对本书的赞誉

“《计算机体系结构：量化研究方法（第5版）》继续发扬传统，为学习计算机体系结构的学生提供了当前计算平台的最新信息，使他们能够洞悉体系结构，便于设计未来系统。这一版的亮点在于大幅修订了数据级并行那一章，用传统的体系结构术语清晰地解读了GPU体系结构。”

——Krste Asanović，加州大学伯克利分校

“《计算机体系结构：量化研究方法》是一部经典，犹如美酒，历久而弥醇。我在本科毕业时第一次购买了本书，它到现在仍然是我最经常参考的书籍之一。第4版问世时，我发现其中包含了如此之多的新材料，为了跟上这一领域的最新趋势，我必须得再买一本。而当审阅第5版时，我发现Hennessy和Patterson再现神奇。全书内容都进行了大量更新，对于希望真正理解云和仓库级计算的人们来说，单凭第6章的内容，这个新版本就值得一读。只有Hennessy和Patterson才可能接触到谷歌、亚马逊、微软等云计算与互联网规模的应用提供商的内部人士，对这一重要领域，业内的介绍材料无出其右。”

——James Hamilton，Amazon Web服务部

“Hennessy和Patterson撰写本书的第一版时，研究生们是在用50 000个晶体管组装计算机。今天，仓库级的计算机集群会包含50 000个服务器，每个服务器中包含数十个处理器和数十亿个晶体管。计算机体系结构一直在不停地快速发展，而《计算机体系结构：量化研究方法》紧跟它的步伐，每个版本都准确地解释和分析了这一领域激动人心的最新重要思想。”

——James Larus，微软研究院

“这一版新增加了一章非常丰富的内容，用来讨论向量、SIMD和GPU体系结构中的数据级并行技术。它解释了应用于大众市场的GPU内部的关键体系结构概念，给出这些概念与传统术语的对应关系，并与向量和SIMD体系结构进行了对比。这一内容非常及时，与业内转向GPU

并行计算的潮流相适应。《计算机体系结构：量化研究方法》继续独领风骚，全面地介绍了体系结构方面的重大新进展！”

——John Nickolls，NVIDIA

“本书已经成为一本经典教科书了，这一版突出介绍了各种显式并行技术（数据、线程、请求）的兴起，各用整整一章来描述。数据并行一章尤为夺目：通过向量 SIMD、指令级 SIMD 和 GPU 的对比，避开每种体系结构的专用术语，揭示了这些体系结构之间的相似与区别。”

——Kunle Olukotun，斯坦福大学

“《计算机体系结构：量化研究方法（第 5 版）》探讨了各种并行概念和它们各自的技术权衡。和过去的几个版本一样，这一新版本中同样涵盖了最新的技术发展趋势。两个重点是个人移动设备（PMD）和仓库级计算（WSC）的爆炸性增长——与原来一味追求性能相比，这里的焦点已经转为更全面地寻求性能与能效之间的平衡。这些趋势刺激了人们不断追求更强劲的处理能力，而这种追求又推动人们在并行道路上走得更远。”

——Andrew N. Sloss，实施顾问，ARM 公司
ARM System Developer's Guide 一书的作者

作者简介

John L. Hennessy 是斯坦福大学的第 10 任校长，从 1977 年开始在该校电子工程与计算机系任教。Hennessy 是 IEEE 和 ACM 会士，美国国家工程院、国家科学院和美国哲学院院士，美国艺术与科学院院士。他获得过众多奖项，如 2001 年度 Eckert-Mauchly 奖，表彰他对 RISC 技术的贡献；2001 年度 Seymour Cray 计算机工程奖；与 David Patterson 共同获得的 2000 年度约翰·冯·诺依曼奖章。他还拥有 7 个荣誉博士学位。

1981 年，John L. Hennessy 带领几位研究生在斯坦福开始 MIPS 项目的研究。1984 年完成该项目之后，他暂时离开大学，与他人共同筹建 MIPS 计算机系统公司（也就是现在的 MIPS 技术公司），这家公司开发了最早的商用 RISC 微处理器之一。到 2006 年，已经有 20 多亿个 MIPS 微处理器被用于视频游戏、掌上电脑、激光打印机和网络交换机等各种设备中。Hennessy 后来领导了 DASH（Director Architecture for Shared Memory，共享存储器控制体系结构）项目，这一项目设计了第一个可扩展缓存一致性多处理器原型，其中的许多重要思想都在现代多处理器中得到了应用。除了参与科研活动、履行学校职责之外，他仍作为前期顾问和投资者参与了无数的创业项目。

David A. Patterson 自 1977 年进入加州大学伯克利分校执教以来，一直讲授计算机体系结构课程，拥有该校计算机科学 Pardee 讲座教授职位。他因为教学成果显著而荣获了加州大学的杰出教学奖、ACM 的 Karlstrom 奖、IEEE 的 Mulligan 教育奖章和本科生教学奖。因为在 RISC 方面的贡献而获得了 IEEE 技术成就奖和 ACM Eckert-Mauchly 奖，他还因为在 RAID 方面的贡献而分享了 IEEE Johnson 信息存储奖，并与 John Hennessy 共同获得了 IEEE 约翰·冯·诺依曼奖章和 C & C 奖金。和 John Hennessy 相似，Patterson 也是美国艺术与科学院院士、美国计算机历史博物馆院士、ACM 和 IEEE 会士。他还被选入美国国家工程院、美国国家科学院和硅谷工程名人堂。Patterson 身为美国总统信息技术顾问委员会委员，同时也是伯克利电子工程与计算机科学系计算机科学分部主任、计算机研究协会主席和 ACM 主席。这一履历使他荣获了 ACM 和 CRA 颁发的杰出服务奖。

在加州大学伯克利分校，Patterson 领导了 RISCI 的设计与实现工作，这可能是第一台 VLSI 精简指令集计算机，为商业 SPARC 体系结构奠定了基础。他曾是廉价磁盘冗余阵列（Redundant

Arrays of Inexpensive Disks，RAID）项目的领导者之一，正是由于这一项目，才有了后来许多公司出品的可靠存储系统。他还参与了工作站网络（Network of Workstations，NOW）项目，因为这一项目而有了因特网公司使用的集群技术和后来的云计算。这些项目获得了 ACM 颁发的三个论文奖。作为“算法-机器-人类”（AMP）实验室和并行计算实验室的主管，他目前在这里开展自己的研究项目。AMP 实验室的目标是开发可扩展的机器学习算法、适用于仓库级计算机的编程模型、能够快速洞悉云中海量数据的众包（Crowd-Sourcing）工具。并行计算实验室的目标是研发先进技术，为并行个人移动设备提供可扩展、可移植、方便快捷的效率软件。

序　言

Hennessy 和 Patterson 合著的《计算机体系结构：量化研究方法》第 1 版是在我刚上研究生时出版的，因此，我属于第一批在本书指导下学习体系结构的人。要写一篇有用的序言，少不了要有作序者自己的独特观点，而我发现自己在这方面有点欠缺，因为我已深受本书前 4 个版本的影响。还有另外一个不利因素，就是我从学生时代就对这两位计算机科学巨匠心存敬畏，尽管后来我有机会与他们合作，近距离了解他们，但也可能正因如此，所以这种敬畏心现在仍未消失。不过，由于我从第一版开始就一直从事这一领域的研究，有机会看到它不断完善，欣赏它持久不变的实用性，这从一定程度上抵消了上述不利因素。

几年前，英特尔取消其 4GHz 单核 CPU 开发项目，转向多核 CPU 研发，标志着业内对更高 CPU 时钟频率的激烈竞争正式结束，两年之后，本书第 4 版出版。经过两年的充分观察，John 和 Dave 在书中将这一变化明确表述为计算技术在过去 10 年中的一个转折点，而不是一次非常随意的生产线升级。第 4 版对指令级并行（ILP）的强调有所降低，增加了线程级并行的相关内容；第 5 版则更进一步，用整整两章的篇幅来讨论线程级和数据级并行，而将 ILP 的讨论压缩为一章。新增加的第 4 章会让刚刚接触新型图形处理引擎的读者受益匪浅，这一章的重点是数据级并行，解释了一些虽有不同但正在趋于一致的解决方案，这些方案是由通用处理器中的多媒体扩展以及可编程性日益增强的图形处理器提供的。这一章还有一些非常实用的内容：如果你一直被 CUDA 术语搞得晕头转向，可以参考表 4-10。（难题："共享存储器"实际上是本地存储器，而"全局存储器"更接近于大众认知的共享存储器。）

多核技术仍处在不断变化之中，但第 5 版还介绍了下一代重大技术：云计算。因特网的无处不在和 Web 服务的发展将人们的注意力引向两个极端：一端是超小型设备（智能手机、平板电脑），一端是超大型设备（仓库级计算系统）。第 3 章的"融会贯通"一节（3.13 节）介绍了 ARM Cortex A8，它是智能手机中的一种常用 CPU，而新增加的整个第 6 章则专门结合仓库级计算系统讨论了请求级并行和数据级并行。在这一章中，John 和 Dave 将这些新出现的大型集群看作一类特殊的新型计算机——欢迎广大计算机架构师一同来推动这一新兴领域的发展。将第 3 版中的 Google 集群体系结构与这一版第 6 章中更现代化的具体实现进行对比，读者就可以欣赏到这一领域在过去 10 年的发展过程。

本书的老读者将会再一次领略两位杰出计算机科学家的贡献，他们在整个职业生涯中，将严谨的学术研究和对前沿产品与技术的深刻理解非常完美地结合在一起，形成了一门艺术。如果你见过 Dave 如何实施他一年两度的项目调整，如何精心筹办研讨会，从校企合作之中萃取最多的精华，如果你能回想起 John 在创办 MIPS 时获得的巨大成功，或者曾经在 Google 公司的走廊里碰到他（我偶尔就会在那里遇到他），那你就不会再为两位作者与业界交流何以如鱼得水而感到惊讶了。

新老读者购买本书绝对物超所值，这可能是最重要的一点了。这本书之所以会成为不朽的经典，是因为它的每个版本都不只是更新，而是一次全面修订，力求向读者展示最新信息，以及作者对这个快速变化的迷人领域最独到的见解。对我来说，在从事这个行业 20 多年之后，它让我又一次有机会体验学生时代对这两位杰出师长的敬仰之情。

——Luiz André Barroso，Google 公司

前　言

本书的目的

本书到现在已经是第 5 个版本了，我们的目标一直没有改变，就是要阐述那些为未来技术发展奠定基础的基本原理。计算机体系结构的各种发展机遇总是让我们激情澎湃，不曾有丝毫消退。我们在第 1 版中就作出过如下的论述："这个学科不是令人昏昏欲睡、百无一用的纸版模型。绝对不是！这是一个受到人们热切关注的学科，需要在市场竞争力与成本-性能-能耗之间作好权衡，从事这个学科既可能导致可怕的失败，也可能带来显赫的成功。"

在编写第 1 版时，我们的主要目的是希望改变人们原来学习和研究计算机体系结构的方式。现今，我们感到这一目标依然正确，依然重要。该领域日新月异，在对其进行研究时，必须采用真实计算机上的测量数据和真实示例，而不是去研究一大堆从来都不需要实现的定义和设计。我们不仅热烈欢迎过去与我们结伴而行的老读者，同样也非常欢迎现在刚刚加入我们的新朋友。不管怎样，我们都保证将采用同样的量化方法对真实系统进行分析。

和前几版一样，在编写这个新版本时，我们力争使其既适用于学习高级计算机体系结构与设计课程的学生，也适用于专业的工程师和架构师。与第 1 版类似，这个版本重点介绍新平台（个人移动设备和仓库级计算机）和新体系结构（多核和 GPU）。这一版还秉承了前几版的做法，希望能够通过强调成本、性能、能耗之间的平衡和优秀的工程设计，揭去计算机体系结构的神秘面纱。我们相信这一领域正在日趋成熟，发展成为一门具备严格量化基础的经典理工学科。

关于第 5 版

我们曾经说过，第 4 版可能因为转向讨论多核芯片而成为自第 1 版以来的最重要版本。但我们收到了这样的反馈意见：第 4 版已经失去了第 1 版重点突出的优点，它一视同仁地讨论所有内容，不分重点和场合。我们非常确信，第 5 版不会再有这样的评价了。

我们相信，最令人激动的地方在于计算规模的两个极端：以移动电话和平板电脑之类的个人移动设备（PMD）为客户端，以提供云计算的仓库级计算机为服务器。（具有敏锐观察力的读者可能已经看出本书封面上云计算的寓意。）尽管这两个极端的规模大小不同，但它们在成本、性能和能效方面的共同主题给我们留下了深刻印象。因此，每一章的讨论背景都是 PMD 和仓

库级计算机的计算，第6章是全新的一章，专门讨论仓库级计算机。

本书的另一条主线是讨论并行的所有不同形式。我们首先在第1章指出了两种应用级别的并行，一个是**数据级并行**（DLP），它的出现是因为有许多数据项允许同时对其进行操作；另一个是**任务级并行**（TLP），它的出现是因为创建了一些可以独立执行并在很大程度上并行的工作任务。随后解释4种开发DLP和TLP的体系结构样式，分别是：第3章介绍的**指令级并行**（ILP），第4章介绍的**向量体系结构**和**图形处理器**（GPU），这一章是第5版新增加的内容；第5章介绍的**线程级并行**；第6章通过仓库级计算机介绍的**需求级并行**（RLP），这一章也是第5版中新增加的。本书中，我们将存储器层次结构的内容提前到第2章，并将存储系统那一章改作附录D。我们对第4章、第6章的内容尤为感到自豪，第4章对GPU的解读是目前最详尽、最清晰的，第6章首次公布了Google仓库级计算机的最新细节。

与前几版相同，本书前三个附录提供了有关MIPS指令集系统、存储器层次结构和流水线的基础知识，如果读者没有读过《计算机组成与设计》之类的书籍，可用作参考。为了在降低成本的同时还能提供一些读者感兴趣的补充材料，我们在网络上提供了另外9个附录，网址为：http://booksite.mkp.com/9780123838728。这些附录的页数之和比本书还要多呢！

这一版继续发扬“以真实示例演示概念”的传统，并增加了全新的“融会贯通”部分。这一版中的“融会贯通”内容包括以下各服务器的流水线组成与存储器层次结构：ARM Cortex A8处理器、Intel core i7处理器、NVIDIA GTX-280和GTX-480 GPU，还有Google仓库级计算机。

主题的选择与组织

和以前一样，我们在选择主题时采用了一种保守的方法，毕竟这个领域中值得讨论的思想实在太多了，不可能在这样一本主要讨论基本原理的书中将其全部涵盖在内。我们没有面面俱到地分析读者可能遇到的所有体系结构，而是将重点放在那些在任何新计算机中都可能涉及的核心概念上。根据一贯坚持的选材标准，本书讨论的思想都经过深入研究并已被成功应用，其内容足以采用量化方法进行讨论。

我们一直重点关注的内容都是无法从其他来源获取的同类资料，因此我们将继续尽可能讨论比较高级的内容。事实上，本书介绍的有些系统，就无法在文献中找到相关描述。如果读者需要了解更为基础的计算机体系结构知识，可以阅读《计算机组成与设计：硬件/软件接口》（*Computer Organization and Design: The Hardware/Software Interface*）一书。

内容概述

这一版对第1章进行了补充，其中包括能耗、静态功率、动态功率、集成电路成本、可靠

性和可用性的计算公式。（封二上也列出了这些公式。）在本书后续部分读者能够一直应用这些公式。除了计算机设计与性能测量方面的经典量化原理之外，还对 PIAT 一节进行了升级，采用了新的 SPECPower 基准测试。

我们认为，与 1990 年相比，指令集体系结构扮演的角色有所弱化，所以我们把这一部分内容作为了附录 A。它仍然采用 MIPS64 体系结构。（为便于快速查看，封三汇总了 MIPS ISA 相关信息。）网站上的附录 K 介绍了 10 种 RISC 体系结构、80x86、DEC VAX 和 IBM 360/370，献给 ISA 爱好者们。

随后，我们在第 2 章开始讨论存储器层次结构，这是因为很容易针对这些内容应用成本-性能-功耗原理，而且存储器是其余各章的关键内容。和上一版一样，附录 B 对缓存机制作了概述，以供读者需要时查阅。第 2 章讨论了对缓存的 10 种高级优化方法。这一章还介绍了虚拟机，它便于提供保护、进行软硬件管理，而且在云计算中也扮演着重要角色。除了介绍 SRAM 和 DRAM 技术之外，这一章还包括了闪存的内容。PIAT 示例选择了 PMD 中使用的 ARM Cortex A8 和服务器中使用的 Intel Core i7。

第 3 章主要研究高性能处理器中的指令级并行开发，包括超标量执行、分支预测、推理、动态调度和多线程。前面曾经提到，附录 C 是关于流水线的一个综述，以备随时查阅之用。第 3 章还研究了 ILP 的局限性。和第 2 章一样，PIAT 示例还是 ARM Cortex A8 和 Intel Core i7。第 3 版包括大量有关 Itanium 和 VLIW 的材料，现在这些内容放在网上的附录 H 中，这表明了我们的观点：这种体系结构未能达到过去所宣称的效果。

多媒体应用程序（比如游戏和视频处理）的重要性在提高，因此，开发数据级并行的体系结构也变得更为重要。具体来说，越来越多的人在关注利用图形处理器（GPU）执行的运算，但很少有架构师了解 GPU 到底是如何工作的。我们决定编写新的一章，主要就是为了揭开这种新型计算机体系结构的奥秘。第 4 章开始介绍向量体系结构，对多媒体 SIMD 指令集扩展和 GPU 的解释就是以此为基础的。（网站上的附录 G 深入地讨论了向量体系结构。）GPU 一节是本书最难写的部分，需要多次反复才能给出一个既精确又容易理解的描述。一个重大挑战就是术语。我们决定使用我们自己的术语，然后给出这些术语与 NVIDIA 官方术语之间的对应关系。这一章介绍了 Roofline 性能模型，然后用它来对比 Intel Core i7、NVIDIA GTX 280 和 GTX 480 GPU。这一章还介绍了供 PMD 使用的 Tegra 2 GPU。

第 5 章介绍多核处理器，探讨了对称、分布式存储器体系结构，考查了组织原理和性能。接下来是有关同步和存储器一致性模型的主题，所采用的示例是 Intel Core i7。对片上互连网络感兴趣的读者可以阅读网站上的附录 F，对更大规模多处理器和科学应用感兴趣的读者可以阅读网站上的附录 I。

前面曾经提到，第 6 章介绍了计算机体系结构中的最新主题——仓库级计算机（Warehouse-Scale Computer，WCS）。依靠 Amazon Web 服务部门和 Google 工程师的帮助，本章整合了有关

WSC设计、成本与性能的详细资料，而以前了解这些内容的架构师寥寥无几。在开始描述WSC的体系结构和物理实现（及成本）之前，首先介绍了MapReduce编程模型。从成本的角度可以解释为什么会有云计算，以及为何在云中使用 WSC 进行计算的成本要低于在本地数据中心的计算成本。PIAT实例是对Google WSC的描述，有些内容是首次公开的。

接下来就是附录A到附录L。[①]附录A介绍ISA的原理，包括MIPS64，附录K介绍Alpha、MIPS、PowerPC和SPARC的64位版本及其多媒体扩展。其中还包括一些经典体系结构（80x86、VAX和IBM 360/370）和流行的嵌入指令集（ARM、Thumb、SuperH、MIPS16和Mitsubishi M32R）。附录H与其相关，介绍了VLIW ISA的体系结构和编译器。

前面曾经提到，附录B和附录C是缓存与流水线基本概念的教程。建议对缓存不够熟悉的读者在阅读第2章之前先阅读附录B，新接触流水线的读者在阅读第3章之前先阅读附录C。

附录D“存储系统”包括：进一步讨论可靠性和可用性，以RAID 6方案介绍为主体的RAID教程，非常珍贵的真实系统故障统计信息。接下来介绍了排队理论和 I/O 性能基准测试。我们评估了一个真实集群Internet Archive的成本、性能和可靠性。“融会贯通”部分以NetApp FAS6000文件管理程序为例。

附录E由Thomas M. Conte撰写，汇总了嵌入式系统的相关内容。

附录F讨论网络互连，由Timothy M. Pinkston和José Duato进行了修订。附录G最初由Krste Asanović撰写，其中详细介绍了向量处理器。就我们所知，这两个附录是其各自相关主题的最好材料。

附录H详细介绍了VLIW和EPIC，也就是Itanium采用的体系结构。

附录I详细介绍了大规模共享存储器多处理方面用到的并行处理应用和一致性协议。附录J由David Goldberg撰写，详细介绍了计算机算法。

附录L将第3版每一章中的“历史回顾与参考文献”部分集中在一起。对于各章介绍的思想，它尽量给予一个恰当的评价，并让读者了解这些创造性思想背后的历史。我们希望以此来展现人类在计算机设计方面的戏剧性发展过程。这个附录还提供了一些参考文献，主修体系结构的学生可能会非常喜欢它们。其中提到了本领域的一些经典论文，如果时间允许，建议读者阅读这些论文。直接听原创者讲述他们的思想，在深受教育的同时，也是一种享受。而“历史回顾”是以前版本中最受欢迎的章节之一。

内容导读

所有读者都应当从第1章开始阅读，除此之外并不存在什么唯一的最佳顺序。如果你不想阅读全部内容，可以参考下面这些顺序。

---

① 本书中文版未收录附录D到附录L，这些内容可在英文书网站 http://booksite.mkp.com/9780123838728 获取。——编者注

- **存储器层次结构**：附录 B、第 2 章、附录 D。
- **指令级并行**：附录 C、第 3 章、附录 H。
- **数据级并行**：第 4 章、第 6 章、附录 G。
- **线程级并行**：第 5 章、附录 F、附录 I。
- **请求级并行**：第 6 章。
- **ISA**：附录 A、附录 K。

