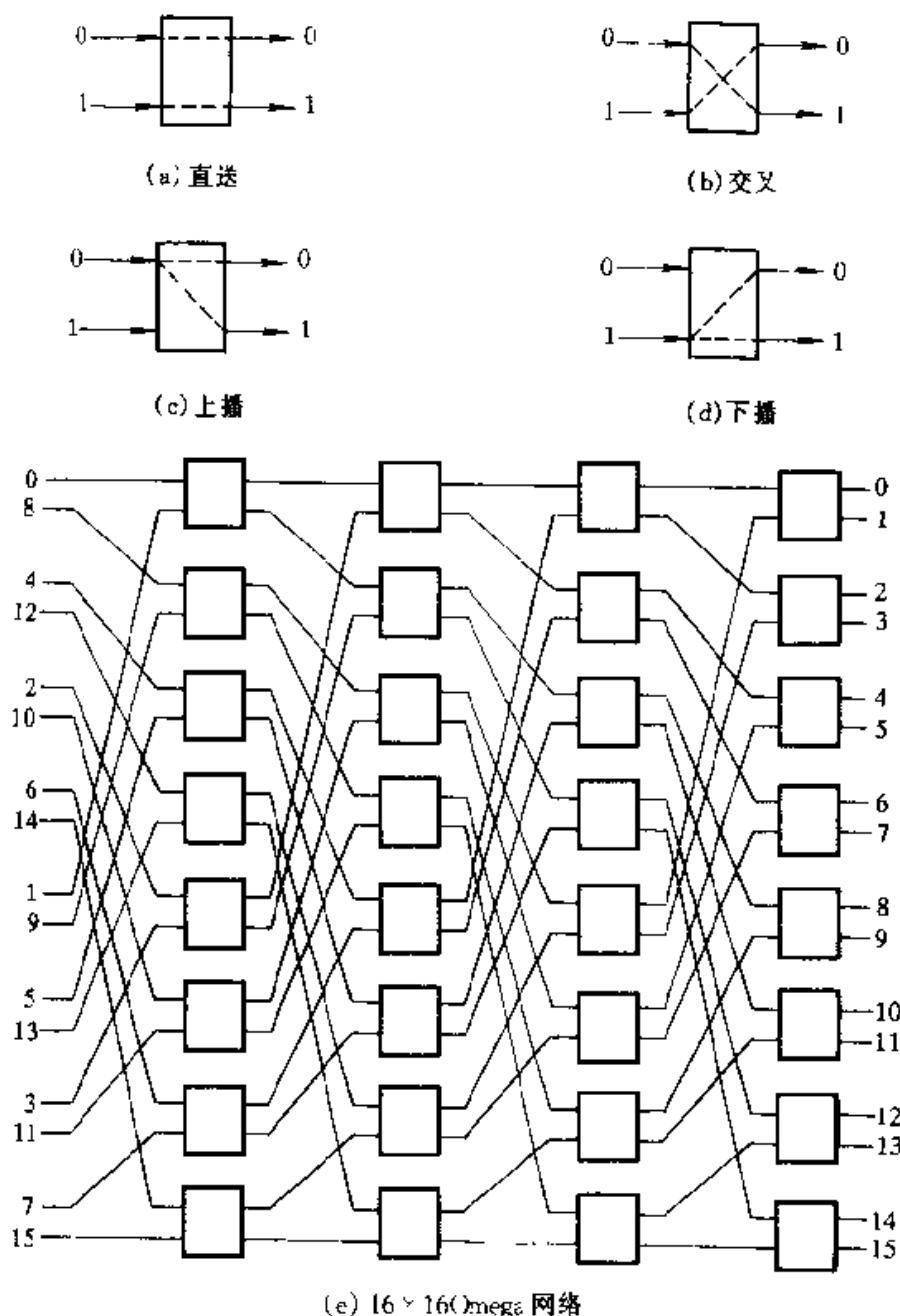


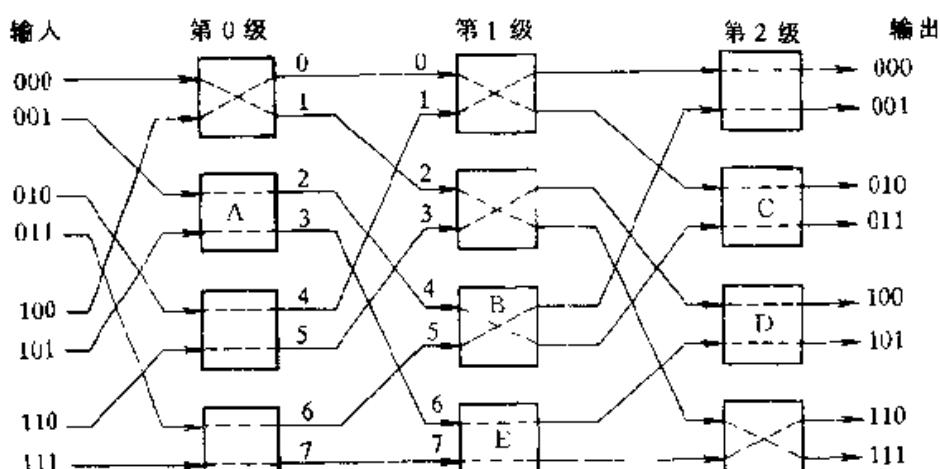
图 7.15 用 2×2 开关和均匀混洗构成的 16×16 Omega 网络

一般说来,一个 n 输入的 Omega 网络需要 $\log_2 n$ 级 2×2 开关,每级要用

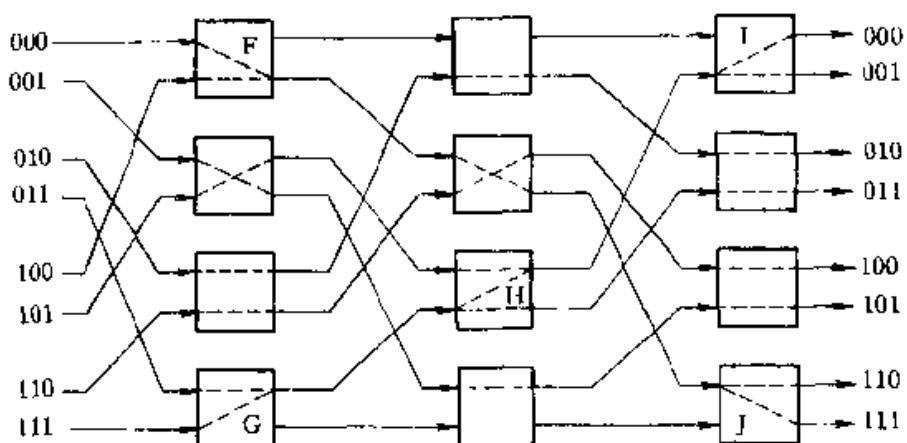
$n/2$ 个开关模块, 网络共需 $n \log_2 n / 2$ 个开关。每个开关模块采用单元控制方式。不同的开关状态组合可实现各种置换、广播或从输入到输出的其他连接。

8 个输入端的 Omega 网络如图 7.16 所示, 从输入级到输出级依次编号为 0 到 $\log_2 n - 1$ 。通过检查二进制目的地址编码来控制数据路径。目的地址编码从高位开始的第 i 位为 0 时, 第 i 级的 2×2 开关的输入端与上输出端连接, 否则输入端与下输出端连接。

下面通过例子来介绍路由算法。图 7.16(a)和图 7.16(b)的开关设置分别对应于置换 $\pi_1 = (0, 7, 6, 4, 2)(1, 3)(5)$ 和置换 $\pi_2 = (0, 6, 4, 7, 3)(1, 5)(2)$ 。



(a) Omega 网络无阻塞地实现置换 $\pi_1 = (0, 7, 6, 4, 2)(1, 3)(5)$



(b) 置换 $\pi_2 = (0, 6, 4, 7, 3)(1, 5)(2)$ 在开关 F, G 和 H 上阻塞

图 7.16 由 2×2 开关组成 8×8 Omega 网络的两种开关设置

图 7.16(a)中的开关设置可以实现 π_1 , 它的映射为 $0 \rightarrow 7, 7 \rightarrow 6, 6 \rightarrow 4, 4 \rightarrow 2, 2 \rightarrow 0, 1 \rightarrow 3, 3 \rightarrow 1, 5 \rightarrow 5$ 。观察一个消息从输入端 001 到输入端 011 的路径。它使用了开关 A、B 和 C。由于目的地址编码 011 的最高位为“0”, 所以开关 A 设

置成直送状态,使输入端 001 与开关 A 的上输出端(编号为 2)连接。011 的中间一位是“1”,开关 B 设置成交换状态,使输入端 4 与下输出端连接。011 的最低位是“1”,开关 C 设置成直送状态。同理,开关 A、E 和 D 设置成对应状态,形成一条消息从输入端 101 到输出端 101 的路径。实现图 7.16(a)的设置 π_1 所要求的开关设置不存在冲突。

现在再来考察 8 个输入端口的 Omega 网络实现置换 π_2 (图 7.16(b))的情况。开关 F、G 和 H 的设置发生了冲突。F 产生冲突是由于 $000 \rightarrow 110$ 和 $100 \rightarrow 111$ 引起的,因为两个目的地址编码的最高位都是“1”,开关 F 的两个输入端都要求与下输出端相连。为了解决这一冲突,必须拒绝一个请求。

同样地, $011 \rightarrow 000$ 和 $111 \rightarrow 011$ 引起开关 G 冲突, $101 \rightarrow 001$ 和 $011 \rightarrow 000$ 引起开关 H 冲突。开关 I 和 J 的设置能实现一个输入端与两个输出端相连的广播功能,其条件是硬件开关要有四种合法状态。以上例子表明并不是所有的置换在 Omega 网络中一次通过便可以实现。

Omega 网络是一种阻塞网络。当出现阻塞时,可以采用几次通过的方法来解决冲突。例如 π_2 ,在第一次通过时实现连接 $000 \rightarrow 110, 001 \rightarrow 101, 010 \rightarrow 010, 101 \rightarrow 001, 110 \rightarrow 100$ 。在第二次通过时实现连接 $011 \rightarrow 000, 100 \rightarrow 111, 111 \rightarrow 011$ 。通常,如果采用 2×2 开关元件, n 个输入端 Omega 网络一次通过可以实现 $n^{n/2}$ 个置换,但总共有 $n!$ 个置换。

$n=8$ 时,意味着 8 个输入端的 Omega 网络一次通过只能实现全部置换的 10.16%,即 $8^4/8! = 4\ 096/40\ 320 = 0.1016 = 10.16\%$ 。所有其他置换将引起阻塞。实现这些置换需要三次通过。一般来说, n 个输入端的 Omega 网络实现非阻塞连接最多需要通过的次数为 $\log_2 n$ 。

构成动态两络的总线、多级网络、交叉开关中,总线的造价最低,但其缺点是每台处理器可用的带宽较窄。总线所存在的另一个问题是容易产生故障。有些容错系统,如用于事务处理的 Tandem 多处理机等,常采用双总线以防止系统产生简单的故障。

由于交叉开关的硬件复杂性以 n^2 倍上升,所以其造价最为昂贵。但是,交叉开关的带宽和路由性能最好。如果网络的规模较小,它就是一种理想的选择。

多级网络则是两个极端之间的折衷。它的主要优点在于采用模块结构,因而可扩展性较好。然而,其时延随网络级数的增高而上升。另外,由于增加了连线和开关复杂性,价格也是一种限制因素。

几种静态拓扑结构针对一些特定的应用,其可扩展性比较好。随着光技术和微电子技术迅速发展,大规模多级两络和交叉开关网络在建立通用计算机的动态连接时也会变得更加经济合理。

7.4 同步与通信

7.4.1 同步机制

同步机制通常是基于硬件提供的有关同步指令,通过用户级软件例程来建立的。在大规模机器或竞争激烈的情况下,同步会成为性能的瓶颈,导致较大的延迟开销。

1. 基本的硬件原语

在多处理机中实现同步,所需的主要功能是一组能自动读出并修改存储单元的硬件原语。如果没有这种功能,建立基本的同步原语的代价将会是非常大,并且这种代价随处理器个数的增加而增加。基本硬件原语有几种形式可供选择,它们能够自动读/修改单元。这些原语作为基本构件,被用来构造各种各样的用户级同步操作。通常情况下,不希望用户直接使用硬件原语,这些原语主要是供系统程序员用来编制同步库函数。

用于构造同步操作的一个典型操作是原子交换(atomic exchange),它的功能是将一个存储单元的值和一个寄存器的值进行交换。为此我们要将一个存储单元定义为锁,该锁的值为“0”表示锁是开的,为“1”表示已上锁。当处理器要给该锁上锁时,是将对应于该锁的存储单元的值与存放在某个寄存器中的“1”进行交换。如果别的处理器早已锁住了该单元,则交换指令返回的值为“1”,否则为“0”。在后一种情况下,该锁的值会从“0”变成“1”,这样,其他竞争的交换指令的返回值就不会是“0”。

例如两个处理器同时进行交换操作,只有一个处理器会先执行成功而返回“0”,而第二个处理器执行会返回“1”。采用原子交换原语实现同步的关键是操作的原子性:交换是不可分的,两个并发的交换操作写的顺序性保证了两个处理器不可能同时获得同步变量锁。

还有一些别的原语可用来实现同步,它们均可读和更新存储单元值,并指示出这两个操作是否已执行成功。在许多机器上常有的一个操作是测试并置定(test-and-set),即先测试一个值,如果符合条件则修改其值。例如我们定义一个操作来检测某个值是否为“0”,如果为“0”则置“1”,这和刚才的原子交换类似,另一个同步原语是读取并加1(fetch-and-increment),它返回存储单元的值并自动增加该值。

现在一些机器上用到的原子读-修改操作略有不同。在一次单独的访存内完成全部的操作会有一些困难,因为它需要在一条单独的不可中断的指令中完成一次存储器读和一次存储器写,在这一过程中不允许进行其他的访存操作,这

带来了硬件实现上的复杂性。

一种可供选择的方法是使用指令对,从第二条指令的返回值可以判断该指令对的执行是否成功。这里两条指令执行等价于原子操作。所谓等价于原子操作是指所有别的处理器进行操作可认为是在本指令对前或后进行的,从而本指令对看上去是原子执行的,在两条指令间无别的处理器改变锁单元的数据值。

指令对包括一条特殊的称为 LL(Load Linked 或 Load Locked)的取指令,及一条称为 SC(Store Conditional)的特殊存指令。指令顺序执行:如果由 LL 指明的存储单元的内容在 SC 对其进行写之前已被其他指令改写过,则第二条指令 SC 执行失败;如果在两条指令间进行切换也会导致 SC 执行失败。SC 将返回一个值来指出该指令操作是否成功。LL 返回初始值,SC 如果执行成功返回“1”,否则返回“0”。下面代码序列实现对由 R1 指出的存储单元进行原子交换操作:

```
TRY: MOV R3,R4           ;送交换值
      LL   R2,0(R1)       ;Load Linked
      SC   R3,0(R1)       ;Store Conditional
      BEQZ R3,TRY         ;存失败转移
      MOV  R4,R2           ;将取的值送 R4
```

最终 R4 和由 R1 指向的单元值进行原子交换,在 LL 和 SC 之间如有别的处理器插入并修改了存储单元的值,SC 将返回“0”并存入 R3 中,从而使指令序列再重新循环。

LL/SC 机制的一个优点是可用来构造别的同步原语。例如,原子的 fetch-and-increment:

```
TRY: LL   R2,0(R1)       ;Load Linked
      ADDI R2,R2,#1        ;增加 1
      SC   R2,0(R1)       ;Store Conditional
      BEQZ R2,TRY         ;存失败转移
```

这些指令的实现必须跟踪地址。通常 LL 指令关联着一个寄存器,该寄存器存放需要跟踪的地址,这个寄存器常称为连接寄存器(link register)。如果发生中断切换或与连接寄存器中值匹配的 Cache 块被作废(比如被别的 SC 指令访问),连接寄存器则清零,SC 指令检查它的存地址与连接寄存器内容是否匹配,如匹配则 SC 继续执行,否则执行失败。既然别的处理器对连接寄存器所指单元的写或任何异常指令都会导致 SC 失败,应该特别注意在两条指令间插入其他指令的选择。一般情况下,只有寄存器-寄存器指令才能安全地通过,否则极有可能产生死锁从而处理器永远不能完成 SC。此外,LL 和 SC 之间的指令数应尽量少,从而减少由无关事件或竞争的处理导致 SC 执行失败。

2. 使用一致性实现锁

有了原子操作,我们可以采用多处理机的一致性机制来实现旋转锁(spin locks)。旋转锁是指处理器环绕一个锁不停地旋转而请求获得该锁。当锁的占用时间很少以及加锁过程延迟很低时可采用旋转锁。因为旋转锁要求处理器必须在循环中等待获得锁,因此在有些条件下不适用。

在无 Cache 一致性机制的条件下,可采用的最简单的实现方法是在存储器中保存锁变量,处理器可以不断地通过一个原子操作请求加锁,比如先交换,再测试返回值从而知道锁的状况。释放锁的时候,处理器可简单地将锁置为“0”,下面是一个旋转锁的代码段,R1 中的地址对应为用来进行原子交换的锁。

```

LI      R2, #1
LOCKIT: EXCH  R2,0(R1)      ;原子交换
          BNEZ R2, LOCKIT    ;是否已加锁

```

如果机器支持 Cache 一致性,则可以将锁缓冲进入 Cache,并通过一致性机制使锁值保持一致。这有两个好处:第一,可使“环绕”的进程(不停测试请求锁的循环)对本地 Cache 块进行操作,而不用每次请求锁时必须先进行一次全局的存储器访问;第二个好处是可利用锁访问的局部性,即处理器最近使用过的锁不久又会使用,这种状况下锁可驻留在那个处理器的 Cache 中,大大降低了请求的时间。

要获得第一条好处,需对上面简单的环绕过程进行一点改动,上面循环中每次交换均需一次写操作,如果有多个处理器都请求加锁,则大多数写会导致写失效。因而可改进旋转锁过程,使其环绕过程仅对本地 Cache 中锁的拷贝进行读,直到它返回“0”确认锁可用,然后再进行加锁交换操作,将“1”存入锁变量。获得锁的处理器执行自己的代码完毕后,将锁变量置“0”释放锁,从而竞争又开始进行。下面是这种旋转锁的代码(“0”为开,“1”为关)。

```

LOCKIT: LW      R2,0(R1)      ;取锁值
          BNEZ R2,LOCKIT    ;锁不可用
          LI      R2, #1        ;存入锁值
          EXCH  R2,0(R1)      ;交换
          BNEZ R2,LOCKIT    ;如果锁不为0 转移

```

让我们看一下这种旋转锁是怎样使用 Cache 一致性机制的。表 7.5 给出了对于三个处理器竞争锁情况下的操作。一旦一个处理器使用锁完毕并存入“0”释放该锁时,所有别的 Cache 对应块均被作废,必须取新的值来更新它们锁的拷贝。其中一个处理器 Cache 会先获得未加锁值并执行交换操作,当别的 Cache 失效处理完后,它们会发现已被加锁,所以又必须不停地环绕测试。

表 7.5 三个处理器竞争锁的操作

步骤	处理器 P0	处理器 P1	处理器 P2	锁状态	总线/目录操作
1	占有锁	环绕测试 Lock = 0	环绕测试 Lock = 0	Shared	无
2	将锁置为 0 (收到作废命令)	(收到作废命令)	Exclusive	P0 发锁变量作废消息	
3		Cache 失效	Cache 失效	Shared	总线/目录处理 P2 Cache 失效, 锁从 P0 写回
4		总线/目录忙则等待	Lock = 0	Shared	P2 Cache 失效处理
5		Lock = 0	执行交换, 导致 Cache 失效	Shared	P1 Cache 失效处理
6		执行交换, 导致 Cache 失效	交换完毕, 返回 0 置 Lock = 1	Exclusive	总线/目录处理 P2 Cache 失效, 作废消息
7		交换完毕, 返回 1	进入关键处理段	Shared	总线/目录处理 P2 Cache 失效, 写回
8		环绕测试 Lock = 0			无