附录 E 可以随时阅读，但在 ISA 和缓存序列之后阅读，效果可能会更好一些。附录 J 可以在涉及运算时阅读。附录 L 的各部分内容应当在读完正文中相应章节后阅读。

章节安排

我们根据一种统一的框架安排内容，使各章在结构方面保持一致。首先会介绍一章的主题思想，然后是“交叉问题”部分，说明本章介绍的思想与其他各章有什么相互关系。接下来是“融会贯通”部分，通过展示如何在实际计算机中应用这些思想，将它们串在一起。

再下面是“谬论与易犯错误”，让读者从他人的错误中汲取教训。我们将举例说明一些常见误解与体系结构陷阱，要避免犯错是非常困难的，哪怕你明明知道它们就在前面等着你。“谬论与易犯错误”部分是本书最受欢迎的内容。每一章都以一个“结语”节结束。

案例研究与练习

每一章的最后都有案例研究和练习。这些案例研究由业内和学术界的专家编撰而成，通过难度逐渐增大的练习来探讨该章的关键概念，检验读者的理解程度。教师们会发现这些案例研究都非常详尽和完善，完全可以针对它们设计出一些练习。

每个练习中用尖括号括起的内容（<章.节>）指明了做这道题应该阅读哪部分正文内容。我们这样做的目的，一方面是为了提供复习内容，另一方面是希望帮助读者避免在还没有阅读相应正文的情况下去做一些练习。为了使读者大致了解完成一道题需要多长时间，我们为这些练习划定了不同等级：

[10] 短于 5 分钟（阅读和理解时间）；

[15] 5~15 分钟给出完整答案；

[20] 15~20 分钟给出完整答案；

[25] 在 1 小时内给出完整的书面答案；

[30] 小型编程项目：时间短于 1 整天；

[40] 大型编程项目：耗时 2 周；

[讨论] 与他人一起讨论的主题。

在 textbooks.elsevier.com 注册的老师可以得到案例研究与习题的解答。

补充材料

我们还通过网络提供了多种资料，网址为 http://booksite.mkp.com/9780123838728/，内容包括：

- 参考附录——涵盖了一系列高级主题，由相关领域的专家撰写；
- 历史材料，考察了正文各章所介绍的关键思想的发展形成过程；
- 供老师使用的 PowerPoint 幻灯片；
- PDF、EPS 和 PPT 格式的书中插图；
- 网上相关材料的链接；
- 勘误表。

我们会定期补充新材料和网上其他可用资源的链接。

帮助改进本书

如果你阅读后面的“致谢”部分，将会看到我们已经下了很大的功夫来纠正错误。由于一本书会进行多次印刷，所以我们有机会进行更多的校订。如果你发现了任何遗留错误，请通过电子邮件联系出版商（ca5bugs@mkp.com）。[①]

我们欢迎你对本书给出其他意见，请将它们发送到另一个电子信箱：ca5comments@mkp.com。

结语

本书仍然是一本真正的合著作品，我们每人编写的章节和附录各占一半。如果没有对方完成另一半工作，如果没有对方在任务似乎无望完成时给予鼓励，如果没有对方点透某个难以表述的复杂概念，如果没有对方花费周末时间来审阅书稿，又如果没有对方在自己因为其他繁重职责而难以提笔时给予宽慰（从简历可以看出，这些职责是随着本书的版本号以指数形式增加的），我们无法想象这本书要花费多长时间才能完成。当然，对于你将要读到的内容，其中若有不当之处，我们也负有同等责任。

John Hennessy

David Patterson

① 读者可以到图灵社区本书主面（www.ituring.com.cn/book/888）提交中译本勘误。——编者注

致　　谢

尽管本书仅正式发布了5个版本，但我们实际上已经写出过10个不同版本：第1版有3个版本（alpha版、beta版和最终版），第2、3、4版各有2个版本（beta版和最终版）。一路走来，我们得到了数百位审阅者和用户的帮助，他们每一位都让这本书变得更好。因此，我们决定列出所有对本书各版本作出贡献的人员名单。

第5版的贡献者

和前几版一样，第5版也是一个有许多志愿者参与的集体成果。没有这些志愿者的帮助，这一版就不可能保持一贯的品质。

审阅者

南卡罗来纳大学的Jason D. Bakos、加州大学圣巴巴拉分校的Diana Franklin、HP实验室的Norman P. Jouppi、田纳西大学的Gregory Peterson、HP实验室的Parthasarathy Ranganathan、克莱姆森大学的Mark Smotherman、威斯康星大学麦迪逊分校的Gurindar Sohi、西班牙加泰罗尼亚理工大学的Mateo Valero以及新泽西理工学院的Sotirios G. Ziavras。

加州大学伯克利分校并行实验室和RAD实验室的成员（他们多次审阅第1、4、6章，并使我们对GPU和WSC的解释部分得以成形）：Krste Asanović、Michael Armbrust、Scott Beamer、Sarah Bird、Bryan Catanzaro、Jike Chong、Henry Cook、Derrick Coetzee、Randy Katz、Yunsup Lee、Leo Meyervich、Mark Murphy、Zhangxi Tan、Vasily Volkov以及Andrew Waterman。

顾问团

Google公司的Luiz André Barroso、R&E Colwell & Assoc.公司的Robert P. Colwell、ARM公司研发副总裁Krisztian Flautner、宾州州立大学的Mary Jane Irwin、NVIDIA公司的David Kirk、Tensilica首席科学家Grant Martin、威斯康星大学麦迪逊分校的Gurindar Sohi以及西班牙加泰罗尼亚理工大学的Mateo Valero。

附录

加州大学伯克利分校的 Krste Asanović（附录 G）、北卡罗来纳州立大学的 Thomas M. Conte（附录 E）、西班牙巴伦西亚理工大学的 José Duato（附录 F）、施乐帕洛阿尔托研究中心的 David Goldberg（附录 J）、南加州大学的 Timothy M. Pinkston（附录 F）。

瓦伦西亚理工大学的 José Flich 为附录 F 的更新作出了重大贡献。

案例研究与练习

南卡罗来纳大学的 Jason D. Bakos（第 3 章和第 4 章）、加州大学圣巴巴拉分校的 Diana Franklin（第 1 章和附录 C）、HP 实验室的 Norman P. Jouppi（第 2 章）、HP 实验室的 Naveen Muralimanohar（第 2 章）、田纳西大学的 Gregory Peterson（附录 A）、HP 实验室的 Parthasarathy Ranganathan（第 6 章）、圣克拉拉大学的 Amr Zaky（第 5 章和附录 B）。

Jichuan Chang、Kevin Lim 和 Justin Meza 帮助制定和测试了第 6 章的案例研究与练习。

补充材料

NVIDIA 公司的 John Nickolls、Steve Keckler 和 Michael Toksvig（第 4 章 NVIDIA GPU），Intel 公司的 Victor Lee（第 4 章 Core i7 与 GPU 的对比），美国劳伦斯伯克利国家实验室（LBNL）的 John Shalf（第 4 章最新向量体系结构），LBNL 的 Sam Williams（第 4 章中的 Roofline 计算机模型），澳大利亚国立大学的 Steve Blackburn 和得克萨斯大学奥斯汀分校的 Kathryn McKinley（第 5 章的 Intel 性能与功耗测量），Google 公司的 Luiz Barroso、Urs Hölzle、Jimmy Clidaris、Bob Felderman 和 Chris Johnson（第 6 章的 Google WSC），Amazon Web 服务部的 James Hamilton（第 6 章的功率分配与成本模型）。

南卡罗来纳大学的 Jason D. Bakos 为这一版制作了新的授课幻灯片。

最后，要再次特别感谢克莱姆森大学的 Mark Smotherman，他对我的最终手稿进行了技术审阅，发现了大量错误和含糊不清的地方，使本书的表述清晰了许多。

当然，没有出版商，本书也不可能出版，所以我们要感谢 Morgan Kaufmann/Elsevier 全体员工的努力和支持。关于第 5 版，我们特别要感谢本书的编辑 Nate McFadden 和 Todd Green，他们协调了从开展调查、组织顾问团、制定案例研究与练习、组成焦点小组、手稿审读到附录更新的整个过程。

我们还得感谢我们学校的员工 Margaret Rowland 和 Roxana Infante，在编写本书期间，她们为我们处理了无数的快递邮件，帮助我们化解了斯坦福和伯克利的许多紧急情况。

最后要感谢我们的妻子，感谢她们容忍我们越来越早起进行阅读、思考和写作。

前几版的贡献者

审阅者

普渡大学的 George Adams，伊利诺伊大学香槟分校的 Sarita Adve，杨百翰大学的 Jim Archibald，麻省理工大学的 Krste Asanović，华盛顿大学的 Jean-Loup Baer，东北大学的 Paul Barr，得克萨斯大学圣安东尼奥分校的 Rajendra V. Boppana，密歇根大学的 Mark Brehob，得克萨斯大学奥斯汀分校的 Doug Burger，SGI 的 John Burger，Michael Butler，Thomas Casavant，Rohit Chandra，密歇根大学的 Peter Chen，纽约州立大学石溪分校、卡内基·梅隆大学、斯坦福大学、克莱姆森大学和威斯康星大学的教学班、Vitesse 半导体公司的 Tim Coe，Robert P. Colwell，David Cummings，Bill Dally，David Douglas，西班牙巴伦西亚理工大学的 José Duato，东南密苏里州立大学的 Anthony Duben，华盛顿大学的 Susan Eggers，Joel Emer，达特茅斯学院的 Barry Fagin，加州大学圣克鲁斯分校的 Joel Ferguson，Carl Feynman，David Filo，HP 实验室的 Josh Fisher，DIKU 的 Rob Fowler，华盛顿大学圣路易斯分校的 Mark Franklin，Kourosh Gharachorloo，哈佛大学的 Nikolas Gloy，施乐帕洛阿尔托研究中心的 David Goldberg，Intel 和西班牙加泰罗尼亚理工大学的 Antonio González，威斯康星大学麦迪逊分校的 James Goodman，弗吉尼亚大学的 Sudhanva Gurumurthi，哈维姆德学院的 David Harris，John Heinlein，斯坦福大学的 Mark Heinrich，加州大学圣克鲁斯分校的 Daniel Helman，威斯康星大学麦迪逊分校的 Mark D. Hill，IBM 公司的 Martin Hopkins，HP 实验室的 Jerry Huck，伊利诺伊大学香槟分校的 Wen-mei Hwu，宾州州立大学的 Mary Jane Irwin，Truman Joe，Norm Jouppi，东北大学的 David Kaeli，内布拉斯加大学的 Roger Kieckhafer，加拿大瑞尔森大学的 Lev G. Kirischian，Earl Killian，普渡大学的 Allan Knies，Don Knuth，斯坦福大学的 Jeff Kuskin，微软研究院的 James R. Larus，多伦多大学的 Corinna Lee，Hank Levy，普林斯顿大学的 Kai Li，阿拉斯加大学费尔班克斯分校的 Lori Liebrock，威斯康星大学麦迪逊分校的 Mikko Lipasti，北卡罗来纳大学教堂山分校的 Gyula A. Mago，Bryan Martin，Norman Matloff，David Meyer，伍斯特理工学院的 William Michalson，James Mooney，密歇根大学的 Trevor Mudge，得克萨斯大学奥斯汀分校的 Ramadass Nagarajan，卡内基·梅隆大学的 David Nagle，Todd Narter，Victor Nelson，加州大学伯克利分校的 Vojin Oklobdzija，斯坦福大学的 Kunle Olukotun，宾州州立大学的 Bob Owens，Sun 公司的 Greg Papadapoulous，Joseph Pfeiffer，康奈尔大学的 Keshav Pingali，南加利福尼亚大学的 Timothy M. Pinkston，加拿大滑铁卢大学的 Bruno Preiss，Steven Przybylski，Jim Quinlan，Andras Radics，佐治亚理工学院的 Kishore Ramachandran，得克萨斯大学奥斯汀分校的 Joseph Rameh，康奈尔大学的 Anthony Reeves，密歇根州立大学的 Richard Reid，密歇根大学的 Steve Reinhardt，加州大学洛杉矶分校的 David Rennels，马萨诸塞大学阿默斯特分校的 Arnold L. Rosenberg，普渡大学的 Kaushik Roy，Unysis

的 Emilio Salgueiro，得克萨斯大学奥斯汀分校的 Karthikeyan Sankaralingam，Peter Schnorf，Margo Seltzer，南卫理公会大学的 Behrooz Shirazi，卡内基·梅隆大学的 Daniel Siewiorek，普林斯顿大学的 J. P. Singh，Ashok Singhal，威斯康星大学麦迪逊分校的 Jim Smith，哈佛大学的 Mike Smith，克莱姆森大学的 Mark Smotherman，威斯康星大学麦迪逊分校的 Gurindar Sohi，华盛顿大学的 Arun Somani，克莱姆森大学的 Gene Tagliarin，圣母大学的 Shyamkumar Thoziyoor，俄勒冈大学的 Evan Tick，北卡罗来纳大学教堂山分校的 Akhilesh Tyagi，弗吉尼亚大学的 Dan Upton，西班牙加泰罗尼亚理工大学的 Mateo Valero，加州大学圣克鲁斯分校的 Anujan Varma，康奈尔大学的 Thorsten von Eicken，得克萨斯 A&M 大学的 Hank Walker，施乐帕洛阿尔托研究中心的 Roy Want，Sun 公司的 David Weaver，以色列特拉维夫大学的 Shlomo Weiss，David Wells，克莱姆森大学的 Mike Westall，Maurice Wilkes，Eric Williams，普渡大学的 Thomas Willis，Malcolm Wing，纽约州立大学石溪分校的 Larry Wittie，佐治亚理工学院的 Ellen Witte Zegura，新泽西理工学院的 Sotirios G. Ziavras。

附录

向量附录由麻省理工学院的 Krste Asanović 修订。浮点附录最初由施乐帕洛阿尔托研究中心的 David Goldberg 编写。

练习

普渡大学的 George Adams，威斯康星大学麦迪逊分校的 Todd M. Bezenek（纪念他的祖母 Ethel Eshom），Susan Eggers，Anoop Gupta，David Hayes，Mark Hill，Allan Knies，加州大学圣克鲁斯分校的 Ethan L. Miller，康柏西部研究实验室的 Parthasarathy Ranganathan，威斯康星大学麦迪逊分校的 Brandon Schwartz，Michael Scott，Dan Siewiorek，Mike Smith，Mark Smotherman，Evan Tick，Thomas Willis。

案例研究与练习

威斯康星大学麦迪逊分校的 Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau，R&E Colwell & Assoc.公司的 Robert P. Colwell，加州州立理工大学圣路易斯奥比斯波分校的 Diana Franklin，伊利诺伊大学香槟分校的 Wen-mei W. Hwu，HP 实验室的 Norman P. Jouppi，伊利诺伊大学香槟分校的 John W. Sias，威斯康星大学麦迪逊分校的 David A. Wood。

特别感谢

国防部高级研究计划署的 Duane Adams，Tom Adams，伊利诺伊大学香槟分校的 Sarita Adve，Anant Agarwal，罗彻斯特大学的 Dave Albonesi，Mitch Alsup，Howard Alt，Dave Anderson，Peter Ashenden，David Bailey，国防部高级研究计划署的 Bill Bandy，康柏西部研究实验室的 Luiz

Barroso，Andy Bechtolsheim，C. Gordon Bell，Fred Berkowitz，IBM公司的John Best，Dileep Bhandarkar，BDTI公司的Jeff Bier，Mark Birman，David Black，David Boggs，JimBrady，Forrest Brewer，加州大学伯克利分校的Aaron Brown，康柏西部研究实验室的E. Bugnion，罗彻斯特大学的Alper Buyuktosunoglu，Mark Callaghan，Jason F. Cantin，Paul Carrick，Chen-Chung Chang，罗彻斯特大学的Lei Chen，Pete Chen，Nhan Chu，普林斯顿大学的Doug Clark，Bob Cmelik，John Crawford，Zarka Cvetanovic，得克萨斯大学奥斯汀分校的Mike Dahlin，Merrick Darley，DEC西部研究实验室的员工，John DeRosa，Lloyd Dickman，J. Ding，华盛顿大学的Susan Eggers，罗彻斯特大学的Wael El-Essawy，Mills公司的Patty Enriquez，Milos Ercegovac，Robert Garner，康柏西部研究实验室的K. Gharachorloo，Garth Gibson，Ronald Greenberg，Ben Hao，康柏公司的John Henning，威斯康星大学麦迪逊分校的Mark Hill，Danny Hillis，David Hodges，Google公司的Urs Hölzle，David Hough，Ed Hudson，伊利诺伊大学香槟分校的Chris Hughes，Mark Johnson，Lewis Jordan，Norm Jouppi，William Kahan，Randy Katz，Ed Kelly，Richard Kessler，Les Kohn，康柏计算机公司的John Kowaleski，Dan Lambright，Sun公司的Gary Lauterbach，Corinna Lee，Ruby Lee，Don Lewine，Chao-Huang Lin，国防部高级研究计划署的Paul Losleben，Yung-Hsiang Lu，国防部高级研究计划署的Bob Lucas，Ken Lutz，Intel伯克利研究实验室的Alan Mainwaring，Al Marston，罗格斯大学的Rich Martin，John Mashey，Luke McDowell，Trimedia公司的Sebastian Mirolo，Ravi Murthy，Biswadeep Nag，Sun公司的Lisa Noordergraaf，国防部高级研究计划署的Bob Parker，因特网研究中心的Vern Paxson，Lawrence Prince，Steven Przybylski，国防部高级研究计划署的Mark Pullen，Chris Rowen，Margaret Rowland，罗彻斯特大学的Greg Semeraro，Bill Shannon，Behrooz Shirazi，Robert Shomler，Jim Slager，克莱姆森大学的Mark Smotherman，华盛顿大学的SMT研究组，国防部高级研究计划署的Steve Squires，Ajay Sreekanth，Darren Staples，Charles Stapper，Jorge Stolfi，Peter Stoll，我们首次尝试编写本书时包容我们的斯坦福大学和加州大学伯克利分校的学生们，Bob Supnik，Steve Swanson，Paul Taysom，Shreekant Thakkar，新泽西理工学院的Alexander Thomasian，国防部高级研究计划署的John Toole，Trimedia公司的Kees A. Vissers，Willa Walker，David Weaver，EMC公司的Ric Wheeler，Maurice Wilkes，Richard Zimmerman。

John Hennessy
David Patterson

1 量化设计与分析基础

个人计算机是人类迄今创造的最强大工具，我认为这种说法并无偏颇。它们是通信工具，是创造工具，是可以由用户定制的工具。

——比尔·盖茨，
2004年2月24日

1.1 引言

自从第一台通用电子计算机问世以来，计算机技术在大约65年间取得了令人难以置信的发展。今天，花费不到500美元购买的一台便携计算机，在性能、主存储器和磁盘存储方面都要优于1985年花费100万美元购买的计算机。这种快速发展既得益于计算机生产技术的发展，也得益于计算机设计的创新。

生产技术的进步一直都相当稳定，而计算机体系结构的改进对这一快速发展的贡献就远没有那么稳定了。在电子计算机前25年的发展中，这两种力量都作出了巨大贡献，使计算机的性能每年提高大约25%。20世纪70年代后期出现了微处理器。依靠集成电路技术的进步，微处理器也使计算机性能进入快速发展期——每年大约增长35%。

这种高增长速度，再加上微处理器大量生产带来的成本优势，提高了微处理器业务在计算机行业内所占的份额。另外，计算机市场的两个重大变化也使新体系结构更容易在商业上获得成功。第一个重大变化是人们几乎不再使用汇编语言进行编程，从而降低了对目标代码兼容性的要求。第二个重大变化是出现了独立于厂商的标准化操作系统（比如UNIX和它的克隆版本Linux），降低了引入新体系结构的成本和风险。

正是由于这些变化，人们才有可能在20世纪80年代早期成功地开发了一组指令更为简单的新体系结构——RISC（精简指令集计算机）体系结构。设计人员在设计RISC计算机时，将主要精力投注在两种关键的性能技术上，即**指令级并行**的开发（最初是通过流水线，后来是通过多指令发射）和缓存的使用（最初采用一些很简单的形式，后来使用了更为复杂的组织与优化方式）。

基于RISC的计算机抬高了性能指标，过去的体系结构要么快速跟上，要么就被淘汰。Digital Equipment Vax未能跟上时代的脚步，所以被一种RISC体系结构替代。Intel则接受挑战，主要是在内部将80x86指令转换为类似于RISC的指令，使它能够采用许多最初由RISC设计倡导的新技术。20世纪90年代后期，晶体管的数目飞速增长，所以在转换更复杂x86体系结构时的硬件开销可以忽略不计。在低端应用中，比如在手机中，由于x86转换开销所带来的功耗与硅面积成本，促使一种RISC体系结构逐渐成为主流，这就是ARM。

图1-1表明，体系结构与组织方式的发展一起促成了计算机性能以超过50%的年增长率持续增长17年，这一速率在计算机行业内是空前的。

20世纪的这一飞速发展共有四重效果。第一，它显著增强了可供计算机用户使用的功能。对许多应用来说，当今性能最高的微处理器比不到10年前的超级计算机还要优秀。

2

第二，性价比的这种大幅提高导致了新型计算机的出现。20世纪80年代出现了个人计算机和工作站，这是因为有微处理器可供使用。在刚刚过去的10年里，人们见证了智能手机和平板电脑的崛起，许多人把它们作为自己的主要计算平台，代替了个人计算机。这些移动客户端设备越来越多地通过因特网来访问包含数万个服务器的仓库，这些仓库的设计使它们看起来就像单个巨型计算机一样。

第三，根据摩尔定律的预测，半导体制造业的持续发展已经使基于微处理器的计算机在整个计算机设计领域中占据了主导地位。传统上使用现成逻辑电路或门阵列制造的小型机已经被使用微处理器制造的服务器所取代，甚至大型计算机和高性能的超级计算机也都是由微处理器组合而成。