这里假设是采用写作废机制, P0 开始占据锁(第一步); P0 退出后释放锁(第二步); P1 和 P2 竞争锁(第三至五步); P2 赢, 进入关键处理段(第六、七步); P1 失败后开始环绕等待(第七、八步)。在实际系统中, 这些事件耗费的时间远大于八个时钟周期, 因为总线请求及失效处理时间会长得多。

这个例子也说明了 LL/SC 原语的另一个状态: 读写操作明显分开。LL 不产生总线数据传送, 这使下面代码与使用经过优化交换的代码具有相同的特点(R1 中保存锁的地址):

```

LOCKIT:   LL      R2,0(R1)      ; Load Linked
          BNEZ    R2,LOCKIT   ; 锁无效
          LI      R2,,±1       ; 加锁值
          SC      R2,0(R1)      ; 存
          BEQZ    R2,LOCKIT   ; 如果存失败则转移

```

第一个分支形成环绕的循环体, 第二个分支解决了两个同时请求锁的处理器竞争问题。尽管旋转锁机制简单并且具有强制性, 但难以将它扩展到处理器数量很多的情况, 因为竞争锁时会产生大量的通信问题。

3. 同步的性能

上面讲的简单旋转锁不能很好地适应可缩放性。大规模机器中所有的处理

器竞争同一个锁，目录或总线对所有处理器的请求服务是串行的，从而会产出大量的竞争问题。下面的例子可以看出情况有多么严重。

例 7.3 设总线上有 20 个处理器同时准备对同一变量加锁。假设每个总线事务处理(读失效或写失效)是 50 个时钟周期，忽略实际的 Cache 块锁的读写时间以及加锁的时间，求 20 个处理器请求加锁所需的总线事务数目。设时间为 0 时锁已释放并且所有处理器在旋转，求处理这 20 个请求时间为多长？假设总线在新的请求到达之前已服务完挂起的所有请求，并且处理器速度相同。

解 表 7.6 给出了从一次释放锁到下一次释放锁之间发生的事件。当然，每次获得锁后竞争的处理器数目也会减 1，从而使平均开销降为 1 525 个周期。20 个处理器通过锁的总时间会超过 30 000 多个周期。整体上看处理器平均有一半时间处于闲置状态，只是简单地在请求获得锁，所用的总线事务数超过了 400(基于第二、第三和第四项： $(40+1)/2 \times 20 = 410$)。

表 7.6 20 个处理器竞争同一锁时，从请求到释放的时间

事件	时间
所有等待的处理器取锁产生的读失效(20×50)	1 000
释放锁的处理器导致的写失效及作废	50
所有等待的处理器的读失效(20×50)	1 000
所有等待的处理器写失效，一个成功获得锁(50)，并作废其余的锁拷贝(19×50)	1 000
一个处理器获得和释放锁的总时间	3 050 个时钟周期

上表中假设每个总线事务处理时间为 50 个时钟周期，因为总线的公平性，第一个处理器释放锁后，其余 19 个处理器来竞争取锁。本例中问题的根源是锁的竞争、存储器中锁访问的串行性以及总线访问的延迟，总线的公平性使得这些情况更为突出。旋转锁的主要优点是对于总线或网络开销较低，同样的处理器反复使用锁时性能较好。但这两点在上述例子中均没有得到体现，下面将讨论对旋转锁实现上的改进。

4. 栅栏(barrier)同步

并行循环程序中另一个常用的同步操作是栅栏。栅栏强制所有到达该栅栏的进程进行等待，直到全部的进程到达栅栏，然后释放全部的进程，从而形成同步。栅栏的典型实现是要用两个旋转锁：一个用来记录到达栅栏的进程数，另一个用来封锁进程直至最后一个进程到达栅栏。栅栏的实现中要不停地探测指定的变量，直到它满足规定的条件。下面的程序是一种典型的实现，其中 lock 和 unlock 提供基本的旋转锁，total 规定了要到达栅栏的进程总数。

```

lock(counterlock);           /* 确保更新的原子性 */
if(count == 0) release = 0;   /* 第一个进程则重置 release */
count = count + 1;           /* 到达进程计数 */
unlock(counterlock);         /* 释放锁 */
if(count == total){          /* 进程全部到达 */
    count = 0;                /* 重置计数器 */
    release = 1;               /* 释放进程 */
}
else {                       /* 还有进程未到达 */
    spin(release = 1);        /* 等待别的进程到达 */
}

```

对 counterlock 加锁保证增量操作的原子性, 变量 count 记录着已到达栅栏的进程数, 变量 release 用来封锁进程直到最后一个进程到达栅栏。函数 spin(release=1)使进程等待直到全部的进程到达栅栏。

实际情况中, 另外一个问题使栅栏的实现稍微复杂了一些: 通常反复使用一个栅栏, 栅栏释放的进程运行一段后又会再次返回该栅栏, 这样有可能出现某个进程永远离不开栅栏的状况(它停在旋转操作上)。例如 OS 调度进程, 可能一个进程速度较快, 当它第二次到达栅栏时, 第一次的进程还挂在栅栏中未能离开。这样所有的进程在这个栅栏的第二次使用中都处于无限等待状态, 因为进程的数目总是达不到 total。一种解决方法是当进程离开栅栏时进行计数(和人口一样), 在上次栅栏使用中的所有进程离开之前不允许任何进程重用并初始化本栅栏。另一种解决办法是 sense-reversing 栅栏, 每个进程均使用一个私有变量 local-sense, 该变量初始化为 1。下面的程序给出了 sense-reversing 栅栏的代码, 这种方法使用安全。但是正如下面例子所示, 其性能仍然很差。

```

local-sense = ! local-sense;      /* local-sense 取反 */
lock(counterlock);              /* 确保更新的原子性 */
count++;                         /* 计算到达进程数 */
unlock(counterlock);             /* 解锁 */
if(count == total){              /* 进程全部到达 */
    count = 0;                   /* 重置计数器 */
    release = local-sense;        /* 释放进程 */
}
else {                           /* 还有进程未到达 */
    spin(release = local-sense); /* 等待信号 */
}

```

例 7.4 假设总线上 20 个处理器同时执行一个栅栏, 设每个总线事务需 50 个时钟周期, 忽略 Cache 块中锁的读、写实际花费的时间以及栅栏实现中非同步操作的时间, 计算 20 个处理器全部到达栅栏、被释放及离开栅栏所需的总线事务数。设总线完全公平, 整个过程需多长时间?

解 下面表 7.7 给出一个处理器通过栅栏产生的事件序列, 设第一个获得总线的进程还没有获得锁。

表 7.7 一个处理器通过栅栏产生的事件序列

事 件	每个处理器时间	20 个处理器时间
每个处理器获得锁、增量、释放锁的时间	1 525	30 500
执行释放的时间	50	50
每个处理器获得释放标志的时间	50	1 000
总和	1 625	31 550

栅栏操作比前面讲的 20 个处理器加锁解锁操作所需时间稍长, 所需总的总线事务数约为 440。

由前面这些例子可见, 当多进程之间竞争激烈时, 同步会成为瓶颈。当竞争不激烈且同步操作较少时, 我们主要关心的是一个同步原语操作的延迟, 即单个进程要花多长时间才能完成一个同步操作。基本的旋转锁操作可以在两个总线周期内完成: 一个读锁, 一个写锁。可以采用多种方法进行改进, 使它在单个周期内完成操作。例如可简单地进行检测交换操作。如果锁已被占用, 将会导致大量的总线事务, 因为每一个想取得锁的进程均需要一个总线周期。实际上, 旋转锁的延迟并不像前面例子中所示的那样糟糕, 因为在实现中可以对 Cache 的写失效加以优化。

同步操作最严重的问题是进程的串行性。当出现竞争时, 就会出现串行性问题。它极大地增加了同步操作的时间。比如, 在无竞争的条件下, 20 个处理器加解锁的同步操作仅需 2 000 个周期(40 个总线事务)。总线的使用是这个问题的关键所在。在基于目录的 Cache 一致性机器中, 串行性问题也同样严重, 它的延迟时间更长, 下面给出在竞争程度高或处理器个数较多的情况下一些有效的解决办法。

7.4.2 大规模机器的同步

我们所希望的同步机制是在无竞争的条件下延迟较小, 在竞争激烈时串行性小。下面来看在竞争较大时怎样通过软件实现来提高锁和栅栏的性能, 并探讨两种基本的硬件原语, 用以在保持低延迟的条件下尽可能减少串行度。

1. 软件实现

旋转锁实现的主要问题是当多个进程检测并竞争锁时引起的延迟。一种解决办法是当加锁失败时就人为地推延这些进程的等待时间。这可通过当 SC 操作失败时就推迟进程再次请求加锁的时间来实现。一般是在失败时，延迟的时间指数增大。下面的程序给出了具有指数延迟(exponential back-off)的旋转锁代码。

```

        LI      R3,1           ;R3=初始延迟值;
LOCKIT: LL      R2,0(R1)     ;Load Linked
        BNEZ   R2,LOCKIT    ;无效
        ADDI   R2,R2,1       ;取到加锁值
        SC      R2,0(R1)     ;Store Conditional
        BNEZ   R2,GOTIT    ;存成功转移
        SLL    R3,R3,1       ;将延迟时间增至 2 倍(左移 1 位)
        PAUSE  R3           ;延迟 R3 中时间值
        J      LOCKIT

GOTIT: 使用加锁保护的数据

```

当 SC 失败时，进程推延 R3 个时间单位。R3 的值是由具体机器而定的，它的开始值约为执行关键程序段并释放锁的时间。PAUSE R3 指令的功能是等待 R3 个时间单位。R3 的值在每次存失败后以 2 为因子增长，从而使进程所等待的时间是前一次的 2 倍。

实现锁的另一种技术是排队锁，稍后将讨论用硬件来实现排队锁，这里先讨论采用数组进行的软件实现。为此我们给出一种更好的栅栏实现代码。前面栅栏机制实现中，所有的进程必须读取 release 标志而形成冲突。我们可通过组合树(combining tree)来减少冲突。组合树是多个请求在局部结合起来形成树的一种分级结构，局部组合的分枝数量大大小于总的分枝数量，因此组合树降低冲突的原因是将大冲突化解成为并行的多个小冲突。