3

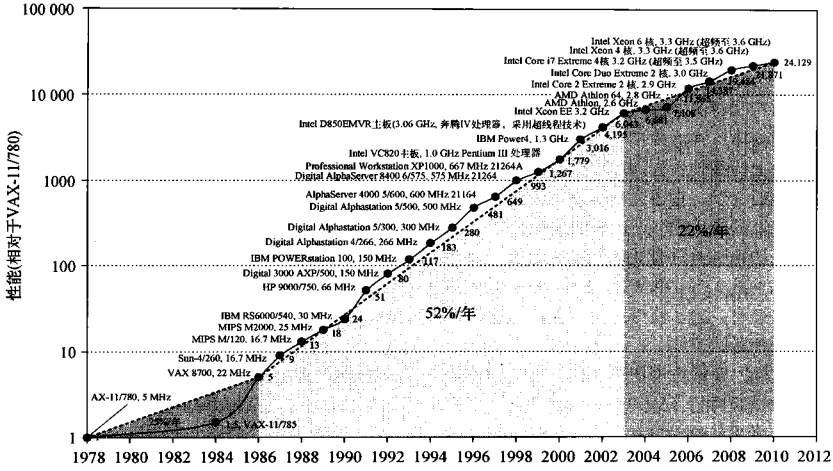


图 1-1 **自 20 世纪 70 年代后期以来处理器性能的增长。**这个图表绘制了相对于 VAX 11/780 的性能曲线，测试数据由 SPEC 基准测试测得（见 1.8 节）。在 20 世纪 80 年代中期之前，处理器性能的增长主要由技术驱动，平均大约每年增长 25%。在此之后的年增长速度为大约 52%，这一高速增长应当归功于更高级的体系结构和组织思想。这样持续发展到 2003 年时，如果一直以 25%的速度增长，处理器性能要比实际性能低 25 倍。2003 年之后，功耗和可用指令级并行的限制减慢了单核处理器性能的增长速度，年增长速度不超过 22%，到 2010 年，如果仍然保持每年 52%的增长速度，处理器性能要比实际性能高出大约 5 倍。（从 2007 年起，单个芯片上的核心数目每年都在增加，最快速的 SPEC 性能测试已经启用了自动并行，所以很难再测试单核处理器的速率。这些结果仅限于单套接字系统，以降低自动并行造成的影响。）图 1-4 给出了时钟速率在上述三个时期的增长速度。由于 SPEC 这些年也发生了变化，所以在评估新机器的性能时，对测试数据进行了换算，换算因数与两种不同 SPEC 版本（比如 SPEC89、SPEC92、SPEC95、SPEC2000 和 SPEC2006）的性能有关

硬件方面的上述创新导致了计算机设计的复兴，既强调体系结构方面的创新，也重视技术改进的高效运用。实际增长速度中已经包含了这两方面的因素，所以到 2003 年，高性能微处理器大约要比仅依靠技术改进（包含电路设计的改进）快 7.5 倍；实际年增长速度为 52%，仅依靠技术改进的年增长速度为 35%。

这一硬件复兴还有第四个影响，那就是对软件开发的影响。自 1978 年以来，硬件性能提高了 25 000 倍（见图 1-1），这样就允许今天的程序员以性能换取生产效率。今天，绝大多数的编程是使用诸如 Java 和 C#之类的托管编程语言来完成的，代替了以提高性能为目的的 C 语言和 C++语言。此外，Python 和 Ruby 之类的脚本语言（它们的生产效率可能更高），连同 Ruby Rails 之类的编程框架，也正在日益普及。为了保持生产效率并尝试缩小性能差距，采用即时（Just-In-Time）编译器和跟踪编译（Trace-based Compiling）的解释器正在取代过去的传统编译器和链接器。软件部署也在发生变化，因特网上使用的“软件即服务”（Software as a Service，SaaS）取代了必须在本地计算机上安装和运行的盒装光盘套装（shink-wrapped）软件。

应用程序的本质也在变化。语言、音效、图像、视频正在变得愈加重要，响应时间对于提供良好的用户体验非常关键，使其保持在可预测范围之内也同样变得愈加重要。Google Goggles就是一个激动人心的例子。当用户拿起手机，把手机上的镜头对准物体时，这个应用程序可以通过因特网将图像无线传送到一个仓库级计算机（warehouse-scale computer）上，这个计算机会识别物体，告诉用户它的一些有趣信息。它可以将物体上的文字翻译为另一种语言；读取书籍封面上的条码，告诉用户这本书在网上是否有售，价格多少；或者，如果用户摇动镜头，拍摄周围的全景，它还会告诉用户附近有什么商店以及它们的网站、电话号码和方位。

遗憾的是，图1-1还表明这一长达17年的硬件复兴结束了。从2003年开始，由于风冷芯片最大功耗和无法有效地开发更多指令级并行这两大孪生瓶颈，单处理器的性能提高速度下降到每年不足22%。事实上，Intel在2004年取消了自己的高性能单核处理器项目，转而和其他公司一起宣布：为了获得更高性能的处理器，应当提高一个芯片上集成的核心数目，而不是加快单核处理器的速度。

这是一个标志着历史性转折的里程碑信号，处理器性能的提高从单纯依赖指令级并行（ILP）转向**数据级并行**（DLP）和**线程级并行**（TLP），ILP是本书前三个版本的重点，第4版开始重点介绍DLP和TLP，第5版进一步加以扩展。这一版还增加了仓库级计算机和**请求级并行**（RLP）的内容。编译器和硬件都是隐式开发ILP的，不会引起程序员的注意，而DLP、TLP和RLP则是显式并行的，需要调整应用程序的结构才能开发显式并行。在某些情况下，这一调整比较容易，但在大多数情况下，它会成为程序员的主要新增负担。

本书主要讨论使上世纪飞速增长成为可能的体系结构思想和相关的编译器改进、这些急剧变化的原因，以及21世纪体系结构思想、编译器和解释器面对的挑战和一些富有前景的方法。本书的核心内容是一种量化的计算机设计与分析方法，所采用的工具是对程序的经验观察、试验和模拟。本书中就反映了这种计算机设计风格与方法。本章的目的是为后续各章及附录奠定量化基础。

编写本书的目的不只是为了解释这种设计风格，还希望激励读者为这一进程作出贡献。这种量化方法对过去的隐式并行计算机是有效的，我们相信这种方法对未来的显式并行计算机也同样有效。

1.2 计算机的分类

由于上述变化，我们在新世纪里看待计算、计算应用程序和计算机市场的观点也发生了巨大变化。自个人计算机诞生以来，我们还没有见到过计算机在外观和操作方式上发生如此之大的变化。计算机使用方式的变化推动形成了5种不同的计算市场，每一种都有自己不同的应用、需求和计算技术。表1-1总结了主流计算环境及其重要特征。

表1-1 5个主流计算类别及其系统特性汇总

特 征	个人移动设备（PMD）	台 式 机	服 务 器	集群/仓库级计算机	嵌 入 式
系统价格	100～1000美元	300～2500美元	5000～10 000 000美元	100 000～200 000 000美元	10～100 000美元
微处理器价格	10～100美元	50～500美元	200～2000美元	50～250美元	0.01～100美元
关键的系统设计问题	成本、能耗、媒体性能、响应率	性价比、能耗、图形性能	吞吐量、可用性、可扩展性、能耗	性价比、吞吐量、能耗均衡性	价格、能耗、应用的特有性能

* 2010年的销售量包括大约18亿个PMD（90%为手机）、3.5亿个台式PC和2000万个服务器。嵌入式处理器的总销售量接近190亿。2010年共交付了61亿个基于ARM技术的芯片。注意服务器和嵌入器系统的系统价格跨度极大，包含了从USB密钥到网络路由器在内的各种设备。对于服务器，这一变化范围主要是因为需要超大规模多处理器系统来完成高端事务处理。

5

1.2.1 个人移动设备

个人移动设备（PMD）是指一类带有多媒体用户界面的无线设备，比如手机、平板电脑等。由于整个产品的零售价格为数百美元，所以成本成为一个关键因素。尽管经常会因为使用电池而需要强调能效，但由于需要使用相对便宜的外壳（由塑料或陶瓷制成），而且缺少冷却风扇，所以也限制了总功耗。我们将在1.5节更详细地研究能耗和功率。PMD上的应用程序经常是基于Web应用、面向媒体的，比如上面提到的Google Goggles。能耗与尺寸要求决定了要采用闪存而不是磁盘来作为存储方式（第2章）。

响应性能和可预测性能是多媒体应用程序的关键特性。**实时性能**需求是指应用程序的一个程序段有一个确定的最大执行时间。例如，在PMD上播放一段视频时，由于处理器必须在短时间内接收和处理下一个视频帧，所以对每个视频帧的处理时间是有限的。某些应用程序中还有一个更具体的需求：当超出某一最大时间时，会限制一项特定任务的平均时间和实例数目。如果仅仅是偶尔违反一个事件的时间约束条件（而非过多地发生这种情况），就可以采用有时被称为**软实时**的方法。实时性能往往严重依赖于具体的应用程序。

许多PMD应用程序中还有其他一些关键特性：需要将存储器占用减至最少，需要高效利用能量。电池容量和散热问题都需要提高能耗效率。存储器可能在系统成本中占有很大的比例，在这种情况下，存储器优化是非常重要的。由于应用程序已经决定了数据规模，所以重视存储器用量其实就是要重视代码规模。

1.2.2 桌面计算

以资金而论，一级市场（可能仍然是最大的市场）是桌面计算市场。桌面计算覆盖了从低端到高端的整个产品范围，既有售价不到300美元的低端上网本，也有售价可能达2500美元的高端高配工作站。从2008年开始，每年生产的桌面计算机中有一半以上是由电池供电的笔记本计算机。

在整个价格与性能范围内，桌面计算机市场都有优化性价比的趋势。系统的性能（主要以计算性能和图形性能来衡量）和价格对这个市场中的客户来说是最重要的，因此对计算机架构师也是最重要的。结果，最新、最高性能的微处理器和低成本微处理器经常首先出现在桌面系统中。（1.6节讨论了影响计算机成本的问题。）

尽管以Web为中心的互动应用日益增多，为性能评估带来了新的挑战，但根据应用程序基准测试还是能够较好地刻画桌面计算的特征。

6

1.2.3 服务器

自20世纪80年代开始转向桌面计算机以来，服务器的角色逐渐变为提供更大规模、更可靠的文件和计算服务。这些服务器已经代替传统的大型机，成为大规模企业计算的中枢。

对服务器而言，所强调的特征不同于桌面计算机。首先，可用性是至关重要的。（我们将在1.7节讨论可用性。）考虑一下运行银行ATM机或者航班订票系统的服务器。由于这些服务器必须每周7天、每天24小时不间断工作，所以此类服务器系统发生故障时产生的灾难性后果要远比单个桌面计算机故障严重。表1-2估算了服务器应用程序因宕机所造成的收益成本。

服务器系统的第二个关键特征是可扩展性。服务器系统经常需要扩展，以满足其所支持服务的增长需求，或者对功能的增长需求。因此，服务器扩展计算容量、内存、存储器和I/O带宽的能力极为重要。

最后一个特征，服务器的设计应使其具有很高的吞吐能力。也就是说，服务器的整体性能（每分钟处理的事务数或者每秒提供的网页数）才是最重要的。尽管对单个请求的响应速度依然重要，但总体效率和成本效益（由单位时间内能够处理的请求数决定）才是大多数服务器的关键度量。我们将在1.8节再来讨论如何评估不同计算环境类型的性能问题。

表1-2 假定有3种不同可用性级别且宕机时间均匀分布，通过分析宕机成本（直接收入损失），给出由于系统不可用而带来的成本（四舍五入至10万美元）

应用程序	每小时的宕机成本（千美元）	不同宕机率造成的年度亏损（百万美元）		
		1%（87.6小时/年）	0.5%（43.6小时/年）	0.1%（8.8小时/年）
经纪业务	6450	565	283	56.5
信用卡授权	2600	228	114	22.8
包裹运输服务	150	13	6.6	1.3
家庭购物频道	113	9.9	4.9	1.0
目录销售中心	90	7.9	3.9	0.8
航空订票中心	89	7.9	3.9	0.8
手机服务激活	41	3.6	1.8	0.4
网络在线费用	25	2.2	1.1	0.2
ATM服务费	14	1.2	0.6	0.1

* 数据来源：Kembel[2000]，由Contingency Planning Research收集、分析。

1.2.4 集群/仓库级计算机

软件即服务（SaaS）应用（比如搜索、社交网络、视频分享、多人游戏、在线销售等）的发展已经推动了一类被称为**集群**的计算机的发展。集群是指一组桌面计算机或服务器通过局域网连接在一起，运转方式类似于一个更大型的计算机。每个节点都运行自己的操作系统，节点之间使用网络协议进行通信。最大规模的集群称为**仓库级计算机**（WSC），它们的设计方式使数万个服务器像一个服务器一样运行。第6章详细介绍了这类超大型计算机。

WSC是如此之大，因而性价比和功耗非常关键。第6章将解释一个价值9000万美元的仓库级计算机，其成本中有80%与计算机内部的功耗和冷却技术有关。这些计算机本身和网络设备另外需要7000万美元，每隔几年就必须更换一次。在购买这样大规模的计算设备时，一定要非常精明，因为性价比提高10%就意味着可以节省700万美元（7000万美元的10%）。

WSC 与服务器的相通之处在于它们都非常看重可用性。例如，Amazon.com 在 2010 年第四季度的销售额为 130 亿美元。一个季度大约有 2200 个小时，每小时的平均收入差不多是 600 万美元。在圣诞节购物的高峰时间，潜在损失可能要多出许多倍。第 6 章将会解释，WSC 与服务器的区别在于，WSC 以很多廉价组件为构建模块，依靠软件层来捕获和隔离在这一级别进行计算时发生的许多故障。注意，WSC 的可扩展性是由连接这些计算机的局域网实现的，而不是像服务器那样，通过集成计算机硬件来实现。

超级计算机与 WSC 的相通之处在于它们都非常昂贵，需要花费数千万美元，但超级计算机的不同之处在于它强调浮点性能，会运行大型的、通信密集的批程序，这些程序可能会一次运行几个星期。这种紧耦合性决定了超级计算级要使用非常快速的内部网络。而 WSC 则不同，它重视互动应用程序、大规模存储、可靠性和很高的因特网带宽。

1.2.5 嵌入式计算机

嵌入式计算机在日用电器中随处可见。微波炉、洗衣机、大多数打印机、大多数网络交换机和所有汽车中都有简单的嵌入式微处理器。

PMD 中的处理器经常被看作是嵌入式计算机，但我们仍然把它们看作一个不同类别，这是因为 PMD 是一些可以运行外部开发软件的平台，它们与桌面计算机有许多共同特征。其他嵌入式设备在硬件和软件复杂性方面都有很大的限制。我们以能否运行第三方软件作为区分嵌入式和非嵌入式计算机的分界线。

嵌入式计算机的处理能力和成本差别最大。它们既包括只需要 0.1 美元的 8 位和 16 位处理器，也有可以每秒执行 1 亿条指令、价格低于 5 美元的 32 位微处理器，还有用于网络交换机的高端处理器，它们的售价高达 100 美元，每秒可以执行数十亿条指令。尽管嵌入式计算市场中的计算能力相差很大，但价格仍然是此类计算机设计的关键因素。性能要求当然是的确存在的，但主要目标通常是以最低价格满足性能需要，而不是以更高的价格来获得更高的性能。

8

本书的大多数内容适用于嵌入式处理器的设计、使用和性能，无论是现成的微处理器，还是与其他专用硬件组装在一起的微处理器核心，都是适用的。实际上，本书第 3 版还包含了一些嵌入式计算示例，用来演示每章的思想。

遗憾的是，大多数读者对第 3 版中的示例感到不满意，这是因为计算机的量化设计与评估需要数据来推动，而嵌入计算领域的相关数据还不够充足（参见 1.8 节中 EEMBC 的挑战）。因此，我们现在只给出定性描述，这种方式与本书其余部分不是特别一致。有鉴于此，在上一版和这一版中，我们将嵌入计算机的相关材料合并到附录 E 中。我们认为把它们放在一个单独的附录中，既便于在正文中流畅地表达思想，又可以让读者了解这些不同需求是如何影响嵌入式计算的。

1.2.6 并行度与并行体系结构的分类

在所有 4 个计算机类别中，多种级别的并行度现在已经成为计算机设计的推动力量，而能耗和成本则是主要约束条件。应用程序中主要有以下两种并行。

(1) **数据级并行**（DLP），它的出现是因为可以同时操作许多数据项。

(2) **任务级并行**（TLP），它的出现是因为创建了一些能够单独处理但大量采用并行方式执行的工作任务。

计算机硬件又以如下4种主要方式来开发这两种类型的应用并行。

(1) **指令级并行**在编译器的帮助下，利用流水线之类的思想适度开发数据级并行，利用推理执行之类的思想以中等水平开发数据级并行。

(2) **向量体系结构和图形处理器**（GPU）将单条指令并行应用于一个数据集，以开发数据级并行。

(3) **线程级并行**在一种紧耦合硬件模型中开发数据级并行或任务级并行，这种模型允许在并行线程之间进行交互。

9 (4) **请求级并行**在程序员或操作系统指定的大量去耦合任务之间开发并行。

硬件支持数据级并行和任务级并行的这4种方式可以追溯到50年前。Michael Flynn [1966] 在20世纪60年代研究并行计算工作量时，提出了一种简单的分类方式，我们今天仍在使用这种分类的缩写。他研究了多处理器最受约束组件中的指令流及数据流的并行，并据此将所有计算机分为以下4类。

(1) **单指令流、单数据流**（SISD）：这个类别是单处理器。程序员把它看作标准的顺序计算机，但可以利用指令级并行。第3章介绍了采用ILP技术（比如超标量和推理执行）的SISD体系结构。

(2) **单指令流、多数据流**（SIMD）：同一指令由多个使用不同数据流的处理器执行。SIMD计算机开发**数据级并行**，对多个数据项并行执行相同操作。每个处理器都有自己的数据存储器（也就是SIMD中的MD），但只有一个指令存储器和控制处理器，用来提取和分派指令。第4章介绍DLP和3种开发DLP的不同体系结构：向量体系结构、标准指令集的多媒体扩展、GPU。

(3) **多指令流、单数据流**（MISD）：到目前为止，还没有这种类型的商用多处理器，但包含这种类型之后，这种简单的分类方式变得更完整。

(4) **多指令流、多数据流**（MIMD）：每个处理器都提取自己的指令，对自己的数据进行操作，它针对的是任务级并行。一般来说，MIMD要比SIMD更灵活，所以适用性也更强，但它要比SIMD更贵一些。例如，MIMD计算还能开发数据级并行，当然，其开销可能要比SIMD处理器高一些。这种开销是指粒度大小要足够大，以便高效地开发并行度。第5章介绍紧耦合MIMD体系结构，由于多个互相协作的线程是并行操作的，所以它开发了**线程级并行**。第6章介绍开发**请求级并行**的松耦合MIMD体系结构（具体来说，就是**集群**和**仓库级计算器**），在这种情况下，可以很自然地并行执行许多独立任务，几乎不需要通信和同步。

这种分类模型很粗略，许多并行处理器是SISD、SIMD和MIMD的混合类型。不过，我们
10 可以用它为本书将要介绍的计算机设计空间设定一个框架。

1.3 计算机体系结构的定义

计算机设计人员面对的是一个非常复杂的任务：判断哪些属性对于一种新计算机来说是至关重要的，然后在设计这种计算机时使其性能和能耗效益达到最佳，同时还要满足成本、功耗和可用性约束条件。这项任务包括许多方面：指令集设计、功能组织、逻辑设计、实现方式。实现方式可能包括集成电路设计、包装、电池和冷却。为使设计方案达到最优效果，设计人员需要熟悉从编译器、操作系统到逻辑设计与包装等广泛技术。

几年前，**计算机体系结构**一词通常仅包括指令集设计。计算机设计的其他方面称为**实现**，言下之意通常就是实现方式不太重要，或者说没有什么挑战性。

我们认为这种观点是错误的。架构师的工作远不止指令集设计一项，在项目其他方面遇到的技术障碍很可能比指令集设计中遇到的障碍更具挑战性。在介绍计算机架构师的更多挑战之前，先来快速回顾一下指令集体系结构。

1.3.1 指令集体系结构：计算机体系结构的近距离审视

我们在本书中用**指令集体系结构**（ISA）一词来指代程序员可以看到的实际指令集。ISA 的作用相当于区分软件和硬件的界限。在下面对 ISA 的快速回顾中，将使用 80x86、ARM 和 MIPS 的例子从 7 个方面来介绍 ISA。附录 A 和附录 K 更详细地介绍了这 3 种 ISA。

(1) **ISA 分类**。现今几乎所有的 ISA 都划分到通用寄存器体系结构中，在这种体系结构中，操作数或者是寄存器，或者是存储器地址。80x86 有 16 个通用寄存器和 16 个通常存入浮点数据的寄存器，而 MIPS 则有 32 个通用寄存器和 32 个浮点寄存器（见表 1-3）。这一类别有两种主流版本，一种是**寄存器-存储器 ISA**，比如 80x86，可以在许多指令中访问存储器；另一种是**载入-存储 ISA**，比如 ARM 和 MIPS，它们只能用载入或存储指令来访问存储器。所有最新 ISA 都采用载入-存储版本。

表1-3 MIPS寄存器和使用规范

名　称	编　号	用　途	在调用之间是否保留
$zero	0	常量值0	不可用
$at	1	为汇编程序保存的临时寄存器	否
$v0-$v1	2～3	函数返回值和表达式计算结果的值	否
$a0-$a3	4～7	实参	否
$t0-$t7	8～15	临时变量	否
$s0-$s7	16～23	已保存的临时变量	是
$t8-$t9	24～25	临时变量	否
$k0-$k1	26～27	为操作系统内核保留	否
$gp	28	全局指针	是
$sp	29	栈指针	是
$fp	30	帧指针	是
$ra	31	返回地址	是