组合树采用预定义的 n 叉树结构。这里用变量 k 表示扇入数目，实际中 $k = 4$ 效果较好。当 k 个进程都到达树的某个结点时，则发信号进入树的上一层。当全部进程到达的信号汇集在根结点时，释放所有的进程。如同前面一样，我们采用 sense-reversing 技术来给出下面的基于组合树的栅栏代码。

```

struct node {                      /* 组合树中一个结点 */
    int counterlock;               /* 本结点锁 */
    int count;                     /* 计数本结点 */
    int parent;                    /* 树中父结点=0..p-1,根结点=-1 */

```

```

    ;
struct node tree[0..p-1];      /* 树中各结点 */
int local-sense;              /* 每个处理器的私有变量 */
int release;                  /* 全局释放标志 */

/* 栅栏实现函数 */
barrier(int mynode) {
    lock(tree[mynode].counterlock); /* 保护计数器 */
    count++;                      /* 计数增量 */
    unlock(tree[mynode].counterlock); /* 解锁 */
    if(tree[mynode].count == k){   /* 本结点进程全部到达 */
        if(tree[mynode].parent) >= 0{
            barrier(tree[mynode].parent);
        }else{
            release = local-sense;
        }
        tree[mynode].count = 0;      /* 为下次初始化 */
    }else{
        spin(release = local-sense); /* 等待 */
    }
}
;

/* 加入栅栏的进程执行代码 */
local-sense = !local-sense;
barrier(mynode);

```

树是根据 tree 数组中的结点预先静态建立的,树中每个结点组合 k 个进程,提供一个单独的计数器和锁,因而在每个结点有 k 个进程进行竞争。当第 k 个进程都到达树中对应结点时则进入父结点,然后递增父结点的计数器,当父结点计数到达 k 时,置 release 标志。每个结点的计数器在最后一个进程到达时被初始化。

以上讨论的是通过软件来实现的栅栏。一些大规模并行处理系统(如 CRAY T3D、CM-5)已引入了栅栏的硬件支持。

2. 硬件原语支持

这里介绍两种硬件同步原语,第一种针对锁,第二种针对栅栏和要求进行计数或提供明确索引的某些用户级操作。这两种硬件同步原语产生的延迟与前面所讨论的情况基本上相同,但可以大大减少串行性。

上面给出的锁的实现中,最主要的问题是它带来了大量无用的竞争。比如,

当锁被释放时,尽管只有一个进程能成功地获得其状态值,但所有的进程都会产生读失效和写失效。我们可以排队记录等待的进程,当锁释放时送出一个已确定的等待进程,这种机制称为排队锁(queuing lock)。排队锁可用硬件实现,也可用记录等待进程的排队数组软件实现。硬件实现一般是在基于目录的机器上,通过硬件向量等方式来进行排队和同步控制。在基于总线的机器中要将锁从一个进程显式地传给另一个进程,软件实现会更好一些。

排队锁的工作过程如下:在第一次取锁变量失效时,失效被送入同步控制器。同步控制器可集成在存储控制器中(基于总线的系统)或目录控制器中。如果锁空闲,将其交给该处理器;如果锁忙,控制器就产生一个结点请求记录(比如可以是向量中某一位),并将锁忙的标志返回给处理器,然后该处理器进入旋转等待。当该锁被释放时,控制器从等待的进程排队中选出一个使用该锁。这可以通过更新所选进程 Cache 中的锁变量或者作废锁的拷贝来完成。

例 7.5 如果在排队锁的使用中,失效时进行锁更新,求 20 个处理器完成 lock 和 unlock 所需的时间和总线事务数。假设条件与前面例子相同。

解 每个处理器初始加锁及随后释放锁各产生 1 次失效,所以总共需 40 个总线周期。20 个初始失效需 1 000 个时钟周期,接着是每次需要 50 个周期的 20 次释放,总和为 2 050 个周期。与通常的基于 Cache 一致性的旋转锁相比,性能大大提高。

排队锁功能实现中有一些要考虑的关键问题。首先,需要识别出对锁进行初次访问的进程,从而对其进行排队操作。其次,等待进程队列可通过多种机制实现,在基于目录的机器中,队列为共享集合,需用类似目录向量的硬件来实现排队锁的操作。最后,必须有硬件来回收锁,因为请求加锁的进程可能在切换时被切出,并且在同一处理器上有可能不再被调度切入。

排队锁可以用来提高栅栏操作的性能。此外,还可以引进一种原语来减少栅栏记数时所需的时间,从而减小串行形成的瓶颈,其性能可与排队锁相比。这种原语还可以用来构造其他的同步操作。Fetch-and-increment 就是这样一种原语,它可以原子地取值并增量,返回的值可以为增量后的值,也可以为取出的值。使用 fetch-and-increment 可以很好地改进栅栏的实现。

例 7.6 写出采用 fetch-and-increment 栅栏的代码。条件与前面假设相同,并设一次 fetch-and-increment 操作也需 50 个时钟周期。计算 20 个处理器通过栅栏的时间,及所需的总线周期数。

解 下面的程序段给出栅栏的代码,这种实现需要进行 20 次 fetch-and-increment 操作,释放时有 20 次 Cache 失效,总共需时间 2 000 个时钟周期,40 个总线事务操作。与前面实现的栅栏机制相比,前面所需时间长达 15 倍,总线操作多达 10 倍。当然,实现组合树栅栏时也可采用 fetch-and-increment 来降低树

中每个结点的串行竞争。

```

local-sense = ! local-sense;           /* local-sense 变反 */
fetch-and-increment(count);          /* 原子性更新 */
if(count == total){                  /* 进程全部到达 */
    count = 0;                      /* 初始化计数器 */
    release = local-sense;          /* 释放进程 */
}
else {
    spin(release = local-sense);    /* 还有进程未到达 */
}

```

我们已经看到,由同步、存储延迟和负载不平衡将会产生许多问题。由此可以理解要有效地利用大规模并行机器所面临的困难有多么大。

程序同步意味着对共享数据的访问被同步操作有序化。实际中我们希望大多数程序是同步的。这是因为如果访问不同步,就很难决定程序的行为。程序员可通过构造自己的同步机制来保证有序性,但这需要很大的技巧性,并可能在体系结构上得不到支持,从而不能保证它们在大规模并行的机器上高效运行。因此,几乎所有的程序员都选择使用同步库。这不但保证了正确性,而且保证了同步的优化。

7.5 并行化技术

本节将专门讨论有关并行程序设计的基本模型以及有关语言和编译问题。要研究的模型包括共享变量、消息传递、面向对象、数据并行、函数和逻辑程序设计等。

7.5.1 并行化的基本策略

程序设计模型是一种程序抽象的集合,给程序员提供了一幅计算机硬件/软件系统透明的简图。并行程序设计模型是专门为多处理机、多计算机或向量/SIMD 计算机而设计的。下面介绍的就是为这些计算机设计的以不同执行规范来开发并行性的五种模型。

1. 共享变量模型

多处理机程序设计的基础是利用共享变量来实现进程间的通信。这需要使用共享存储器以及访问同一组共享变量的多个进程间的同步特性。

在紧耦合的多处理机中要开发细粒度 MIMD 并行性。进程间的同步可以

无条件也可以有条件地实现,这与所采用的机制有关。

使用这种模型所遇到的主要问题包括临界区的保护性访问、存储器的一致性、存储操作的可分性、快速同步、共享的数据结构以及快速数据迁移技术等。

在保护数据一致性安全和保持事件的顺序方面,其实现复杂程度从单道程序设计到多任务处理、多道程序设计、多重处理以及多线程处理依次递增。必须开发存储器管理和特殊的保护机制,以确保程序并行运行的正确性及数据的完整性。

2. 消息传递模型

驻留在不同处理机结点上的两个进程可以通过网络传递消息的方式来相互通信。消息可以是指令、数据、同步信号或中断信号等。消息传递所产生的通信延迟比访问共享存储器的延迟要长得多,消息传递程序设计模型分为同步消息传递和异步消息传递。

同步消息传递在时间和空间上必须对发送进程和接收进程实现同步,就像使用线路交换技术通电话一样。由于通道一次只允许传送一个消息,因而同步通信常因通道忙或出错而被阻塞。

遵循同步设计规范的消息传递必须对发送进程和接收进程进行同步。发送者和接收者除在时间上有联系外,在空间上也必须用物理通信通道连接起来,也就是必须为通道准备好递路使它们之间能传递消息。

对于异步消息传递,通道中常用缓冲区。倘若缓冲区足够大或网络通信量不饱和,则消息传递就不会阻塞。不管接收者是否准备好,都允许发送者发送消息而不被阻塞。

异步通信要求在连接信道的通路上使用缓冲区来保存消息。由于通道缓冲区有限,因此发送者有可能会被阻塞。在同步多计算机中并不需要缓冲区,因为通道一次只允许一个消息通过。

用这种模型进行程序设计的关键问题是如何把程序代码和数据分布或复制到所有处理结点上。此外还必须考虑计算时间和通信开销之间的权衡问题。

3. 数据并行模型

数据并行程序要求使用预先分布好的数据集。因此,并行数据结构的选择会使数据并行程序设计差别很大。互连的数据结构需要进行数据交换操作。总之,数据并行程序设计强调的是局部计算和数据路由操作(如置换、复制、归约等)。它比较适合于使用规则网络、模板和多维信号/图像数据集来求解细粒度的应用问题。

数据并行性取决于粒度大小以及所采用的操作方式。数据并行通常研究的是有几千个并发数据操作的高度并行性问题。这与只在指令级提供低度并行控制的并行性完全不同。

数据并行操作的同步是在编译时而不是在运行时完成的。

4. 面向对象模型

在这种模型中,对象是动态建立和控制的。处理是通过在对象间发送和接收消息来完成的。并发程序设计模型是将低级对象(如进程、队列和信号灯)组合起来形成高级对象(如管程和程序模块)。

对象是把数据和操作封装在一个计算单元中的程序实体。已经证明并发是对象概念的一个很自然的结果。事实上,传统程序设计中的并发协同程序与对象程序中对象的并发操作很类似。

并发面向对象程序设计的研究为多处理机或多计算机的并发计算提出了一种新模型。不同的对象模型在描述对象行为和交互方式等方面都是不一样的。

5. 函数和逻辑模型

函数程序设计模型强调程序的函数性,程序在执行后不应该产生副作用。在函数程序中没有存储、赋值以及转移的概念。换句话说,在函数表达式求值之前任何计算的历史都不应与表达式有任何关系。