* 除了 32 个通用寄存器（R0-R31）之外，MIPS 还有 32 个浮点寄存器（F0-F31），可以保存一个 32 位单精度数或一个 64 位双精度数。

(2) **存储器寻址**。几乎所有桌面计算机和服务器计算机（包括 80x86、ARM 和 MIPS）都使用字节寻址来访问存储器操作数。有些体系结构（像 ARM 和 MIPS）要求操作对象必须是对齐的。一个大小为 s 的对象，其字节地址为 A，如果 A mod s=0，则对这个对象的访问是对齐的。80x86（见附录图 A-2）不需要对齐，但如果操作数是对齐的，访问速度通常会更快一些。

(3) **寻址模式**。除了指定寄存器和常量操作数之外，寻址模式还指定了一个存储器对象的地址。MIPS 寻址模式为：寄存器（寻址）、立即数（寻址）和位移量（寻址）。立即数寻址用于常数寻址，在位移量寻址模式中，将一个固定偏移量加到寄存器，得出存储器地址。80x86 支持上述 3 种模式，再加上位移量的 3 种变化形式，即：无寄存器（绝对数）、两个寄存器（用位移量进行基址寻址）、两个寄存器，其中一个寄存器的内容乘以操作数的字节大小——用比例索引和位移量进行变址寻址。它与上述 3 种寻址方式类似，只是要减去位移量字段，加上寄存器间

接寻址、基址寻址和变址寻址。ARM拥有3种MIPS寻址模式再加上相对PC（程序计数器）的寻址方式、两个寄存器之和，还有一种方式也是两个寄存器之和，但其中一个寄存器的内容要乘以操作数的字节大小。它还有自动递增寻址和自动递减寻址：计算得到的地址会被放在用于构造该地址的一个寄存器中，替代其中的内容。

(4) **操作数的类型和大小**。和大多数ISA类似，80x86、ARM和MIPS支持的操作数大小为8位（ASCII字符）、16位（Unicode字符或半个字）、32位（整数或字）、64位（双字或长整型）以及IEEE 754浮点数，包括32位（单精度）和64位（双精度）。80x86还支持80位浮点（扩展双精度）。

(5) **操作指令**。常见的操作类别为：数据传输指令、算术逻辑指令、控制指令（下面进行讨论）和浮点指令。MIPS是一种简单的、易于实现流水化的指令集体系结构，它是2011年采用RISC体系结构的代表。表1-4总结了MIPS ISA。80x86的操作指令集要丰富得多，大得多（参见附录K）。

12

表1-4　MIPS64中的部分指令

指令类型/操作码	指令含义
数据传输	**在寄存器和存储器之间，或者在整数和FP或特殊寄存器之间移动数据；唯一的存储器寻址模式是16位位移量加上GPR的内容**
LB、LBU、SB	载入字节、载入无符号字节、存储字节（至/自整数寄存器）
LH、LHU、SH	载入半字、载入无符号半字、存储半字（至/自整数寄存器）
LW、LWU、SW	载入字、载入无符号字、存储字（至/自整数寄存器）
LD、SD	载入双字、存储双字（至/自整数寄存器）
L.S、L.D、S.S、S.D	载入SP浮点、载入DP浮点、存储SP浮点、存储DP浮点
MFC0、MTC0	在GPR与特殊寄存器之间复制数据
MOV.S、MOV.D	将一个SP或DP FP寄存器复制到另一个FP寄存器
MFC1、MTC1	在FP寄存器与整数寄存器之间复制32位
算术/逻辑	**对GPR中的整数或逻辑数据进行操作；带符号算术运算溢出时进行陷阱捕获**
DADD、DADDI、DADDU、DADDIU	加，加立即数（所有立即数为16位），有符号和无符号
DSUB、DSUBU	减，有符号和无符号
DMUL、DMULU、DDIV、DDIVU、MADD	乘和除，有符号和无符号；乘-加；所有运算的操作数和结果都是64位数值
AND、ANDI	与，和立即数相与
OR、ORI、XOR、XORI	或，和立即数求或，异或，和立即数求异或
LUI	载入高位立即数；将立即数载入到寄存器的32～47位，然后进行符号扩展
DSLL、DSRL、DSRA、DSLLV、DSRLV、DSRAV	移位：立即数形式（DS__）和变量形式（DS__V）；移位为左逻辑移位、右逻辑移位、右算术移位
SLT、SLTI、SLTU、SLTIU	若小于操作数则置位、若小于立即数则置位、有符号和无符号
控制	**控制分支和跳转，相对于PC寄存器或通过寄存器控制**
BEQZ、BNEZ	GPR等于/不等于0时转移、相对于PC+4偏移16位偏移量
BEQ、BNE	GPR相等/不相等时转移、相对于PC+4转移16位偏移量
BC1T、BC1F	测试FP状态寄存器中的对比位，并转移；相对于PC+4转移16位偏移量
MOVN、MOVZ	如果第三个GPR为负数/零，则将第一个GPR复制到第二个GPR
J、JR	跳转至与PC+4偏移26位偏移量的位置（J）、跳转至寄存器中的目标位置（JR）

（续）

指令类型/操作码	指令含义
JAL、JALR	跳转和链接：将PC+4保存在R31中，目标为相对于PC（JAL）或寄存器（JALR）
TRAP	转移到操作系统的一个向量地址
ERET	从异常中返回用户代码，恢复用户模式
浮点	**对DP和SP格式执行FP操作**
ADD.D、ADD.S、ADD.PS	DP、SP数相加，一对SP数相加
SUB.D、SUB.S、SUB.PS	DP、SP数相减，一对SP数相减
MUL.D、MUL.S、MUL.PS	DP、SP浮点数相乘，一对SP数相乘
MADD.D、MADD.S、MADD.PS	DP、SP浮点数相乘加，一对SP数相乘加
DIV.D、DIV.S、DIV.PS	DP、SP浮点数相除，一对SP数相除
CVT._.._	转换指令：CVT.x.y从类型x转换为类型y，其中x和y为L（64位整数），W（32位整数），D（DP）或S（SP）。两个操作数都是FRP
C._.D、C.__.S	DP和SP对比："__"=LT，GT，LE，GE，EQ，NE，在FP状态寄存器中置位

* SP=单精度、DP=双精度附录 A 给出了有关 MIPS64 的更多详细信息。对数据而言，最高有效位编号为 0，最低有效位为 63。

13

(6) **控制流指令**。几乎所有 ISA，包括上述三种在内，都支持条件转移、无条件跳转、过程调用和返回。所有这三种都使用相对于 PC 的寻址方式，其中的分支地址由一个地址字段指定，该地址将被加到 PC。这三种 ISA 之间有一些微小的区别。MIPS 条件分支（BE、BNE 等）检验寄存器中的内容，而 80x86 和 ARM 分支测试条件代码位，这些位是在执行算术/逻辑运算时顺带置位的。ARM 和 MIPS 过程调用将返回地址放在一个寄存器中，而 80x86 调用（CALLF）将返回地址放在存储器中的一个栈内。

(7) **ISA 的编码**。有两种基本的编码选择：**固定长度**和**可变长度**。所有 ARM 和 MIPS 指令的长度都是 32 位，从而简化了指令译码。图 1-2 给出了 MIPS 指令格式。80x86 编码为可变长度，变化范围为 1～18 个字节。与固定长度的指令相比，可变长度的指令可以占用较少的空间，所以为 80x86 编译的程序通常要小于为 MIPS 编译的相同程序。注意，上面提到的编码选择会影响将指令转换为二进制编码的方式。例如，由于寄存器字段和寻址模式字段可以在一条指令中出现许多次，所以寄存器的数目和寻址模式的数目都对指令的大小具有显著影响。（注意，ARM 和 MIPS 后来都进行了扩展，支持长 16 位的指令，以便缩小程序规模，这两种扩展分别叫做 Thumb 或 Thumb-2、MIPS16。）

14

除了 ISA 设计方面的挑战之外，如果指令集之间的区别很小，如果存在不同的应用领域，那计算机架构师面对的其他挑战将更加严重。因此，从本书上一版开始，除了这里给出的快速回顾之外，还在附录中提供了大量有关指令集的材料（参见附录 A 和网站上的附录 K）。

本书以 MIPS64 的子集作为 ISA 的例子，是因为它既是网络领域的主导 ISA，又是前面提到的 RISC 体系结构的出色示例，ARM（Advanced RISC Machine，高级 RISC 机器）是 RISC 体系结构的最流行示例。2010 年出产的芯片中有 61 亿个采用了 ARM 处理器，与采用 80x86 处理器的芯片相比，大约是其 20 倍。

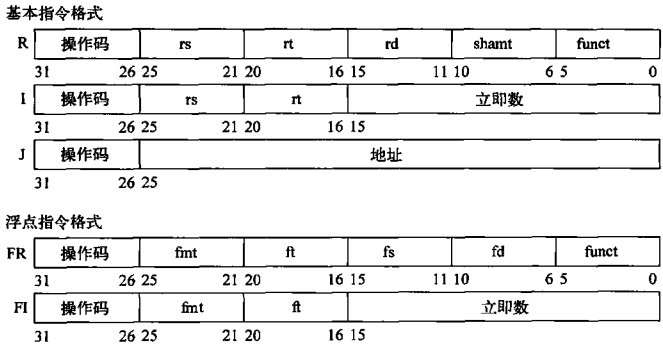


图 1-2 MIPS64 指令集体系结构格式。所有指令的长度都是 32 位。R 格式用于整数寄存器至寄存器操作，比如 DADDU、DSUBU 等。格式 I 用于数据传送、分支和立即数指令，比如 LD、SD、BEQZ 和 DADDI。格式 J 用于跳转指令，FR 格式用于浮点操作，FI 格式用于浮点分支

1.3.2 真正的计算机体系结构：设计满足目标和功能需求的组成和硬件

计算机的实现包括两个方面：组成和硬件。**组成**一词包含了计算机设计的高阶内容，比如存储器系统，存储器互连，设计内部处理器或 CPU（中央处理器——算术、逻辑、分支和数据输送功能都在这里实现）。有时也使用**微体系结构**一词来代替“组成”。例如，AMD Opteron 和 Intel Core i7 是两个指令集体系结构相同但组成不同的处理器。这两种处理器都实现 x86 指令集，但它们的流水线和缓存组成有很大不同。

由于单个微处理器上开始采用多个处理器，所以人们开始使用**核心**一词来称呼处理器。人们一般不说“多处理器微处理器”，而是使用“多核”。由于现今几乎所有芯片都有多个处理器，所以人们不怎么使用中央处理器（或 CPU）一词了。

硬件是指一个计算机的具体实现，包括计算机的详尽逻辑设计和封装技术。同一系列的计算机通常具有相同的指令集体系结构和几乎相同的组成，但在具体硬件实现方面有所不同。例如，Intel Core i7（见第 3 章）和 Intel Xeon 7560（见第 5 章）基本相同，但提供不同的时钟速率和不同的存储器系统，Xeon 7560 更适用于服务器计算机。

在本书中，**体系结构**涵盖了计算机设计的所有三个方面：指令集体系结构、组成或微体系结构、硬件。

计算机架构师设计的计算机必须满足功能需求，并达到价格、功耗、性能和可用性指标。表 1-5 总结了在设计新计算机时要考虑的要求。通常，架构师还必须判断有什么样的功能要求，其中哪一项可能是主要任务。需求可能是由市场驱动的特定功能。应用软件决定了计算机的使用方式，从而经常会推动特定功能需求的选择。如果存在大量为一特定指令集体系结构设计的软件，那架构师可能会决定：新计算机应当实现这种已有指令集。如果某类应用程序拥有庞大的市场，那可能会鼓励设计人员整合一些需求，使计算机在这一市场上具有更强的竞争力。后面章节将深入研究大量此类需求和功能。

表 1-5 架构师面对的一些最重要功能需求汇总

功能需求	应当具备或支持的典型特性
应用领域	**计算机的目标**
个人移动设备	一系列任务的实时性能，包括图形、视频和音频的交互性能；能效（第2、3、4、5章，附录A）
通用桌面计算机	一系列任务的均衡性能，包括图形、视频和音频的交互性能；能效（第2、3、4、5章，附录A）
服务器	支持数据库和事务处理，可靠性和可用性的增强，支持可伸缩性（第2、5章，附录A、D、F）
集群/仓库级计算机	许多独立任务的吞吐量性能，存储器的纠错功能，能耗均衡（第2、6章，附录F）
嵌入式计算	通常需要对图形或视频提供特殊支持（或者其他专用扩展），可能需要功耗限制和电源控制，实时约束（第2、3、5章，附录A、E）
软件兼容级别	**决定计算机的现有软件数目**
在编程语言级别	对设计人员来说最为灵活，需要新编译器（第3、5章，附录A）
目标代码或二进制代码兼容性	指令集体系结构完全确定（几乎没有灵活性），但不需要在软件或端口程序中进行投入（附录A）
操作系统需求	**支持选定OS所需要的特性（第2章、附录B）**
地址空间的大小	非常重要的特性（第2章），可能会限制应用
存储器管理	为现在操作系统所必需，可能进行分页或分段（第2章）
保护	不同的操作系统和应用需要：分页或分段，虚拟机（第2章）
标准	**市场可能要求特定的标准**
浮点	格式和算法：IEEE 754标准（附录J），用于图形处理或信号处理的特殊算法
I/O接口	对于I/O设备：串行ATA，串行连接SCSI、PCI Express（附录D、F）
操作系统	UNIX、Windows、Linux、CISCO IOS
网络	支持不同网络：以太网、Infiniband（附录F）
编程语言	语言（ANSI C、C++、Java、Fortran）影响指令集（附录A）

* 左列描述需求的类别，右列给出特定示例，还包含讨论相应主题的章节和附录。

架构师还必须了解技术和计算机应用这两方面的重要趋势，因为这些趋势不仅会影响未来的成本，还会影响到体系结构的寿命。

16

1.4 技术趋势

一种指令集体系结构要取得成功，它的设计必须能够适应计算机技术的快速变化。毕竟，一种成功的新指令集体系结构可能要持续几十年——例如，IBM 大型机的核心已经使用了将近 50 年。一种成功的计算机可能会因为技术变革而延长寿命，架构师必须提前为此类技术变革做出应对计划。

要为计算机的发展作长远计划，设计人员必须了解实现技术的快速变化。以下 5 种实现技术是现代计算机实现不可或缺的，它们都在发生急剧的变化。

- **集成电路逻辑技术**。晶体管密度每年大约增加 35%，差不多每四年时间翻两番。晶片大小的增长速度要慢一些，也比较难以预测，每年在 10%到 20%之间。两者综合起来，一个芯片上的晶体管数目每年大约增长 40%～55%，或者说每 18～24 个月翻番。这就是人们熟悉的摩尔定律。器件的增长速度要慢一些，下面将进行讨论。
- **半导体 DRAM**（动态随机访问存储器）。由于大多数 DRAM 芯片主要是以 DIMM 模块形式交付的，所以很难跟踪芯片容量，因为 DRAM 制造商通常会同时提供几种容量产品，

以与DIMM容量相匹配。最近几年来，单个DRAM芯片的容量每年增加大约25%～40%，大约每2～3年翻一番。这一技术是主存储器的基础，我们将在第2章详细讨论。注意，从表1-6可以看出，这一增长速度一直在随着本书的反复再版而下降。由于高效制造更小型DRAM单元的难度持续增大，人们甚至在担心，在5~7年内会不会停止增长[Kim 2005]。第2章提到了其他几种技术，可以在DRAM碰到容量壁垒时取代它。

17

- **半导体闪存**（电可擦编程只读存储器）。这种永久性半导体存储器是PMD中的标准存储器件，普及率的迅速提高刺激了其容量的快速增加。最近几年，单个闪存芯片的容量以每年大约50%～60%的速度增长，大约每两年翻一番。2011年，每比特的闪存价格大约是DRAM价格的1/15～1/20。第2章将详细探讨闪存。
- **磁盘技术**。在1990年之前，磁盘密度每年提高大约30%，3年翻一番。之后每年增长60%，1996年的增速为100%。从2004年开始，已经回落到大约40%，每3年翻一番。每比特的磁盘价格大约是闪存价格的1/15～1/25。由于DRAM的增长速度缓慢，所以每比特的磁盘价格现在大约是DRAM的1/300～1/500。这一技术是服务器和仓库级存储的核心技术，附录D中详细讨论了这些变化趋势。
- **网络技术**。网络性能取决于交换机性能和传输系统的性能。附录F中详述了网络技术的趋势。

表1-6 DRAM容量增长速度随时间的变化

CA:AQA版本	年 份	DRAM增长速率（年）	DRAM容量的增长特征
1	1990	60%	每3年翻两番
2	1996	60%	每3年翻两番
3	2003	40%～60%	每3～4年翻两番
4	2007	40%	每2年翻一番
5	2011	20%～40%	每2～4年翻一番

* 前两个版本甚至将这一速率称为DRAM增长经验定律，因为从1977年的16Kbit DRAM到1996年的64Mbit DRAM，始终都是以这一速率增长的。由于大容量三维DRAM单元的制造难度，有人开始担心DRAM容量在5~7年之内是否会完全停止增长[Kim 2005]。

这些快速发展的技术左右着计算机设计的命运，由于计算速度的激增和技术的迅猛发展，一种计算机设计的生存周期可能只有3～5年。DRAM、闪存和磁盘等关键技术的发展如此之快，设计人员必须为应对这些变化作好打算。事实上，设计人员在设计时经常要考虑到下一代技术，因为他们知道，当一件产品开始大量交付时，下一代技术可能是最具成本效益的，或者可能拥有性能优势。一般来说，成本的下降速度与密度的增长速度大体相当。

尽管技术进步是连续性的，但只有当技术积累到一定程度，为新功能的出现作好准备时，才会产生跳跃性的、不连续的影响。比如，当MOS技术在20世纪80年代早期发展到可以在单片芯片上集成25 000到50 000个晶体管时，才使单片32位微处理器的制造成为可能。到20世纪80年代后期，一级缓存可以出现在一个芯片上。通过消除处理器内部以及处理器与缓存之间的芯片交叉，使成本效率和能耗效率的大幅提高成为可能。在技术发展到一定程度之前，这种设计就无法付诸实现。随着多核微处理器的出现以及每一代核心数目的增加，甚至服务器计算机也开始追求将所有处理器放在一片芯片上。这类技术门槛绝非罕见，对众多设计决策都有着重要影响。

1.4.1 性能趋势：带宽胜过延迟

在 1.8 节将会看到，带宽和吞吐量是指在给定时间内完成的总工作量，比如在进行磁盘传送时每秒完成的 MB 数。与之相对，延迟或响应时间是指一个事件从开始到完成所经历的时间，比如一次磁盘访问需要的毫秒数。图 1-3 绘制了微处理器、存储器、网络和磁盘等各项技术在产生里程碑式进步时，带宽与延迟的相对改进曲线。表 1-7 更详细地描述了这些示例和里程碑。 18

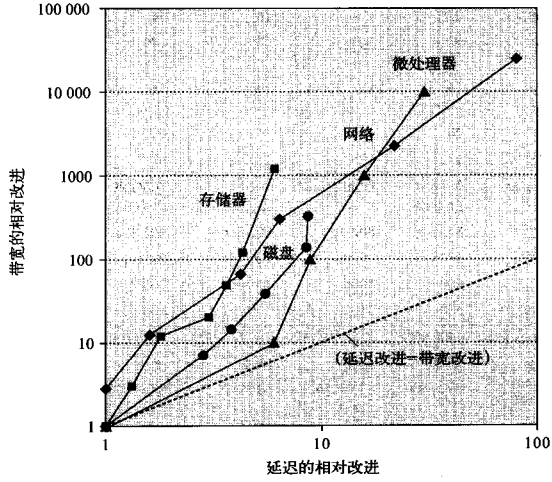


图 1-3 表 1-7 中各个带宽与延迟里程碑相对于第一个里程碑的双对数曲线。注意，延迟的改进为 6～8 倍，而带宽的改进为 300～25 000 倍。更新自 Patterson[2004]

表 1-7 微处理器、存储器、网络和磁盘在过去 20～40 年里的性能里程碑

微处理器	16位地址/总线、微编码	32 位 地址/总线、微编码	5级流水线、片上I&D缓存、FPU	2路超标量、64位总线	乱序3路超标量	乱序超流水线、片上L2缓存	多核OOO 4路片上L3缓存、Turbo
产品	Intel 80286	Intel 80386	Intel 80486	Intel Pentium	Intel Pentium Pro	Intel Pentium 4	Intel Core i7
年份	1982	1985	1989	1993	1997	2001	2010
晶片大小(平方毫米)	47	43	81	90	308	217	240
晶体管数	134 000	275 000	1 200 000	3 100 000	5 500 000	42 000 000	1 170 000 000
处理器数/芯片	1	1	1	1	1	1	4
管脚	68	132	168	273	387	423	1366
延迟(时钟周期)	6	5	5	5	10	22	14
总线宽度（位）	16	32	32	64	64	64	196