没有副作用可为开发并行性提供更多的可能性。函数计算应产生相同的值而与它的参数计算次序无关。这意味着在一个函数程序动态创建的结构中所有参数都可以并行计算。所有的单赋值和数据流语言自然都是函数语言。这说明函数程序设计模型可以很容易地应用于数据驱动多处理机。函数模型强调的是开发细粒度 MIMD 并行性。例如并行 LISP 语言即属于该类型。

逻辑程序设计模型以谓词逻辑为基础,适用于涉及大型数据库的知识处理。这种模型采用隐式搜索策略并支持逻辑推理过程中的并行性。如果在数据库中找到了要匹配的事实,那么问题就得到了解答。如果两个事实的谓词和有关的参数都相同,则说明这两个事实匹配。在一定的条件下,匹配和合一的进程可以并行化。

Shapiro 于 1986 年研制的 Concurrent Prolog 和 Clark 于 1987 年提出的 Parlog 是两种并行逻辑程序设计语言。

7.5.2 并行语言与编译器

并行计算机比串行计算机更加需要语言环境的支持。推动并行计算机发展的软件仍处于早期的开发阶段,用户还不能使用高层抽象来致力于程序并行性的开发,他们仍不得不花费很多时间在了解硬件细节之后再去进行程序设计。

要打破硬件/软件的壁垒,需要一个并行软件开发环境。它要能为用户开发并行性和调试程序提供良好的工具。最近开发的大多数软件工具仍处于研究和测试阶段,很少已成为商品并提供市场。

1. 并行性的语言特征

Chang 和 Smith 在 1990 年把并行程序设计的语言特性根据功能分成六类。这些特性对一般的应用来说可能过于理想。实际上,有些实用语言可能只有一部分或者根本就没有这些特性。某些特性用现有的语言/编译器开发工具是可以识别出来的。下面列举的特性可作为开发用户友好程序设计环境的设计要求。

(1) 优化特性 这些特性在把顺序程序转换成并行形式时可用于程序的重构和编译,从而使目标机中的软件并行性与硬件并行性相匹配。最好的是自动并行化程序,但目前还无法做到。现在可做的是需要编译命令或程序员交互作用的半自动并行化程序,以及通过静态分析程序、运行时统计、数据流图和重构 FORTRAN 代码的代码翻译程序等进行的交互式重构支持。

(2) 可用性 这些特性能增进用户友善性,使语言可移植到很多类型的并行计算机上并扩充了软件库的适用范围。它包括可扩展性、兼容性和可移植性。

(3) 同步/通信特性 这一语言特性主要包括了共享变量(锁)的支持、消息传递的发送/接收、远程过程调用、栅栏同步机制和邮箱等。

(4) 并行性控制 包括以各种形式确定并行性的控制结构。它们主要处理:各种粒度的并行性、显式并行性与隐式并行性、整个程序中的全局并行性、迭代中的循环并行性、任务分割并行性、共享任务队列以及共享抽象数据类型等。

(5) 数据并行性 数据并行性用来说明访问数据以及把数据分布在多处理计算机上的方法。主要包括:无需用户干预的运行时数据自动分布;为用户提供一种能指定通信模式或说明数据和进程映射到硬件上的方法;编译器将虚处理机动态或静态地映射到物理处理机上的虚拟处理机支持;共享数据可被直接访问而不用编程控制;SPMD(Single Program Multiple Data)程序设计支持。

(6) 进程管理特性 这些特性用来支持并行进程的高效创建、多线程处理和多任务处理的实现、程序划分和复制以及运行时的动态负载平衡。

倘若没有编译器的支持、操作系统的协助以及同现有环境的集成,上述语言特性是无法实现的。用传统语言写成的软件资源是建成高效并行程序设计环境的基础。

优化特性强调的是编译时代码的并行化和向量化。可用性拓宽了语言的应用领域并使它与机器无关。

实现同步特性必须由有效的硬件和软件机制给予支持。控制特性常常要在粒度大小、存储需求、通信和调度开销之间进行权衡。进程管理特性与所提供的 OS 功能有很密切的关系。因此,语言、编译器和 OS 必须用集成的方式联合开发。

2. 并行语言结构

下面将介绍特定的语言结构和数组表达式用于开发程序的并行性。首先说明 FORTRAN 90 的数组表示符,然后描述程序流控制的公用并行结构。

(1) FORTRAN 90 数组表示符 多维数组用一个带下标的三元组序列的

数组名来表示,每维一个三元组,不同维的三元组用逗号隔开,例如:

```
e1:e2:e3  
e1:e2  
e1:* :e3  
e1 : *  
e1  
*
```

这里每个 e_i 一定是能产生一个标量整数值的算术表达式。第一个表达式 e_1 是下界,第二个表达式 e_2 是上界,第三个 e_3 是增量(步距)。例如, $B(1:4:3,6:8:2,3)$ 表示一个三维数组的四个元素 $B(1,6,3), B(4,6,3), B(1,8,3), B(4,8,3)$ 。

当三元组中的第三个表达式省去时,就认为步距是 1。在第二个表达式中的表示符 * 表明该维中所有的元素都从 e_1 开始,如果 e_1 也省去,则表示整个维。当 e_2 和 e_3 都省去时, e_1 仅表示该维中的一个元素,例如, $A(5)$ 代表数组 $A(3:7:2)$ 中的第五个元素。这种表示符可以用来选择数组段或特定的数组元素。

在下述约束条件下可对数组赋值:右边的数组表达式必须和左边的数组具有相同的形式和相同数量的元素。例如,赋值 $A(2:4,5:8)=A(3:5,1:4)$ 是有效的,而赋值 $A(1:4,1:3)=A(1:2,1:6)$ 是无效的,虽然它们两边都有 12 个元素。将一个标量赋给数组,就是将这个标量的值赋给数组中的每个元素。例如,语句 $B(3:4,5)=0$ 即是将 $B(3,5)$ 和 $B(4,5)$ 都设置为 0。

(2) 并行流控制 传统 FORTRAN 的 Do 循环说明(Do,Enddo)语句对中的所有标量指令都顺序执行,因而是连续的迭代。为了说明并行活动,我们使用(Doall,Endall)语句对。倘若 Doall 循环中的所有迭代彼此都互不依赖,而且又有足够的处理机来处理不同的迭代,则它们可以并行执行。但是,每次迭代内部的计算仍按程序中的次序顺序执行。

当一个循环嵌套的连续迭代彼此相互依赖时,可以用(Doacross,Endacross)语句对来说明循环体间相关并行性。彼此相互依赖的迭代之间必须实行同步。例如,在下面的程序中,J 维方向上存在一致性。我们用 Doacross 来说明 I 维方向上的并行性,但是要求迭代之间同步。(Forall,Endall)和(Pardo,Parend)命令可以解释为 Doall 循环或 Doacross 循环。

```
Doacross I=2,N  
  Do J=2,N  
    SI:A(I,J)=(A(I(J-1))+A(I,J+1))/2  
  Enddo  
Endacross
```

另一种程序结构是(Cobegin,Coend)语句对。所有在程序块内指定的计算都能并行执行。但是并行进程的创建在实际执行中会有一点时间差别。这和Doall 循环或 Doacross 循环结构的语义完全不同，在这些语句对中创建的并发进程间的同步是隐含的。形式上，命令

Cobegin

P₁

P₂

P_n

Coend

使进程 P₁,P₂,...,P_n 同时开始并且并发地进行处理，直到它们全部结束。命令(Parbegin,Parend)的意义与此相同。

最后，在下面的例子中介绍一下 Fork 和 Join 命令。在进程 P 的执行过程中，可以使用 Fork Q 命令来产生一个新的进程 Q：

Process P

Process Q

Fork Q

Join Q

End

Join Q 命令把两个进程重新组合成一个进程。当执行 P 中的 Fork Q 语句时，进程 Q 即被启动。然后程序 P 和 Q 并发执行，直到 P 执行 Join Q 语句或者 Q 结束为止，在它们重新联接之前，无论哪一个先结束都必须等待另一个执行完成。

在 UNIX 环境中，Fork-Join 语句提供了一种动态创建进程并对其多次激活的机制。而 Cobegin-Coend 语句提供的是一个结构化的单入、单出控制命令，它和动态的 Fork-Join 不一样。(Parbegin,Parend)命令相当于(Cobegin,Coend)命令。

3. 并行优化编译器

由于源代码用高级语言编写，因此在现代计算机中，编译器已经成为一个必不可少的软件。编译器的任务是为程序员解除程序优化和代码生成的负担。并行化编译器则由下述三个主要部分组成：流分析、优化和代码生成，如图 7.17 所示。

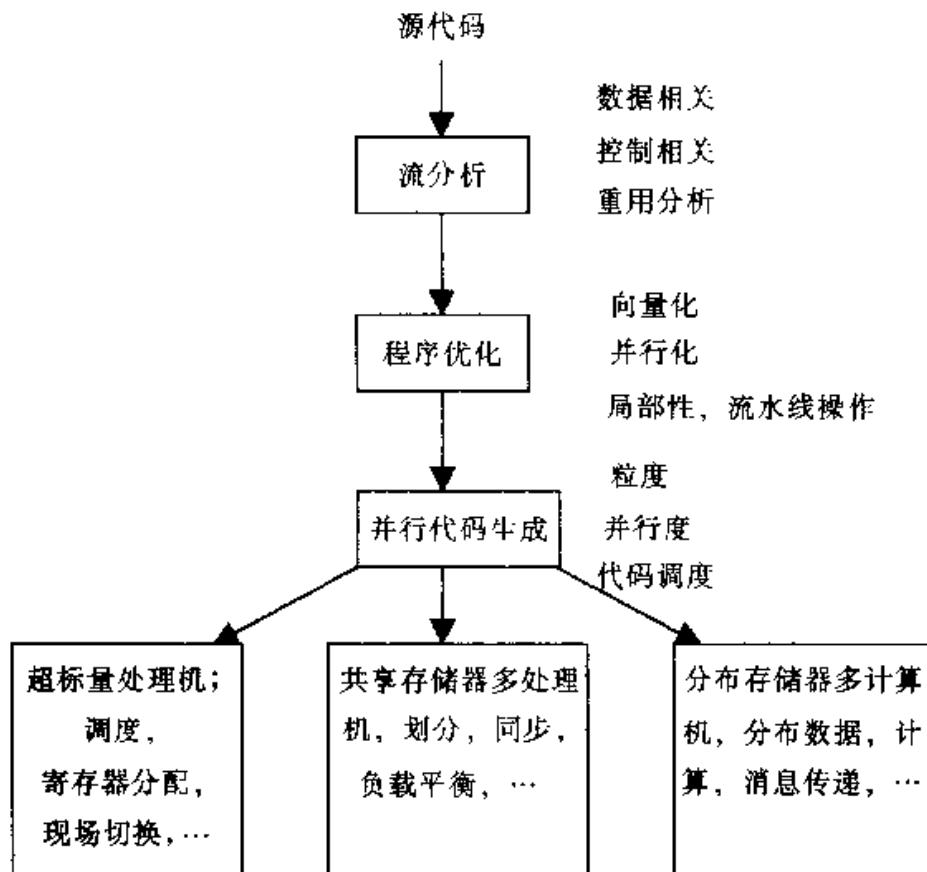


图 7.17 并行代码生成的编译过程

(1) 流分析 为确定源代码中的数据和控制一致性,这部分要将程序流的模式分析出来。前面已经讨论过了标量型指令间的数据依赖关系。下面要将标量一致性分析进一步推广,以构造数组或矩阵。并行性粒度的开发是很不一样的,与机器的结构有关。因此,在不同的并行计算机上,流分析将在不同的执行层次上进行。