（续）

时钟频率(MHz)	12.5	16	25	66	200	1500	3333
带宽（MIPS）	2	6	25	132	600	4500	50 000
延迟（ns）	320	313	200	76	50	15	4
存储器模块	DRAM	分页模式DRAM	快速分页模式DRAM	快速分页模式DRAM	同步DRAM	双数据率SDRAM	DDR3 SDRAM
模块宽度（位）	16	16	32	64	64	64	64
年份	1980	1983	1986	1993	1997	2000	2010
Mbit/DRAM芯片	0.06	0.25	1	16	64	256	2048
晶片大小（平方毫米）	35	45	70	130	170	204	50
管脚/DRAM芯片	16	16	18	20	54	66	134
带宽（MB/s）	13	40	160	267	640	1600	16 000
延迟（ns）	225	170	125	75	62	52	37
局域网	以太网	快速以太网	Gbit以太网	10Gbit以太网	100Gbit以太网		
IEEE标准	802.3	803.3u	802.3ab	802.3ac	802.3ba		
年份	1978	1995	1999	2003	2010		
带宽（Mbit/s）	10	100	1000	10 000	100 000		
延迟（μs）	3000	500	340	190	100		
硬盘	3600 RPM	5400 RPM	7200 RPM	10 000 RPM	15 000 RPM	15 000 RPM	
产品	CDC WrenI 94145-36	希捷 ST41600	希捷 ST15150	希捷 ST39102	希捷 ST373453	希捷 ST3600057	
年份	1983	1990	1994	1998	2003	2010	
容量（GB）	0.03	1.4	4.3	9.1	73.4	600	
磁盘物理尺寸	5.25英寸	5.25英寸	3.5英寸	3.5英寸	3.5英寸	3.5英寸	
介质直径	5.25英寸	5.25英寸	3.5英寸	3.0英寸	2.5英寸	2.5英寸	
接口	ST-142	SCSI	SCSI	SCSI	SCSI	SAS	
带宽（MB/s）	0.6	4	9	24	86	204	
延迟（ms）	48.3	17.1	12.7	8.8	5.7	3.6	

* 微处理器里程碑是几代IA-32处理器，从16位总线、微编码80286到64位总线、多核、乱序执行、超流水线Core i7。存储器模块里程碑包括从16位宽、纯DRAM到64位宽双数据率第3版同步DRAM。以太网从10Mbit/S发展到100Gbit/s。磁盘里程碑是以旋转速度为标志的，从3600 RPM至15 000 RPM。每种情况都是指最佳带宽，延迟就是假定没有争用时执行简单操作的时间。更新自Patterson[2004]。

性能是区分微处理器和网络的主要指标，所以它们的进步最大：带宽为10 000～25 000倍，延迟为30～80倍。对容量和磁盘来说，容量通常要比性能更重要，所以容量的提高最多，但带宽发展了300～1200倍，仍然要远远高于延迟的6～8倍。

显然，在这些技术的发展过程中，带宽完胜延迟，而且这一趋势很可能会继续下去。一个简单的经验法则是：带宽的增长速度至少是延迟改进速度的平方。计算机的设计人员应当制定相应规划。

1.4.2 晶体管性能与连线的发展

集成电路的制造工艺是用特征尺寸来衡量的，所谓特征尺寸就是一个晶体管或一条连线在 x 方向或 y 方向的最小尺寸。特征尺寸已经从 1971 年的 10 微米下降到 2011 年的 0.032 微米；事实上，我们已经改变了单位，所以 2011 年的产品工艺被称为“32 纳米”，22 纳米的芯片也在研发之中。由于每平方毫米硅片上的晶体管数目是由单个晶体管的表面积大小决定的，所以当特征尺寸线性下降时，晶体管密度将以平方曲线上升。

19 ~ 20

不过，晶体管性能的增长就要更复杂了。当特征尺寸缩小时，器件在水平方向以平方关系缩小，在垂直方向上也会缩小。垂直方向上的缩小需要降低工作电压，以保持晶体管的正常工作和可靠性。缩放因子的这种组合效果使晶体管性能和工艺特征尺寸之间产生了复杂的相互关系。大致来说，晶体管性能的提高与特征尺寸的下降成线性关系。

当特征尺寸下降时，晶体管性能以线性提升，而晶体管数目却是以平方曲线增长，这一事实既是挑战，也是机遇，计算机架构师就是为此而生的！在微处理器发展的早期，借助晶体管密度的这种快速增长，微处理器迅速从 4 位发展到 8 位、16 位、32 位乃至 64 位。最近几年，密度的增长已经足以支持在一个芯片上引入多个处理器，支持更宽的 SIMD 单元、推理执行和缓存中的许多创新，在第 2、3、4、5 章将会讨论这些内容。

尽管晶体管的性能通常会随着特征尺寸的缩小而得到提升，但集成电路中的连线却不会如此。具体来说，一段连线的信号延迟与其电阻、电容的乘积成正比。当然，当特征尺寸缩小时，连线会变短，但单位长度的电阻和电容都会变差。这种关系很复杂，这是因为电阻和电容都依赖于工艺的具体细节、连线的几何形状、连线的负载，甚至取决于与其他结构的邻近程度。偶尔也会存在一些工艺方面的改进，比如铜的引入，这些改进会一次性地改善连线延迟性能。

一般来说，与晶体管性能相比，连线延迟方面的改进小得可怜，增加了设计人员面临的挑战。在过去几年里，除了功率耗散方面的限制之外，连线延迟已经成为大型集成电路的主要设计限制，往往比晶体管开关延迟还要关键。越来越多的时钟周期被消耗在信号在连线上的传播延迟上，而功率现在扮演的角色甚至比连线延迟还要重要。

1.5 集成电路中的功率和能耗趋势

今天，对于几乎所有类型的计算机来说，功率都是计算机设计人员面对的最大挑战。第一，必须将功率引入芯片，并进行分配，而现代微处理器仅仅为电源和接地就使用了数以百计的管脚和多个互连层。第二，功率以热的形式耗散，必须消除。

1.5.1 功率和能耗：系统观点

系统架构师或用户应当如何考虑性能、功率和能耗呢？从系统架构师的角度来看，共有三个主要关注事项。

21

第一，一个处理器需要的最大功率是多少？满足这一要求对于确保正确操作非常重要。例如，如果处理器试图汲取的功率大于电源系统能够提供的功率（也就是试图汲取的电流大于电源系统能够提供的电流），其结果通常会导致电压下降，而电压下降可能会导致器件无法正常工作。现代处理器在峰值电流时的功耗变化范围很大，因此提供了电压指数方法，允许处理器减缓速度，在更大幅度内调整电压。显然，这样会降低性能。

第二，持续功耗是多少？这个度量普遍称为**热设计功耗**（TDP），这是因为它决定了冷却需求。TDP既不是峰值功率（峰值功率通常要高1.5倍），也不是在一个给定计算期间消耗的实际平均功率（它可能还要更低一些）。在为一个系统设定典型电源型时，其功率通常要大于TDP，而一个冷却系统通常设计为与TDP相匹配或大于TDP。如果不能提供足够的冷却功率，可能会使处理器中的结点温度超出最大值，导致器件故障，甚至永久损坏。由于最大功率可能超出TDP指定的长期平均值（从而使热量和温度上升），所以现代处理器提供了两项功能来帮助管理热量。第一，当温度接近结点温度上限时，电路降低时钟频率，从而减小功率。如果这一技术不成功，则启用第二热过载启动装置，以降低芯片的功率。

设计者和用户需要考虑的第三个因素是能耗和能耗效率。回想一下，功率就是单位时间的能耗：1瓦=1焦/秒。哪个度量更适合对比处理器呢：能耗，还是功率？一般来说，能耗总是要更好一些，因为它与特定任务以及该项任务所需要的时间结合在一起。具体来说，执行一项工作负载的能耗等于平均功率乘以此项工作负载的执行时间。

因此，如果我们想知道两种处理器中的哪一个对于某一给定任务更为高效，应当对比执行该项任务的能耗（而不是对比功率）。例如，处理器A可能比处理器B的平均功耗高20%，但如果A执行该任务的时间仅为B所需时间的70%，它的能耗就是$1.2 \times 0.7=0.84$，显然要更好一些。

有人可能会说，在一个大型服务器或者云中，由于工作负载经常被认为是无限的，所以考虑平均功率就足够了，但这是一种误导。如果云中填充的是处理器B，而不是处理器A，那这个云在消耗相同能量的情况下，所做的工作会更少一些。使用能耗进行对比可以避免这一谬误。只要我们的工作负载是固定的，那无论是仓库规模的计算机，还是智能手机，在对比处理器时使用能耗指标总是正确的方法，因为无论是云的电费单还是智能手机的电池寿命，都是由所消耗的能量决定的。

那功率消耗什么时候才是一种有用的衡量指标呢？它的主要合理用途是作为一种约束条件：比如，一个芯片的功率可能被限定为不得超过100瓦。如果工作负载是固定的，那就可以用它来进行度量，但在这种情况下，它只不过是平均任务能耗这一真正度量的变体。

22

1.5.2 微处理器内部的能耗和功率

对CMOS芯片来说，传统的主要能耗源是开关晶体管，也称为**动态能耗**。每个晶体管所需要的能耗与该晶体管驱动的容性负载与电压平方的乘积成正比：

$$能耗_{动态} \propto 容性负载 \times 电压^2$$

这个公式的计算结果就是逻辑转变脉冲0→1→0或1→0→1的能耗。那么一次转换（0→1或1→0）的能耗就是：

$$能耗_{动态} \propto 1/2 \times 容性负载 \times 电压^2$$

每个晶体管所需要的功率就是一次转换的能耗与转换频率的乘积：

$$功率_{动态} \propto 1/2 \times 容性负载 \times 电压^2 \times 开关频率$$

对于一项固定任务，降低时钟频率可以降低功率，而不会降低能耗。

显然，通过降低电压可以大幅降低动态功率和能耗，所以在20年里，电压已经从5V降低到1V以下。容性负载的大小取决于输出端连接的晶体管数目及所用技术，这种技术决定了连线和晶体管的电容。

例题 现在的一些微处理器设计采用可调电压，电压降低15%可能导致频率下降15%。这对动态能耗和动态功率有什么影响？

解答 由于电容值不变，所以能耗变化就是电压平方之比：

$$\frac{能耗_{新}}{能耗_{原}}=\frac{(电压\times 0.85)^2}{电压^2}=0.85^2=0.72$$

因此，能耗大约降为原能耗的 72%。对于功率，需要考虑频率的比值：

$$\frac{功率_{新}}{功率_{原}}=0.72\times\frac{(开关频率\times 0.85)}{开关频率}=0.61$$

缩小到原功率的大约 61%。

23

当我们从一种制造工艺转向另一种工艺时，晶体管的开关次数以及其开关频率的增高强于负载电容和电压的下降，从而导致功耗和能耗的总体上升。第一代微处理器消耗的功率低于 1 瓦，第一代 32 位微处理器（比如 Intel 80286）消耗大约 2 瓦，而 3.3GHz Intel Core i7 消耗 130 瓦。如果这些热量必须从旁边的一个大约为 1.5 cm 的芯片上消散出去，那实际已经达到了风冷所能达到的极限。

根据上述公式，如果不能降低电压或提高每个芯片的功率，那可能就要减缓时钟频率的增长速度。图 1-4 表明，实际上从 2003 年开始就已经是这种局势了，图 1-1 中的微处理器都是每一年度的最佳处理器，它们也无一例外。注意，图 1-4 中具有平坦时钟频率曲线的那段时期与图 1-1 中性能改进缓慢的时期相对应。

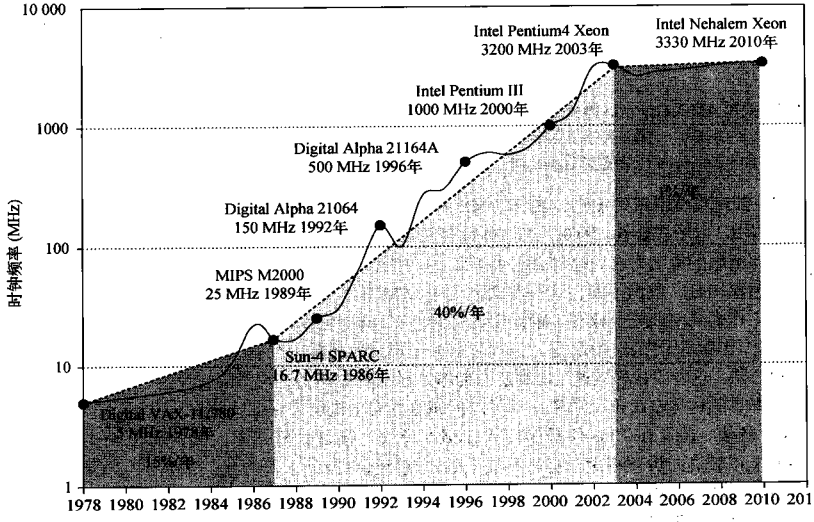


图 1-4 图 1-1 中各微处理器时钟频率的增长。从 1978 年到 1986 年，时钟频率的年增长速度低于 15%，而性能的年增长速度为 25%。在 1986 年到 2003 年性能年增长速度达到 52%的“复兴时期”，时钟频率飞速增长，几乎达到每年 40%。之后，时钟速率几乎停滞不前，每年的增长速度低于 1%，而单处理器性能的年增长速度低于 22%

24

分配功率、消散热量和防止热区已经变成难度日增的挑战。功率是现在使用晶体管的主要限制，在过去，主要约束体现在原料硅区域。因此，现代微处理器提供了许多技术，试图在时钟频率和电源电压保持不变的情况下，提高能耗效率。

(1) **以逸待劳**。今天的大多数微处理器都会关闭非活动模块的时钟，以节省能耗和动态功率。例如，如果没有正在执行浮点指令，浮点单元的时钟将被禁用。如果一些核心处于空闲状态，它们的时钟也会被停止。

(2) **动态电压-频率调整**（DVFS）。第二种技术直接来自上述公式。个人移动设备、膝上型电脑、甚至服务都会有一些活跃程度较低的时期，在此期间不需要以最高时钟频率和电压运转。现代微处理器通常提供几种能够降低功率和能耗的工作时钟频率和工作电压。图 1-5 绘制了当工作负载降低时，服务器通过 DVFS 可能节省的功率，三种不同时钟频率为：2.4 GHz、1.8 GHz 和 1 GHz。在这两个步骤的每一步中，服务器节省的总功率大约为 10% ~ 15%。

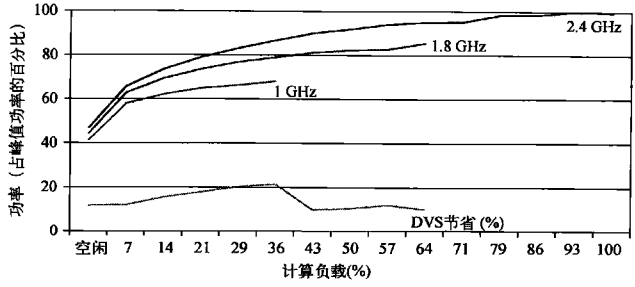


图 1-5 采用 AMD Opteron 微处理器、8 GB DRAM、一个 ATA 磁盘的服务器节省的能耗。工作频率为 1.8 GHz 时，服务器在不降低服务水平的情况下最多只能处理三分之二的工作负载，当工作频率为 1.0 GHz 时，只能安全地处理三分之一的工作负载（Barroso、Hölzle[2009]书中的图 5-11）

(3) **针对典型情景的设计**。由于 PMD 和膝上型电脑经常空闲，所以内外存储器都提供了低功率模式，以节省能耗。例如，DRAM 具有一系列功率逐渐降低的低功率模式，用于延长 PMD 和膝上型电脑的电池寿命，针对磁盘也提出了一些建议，在空闲时使其采用低转速模式，以节省功率。遗憾的是，我们不能在这些模式下访问 DRAM 和磁盘，无论访问速度有多低，都必须返回全速工作模式才能进行读写。前面曾经提到，PC 微处理器的设计已经考虑了一种更典型的情景：在高工作温度下密集使用，这种设计依靠片上温度传感器检测应当在什么时候自动减少活动，以避免过热。这种“紧急减速”使制造商能够针对一种更典型的情景进行设计，如果运行程序消耗的功率真的远远超出典型功率，则可以依靠这种安全机制来保证安全。

25

(4) **超频**。Intel 在 2008 年开始提供 Turbo 模式，在这种模式中，芯片可以判定在少数几个核心上以较高时钟频率短时运行是安全的，直到温度开始上升为止。例如，3.3 GHz Core i7 可以在很短的时间内以 3.6 GHz 的频率运行。实际上，从 2008 年开始，图 1-1 中每一年度最高性能的微处理器都提供了短时超频功能，超频频率大约比标称时钟频率高 10%。在执行单线程代码时，这些微处理器可以仅留下一个核心，并使其以更高时钟频率运行，而其他所有核心均被关闭。注意，操作系统可以关闭 Turbo 模式，而且在启用时也没有通知，所以程序员可能会惊奇地发现，他们的程序可能会因为室温而发生性能变化。

尽管通常认为动态功率是CMOS中的主要功率耗散源，但由于即使晶体管处于截止状态时也存在泄漏电流，所以静态功率也正在成为一个重要问题：

$$功率_{静态} \propto 电流_{静态} \times 电压$$

也就是说，静态功率与器件数目成正比。

因此，如果增大晶体管的数目，即使它们处于空闲状态也会增加功率，当晶体管的尺寸较小时，处理器中的泄漏电流会增大。所以，极低功率的系统甚至会关闭非活动模块的电源（电源门控），以控制由于泄漏电流导致的损失。2011年，泄漏目标是总功耗的25%，而高性能设计中的泄漏有时会远远超过这一目标。对于此类芯片，泄漏可能高达50%，部分原因是大型SRAM缓存需要功率来维持其存储值。（SRAM中的S表示“静态”，即static）。停止泄漏的唯一手段就是关闭部分芯片的电源。

最后，由于处理器只是系统整体能耗中的一部分，所以如果使用一个速度较快但能耗效率较低的处理器，使系统的其他部分能够进入睡眠模式，那也可能有助于降低整体能耗。这种策略被称为**竞相暂停**（race-to-halt）。

由于功率和能耗的重要性，人们在评价一项创新时，更加重视对其效率的审核。因此，现在的主要评价指标是每焦耳完成的任务数或者每瓦特实现的性能，而不再是每平方毫米的硅所实现的性能。这一新的度量影响了并行化方法，在第4章和第5章将会读到这一内容。 26

1.6 成本趋势

尽管成本趋势在一些计算机设计中不是特别重要（特别是在超级计算机中），但对成本敏感的设计正在变得越来越重要。事实上，在过去30年里，通过技术改进来降低成本（以及提高性能）已经成为计算机行业的一个重要主题。

教科书中经常会忽略“成本-性能”中的成本部分，这是因为成本的不断变化会使书中内容变得过时，还有一个原因是，这些问题非常微妙，因行业部门的不同而不同。但对计算机架构师来说，了解成本及其因素是必不可少的，在存在成本问题时，可以帮助他们明智地决定是否要包含某一项新功能。（想象一下，如果摩天大楼的设计师完全不了解钢梁和混凝土的成本，那会是一种什么结果！）

本节讨论影响计算机成本的主要因素，以及这些因素如何随时间变化。

1.6.1 时间、产量和大众化的影响

即使基础实现技术没有发生任何重大进步，计算机组件的制造成本也会随着时间的推移而降低。推动成本走低的基础原理是**学习曲线**——制造成本随时间的推移而降低。学习曲线本身是根据**成品率**的变化测得的，所谓**成品率**是指成功通过测试程序的器件占所生产器件总数的百分比。无论是芯片、主板，还是系统，使成品率加倍的设计就能使成本减半。

了解学习曲线如何提高成品率，对于在一件产品生存周期的不同阶段控制成本非常重要。比如，长期以来，每兆字节的DRAM价格一直在下降。由于DRAM的定价往往与成本密切关联（出现供给不足或过度供给的时期除外），所以DRAM的价格与成本变化趋势基本一致。

微处理器的价格也随时间的推移而降低，但由于它们的标准化程度弱于DRAM，所以价格与成本之间的关系要更复杂一些。当竞争非常激烈时，价格的变化趋势与成本非常一致，当然，微处理器销售商赔钱销售的可能性很少。

产量是决定成本的第二个重要因素。产量的提高会以几种不同方式对成本产生影响。第一，产量的提高缩短了降低学习曲线所需要的时间，这一时间在一定程度上与系统（或芯片）的制造数量成正比例关系。第二，由于产量的增加会提高购买与制造效率，所以产量会降低成本。一些设计人员根据经验估计：产量每增加一倍，会使成本下降大约 10%。此外，产量的增加还降低了必须分摊到每台计算机上的开发成本，使成本与销售价格更为接近。

大众化商品是指有多家销售商大量出售且基本相同的产品。在杂货商店货架上出售的几乎所有产品都是大众化商品，标准的 DRAM、闪存、磁盘、监视器和键盘也都是大众化商品。在过去 25 年里，个人计算机行业的大部分领域已经变为一项大众化商品业务，主要生产运行 Microsoft Windows 的桌面式计算机和膝上型计算机。

27

因为许多供应商都提供几乎完全相同的产品，所以市场竞争非常激烈。这种竞争当然会缩小成本与销售价格之间的距离，而且还会降低成本。由于大众化商品市场既拥有很大的产量，又有明确的产品定义，这样可以让那些为大众化产品制造组件的供应商展开竞争，从而降低成本。因此，由于组件供应商之间的竞争，以及销售商所能达到的产量效率，会降低产品的总成本。与其他部门相比，这种竞争使计算机行业的低端能够获得更好的性价比，得以更快速地增长，当然，它们的利润率非常有限（在所有大众化产品行业内，通常都是如此）。

1.6.2 集成电路的成本

一本讲解计算机体系结构的书中为什么会有一节内容来讨论集成电路的成本呢？在竞争日益激烈的计算机市场上，标准零件（磁盘、闪存、DRAM 等）在系统成本中的份额越来越高，集成电路成本成为计算机成本差异的重要因素，在大产量、价格敏感的市场中尤为如此。事实上，由于个人移动设备越来越依赖于整体片上系统（SOC），所以集成电路的成本成为 PMD 成本的主体部分。因此，计算机设计人员必须了解芯片的成本，才能理解当前计算机的成本。

尽管集成电路的成本以指数形式下降，但基本的硅制造工艺没有变化：仍然要对晶圆（wafer）进行测试，切割成晶片（die）进行封装（见图 1-6、图 1-7 和图 1-8）。因此，一个已封装集成电路的成本为：

$$\text{集成电路的成本}=\frac{\text{晶片成本}+\text{晶片测试成本}+\text{封装与最终测试成本}}{\text{最终测试成品率}}$$