一般说来,在超标量或 VLSI 处理机上并发的是指令级并行性;在 SIMD 机或向量机上开发的是循环级并行性;在多处理机、多计算机或工作站网络上开发的是任务级并行性。当然也有例外。例如,细粒度并行性正在下移让下一代带全局共享地址空间的多计算机来开发。流分析还必须指明某些代码/数据的重用以及存储器的访问模式。

(2) 程序优化 这关系到用户程序的转换,目的是尽可能多地挖掘硬件的潜能。转换可以在循环级、局部级或预取级进行,最终目标是要达到全局优化。优化常常是同一种语言将代码转换成与之等价而高效的形式。这些转换应与机器无关。

实际上,大多数转换受机器的结构限制。这就是很多编译器依赖于机器的主要原因。至少,我们希望设计的编译器只要做少量修改就能在大多数机器上运行。因此,要在全局优化之前进行一定的转换。这就需要源到源的优化(有时

称为预编译器),实现程序从一种高级语言到另一种高级语言的转换,在这之后再去利用目标机上第二种语言的编译器。

程序优化的最终目标是使代码执行速度达到最高。这就要求代码的长度最短、存储器的访问次数最少以及程序并行性得到较好的开发。优化技术包括用流水线硬件完成向量化以及同时用多台处理机实现并行化。所设计成的编译器应当能使其减少运行时间而用的资源却最小。其他优化还要求扩充例程或者过程并集成在一起。在大多数程序中局部优化和全局优化都是需要的。有时优化应当在算法级进行,有时还需要程序员参与。

依赖于机器的转换是希望机器资源如存储器、寄存器和功能部件的分配更有效,也常常用简单操作来替换复杂操作。其他优化措施还包括取消不必要的转换或公用表达式,也可以用指令调度来消去执行连续指令中流水操作或存储器的延迟。

(3) 并行代码生成 代码生成通常涉及从一种描述到另一种称之为中间形式描述的转换。一定要选择好一种代码模型作为中间形式。因为要包含有并行结构,所以就更加需要并行代码。代码生成与所用的指令调度策略是紧密联系在一起的。此外还要常常对控制流命令连接的基础程序块进行优化以获得较高的并行性,而程序块也需要用特殊的数据结构来表示。

不同类型计算机的并行代码生成也是很不一样的。例如,超标量处理机可能用软件调度实现也可能用硬件调度实现。在 RISC 或超标量处理机上如何优化寄存器分配,在多处理机上划分代码时如何降低开销,当代码/数据在多计算机上分布(或复制)时如何实现消息传递命令,这些问题给并行代码生成增添了不少困难。当代码自动生成不易实现时,还可以使用编译命令来帮助生成并行代码。

近十年来,已研制成两种研究性的优化编译器:一种是伊利诺依大学的 Parafrase,另一种是赖斯大学的 PFC(Parallel FORTRAN Converter)。

7.6 多处理机实例

本节介绍两个多处理机实例 Challenge 和 Origin 2000,前者为集中共享存储器结构,后者为分布共享存储器结构。

7.6.1 Challenge 多处理机系统

SGI(Silicon Graphics Inc.)多处理机系统 Challenge 具有共享主存基于总线的 Cache 一致性结构,高速总线上可容纳最多 36 个 MIPS R4400 处理器。系统中每块板上有 4 个处理器,存储器为 8 路交叉访问,总容量 16 GB。该系统可升档为 R8000 处理器,从而形成 Power Challenge 系统,这时每块主板上有 2 个处

理器,整个系统最多可有 18 个处理器。由于我们主要关心总线、一致性硬件和同步支持,所以下面主要讨论基于 150 MHz 的 R4400 系统。

Challenge 设计的核心是 POWERpath-2 总线,它有 256 位数据宽度,40 位地址,频率为 50 MHz,该总线连接了系统的所有主要部件。系统结构如图 7.18 所示。

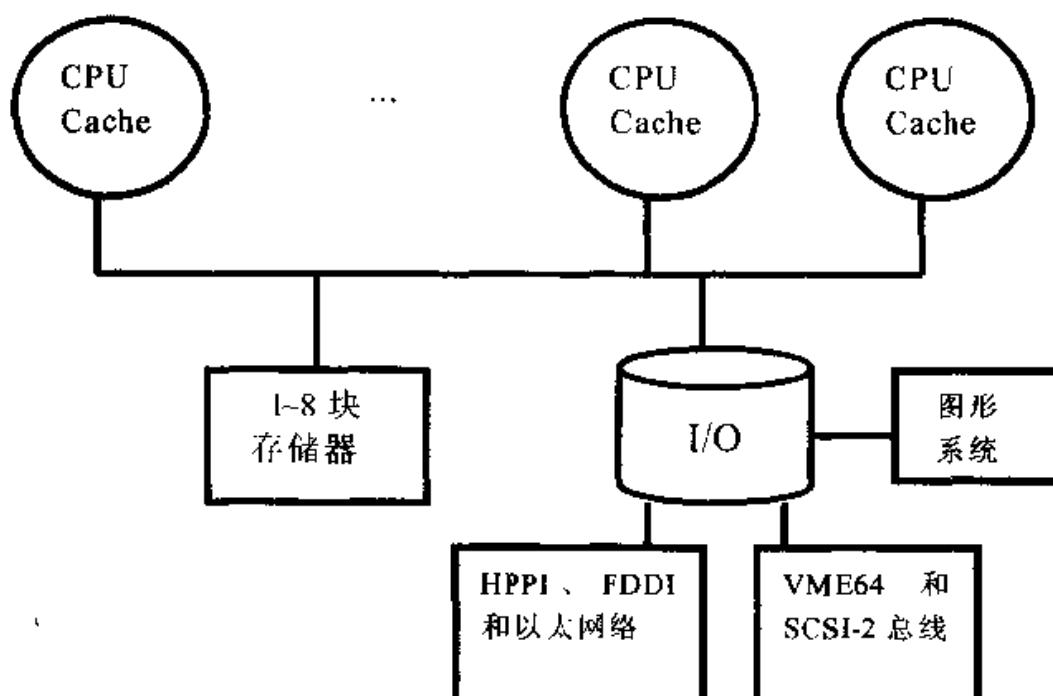


图 7.18 Challenge 系统结构

该系统采用写作废模式。Cache 数据块有 4 个状态,其中三个同前面所述,另增加一个干净专有(clean)状态,当该块被写过时成为脏专有状态。当块不是共享读出时,进入干净专有状态,其他处理器的读失效使其成为共享状态。从干净专有状态到脏专有状态不产生写失效或写作废。这把潜在要进行共享但还没有开始共享的数据视为私有数据,从而在实现上改善了性能。

POWERpath-2 可同时处理 8 个未完成的存储器读操作,包括写失效的读出。资源标识符(resource identifiers)分配到挂起的读上,用于标识读出的结果,使处理器知道它们的响应何时出现在总线上。

如果对同一 Cache 块有两个写,或一个写和一个读在总线上交叉,那么前面所讲 Cache 一致性算法就不能正确地工作。为防止这种情况发生,当与未完成的读有同一地址时,处理器不流出写作废(或写失效)。因此处理器的控制模块要追踪 8 个未完成读的请求地址。如果对未完成的读有同样地址的写失效出现,则处理器停顿,写被延迟。如果处理器检测到对一个未完成的读地址出现两个读失效时,则后一个读能与前一个读合并,也就是在前一个读的响应出现时,

总线接口将此数据同时送到后一个 Cache 读失效。

8 个读资源中每个都有一条禁止信号线, 用于指出一个读请求是否应由存储器响应。这一信号是实现一致性的关键。当一个处理器监听到请求时, 如果发现自己 Cache 没有该块, 或发现该块是干净状态, 该处理器就置禁止信号。如果所有处理器都已置禁止信号, 则存储器将响应, 因为所有处理器拷贝都为干净。如果一个处理器发现自己有该块已被写过的拷贝, 它请求总线并把该数据块放到总线上, 之后取消禁止信号。发出请求的处理器和存储器都收到该数据块并进行写入。

数据和地址总线分别仲裁和使用。写回请求同时使用两个总线, 读响应只使用数据总线, 写回和读请求只使用地址总线。这样就可形成并行: 同时接收一个新的读请求和发出一个对先前请求的响应。

POWERpath-2 总线事务由每个为 21 ns 的 5 个时钟构成。5 个时钟为仲裁、分解、寻址、译码和确认。在地址总线上发送一个地址需要 5 个时钟。在数据总线上发送 4 个 256 位数需要 5 时钟总线事务, 这时的传输率为

$$(256/8) \times (1\,000/21) \times (4/5) = 1.22 \text{ GB/s}$$

每一块存储板上有到 DRAM 的 576 位的传输路径 (512 位数据, 64 位纠错), 这使一个存储周期内可提供出三次总线传输的数据。所以在二路交叉下, 一个存储板可提供出完全的总线带宽。所需的总时间为 22 个总线时钟, 可满足无竞争下 128 字节 Cache 线的读失效。22 个时钟的构成为

1. 启动读请求为一个 5 总线时钟周期的总线事务;
2. 存储器读出并准备传输的等待时间为 12 个总线时钟;
3. 传输所有 128 字节(4 个 256 位)需要 5 个总线时钟。

为计算总的失效开销, 需要考虑从存储器地址流出点开始的每一步所需的等待时间, 直到失效处理完, 处理器重新启动:

1. 失效的初始检测及处理器访问存储器请求的生成。这一过程由三步构成: 检测在片上的初级 Cache; 启动片外二级 Cache 访问, 检测二级 Cache 的失效; 驱动全地址信号到系统总线。这一过程共用 40 个处理器时钟周期。
2. 总线和存储器系统所用部分, 由上可知为 22 个总线时钟。
3. 取数据到 Cache, 由于 R4400 停顿直到全部的 Cache 块重新装入, 所以装入的时间加到失效时间上。处理器上存储器接口为 64 位, 在外部总线 50 MHz 下工作, 装入 128 字节到 Cache 中需要 16 个总线时钟。
4. 10 个处理器时钟用于装入初级 Cache, 启动流水线。

所以总的失效开销为 50 处理器时钟加 38 总线时钟。在 150 MHz 的 R4400 的情况下, 每一总线时钟(20 ns)是三个处理器时钟(6.67 ns), 最后失效开销时间为 164 处理器时钟, 也就是 1 093 ns。

7.6.2 Origin 2000

Origin 是 SGI 公司分布共享存储器结构的大规模并行多处理机系统,采用超结点(每个结点 2 个处理器)的模块结构,可以从 1 个处理器扩展到 128 个处理器而维持系统的性能价格比不变,因此具有良好的可扩展性,能很好地满足当前网络计算环境的要求。Origin 2000 采用超标量 MIPS R10000 处理器,运行基于 UNIX 的 64 位 IRIX 操作系统。

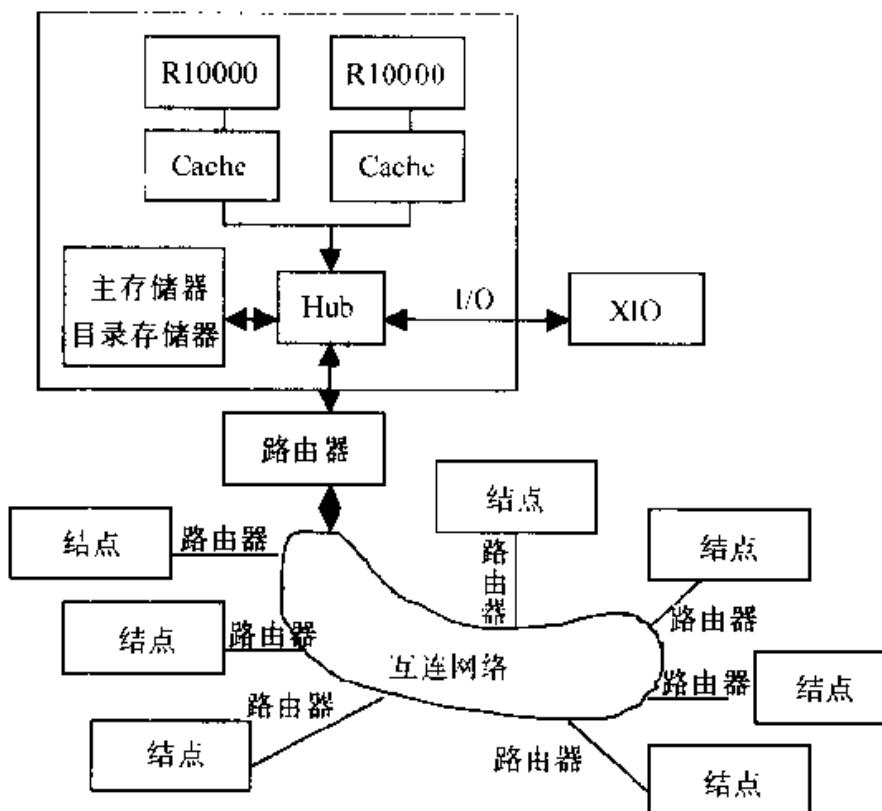


图 7.19 Origin 系统结构

Origin 基于 NUMA 体系结构。图 7.19 为 Origin 系统,该系统由结点、I/O 子系统、路由器和互连网络构成。每个结点可安装 1 个或 2 个 MIPS R10000 微处理器(内含第一级高速缓存,即 L1 Cache)、第二级高速缓存(L2 Cache)、主存储器、目录存储器及 Hub 等,Hub 用于连接微处理器、存储器、I/O 和路由器等。Origin 存储器系统每个结点的主存储器容量为 4 GB。结点的 Hub 内含 4 个接口和交叉开关,存储器接口能双向传送数据,最大传输率为 780 Mb/s,I/O 和路由器接口各有二个半双工传送端口,最大传输率为 2×780 Mb/s,即 1.56 Gb/s。每个 Hub 接口连接 2 个先进先出(FIFO)缓冲器,分别用于输入和输出的缓冲。

Origin 的路由器有 6 个端口,用于连接结点或其他路由器。Origin 的路由器和互连网络是 ASIC 芯片,通过芯片内部的交叉开关选择数据传送路径。

Origin 系统可由 1~128 个处理器组成,2 结点互连的情况如图 7.20 所示。

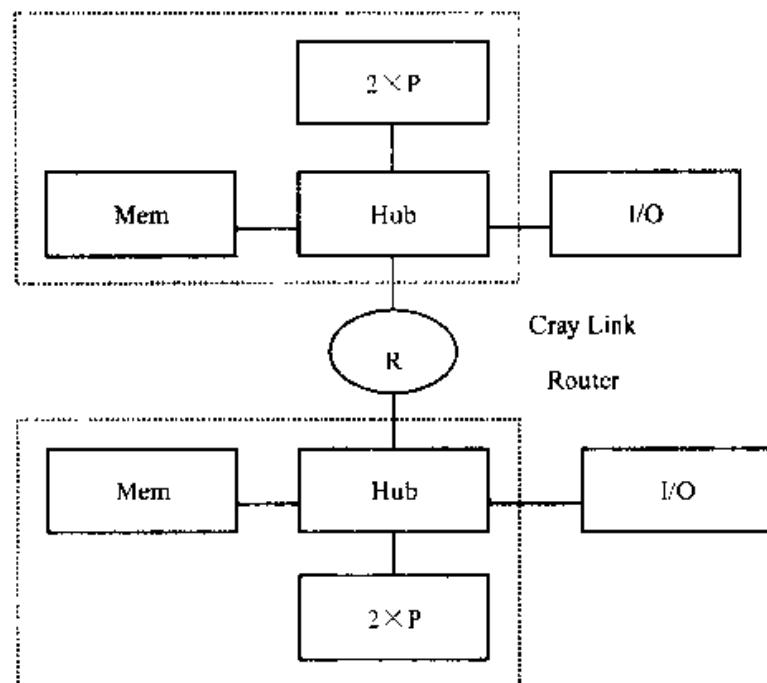


图 7.20 4 处理器系统

图 7.21 为由 8 结点构成的 16 处理器系统。为了减少数据在路由器之间的传送延迟,加快传送速度,可将图 7.21 中处于对角位置的路由器进行连接,如虚线所示。图 7.22 为 128 处理器系统。

128 处理器构成的 Origin 2000 系统由 4 个立方体组成,在立方体之间传送数据多经过了一级路由器。

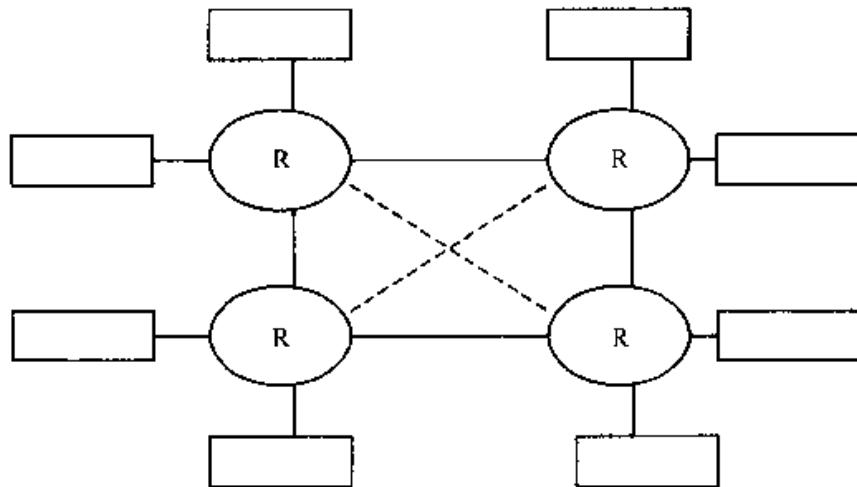


图 7.21 16 处理器系统

从上述的系统结构可以看到:在结点内部实现的是 SMP(对称多处理器)结

构,由于只有两个处理器,所以不存在 SMP 结构的总线瓶颈问题。在结点之间实现的是大规模并行处理结构,但又解决了共享存储器问题。因此在 Origin 系统中,无论是访问存储器的时间还是结点间传送数据的频带宽度都很理想。