这一节主要讨论晶片成本，最后总结测试和封装中的关键问题。

要学习如何预测一个晶圆上的正品芯片数目，需要首先了解一个晶圆上可以放多少个晶片，然后了解如何预测正常工作晶片的百分比。知道了这些数据，预测成本就很简单了：

$$\text{晶片成本}=\frac{\text{晶圆成本}}{\text{每个晶圆上的晶片数}\times\text{晶片成品率}}$$

28

晶片成本公式第一项中的最重要特征是它对晶片尺寸非常敏感，如下所示。

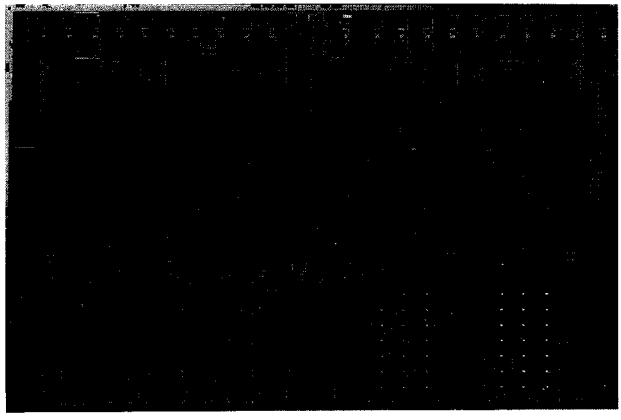


图 1-6 Intel Core i7 微处理器晶片的相片，在第 2 章到第 5 章将对其进行评价。在 45 nm 工艺中，尺寸为 18.9 mm × 13.6 mm（257mm^2）（感谢 Intel 提供）

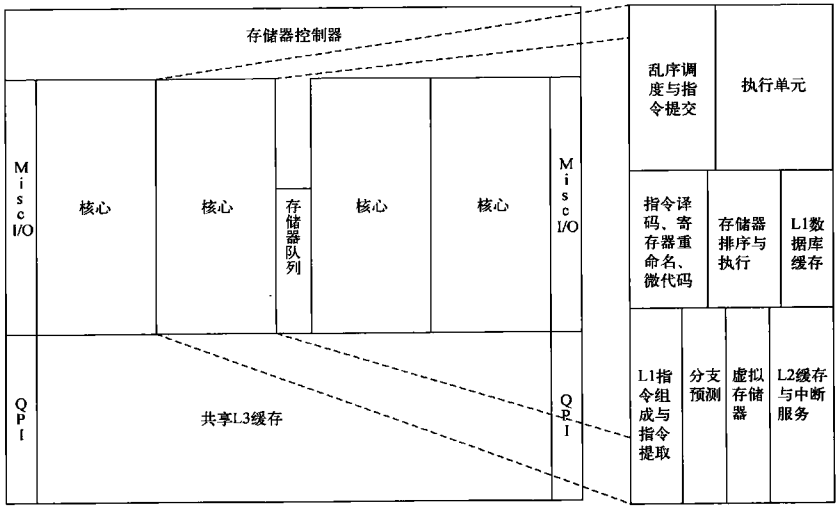


图 1-7 左图为图 1-6 中 Core i7 晶片的布置图，右图为第二核心布置图的特写

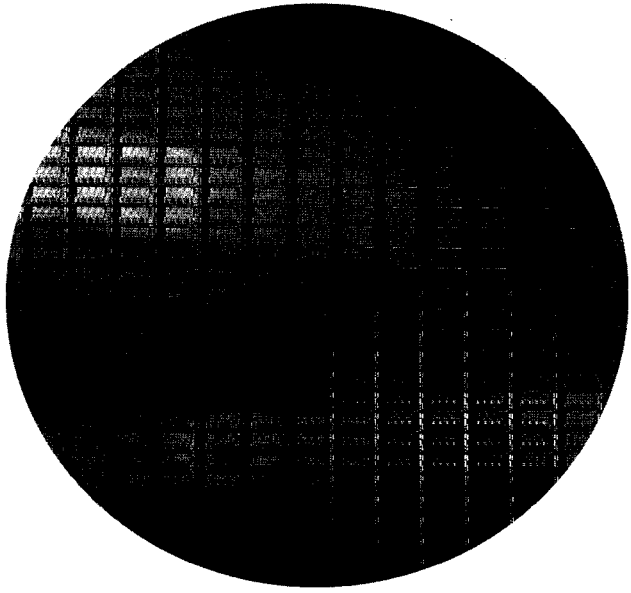


图 1-8　这个 300 mm 的晶圆上包含 280 个全沙桥晶片，采用 32 nm 工艺时，每个晶片的大小为 20.7mm×10.5 mm。（沙桥是 Intel 在 Core i7 中所用 Nehalem 的后续产品）。当晶片大小为 216 mm^2 时，每个晶圆上的晶片数大约为 282 个（感谢 Intel 提供）

每个晶圆上的晶片数目大约等于晶圆面积除以晶片面积。更准确的估算公式为：

$$\text{每个晶圆上的晶片数}=\frac{\pi\times(\text{晶圆直径})/2^2}{\text{晶片面积}}-\frac{\pi\times\text{晶圆直径}}{\sqrt{2}\times\text{晶片面积}}$$

30

第一项是晶圆面积（πr^2）与晶片面积之比。第二项对“方枘圆凿”问题（也就是接近晶圆外围的矩形晶片）作出补偿。将圆周（πd）除以方形晶圆的对角线，大约就是沿边缘排列的晶片数目。

例题　若晶片边长为 1.5 cm，求一个 300 mm（30 cm）晶圆上的晶片数目，若晶片的边长为 1.0 cm，又可以有多少个晶片。

解答　当晶片面积为 2.25 cm^2 时：

$$\text{每个晶圆上的晶片数}=\frac{\pi\times(30/2)^2}{2.25}-\frac{\pi\times30}{\sqrt{2\times2.25}}=\frac{706.9}{2.25}-\frac{94.2}{2.12}=270$$

由于较大晶片的面积大了 2.25 倍，所以每个晶圆上的小晶片数大约多 2.25 倍：

$$\text{每个晶圆上的晶片数}=\frac{\pi\times(30/2)^2}{1.00}-\frac{\pi\times30}{\sqrt{2\times1.00}}=\frac{706.9}{1.00}-\frac{94.2}{1.41}=640$$

但这个公式只给出了每个晶圆上的最大晶片数目。关键问题是：一个晶圆上的合格晶片占多大比例呢？或者说晶片成品率是多少呢？集成电路成品率的一种简单模型假定晶圆上的缺陷是随机分布的，成品率与制造工艺的复杂度成反比，由这一模型可以得出如下结果：

$$晶片正品率=晶圆正品率\times 1/(1+单位面积上的缺陷\times 晶片面积)^{N}$$

这个波斯-爱因斯坦公式是通过研究许多生产线的成品率而得出的经验模型[Sydow 2006]。晶圆成品率考虑了完全损坏而不需要测试的晶圆。为简单起见，我们直接假定晶圆成品率为100%。单位面积上的缺陷数是发生随机制造缺陷的度量。2010年，对于40 nm工艺，这个数值通常为每平方英寸0.1～0.3个缺陷，或者为每平米厘米0.016至0.057个缺陷，具体取决于工艺的成熟度（回想一下前面提到的学习曲线）。最后，N是一个称为工艺复杂度因数的参数，用于衡量制造难度。对于2010年的40 nm工艺，N的范围大约为11.5～15.5。

例题 设缺陷密度为0.031/cm^2，N为13.5，若晶片边长为1.5 cm和1.0 cm，求晶片成品率。

解答 总晶片面积为2.25 cm^{2}和1.00 cm^2。较大晶片的成品率为：

$$晶片成品率=1/(1+0.031\times 2.25)^{13.5}=0.40$$

较小晶片的成品率为：

$$晶片成品率=1/(1+0.031\times 1.00)^{13.5}=0.66$$

即，所有较大晶片中，成品数少于一半，而较小晶片中有三分之二以上是成品。 31

这个计算结果是每个晶圆上的合格晶片数，它等于每个晶圆上的晶片数乘以晶片成品率，将缺陷效应考虑在内。上述示例预测在300 mm晶圆上大约可以制造109个2.25 cm^{2}的合格晶片或者424个1.00 cm^{2}的合格晶片。许多微处理器都介于这两个尺寸之间。低端嵌入式32位处理器有时小至0.10 cm^2，而（在打印机、微波炉等设备内部）用于嵌入式控制的处理器通常小于0.04 cm^2。

由于DRAM和SRAM之类的大众化商品承受着巨大的价格压力，所以设计人员会加上一些冗余，作为提高成品率的方法。很多年来，DRAM中都有规律地包含一些冗余存储器单元，从而可以容许存在一定数目的缺陷。设计人员在标准SRAM中和用作微处理器内部缓存的大型SRAM阵列中使用类似技术。显然，冗余单元的存在可以显著提高成品率。

采用2010年业内领先技术，直径为300 mm（12英寸）的晶圆制造成本介于5000～6000美元。假定晶圆的生产成本为5500美元，1.00 cm^{2}晶片的成本大约为13美元，但每个2.25 cm^{2}晶片的成本将达到大约51美元，尺寸略大于2倍，而成本几乎达到了4倍。

关于芯片成本，计算机设计人员应当记住什么呢？制造工艺决定了晶圆成本、晶圆成品率和单位面积上的缺陷数，设计人员唯一能够控制的就是晶片面积。在实践中，由于单位面积上的缺陷数目很小，所以每个晶圆上合格晶片数的增长速度大致与晶片面积的平方成正比，每个晶片的成本也符合这一规律。计算机设计人员可以影响晶片大小，从而影响成本，方法有两个：决定晶片上包含哪些功能或者排除哪些功能，确定I/O管脚的数目。

必须首先对晶片进行测试（将合格晶片从不合格晶片中分离出来）、封装、封装后的再测试，才能得到一个能够在计算机中使用的零件。这些步骤都增加了成本。

上述分析的重点是生产一个功能晶片的可变成本，适用于大批量生产的集成电路。但是对于小批量生产的集成电路（小于100万），固定成本中有一个非常重要的部分会显著影响到集成

电路的成本，这个重要部分就是掩膜组的成本。集成电路工艺中的每个步骤都需要一个单独的掩膜。因此，对于现在具有4～6个金属层的高密度制造工艺来说，掩膜成本超过100万美元。显然，这一庞大的固定成本影响着原型设计和调试过程的成本，对于小批量生产来说，可能会成为生产成本的一个重要部分。由于掩膜成本可能继续增大，所以设计人员可以采用可重新配置的逻辑，以提高一件零件的灵活性，或者选择使用门阵列（门阵列的自定义掩膜级别较少），从而降低掩膜带来的成本。

1.6.3 成本与价格

32

随着计算机的大众化，一件产品的制造成本与销售价格之间的差额已经缩小了。这些差额主要用来支付公司的研发、营销、销售、制造设备维护、厂房租金、财务、税前利润、税收等各项费用。大多数公司的研发费用只占其收入的4%（大众化PC业务）至12%（高端服务器业务），其中包括了所有工程费用，许多工程师在知道这一数据时都非常惊讶。

1.6.4 制造成本与运行成本

在本书的前四个版本中，成本是指计算机的制造成本，价格是指购买计算机的价格。在仓库级计算机（其中包含数万个服务器）出现之后，除了购买成本之外，这种计算机的运行成本也非常高。

第6章指出，服务器和网络的可分摊购买价格仅略高于仓库级计算机月运行成本的60%（假定这一IT设备的短暂寿命为3～4年）。月运行成本的大约30%是电费以及用于配电和IT设备冷却的可分摊基础设施（尽管这一基础设施的费用可以分摊到10年以上）。因此，要降低仓库级计算机的运行成本，计算机架构师需要高效利用能源。

1.7 可信任度

在历史上，集成电路曾经是计算机中最可靠的组件之一。尽管它们的管脚很容易损坏，在信道中也可能发生错误，但芯片内部的错误率是非常低的。当我们向32 nm乃至更低特征尺寸前进时，上述常识发生了变化，由于暂时错误和永久错误的出现频率都大大增加，所以架构师设计的系统必须能够应对这些挑战。这一节快速回顾了一些可信任度问题，而术语与方法的官方定义请参阅附录D的D.3节。

计算机是在不同抽象层上设计和构建的。我们可以逐级深入计算机的不同层面，将每个组件放大为一个完整的子系统进行查看，直到深入到独立的晶体管为止。尽管有些错误分布得比较普遍，比如掉电，但许多错误都局限于一个模块的单个组件中。因此，在某一层级看来是整个模块完全失效的情况，从更高层级来看，可能只是一个模块中的组件故障。这种层级区分对于找出构建可靠计算机的方法很有帮助。

如何判断一个系统的运行是否正常，这是一个难题。随着互联网服务的普及，这一问题变得更为明确。基础设施供应商开始提供**服务等级协议**（SLA）或**服务等级目标**（SLO），保证他们的网络或电源服务是可靠的。例如，他们在一个月内无法满足协议的时间超过若干小时，就会向客户提供赔偿。因此，可以使用SLA来判断系统是在正常运行，还是已经宕机。

33

系统在SLA规定的两种服务状态之间切换。

(1) **服务实现**，即提供了指定服务

(2) **服务中断**，即所提供服务与 SLA 不一致

两种状态之间的转换由**故障**（由状态 1 至状态 2）或**恢复**（状态 2 至状态 1）导致。对这两种转换进行量化，可以得到可信任度的两种主要度量。

- **模块可靠性**是从一个参考初始时刻开始持续实现服务的度量（一种等价说法是：对发生故障之前的时间度量）。因此，**平均无故障时间**（MTTF）是一种可靠性度量。MTTF 的倒数就是故障率，通常以运行 10 亿小时发生的故障数来表示，或称为 FIT（Failures In Time）。因此，MTTF 等于 1 000 000 小时，相当于 $10^9/10^6$=1000 FIT。服务中断以**平均修复时间**（MTTR）来度量。**平均故障间隔时间**（MTBF）就是 MTTF+MTTR。尽管 MTBF 的使用更为广泛，但 MTTF 通常更为适用。如果一组模块的寿命服务以指数分布，也就是说模块的老化对于故障概率的影响并不是很大，那么这一组模块的整体故障率就是这些模块的故障率之和。
- **模块可用性**是指在服务完成与服务中断两种状态之间切换时，对服务完成的度量。对于可修复的非冗余系统，模块可用性为

$$\text{模块可用性}=\frac{\text{MTTF}}{(\text{MTTF}+\text{MTTR})}$$

注意，可靠性和可用性现在是可量化度量，而不再是可信任度的同义词。从这些定义出发，如果我们对组件的可靠性作出一些假设，并假设故障之间是互相独立的，则可以量化估计一个系统的可靠性。

例题 设磁盘子系统的组件及 MTTF 如下：

- 10 个磁盘，各自的等级为 1 000 000 小时 MTTF
- 1 个 ATA 控制器，500 000 小时 MTTF
- 1 个电源，200 000 小时 MTTF
- 1 个风扇，200 000 小时 MTTF
- 1 根 ATA 电缆，1 000 000 小时 MTTF

采用简化假设：寿命符合指数分布，各故障相互独立，试计算整个系统的 MTTF。 34

解答 故障率之和为：

$$\text{故障率}_{\text{系统}}=10\times\frac{1}{1\,000\,000}+\frac{1}{500\,000}+\frac{1}{200\,000}+\frac{1}{200\,000}+\frac{1}{1\,000\,000}$$

$$=\frac{10+2+5+5+1}{1\,000\,000\text{小时}}=\frac{23}{1\,000\,000}=\frac{23\,000}{1\,000\,000\,000\text{小时}}$$

或 23 000 FIT。系统的 MTTF 就是故障率的倒数：

$$\text{MTTF}_{\text{系统}}=\frac{1}{\text{故障率}_{\text{系统}}}=\frac{1\,000\,000\,000\text{小时}}{23\,000}=43\,500\text{小时}$$

或略低于 5 年。

应对故障的主要方法是冗余，或者是时间冗余（重复操作，以查看是否仍然错误），或者是资源冗余（当一个组件发生故障时，由其他组件接管）。在替换组件、完全修复系统后，认为系统的可信任度与新系统相同。现在用一个例子来量化冗余的好处。

例题 磁盘子系统经常备有冗余电源，以提高可信任度。利用上述组件和MTTF，计算冗余电源的可靠性。假设一个电源足以运行磁盘子系统，而且我们正在添加一个冗余电源。

解答 我们需要一个公式来表明当可以容忍一个故障并仍能提供服务时的情景。为了简化计算，假定组件的寿命为指数分布，而且组件故障之间没有相关性。冗余电源对的MTTF就是两个量的比值，分子是从初始时刻到一个电源发生故障的平均时间，分母是在更换第一个电源之前另一电源也发生故障的几率。因此，如果在修复第一个故障之前发生第二个故障的机会很小，那么电源对的MTTF就很大。

由于我们有两个电源，而且故障独立，则在一个磁盘发生故障之前的平均时间为$MTTF_{电源}/2$。发生第二个故障的概率有一个很好的近似：用MTTR除以另一电源发生故障之前的平均时间。因此，冗余电源对的合理近似为：

$$MTTF_{电源对}=\frac{MTTF_{电源}/2}{\frac{MTTR_{电源}}{MTTF_{电源}}}=\frac{MTTF^2_{电源}/2}{MTTR_{电源}}=\frac{MTTF^2_{电源}}{2\times MTTR_{电源}}$$

使用以上MTTF数字，如果假设操作人员平均需要24小时才能注意到电源发生故障并进行更换，则这个容错电源对的可靠性为：

$$MTTF_{电源对}=\frac{MTTF^2_{电源}}{2\times MTTR_{电源}}=\frac{200\,000^2}{2\times 24}\cong 830\,000\,000$$

35 使电源对的可靠程度比单电源提高大约4150倍。

在计算机的成本、功率和可信用度进行量化之后，我们可以开始量化性能了。

1.8 性能的测量、报告和汇总

如果我们说一台计算机比另一台计算机快，这是什么意思呢？一个台式计算机的用户说一台计算机更快，可能是一个程序的执行时间较短，而Amazon.com管理员说一台计算机更快，可能是它每小时完成的事务较多。计算机用户关心的是缩短响应时间，也就是一个事件从启动到完成的时间，也称为执行时间。仓库级计算机的操作人员可能关心的是吞吐量，也就是在给定时间内完成的总工作量。

在对比不同设计时，我们经常希望找出两台不同计算机（比如说X和Y）性能之间的关系。这里所说的“X比Y快”是指给定任务在X上的响应时间或执行时间短于在Y上的响应时间。具体来说，“X的速度是Y的n倍”是指：

$$\frac{执行时间_Y}{执行时间_X}=n$$

由于执行时间是性能的倒数，所以以下关系成立：

$$n=\frac{执行时间_Y}{执行时间_X}=\frac{\frac{1}{性能_Y}}{\frac{1}{性能_X}}=\frac{性能_X}{性能_Y}$$

“X的吞吐量是Y的1.3倍”指计算机X在单位时间内完成的任务数是Y上完成任务数的1.3倍。

遗憾的是，在对比计算机性能时并非总是使用时间这一度量标准。我们的观点是：唯一稳定、可靠的性能度量就是实际程序的执行时间，以任意其他度量代替时间或者以任意其他被测项目代替实际程序，最终都会在计算机设计中产生误导，甚至是错误。

即使是执行时间，也可以根据我们的测量内容采用不同的定义方式。最直接的时间定义被称为**挂钟时间**，**响应时间**或**已用时间**，也就是完成一项任务的延迟，包括磁盘访问、存储器访问、输入/输出活动、操作系统开销等所有相关时间。在同时运行多个程序的情况下，处理器在等待 I/O 时处理另一个程序，不一定使某一程序的已用时间缩至最短。因此，我们需要有一个术语来表达这一行为。**CPU 时间**可以区分这种不同，它是指处理器执行计算的时间，不包括等待 I/O 或运行其他程序的时间。（显然，用户观测到的响应时间是程序的已用时间，而不是 CPU 时间。）

36

那些定期运行相同程序的计算机用户当然是评估新计算机性能的最佳候选人。如果他们要评估一个新系统的性能，只需要比较其**工作负载**的执行时间就行了（其**工作负载**就是在计算机上运行的程序和操作系统命令）。但很少有用户具备这种得天独厚的条件。大多数用户必须依赖其他方法来评价计算机的性能，还经常需要使用评估软件，希望这些方法能够预测自己在使用新计算机时的性能。

1.8.1 基准测试

测量性能的最佳基准测试方法就是采用实际应用程序，比如 1.1 节的 Google Goggles。人们曾经尝试运行一些远比实际应用程序简单的程序，但这种做法已经导致了性能隐患。这些简单程序的示例包括：

- **程序内核**，即实际应用程序中的短小、关键部分；
- **玩具程序**，为了完成编程入门作业而编写的小程序，通常不超过 100 行，比如快速排序；
- **合成基准测试程序**，为了匹配实际应用程序的特征和行为而编写的虚拟程序，比如 Dhrystone。

今天，所有这三种方法都没有什么好名声，主要是因为编译器的编写人员和架构师可以串通起来，使计算机在执行这些替代程序时能够比运行实际应用程序时显得更快一些。令本书作者感到沮丧的是，合成程序 Dhrystone 仍然是应用最为广泛的嵌入式处理器基准测试程序！（由于我们认为计算机架构师也认同合成程序的名声不佳，所以在本书前四版中就不再使用合成程序来测量计算机性能。）

另外一个问题就是运行基准测试的条件。一种提高基准测试性能的方法是使用基准测试的专用标志；这些标志经常会导致对许多程序的非法转换，还可能降低另外一些程序的性能。为了限制这种情况，并使结果更有意义，基本测试开发人员经常要求销售商对所有使用同一语言（C++或 C）编写的程序使用同一编译器和同一组标志。除了编译器标志的问题之外，还有另外一个问题：是否允许修改源代码。有三种不同方法可以解决这一问题。

(1) 不允许修改源代码。

(2) 允许修改源代码，但基本没有修改可能。例如，数据库基准测试取决于拥有数千万行代码的标准数据库程序。数据库公司几乎不可能为了提高一个特定计算机的性能而进行修改。

(3) 允许修改源代码，只要修改后的版本能够给出相同输出结果即可。

37

在决定是否允许修改源代码时，基准测试设计人员面对的主要问题是这些修改是否会反映

实际做法，能否向用户提供有用的洞察能力，还是只会降低基准测试的准确度，只能作为实际性能的预测值。

为了避免将太多鸡蛋放在一个篮子中所带来的危险，一种流行的做法是采用基准测试应用程序集（称为**基准测试套件**）来衡量处理器处理各种应用程序的性能。当然，这些套件的准确程序不会超过组成该套件的各个基准测试。不过，这种套件的主要优势在于任何一个基准测试的弱点都会因为其他基准测试的存在而变小。基准测试套件的目的是描述两个计算机的相对性能，特别是那些客户可能会运行又未包含在该套件之内的程序。

电子设计新闻杂志嵌入式微处理器基准测试联盟（缩写为 EEMBC，发音与 embassy 相同）基准测试就是前车之鉴。它由 41 个内核程序组成，用于预测不同嵌入式应用程序的性能，这些领域包括汽车/工业、消费应用、网络、办公自动化和电信。EEMBC 报告中给出的性能数据未经任何修改，显得有些“杂乱”，几乎所有信息都包含在内。因为这些基准测试采用内核程序，还因为这些复杂的报告选项，所以 EEMBC 并不能很好地预测业内不同嵌入式计算机的相对性能。正是由于 EEMBC 的不够成功，才使它试图取代的 Dhrystone 一直沿用至今。

在创建标准化基准应用程序套件方面，最成功的尝试之一是 SPEC（标准性能评估机构），它源于 20 世纪 80 年代后期为了更好地对工作站进行基准测试所付出的努力。计算机行业一直在发展之中，所以对不同基准测试套件的需求也在不断发展，现在有许多 SPEC 基准测试，可以涵盖众多应用领域。所有 SPEC 基准测试套件及其测试报告都可以在 www.spec.org 找到。

尽管在下面的许多章节中，我们将主要讨论 SPEC 基准测试，但针对运行 Windows 操作系统的 PC 机，人们已经开发了许多其他基准测试。

1. 桌面基准测试

桌面基准测试分为两大类：处理器密集型基准测试和图形密集型基准测试，不过许多图形基准测试中包含大量处理器行为。SPEC 最初开发了一个针对处理器性能的基准测试集（最初被称为 SPEC89），它现在已经发展到第五代：SPEC CPU 2006，前面还有 SPEC2000、SPEC95、SPEC92 和 SPEC89。SPEC CPU2006 由 12 个整数基准测试（CINT2006）和 17 个浮点基准测试（CFP2006）组成。图 1-9 介绍了目前的 SPEC 基准测试及其以前的各个版本。

38

SPEC 基准测试是一些实际应用程序，这些应用程序经过修改就能够移植，并能在最大程度上降低 I/O 对性能的影响。整数基准测试涉及的范围很广，既有 C 编译器的一部分，又有一个国际象棋程序，还有一个量子计算机的模拟。浮点基准测试包括用于有限元建模的结构化网格代码、用于分子动力学的粒子方法代码，以及用于流体动力学的稀疏线性代数代码。SPEC CPU 套件可用于对桌面系统和单处理器服务器进行处理器基准测试。在本书中，将会看到许多此类程序的相关数据。但是，这些程序与 1.1 节介绍的编程语言、环境以及 Google Goggles 应用程序没有什么共同点。其中 7 个使用 C++、8 个使用 C、9 个使用 Fortran！它们甚至都是静态链接的，这些应用程序本身反应比较迟钝。我们不太清楚 SPECINT2006 和 SPECFP2006 能否把握 21 世纪的计算特点。

1.11 节将介绍在开发 SEEC 基准测试套件时已经出现的失误，以及为了维护一个有用的、具有预测性的基准测试套件所面对的挑战。

SPEC CPU2006 针对的是处理器性能，不过 SPEC 还提供了许多其他基准测试。

基准测试说明	各代SPEC的基准测试名称 SPEC2006	SPEC2000	SPEC95	SPEC92	SPEC89
GNU C compiler					gcc
Interpreted string processing			perl		espresso
Combinatorial optimization		mcf			li
Block-sorting compression		bzip2		compress	eqntott
Go game (AI)	go	vortex	go	sc	
Video compression	h264avc	gzip	ijpeg		
Games/path finding	astar	eon	m88ksim		
Search gene sequence	hmmer	twolf			
Quantum computer simulation	libquantum	vortex			
Discrete event simulation library	omnetpp	vpr			
Chess game (AI)	sjeng	crafty			
XML parsing	xalancbmk	parser			
CFD/blast waves	bwaves				fpppp
Numerical relativity	cactusADM				tomcatv
Finite element code	calculix				doduc
Differential equation solver framework	dealII				nasa7
Quantum chemistry	gamess				spice
EM solver (freq/time domain)	GemsFDTD			swim	matrix300
Scalable molecular dynamics (~NAMD)	gromacs		apsi	hydro2d	
Lattice Boltzman method (fluid/air flow)	lbm		mgrid	su2cor	
Large eddie simulation/turbulent CFD	LESlie3d	wupwise	applu	wave5	
Lattice quantum chromodynamics	milc	apply	turb3d		
Molecular dynamics	namd	galgel			
Image ray tracing	povray	mesa			
Spare linear algebra	soplex	art			
Speech recognition	sphinx3	equake			
Quantum chemistry/object oriented	tonto	facerec			
Weather research and forecasting	wrf	ammp			
Magneto hydrodynamics (astrophysics)	zeusmp	lucas			
		fma3d			
		sixtrack			

图 1-9 SPEC2006 程序及 SPEC 基准测试随时间的演变，粗线上方为整数程序，下方为浮点程序。在 SPEC2006 的 12 个整数应用程序中，9 个用 C 编写、其余 3 个用 C++编写。在浮点程序中，6 个用 Fortran 编写、4 个用 C++编写、3 个用 C 编写、4 个用 C 和 Fortran 混合编写。图中给出了在 1989 年、1992 年、1995 年和 2006 年版本中的所有 70 个程序。左侧的基准测试描述仅针对 SPEC2006，不适用于之前的版本。同一行中不同 SPEC 代的程序之间一般不相关；例如，fpppp 不像 bwaves 那样是 CFD 代码。gcc 是这个组中的高级成员。只有 3 个整数程序和 3 个浮点程序在三个或三个以上的版本中幸存下来。注意，SPEC2006 中的所有浮点程序都是新加入的。尽管有些程序一代一代地沿续下来，但程序的版本在变化，或者是基准测试的输入发生变化，或者是其大小发生变化，这些变化是为了延长其运行时间或者避免 CPU 时间之外的某种因素来干扰执行时间的测量或动摇其主导地位

2. 服务器基准测试

服务器有许多功能，所以也存在多种类型的基准测试。最简单的基准测试可能是面向处理

器吞吐量的基准测试。SPEC CPU2000利用SPEC CPU基准测试构建了一个简单的吞吐量基准测试，这种测试可以测试多处理器的处理速率：运行每个SPEC CPU基准测试的多个副本（副本数目通常与处理器数目相同），并将CPU时间转变为处理速率。这样会得到一个名为SPECrate的度量，它也是1.2节介绍的请求级并行的度量。为了测量线程级并行，SPEC提供了一些围绕OpenMP和MPI的基准测试，称之为高性能计算基准测试。

除了SPECrate之外，大多数服务器应用程序和基准测试都有大量因为磁盘和网络通信流量所产生的I/O行为，包括用于文件服务器系统、Web服务器和数据库与事务处理系统的基准测试，都是如此。SPEC提供了一个文件服务器基准测试（SPECSFS）和Web服务器基准测试（SPECWeb）。SPECSFS基准测试利用文件服务器请求脚本来测试NFS（网络文件系统）性能，它会测试I/O系统（包括磁盘I/O和网络I/O）与处理器的性能。SPECSFS是面向吞吐量的基准测试，但具有一些非常重要的响应时间需求。（附录D详细讨论了一些文件和I/O系统基准测试。）SPECWeb是一种Web服务器基准测试，它模拟多个客户端从服务器请求静态页和动态页，还模拟客户端向服务器张贴数据。SPECjbb测量针对Java的Web应用程序的服务器性能。最新的SPEC基准测试是SPECvirt_Sc2010，它评估虚拟化数据中心服务器的端到端性能，包括硬件、虚拟机层、虚拟化来宾操作系统。另一个比较新的SPEC基准测试用于测量功率，我们将在1.10节研究它。

39～40

事务处理（TP）基准测试测量一个系统处理事务（包括数据库访问与更新）的能力。航空订票系统和银行ATM系统是比较典型的简单TP示例，更高级的TP系统涉及复杂数据库和决策制定。在20世纪80年代中期，一群工程师特地为此组建了独立于供应商的**事务处理委员会**（TPC），尝试为TP创建客观公平的基准测试。www.tpc.org有关于TPC基准测试的介绍。

第一个TPC基准测试TPC-A于1985年发布，它随后被几个不同基准测试取代和增强。TPC-C最初在1992年创建，模拟一种复杂的查询环境。TPC-H对专用决策支持建模——查询之间没有相互关联，不能利用过去查询的相关知识来优化将来的查询。TPC-E是一种新的**联机事务处理**（OLTP）工作负载，它模拟一个代理公司的客户账户。最近发布的基准测试是TPC Energy，它向所有现有TPC基准测试添加了能耗度量。

所有TPC基准测试都以每秒完成的事务数来测试性能。此外，它们还包含响应时间要求，仅在满足响应时间限制时才会测试吞吐量性能。在实际系统建模时，更高的事务率也与更大型的系统关联在一起，这种所说的“更大型”一方面表现在用户数上，另一方面表现在作为事务应用对象的数据库上。最后，基准测试系统的系统成本也必须包含在内，以便准确地对比性价比。TPC修改了它的定价策略，对于所有TPC基准测试只有一个规格，从而可以验证TPC发布的价格。

1.8.2 报告性能测试结果

在报告性能测试结果时，应当坚持一条指导原则——**可重现性**，即列出其他试验者在重现该结果时所需要的全部信息。SPEC基准测试报告需要全面描述计算机和编译器标志，以及所公布的基准性能和优化结果。除了对硬件、软件和基准调优参数的描述之外，SPEC报告还包含了实际**执行倍数**（performance times），以表格和曲线图两种形式给出。TPC基准测试报告甚至更为复杂，因为它必须包含基准测试审核的结果和成本信息。自从制造商在高性能和高性价比方面展开竞争以来，这些报告就成为确定计算系统实际成本的极佳信息源。

1.8.3 性能结果汇总

在实际计算机设计中，必须通过一组相关基准测试来评估大量设计选项的相对量化优势。同样，消费者在选择计算机时，也会依靠一些基准测试的性能测试结果，希望这些基准测试与用户的应用程序相类似。在这两种情况下，拥有一组基准测试的测量结果都是有用的，一些重要应用程序的性能会与套件中一或多个基准测试的性能类似，另一方面也可以了解性能的变化程度。在理想情况下，这个套件类似于应用空间上的统计有效样本，但这样一个样本所需要的基准测试要多于大多数套件的基准测试数，因此需要进行随机采样，基本上没有基准测试套件使用这一方法。

41

一旦我们决定用某一种基准测试套件来测量性能，就希望能够用一个数值来汇总套件的性能结果。计算汇总结果的一种简单方法是对比套件中各个程序执行时间的算术平均值。遗憾的是，一些 SPEC 程序花费的时间要比其他程序长 4 倍，所以如果使用算术均值作为总结性能的唯一数值，那这些程序就太过重要了。一种替代方法是为每个基准测试增加一个加权因子，以加权算术平均作为总结性能的唯一数值。现在的问题是如何选择权重；SPEC 是由一些相互存在竞争的公司组成的联盟，每家公司可能都有自己中意的权重集，这就很难达成一致意见。另一种方法是在选择权重时，使所有程序在某一基准计算机上的执行时间相同，但这样会使测试结果出现偏差，向基准计算机的性能特性靠近。

当然，也可以不选择权重，而是以基准计算机为依据，对执行时间进行归一化：将基准计算机上的执行时间除以待评价计算机上的执行时间，得到一个与性能成正比的比值。SPEC 就是使用这种方法，将这个比值称为 SPECRatio。这是一个特别有用的特性，与本书对比计算机性能的方法相匹配（即比较性能比）。例如，假定在进行基准测试时，计算机 A 的 SPECRation 是计算机 B 的 1.25 倍，于是可以计算：

$$1.25=\frac{\text{SPECRatio}_A}{\text{SPECRatio}_B}=\frac{\dfrac{\text{执行时间}_{基准}}{\text{执行时间}_A}}{\dfrac{\text{执行时间}_{基准}}{\text{执行时间}_B}}=\frac{\text{执行时间}_B}{\text{执行时间}_A}=\frac{\text{性能}_A}{\text{性能}_B}$$

注意，基准计算机上的执行时间会略去不计，所以在以比值形式进行比较时，基准计算机可以任意选择，我们将一直使用这种方法。表 1-8 给出了一个示例。

表 1-8 Sun Ultra 5（SPEC 2000 的基准计算机）的 SPECfp2000 执行时间（秒）和 AMD Opteron、Intel Itanium 2 的执行时间与 SPECRation

基准测试	Ultra 5 时间（秒）	Opteron时间（秒）	SPECRatio	Itanium 2时间（秒）	SPECRation	Opteron/Itanium时间（秒）	Itanium/OpteronSPECRation
wupwise	1600	51.5	31.06	56.1	28.53	0.92	0.92
swim	3100	125.0	24.73	70.7	43.85	1.77	1.77
mgrid	1800	98.0	18.37	65.8	27.36	1.49	1.49
applu	2100	94.0	22.34	50.9	41.25	1.85	1.85
mesa	1400	64.6	21.69	108.0	12.99	0.60	0.60
galgel	2900	86.4	33.57	40.0	72.47	2.16	2.16
art	2600	92.4	28.13	21.0	123.67	4.40	4.40
equake	1300	72.6	17.92	36.3	35.78	2.00	2.00
facerec	1900	73.6	25.80	86.9	21.86	0.85	0.85
ammp	2200	136.0	16.14	132.0	16.63	1.03	1.03
lucas	2000	88.8	22.52	107.0	18.76	0.83	0.83

（续）

基准测试	Ultra 5 时间（秒）	Opteron时间（秒）	SPECRatio	Itanium 2时间（秒）	SPECRation	Opteron/Itanium时间（秒）	Itanium/Opteron SPECRation
fma3d	2100	120.0	17.48	131.0	16.09	0.92	0.92
sixtrack	1100	123.0	8.95	68.8	15.99	1.79	1.79
apsi	2600	150.0	17.36	231.0	11.27	0.65	0.65
几何平均值			20.86		27.12	1.30	1.30

*（SPEC2000将执行时间的比值乘以100，以清除结果中的小数点，所以20.86在报告中显示为2086。）最后两列给出执行时间和SPECRatio的比值。这一数字说明基准计算机与相对性能无关。执行时间之比与SPECRatio之比相同，几何均值之比（27.12/20.86=1.30）与比值的几何均值（1.30）相等。

因为SPECRatio是一个比值，而不是绝对执行时间，所以必须用几何平均来计算它的均值。（由于SPECRatio没有单位，所以以算术方式比较SPECRatio是没有意义的。）几何平均的公式是：

42

$$几何平均 = \sqrt[n]{\prod_{i=1}^{n} 样本_i}$$

对于SPEC，$样本_i$表示第i个程序的SPECRatio。使用几何均值可以确保以下两个重要特性。

(1) 这些比值的几何均值与几何均值之比相等。

(2) 几何均值之比等于性能比值的几何均值，这就意味着与基准计算机的选择无关。

因此，使用几何均值的动机更为充分，特别是在使用性能比值进行对比时，尤为如此。

例题 证明几何均值之比等于性能比值的几何平均，且与SPECRatio基准计算机的选择无关。

43

解答 假定有两个计算机A和B，每个计算机有一组SPECRatio。

$$\frac{几何均值_A}{几何均值_B} = \frac{\sqrt[n]{\prod_{i=1}^{n} SPECRatioA_i}}{\sqrt[n]{\prod_{i=1}^{n} SPECRatioB_i}} = \sqrt[n]{\prod_{i=1}^{n} \frac{SPECRatio_{A_i}}{SPECRatio_{B_i}}}$$

$$= \sqrt[n]{\prod_{i=1}^{n} \frac{\frac{执行时间_{基准}}{执行时间_{A_i}}}{\frac{执行时间_{基准}}{执行时间_{B_i}}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{执行时间_{B_i}}{执行时间_{A_i}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{性能_{A_i}}{性能_{B_i}}}$$

即，A与B的SPECRatio的几何均值等于对A与B执行套件中所有基准测试所得性能比的几何均值。表1-8用SPEC中的示例表明了其有效性。

1.9 计算机设计的量化原理

既然我们已经知道如何对性能、成本、可靠性、能耗和功率进行定义、测量和汇总，现在可以开始研究在计算机设计与分析中非常有用的指导原则了。这一节将介绍有关设计的重要观测结果，以及两个用于评估备选设计的公式。

1.9.1 充分利用并行

充分利用并行是提高性能的最重要方法之一。本书的每一章都有一个如何通过开发并行来提高性能的示例。这里给出三个简单的例子，在后续各章会给出详细解释。

第一个例子是在系统级别开发并行。为了提高在一个典型服务器基准测试（比如 SPECWeb 或 TPC-C）上的吞吐量性能，可以使用多个处理器和多个磁盘。随后可以在处理器和磁盘之间分散处理请求的工作负载，从而提高吞吐量。扩展内存以及处理器和磁盘数目的能力称为**可扩展性**，这对服务器来说是非常有价值的优点。在许多磁盘之间分布数据，以实现并行读写，就可以支持数据级并行。SPECWeb 还依靠请求级并行来使用大量处理器，而 TPC-C 使用线程级并行实现对数据库请求的更快速处理。

在单独的处理器级别，充分利用指令间的并行对于实现高性能非常关键。实现这种并行的最简单方法之一就是通过流水线。（在附录 C 中会更详细地解释流水线，它也是第 3 章的一个重点。）流水线背后的基本思想是将指令执行重叠起来，以缩短完成指令序列的总时间。流水线能够实现的关键是认识到并非所有执行都取决于与其直接相邻的前一条指令，所以有可能完全并行或部分并行地执行这些指令。流水线是人们最熟悉的指令级并行示例。

在具体的数字设计级别也可以开发并行。例如，组相联（Set Associative）缓存使用多组存储器，通常可以对它们进行并行查询，以查找所需项目。现代 ALU（算术逻辑单元）使用先行进位，这种方法使用并行来加快求和过程，使计算时间与操作数位数之间的关系由线性关系变为对数关系。数据级并行的例子还有许多。

1.9.2 局域性原理

人们通过观察程序特性，已经得到了一些重要的基本事实。我们经常用到的一个最重要程序特性是**局域性原理**：程序常常重复使用它们最近用过的数据和指令。一条广泛适用的经验规律是：一个程序 90%的执行时间花费在仅 10%的代码中。局域性意味着我们可以根据一个程序最近访问的指令和数据，比较准确地预测它近期会使用哪些内容。局域性的原理也适应于数据访问，不过不像代码访问那样明显。

人们已经观察到两种局域性类型。**时间局域性**是指最近访问过的内容很可能会在短期内被再次访问。**空间局域性**是指地址相互临近的项目很可能会在短时间内都被用到。我们将会在第 2 章看到这些原理的应用。

1.9.3 重点关注常见情形

最重要、最普遍的计算机设计原则可能就是重点关注常见情形：在进行设计权衡时，常见情形优先于非常见情形。这一原则适用于资源的分配方式，如果某一情形会频繁出现，那对其进行改进会产生更显著的效果。

对于功率、资源分配和性能，重点关注常见情形这一原则都是有效的。处理器中指令提取与译码器的使用可能比乘法器频繁得多，所以应当优先对其进行优化。这一原则也适用于可靠性。如果一个数据库服务器为每个处理器准备 50 个磁盘，那系统可靠性将主要取决于存储可靠性。

此外，常见情形经常要比非常见情景更简单一些，完成速度更快一些。比如，在处理器中对两个数值求和时，可以预料溢出是很少出现的情形，因此可以通过优化没有溢出的更常见情

形来提高性能。强调非溢出情形可能会降低溢出情形的处理速度，但如果很少发生溢出，那整体性能可以通过优化正常情形得以提高。

45

在本书中将会看到这一原则的许多示例。在应用这一简单原则时，我们必须判断常见情形是什么，加快这一常见情形可以使性能提高多少。有一种基本定律可以用来量化这一原则，即Amdahl定律。

1.9.4 Amdahl定律

利用Amdahl定律，可以计算出通过改进计算机某一部分而能获得的性能增益。Amdahl定律表明，使用某种快速执行模式获得的性能改进受限于可使用此种快速执行方式的时间比例。

Amdahl定律定义了使用某一特定功能所获得的加速比（speedup）。加速比是什么？假定我们可以对某一计算机进行某种升级，在采用这一升级时可以提高计算机的性能。加速比的定义为：

$$\text{加速比}=\frac{\text{整个任务在采用该升级时的性能}}{\text{整个任务在未采用该升级时的性能}}$$

或者：

$$\text{加速比}=\frac{\text{整个任务在未采用该升级时的执行时间}}{\text{整个任务在采用该升级时的执行时间}}$$

加速比告诉我们，与原计算机相比，在经过升级的计算机上运行一个任务可以加快多少。

Amdahl定律为我们提供了一种快速方法，用来计算某一升级所得到的加速比，加速比取决于下面两个因素。

(1) **原计算机计算时间中可升级部分所占的比例**。例如，一个程序的总执行时间为60秒，如果有20秒的执行时间可进行升级，那这个比例就是20/60。我们将这个值称为**升级比例**，它总是小于或等于1。