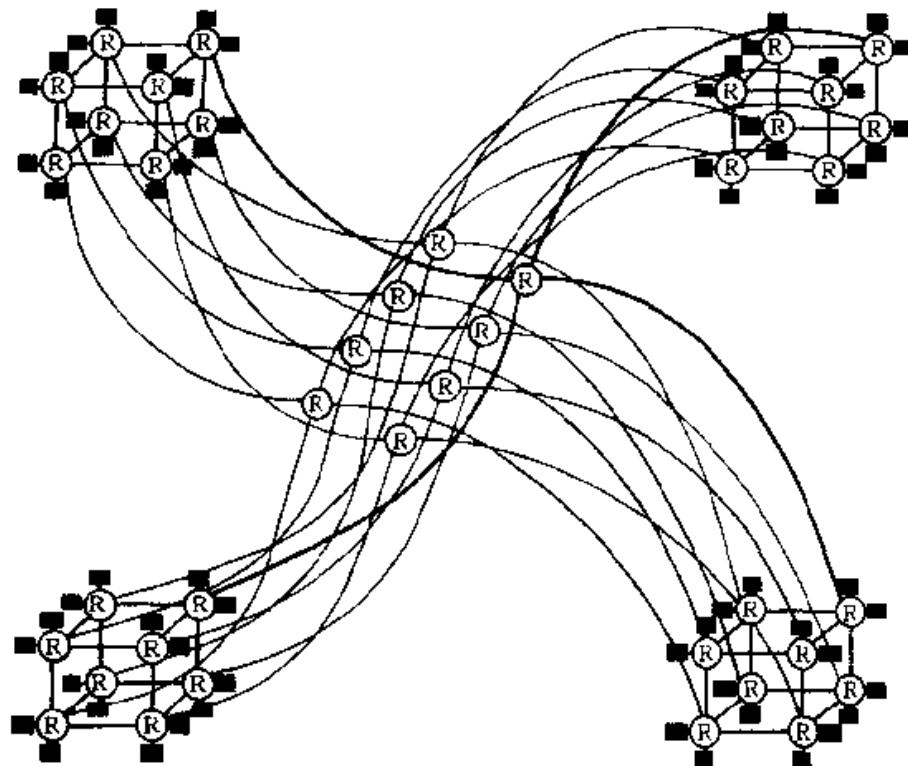


图 7.22 128 处理器系统

表 7.8 列出了 Origin 系统 CPU 访问存储器的延迟时间,假设 CPU 的主频为 195 MHz,Cache 不命中。其中最小延迟时间即是 CPU 访问本结点存储器的时间,最大延迟时间是 CPU 访问距离最远的存储器时间。

表 7.8 访问存储器的延迟时间

系统 CPU 数	最小延迟时间	最大延迟时间	平均延迟时间
2	318 ns	343 ns	343 ns
4	318 ns	554 ns	441 ns
8	318 ns	759 ns	623 ns
16	318 ns	759 ns	691 ns
32	318 ns	836 ns	764 ns
64	318 ns	1 067 ns	851 ns
128	318 ns	1 169 ns	959 ns

表 7.9 列出的是系统频带宽度, 其中每个 Hub 连到路由器和互连网络的最大频宽为 1.56 Gb/s(全双工, 2×780 Mb/s)。

表 7.9 系统频带宽度

系统处理器数	频宽(无快速传送连线)*	频宽(有快速传送连线)*
8	1.56 Gb/s	3.12 Gb/s
16	3.12 Gb/s	6.24 Gb/s
32	6.24 Gb/s	12.5 Gb/s
64	12.5 Gb/s	--
128	25 Gb/s	--

* 相当于图 7.21 中的虚线

Origin 系统的存储器层次结构可分为寄存器、L1 Cache、L2 Cache 和主存储器, 其中寄存器和 L1 Cache 在 R10000 微处理器中。寄存器的存取时间最短, L1 Cache 又分成指令 Cache 和数据 Cache 两部分, 这是为了避免取指令和存/取数据发生冲突。L2 Cache 安装在结点卡中, 统一存放指令和数据, 由 SRAM 组成。Origin 的主存储器地址是统一编址的, 每个处理器通过互连网络可访问系统中任一存储单元。

当一个处理器初次读取某一存储单元数据时, 该数据在提供给 CPU 的同时也拷贝到本结点的 Cache 中。其他处理器也可能读取该数据, 因此同一数据可能存放在几个结点的 Cache 中, 同一数据在各个 Cache 中将保持一致。Origin 系统的 Cache 一致性采用写作废协议。

Origin 采用基于目录的协议。在 Origin 的结点中, 有一个存储器和一个目录存储器。存储器被划分为存储器块(每个存储器块对应于一个 Cache 行)。每块对应一个目录项, 每个目录项包含其对应存储器块的状态信息和系统中各 Cache 共享存储情况的位向量, 根据位向量可以知道本存储器块在哪些 Cache 中有拷贝。当执行写存储器操作时, 根据目录项的位向量可将有关结点中的 Cache 数据作废, 从而实现了 Cache 的一致性。

7.7 小结

多处理机应用发展较为缓慢的主要原因是受限于软件以及使用效率, 这也是多处理机体系结构设计追求的主要目标。本章讨论了多处理机的有关问题, 其中涉及到一致性、互连网络、同步、远程访问、通信延迟和并行程序设计等。在未来, 多处理机将会得到更快的发展。这是因为应用领域及其并行性研究进展

较快；多处理器的性能价格比越来越好；多处理器对多道程序负载的高效性。目前在处理器芯片中已经可以包括 Cache 一致性逻辑，这有力地支持了小规模多处理器的实现，单芯片包含多个 CPU（例如 4 个）的微处理器也正在研究实现中。

摆在多处理器面前的问题是：要建立多大规模的多处理器？当处理器数量较大(>100)时，从硬件到软件带来的问题都将会十分严重。

多处理器的研究问题很多，除在本章中所讨论的之外，我们将正在研究的重要问题列举 4 个如下：

1. 多处理器性能的评测方法。并行处理中最为争论的问题之一就是如何评测并行机的性能。当然，直接的答案是运行测试程序，检测其响应时间。但在并行处理机中仅测试 CPU 时间会产生误导，因为有时处理器虽空闲，但却不可用于其他任务的执行。

2. 降低通信开销和延迟隐藏。在多处理器中，准确估算通信开销、如何使通信开销增长放慢、如何使访问时延在容许范围内增长仍都是有待解决的研究课题。此外，Cache、多流水线和超流水线的使用使失效延迟加长，这意味着延迟隐藏有很大的研究余地和难度。

3. 虚拟共享存储器(Distributed Virtual Memory, DVM)。它用操作系统来获得分布存储器具有一致性的共享地址空间。这种机制主要的不同点在于保持一致性的单位是页，并且用软件来实现一致性算法。

4. 并行软件的开发。包括编译程序、操作系统和应用软件。如何充分利用计算机系统结构提供的各种支持来提高并行性，在研究上还有很大的难度和深度。

习 题 七

7.1 解释术语：

集中式共享多处理器	分布式共享多处理器	通信延迟	计算/通信比
互连网络	静态网络	路由	动态网络
网络直径	结点度	等分带宽	旋转锁
栅栏同步			

7.2 设有一个在 3 种方式下运行的应用：使用所有的处理器、使用一半的处理器和单处理器的串行。设有 0.02% 的时间为串行，总共有 100 个处理器。如果加速比目标是 80，求在使用一半的处理器的方式下所允许的最大时间比例。

7.3 什么是多处理器的一致性？给出解决一致性的监听协议和目录协议的工作原理，并画出它们各自的状态变迁图。

7.4 画出 2 元 6-立方体的拓扑结构图。

7.5 画出用 4×4 交叉开关组成一个 3 级的 16×16 交叉开关网络，其设备量比单级 16×16

的交叉开关节省多少设备？举例说明在输入和输出之间存在着较多的冗余路径。

7.6 具有 $N=2^a$ 个输入端的 Omega 网络，采用单元控制，

- (1) N 个输入总共应有多少种不同的排列？
- (2) 该 Omega 网络通过一次可以实现的置换总共可有多少种是不同的？
- (3) 若 $N=8$ ，计算出一次通过能实现的置换数占全部排列的百分比。

7.7 试证明多级 Omega 网络采用不同大小构造块构造时所具有的下列特性：

- (1) 一个 $k \times k$ 开关模块的合法状态(连接)数目等于 k^k 。
- (2) 试计算用 2×2 开关模块构造的 64 个输入端的 Omega 网络一次通过所能实现置换的百分比。
- (3) 采用 8×8 开关模块构造 64 个输入端的 Omega 网络，重复(2)。
- (4) 采用 8×8 开关模块构造 128 个输入端的 Omega 网络，重复(2)。

7.8 在标准的栅栏同步中，设单个处理器的通过时间(包括更新计数和释放锁)为 C，求 N 个处理器一起进行一次同步所需要的时间。

7.9 采用排队锁和 fetch_and_increment 重新实现栅栏同步，并将它们分别与采用旋转锁实现的栅栏同步进行性能比较。

7.10 有些机器实现了专门的锁广播一致性协议，实现上可能使用不同的总线。假设使用写广播协议，重新给出例 7.3 旋转锁的时间计算。

7.11 在采用 16 个 R4400 的分布式共享多处理机中，设从局部或远程处理器访问地址可用到获得第一个字的存储器访问时间为 150 ns，R4400 的 Cache 失效开销在 7.6.1 节中给出，试求：

- (1) 局部访问比 Challenge 快多少？
- (2) 如果互连网络为 2D 网格，链路为 16 位宽，时钟为 100 MHz，信息起步时间为 5 个周期，网络结点间传输需要 1 个时钟周期，忽略信息竞争的开销(即网络带宽不受限)，求在远程请求为均匀分布时的平均远程访问时间。相比 Challenge 情况怎样？求在分布式共享存储器机器比 Challenge 有更低的平均存储器访问时间时，最大的远程失效比例应该是多少？

7.12 带宽消耗和延迟可引起写作废和写更新模式在性能上的差别。设有 64 字节 Cache 块的一个存储系统，不考虑竞争的影响，

- (1) 写出两段并行代码来说明写作废和写更新模式的带宽不同。
- (2) 写出一段并行代码来说明与写作废模式相比，写更新模式在延迟上的优越。通过例子说明在包含竞争时，写更新模式的延迟情况更差。设基于总线机器中存储器和监听事务需要 50 个时钟。

主要参考文献

- [1] Patterson D A, Hennessy J L. Computer Architecture: A Quantitative Approach. 2nd ed. San Francisco: Morgan Kaufmann Publish, 1996
- [2] Hwang k. Advanced Computer Architecture: Parallelism, Scalability, Programmablity. McGraw-Hill, 1993
- [3] Hennessy J L. Patterson D A. Computer Organization and Design: The Hardware Software Interface. 2nd ed. San Francisco: Morgan Kaufmann Publish, 1998
- [4] 李勇, 刘恩林. 计算机体系统结构. 长沙: 国防科技大学出版社, 1988
- [5] 郑纬民, 汤志忠. 计算机系统结构. 第2版. 北京: 清华大学出版社, 1998