(2) **通过升级执行模式得到的改进，也就是说在为整个程序使用这一执行模式时，任务的运行速度会提高多少倍**。这个值等于原模式的执行时间除以升级模式的执行时间。如果为程序的某一部分采用升级模式后需要2秒，而在原始模式中需要5秒，则提升值为5/2。我们将这个值称为**升级加速比**，它总是大于1。

原计算机采用升级模式后的执行时间等于该计算机未升级部分耗用的时间加上使用升级部分耗用的时间：

46

$$\text{新执行时间}=\text{原执行时间}\times\left((1-\text{升级比例})+\frac{\text{升级比例}}{\text{升级加速比}}\right)$$

总加速比是这两个执行时间之比：

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

例题 假设我们希望升级一个用于提供Web服务的处理器。新处理器执行Web服务应用程序的计算速度是原处理器的10倍。假定原处理器有40%的时间忙于计算，60%的时间等待I/O，进行这一升级后，所得到的总加速比为多少？

解答 升级比例=0.4、升级加速比=10、总加速比$=\dfrac{1}{0.6+\dfrac{0.4}{10}}=\dfrac{1}{0.64}\approx 1.56$

Amdahl 定律阐述了一个回报递减规律：如果仅改进一部分计算的性能，在增加改进时，所获得的加速比增量会逐渐减小。Amdahl 定律有一个重要推论：若某一升级仅对一项任务的一部分适用，则该任务的总加速比不会超过一个数值，该数值即 1 减去未升级部分所占比例，再取其倒数。

在应用 Amdahl 定律时的一个常见错误是混淆“**可升级部分在升级之前**所占时间比例”和“**升级部分在升级之后**所占时间比例”。如果我们测量的不是计算中可以应用该升级的时间，而是测试应用升级之后的时间，结果就是错误的！

Amdahl 定律可用来判断某项升级能使性能提高多少，以及如何分配资源来提高性价比。分配目标显然是：某一部分的升级资源应当与这一部分原来花费的时间成比例。Amdahl 定律对于比较两种系统的整体系统性能尤其有用，不过也可用于比较两种处理器设计，如下面的例子所示。

例题 图形处理器中经常需要的一种转换是求平方根。浮点（FP）平方根的实现在性能方面有很大差异，特别是在为图形设计的处理器中，尤为明显。假设 FP 平方根（FPSQR）占用一项关键图形基准测试中 20%的执行时间。有一项提议：升级 FPSQR 硬件，使这一运算速度提高到原来的 10 倍。另一项提议是让图形处理器中所有 FP 指令的运行速度提高到原来的 1.6 倍，FP 指令占用该应用程序一半的执行时间。设计团队相信，他们使所有 FP 指令执行速度提高到 1.6 倍所需要的工作量与加快平方根运算的工作量相同。试比较这两种设计方案。

47

解答 可以通过计算加速比来对比两种方案：

$$加速比_{FPSQR}=\frac{1}{(1-0.2)+\dfrac{0.2}{10}}=\frac{1}{0.82}=1.22$$

$$加速比_{FP}=\frac{1}{(1-0.5)+\dfrac{0.5}{1.6}}=\frac{1}{0.8125}=1.23$$

提高整体 FP 运算的性能要稍好一些，原因是它的使用频率较高。

Amdahl 定律的适用范围不仅限于性能。我们重做 1.7 节的可靠性例题，通过冗余来提高电源可靠性，将 MTTF 从 200 000 小时提高到 830 000 000 小时，达到 4150 多倍。

例题 磁盘子系统故障率的计算为：

$$故障率_{系统}=10\times\frac{1}{1\ 000000}+\frac{1}{500000}+\frac{1}{200000}+\frac{1}{200000}+\frac{1}{1\ 000000}$$

$$=\frac{10+2+5+5+1}{1\ 000000小时}=\frac{23}{1\ 000000小时}$$

因此，可改进的故障率比例就是 5 次/百万小时占整个系统 23 次/百万小时的比例，即 0.22。

解答 可靠性的改进为：

$$改进_{电源对}=\frac{1}{(1-0.22)+\frac{0.22}{4150}}=\frac{1}{0.78}=1.28$$

尽管一个模块的可靠性提高了4150倍之巨，但从系统的角度来看，这一改变所带来的好处虽然可测，但数值很小。

在上面的几个例子中，我们需要知道改进后的新版本所占比例；这些时间一般是很难直接测量的。在下一节，我们将看到另外一种比较方法：利用一个公式将CPU执行时间分解为三个独立分量。如果我们知道一种候选方案是如何影响这三个分量的，就可以判断它的整体性能。另外，通常可以构建一个模拟器，在实际设计硬件之前先测量这些分量。

1.9.5 处理器性能公式

48

几乎所有计算机都有一个以固定频率运行的时钟。这些离散时间事件称为：**嘀嗒**、**时钟嘀嗒**、**时钟周期**、**时钟**、**脉冲周期**等。计算机设计人员用时钟周期的持续时间（例如，1 ns）或其频率（例如，1 GHz）来描述时钟周期的时间。程序的CPU时间可以有两种表示方法：

CPU时间=程序的CPU时钟周期数×时钟周期时间

或者

$$\text{CPU时间}=\frac{\text{程序的CPU时间周期数}}{\text{时间频率}}$$

除了执行一个程序所需要的时钟周期数之外，我们还会计算所执行的指令数：**指令路径长度**或**指令数**（IC）。如果我们知道时钟周期数和指令数，就可以计算**每条指令时钟周期数**（CPI）的平均值。由于这个指标便于使用，也因为这一章讨论的处理器比较简单，所以我们使用CPI。设计人员有时也将它称为**每时钟周期指令数**（IPC），它是CPI的倒数。

CPI的计算公式为：

$$\text{CPI}=\frac{\text{程序的CPU时钟周期数}}{\text{指令数}}$$

通过这个处理器指标值可以深入了解不同类型的指令集和实现方式，在后续四章中将广泛使用这一指标。

变换以下公式中的指令数，时钟周期可以定义为IC×CPI。这样就可以在执行时间公式中使用CPI：

CPU时间=指令数×CPI×时钟周期时间

将第一个公式按测试单位展开，可以看到各部分是如何组合在一起的：

$$\frac{\text{指令}}{\text{程序}}\times\frac{\text{时钟周期}}{\text{指令}}\times\frac{\text{秒}}{\text{时钟周期}}=\frac{\text{秒}}{\text{程序}}=\text{CPU时间}$$

从这个公式可以看出，处理器性能取决于三个特性：时钟周期（或时钟频率）、每条指令的时钟周期和指令数。此外，CPU时间同等取决于这三个特性；例如，三个特性中任一项改进10%，将使CPU时间改进10%。

遗憾的是，用于改变这三项特性的基本技术都是相互关联的，所以很难在不改变其他两个参数的情况下改变其中一个参数。

- **时钟周期时间**：硬件技术与组成。

- CPI：组成与指令集体系结构。
- 指令数：指令集体系结构和编译器技术。

幸运的是，许多可能采用的性能改进技术都是主要改进处理器性能的一个分量，而对其他两个分量的影响较小或在可预测范围内。 49

有时，在设计处理器时，采用以下公式计算处理器总时钟周期数目会有所帮助：

$$\text{CPU时钟周期}=\sum_{i=1}^{n}\text{IC}_i\times\text{CPI}_i$$

式中，IC_i表示一个程序中第 i 个指令的执行次数，CPI_i表示第 i 个指令的每条指令平均时钟周期数。这一形式可用来将 CPU 时间表示为：

$$\text{CPU时间}=\left(\sum_{i=1}^{n}\text{IC}_i\times\text{CPI}_i\right)\times\text{时钟周期时间}$$

总 CPI 为：

$$\text{CPI}=\frac{\sum_{i=1}^{n}\text{IC}_i\times\text{CPI}_i}{\text{指令数}}=\sum_{i=1}^{n}\frac{\text{IC}_i}{\text{指令数}}\times\text{CPI}_i$$

CPI 的后一种计算形式使用了各个 CPI_i和该指令在一个程序中所占的比例（即，IC_i除以指令数）。CPI_i 应当通过测量得出，而不是根据一个参考手册后面的表格来进行计算，这是因为它必须包括流水线效应、缓存缺失和存储系统的任何其他低效特性。

考虑 1.9.4 节的性能示例，这里改为使用指令执行频率的测试值和指令 CPI 测量值，在实际中，后者是通过模拟或硬件仪器获得的。

例题 假设已经进行以下测量：

FP 操作频率=25%

FP 操作的平均 CPI=4.0

其他指令的平均 CPI=1.33

FPSQR 的频率=2%

FPSQR 的 CPI=20

假定有两种设计方案，一种方案将 FPSQR 的 CPI 降至 2，一种是把所有 FP 操作的平均 CPI 降至 2.5。请使用处理器性能公式对比这两种设计方案。 50

解答 首先，观察到仅有 CPI 发生变化；时钟频率和指令数保持不变。我们首先求出没有任何改进时的原 CPI：

$$\text{CPI}_{\text{原}}=\sum_{i=1}^{n}\text{CPI}_i\times\left(\frac{\text{IC}_i}{\text{指令数}}\right)$$
$$=(4\times25\%)+(1.33\times75\%)=2.0$$

从原 CPI 中减去节省的周期数就可以求出改进 FPSQR 后的 CPI：

$$\text{CPI}_{\text{采用新FPSQR}}=\text{CPI}_{\text{原}}-2\%\times(\text{CPI}_{\text{旧FPSQR}}-\text{CPI}_{\text{仅新FPSQR}})$$
$$=2.0-2\%\times(20-2)=1.64$$

我们可以采用相同方式来计算对所有 FP 指令进行改进后的 CPI，也可以将 FP 和非 FP CPI 相加。采用后一种方法，将得到：

$$CPI_{新FP}=(75\% \times 1.33)+(25\% \times 2.5)=1.625$$

由于总 FP 改进的 CPI 稍低一些，所以它的性能也稍好一点。具体来说，总 FP 改进的加速比为：

$$加速比_{新FP}=\frac{CPU时间_{原}}{CPU时间_{新FP}}=\frac{IC\times时钟周期\times CPI_{原}}{IC\times时钟周期\times CPI_{新FP}}$$

$$=\frac{CPI_{原}}{CPI_{新FP}}=\frac{2.00}{1.625}=1.23$$

令人开心的是，我们使用 1.9.4 节的 Amdahl 定律得到了同一加速比。

要测量处理器性能公式的各个组成部分，通常是可能做到的。与前面例子中的 Amdahl 定律相比，这是使用处理器性能公式的重要优势之一。具体来说，有些东西可能很难测量，比如一组指令的执行时间在总执行时间中所占的比例。在实际计算中，一般是对指令数与集中每一指令的 CPI 乘积进行求和得到。由于计算过程通常是从测量各个指令数目和 CPI 开始的，所以处理器性能公式就更为有用。

要把处理器性能公式用作一种设计工具，需要能够测量各种因素。对于已有处理器来说，很容易通过测量来获得执行时间，而且我们知道默认的时钟速度。问题在于如何求得指令数或 CPI。大多数新处理器中都包含对所执行指令和时钟周期进行计数的计数器。通过定期观察这些计数器，还可以将执行时间和指令数与代码段关联在一起，如果程序员希望了解应用程序的性能并对其进行调优，那可能会有所帮助。通常，设计人员或程序员都希望更深入地了解性能，而不仅限于硬件计数器提供的信息。比如，他们可能希望知道 CPI 为什么是现在这种状况。在这种情况下，就要采用一些模拟技术，这些技术类似于正在设计的处理器所采用的技术。

51

有些能够提高能耗效率的技术，比如动态电压频率调整和超频（见 1.5 节），会增加这个公式的使用难度，在对程序进行测量时，时钟速度可能会发生变化。一种简单的方法是关闭这些功能，从而使结果能够重现。幸运的是，由于性能和能耗效率之间通常具有非常密切的联系（在短时间内完成一个程序，通常就可以节省能量），所以在评估性能时不考虑 DVFS 或超频对结果的影响，也是可行的。

1.10 融会贯通：性能、价格和功耗

在每章末尾的“融会贯通”小节中，我们提供了一些利用该章基本原理的真实示例。本节，我们将研究如何使用 SPECpower 基准测试对一些小型服务器测量性能和功率性能。

表 1-9 给出我们正在评估的三个多核服务器及其价格。为了保持价格对比的公平，所有服务器都是 Dell PowerEdge 服务器。第一台是 PowerEdge R710，它的微处理器为 Intel Xeon X5670，时钟频率为 2.93 GHz。这个 Intel 芯片有 6 个核心和 1 个 12 MB L3 缓存，不同于第 2 章到第 5 章的 Intel Core i7，后者有 4 个核心和一个 8 MB L3 缓存，不过这些核心本身是一样的。我们选择了一个两插槽系统，装有 12G 受 ECC 保护的 1333 MHz DDR3 DRAM。第二台服务器是 PowerEdge R815，它的微处理器是 AMD Opteron 6174 微处理器。这一芯片拥有 6 个核心和一个 6 MB L3 缓存，其运行频率为 2.20 GHz，不过 AMD 在一个插槽中放了 2 个芯片。因此，一个插槽拥有 12 个核心和 2 个 6 MB L3 缓存。我们的第二服务器有两个插槽，共有 12 个核心和 16 GB 受 ECC 保护的 1333 MHz DDR3 DRAM，第三台服务器（也是 PowerEdge R815）有四个插槽，共 48 个核心和 32 GB DRAM。

所有这些服务器都运行 IBM J9 JVM 和 Microsoft Windows 2008 Server x64 企业版。

表1-9 三个Dell PowerEdge被测服务器及2010年8月的各自价格

	系统1		系统2		系统3	
组　件		成本（%成本）		成本（%成本）		成本（%成本）
基本服务器	PowerEdge R710	$653 (7%)	PowerEdge R815	$1437 (15%)	PowerEdge R815	$1437 (11%)
电源	570W		1100W		1100W	
处理器	Xeon X5670	$3738 (40%)	Opteron 6174	$2679 (29%)	Opteron 6174	$5358 (42%)
时钟频率	2.93 GHz		2.20 GHz		2.20 GHz	
核心总数	12		24		48	
插槽数	2		2		4	
核心/插槽	6		12		12	
DRAM	12 GB	$484 (5%)	16 GB	$693 (7%)	32 GB	$1386 (11%)
以太网	双 1Gbit	$199 (2%)	双 1Gbit	$199 (2%)	双 1Gbit	$199 (2%)
磁盘	50 GB SSD	$1279 (14%)	50 GB SSD	$1279 (14%)	50 GB SSD	$1279 (10%)
Windows 操作系统		$2999 (32%)		$2999 (33%)		$2999 (24%)
总值		$9352 (100%)		$9286 (100%)		$12658(100%)
最大ssj_ops	910 978		926 676		1 840 450	
最大ssj_ops/$	97		100		145	

* 我们在计算处理器的成本时减去了第二个处理器的成本。与此类似，通过查看额外存储器的成本来计算存储器的总成本。因此，通过减去默认处理器和存储器的估计成本，调整服务器的基础成本。第 5 章介绍这些多插槽系统是如何连接在一起的。

注意，由于基准测试的压力（见 1.11 节），这些服务器的配置不同寻常。与计算量相比，表 1-9 中的系统在存储器方面比较可怜，只有一个很小的 50 GB 固态磁盘。如果不需要相应增加内存和存储，添加核心的费用是比较便宜的。

SPEC CPU 不是运行静态链接的 C 程序，而是一种使用 Java 编写的更加现代化的软件栈。它的基础是 SPECjbb，代表着业务应用程序的服务器端，性能的测量以每秒完成的事务数为单位，称为 ssj_ops（server side Java operations per second，每秒完成的服务器端 Java 操作）。它不仅和 SPEC CPU 一样，测试服务器的处理器，还会测试缓存、存储器系统，甚至还包括多处理器互连系统。除此之外，它还会测试 Java 虚拟机（JVM），包括 JIT 运行库编译器和垃圾回收器，还有底层操作系统的相关部分。 52

如表 1-9 的最后两行所示，在性能和性价比方面的优胜者是拥有四个插槽和 48 个核心的 PowerEdge R815。它达到了 1.8M ssj_ops，每美元的 ssj_ops 达到了最高的 145。令人惊讶的是，核心数目最多的计算机其成本效率竟然最高。处在第二位的是具有 24 个核心的两插槽 R815，拥有 12 个核心的 R710 处在最后一位。

尽管大多数基准测试（以及大多数计算机架构）都只关心峰值负载时的系统性能，但计算机实际上很少运行在峰值负载状态。事实上，第 6 章的表 6-2 显示了 Google 数万个服务器六个多月使用情况的测量结果，平均利用率为 100%的服务器只有不到 1%。大多数服务器的平均利用率介于 10%到 50%之间。因此，SPECpower 基准测试在收集功率信息时，将目标工作负载从其峰值开始，以 10%为间隔，一直降至 0%，工作负载为 0%时的状态称为“活跃空闲”。

图 1-10 绘制了每瓦的 ssj_ops（SSJ 操作/秒）以及目标负载从 100%变到 0%时的平均功率。在每个目标工作负载级别，Intel R710 的功率总是最低，ssj_ops/瓦总是最佳。一个原因是 R815 的电源功率要大得多，为 1100 瓦，而 R715 的功率为 570 瓦。如第 6 章所示，电源效率在一个计算机的总功率效率中非常重要。由于 1 瓦=焦/耳，这个度量与每焦耳的 SSJ 操作成正比。

53

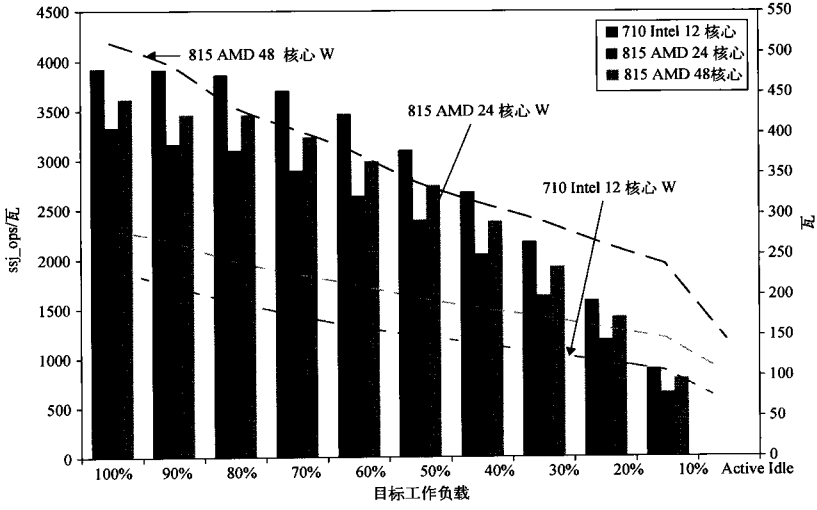


图 1-10 表 1-9 中 3 个服务器的功率效率。ssj_ops/瓦值放在左坐标轴上，有 3 个柱形与它相关联，瓦数表示在右坐标轴上，有三条线与它相关联。水平轴显示目标工作负载，从 100%变化到“活跃空闲”。在每个工作负载级别，Intel 的 R715 都具有最佳 ssj_ops/瓦，而且它在每个级别消耗的功率也最低

$$\frac{\text{ssj_ops}/\text{秒}}{\text{瓦}}=\frac{\text{ssj_ops}/\text{秒}}{\text{焦}/\text{秒}}=\frac{\text{ssj_ops}}{\text{焦}}$$

为了计算出一个数字，用来对比系统的功率效率，SPECpower 使用：

$$\text{总ssj_ops}/\text{瓦}=\frac{\sum\text{ssj_ops}}{\sum\text{功率}}$$

这 3 个服务器的总 ssj_ops/瓦分别是：Intel R710 为 3034、AMD 双插槽 R815 为 2357、AMD 四插槽 R815 为 2696。因此，Intel R710 具有最佳功率效率。除以服务器的价格之后，Intel R710 的 ssj_ops/瓦/1000 美元为 324、双插槽 AMD R815 为 254、四插槽 MD R815 为 213。因此，增加功率会反转性价比竞赛的结果，Intel R710 荣获价格-功率-性能的奖杯，而 48 核的 R815 垫底。

54

1.11 谬论与易犯错误

在每一章都会有这样一节，其目的是解释读者应当避免的常见错误观念或误解。我们将此类错误观念称为**谬论**。我们在讨论谬论时，会尝试给出一个反例。我们还会讨论**易犯错误**。一

般来说，谬论是随意推广一些在特定环境中才成立的原理而造成的。这些小节的目的是帮助读者避免在自己设计的计算机中犯这些错误。

谬论 多处理器是万能钥匙。

2005年左右之所以转向一芯多核，并不是因为取得了什么重大突破，显著简化了并行编程方式，或者使多核计算机的生产变得简单。之所以发生这种变化，就是因为ILP壁垒和功率壁垒的存在而别无选择。在一个芯片中设计多个处理器并不能保证功率较低，而的确有可能设计一种消耗更高功率的多核芯片。其潜力仅仅在于能够用几个低时钟频率的高效核心代替高时钟频率的低效核心，从而有可能继续提高效率。随着技术的发展，晶体管变得更小，使电容和电压都可以稍有下降，从而可以使每一代核心数目都会有适度提高。例如，在最近几年里，Intel每更新一代增加两个核心。

在第4章和第5章将会看到，性能现在已经成为程序员的负担。单靠硬件设计人员，不费吹灰之力就能加快程序运行速度的La-Z-Boy[①]程序员时代，已经正式结束了。如果程序员希望自己的程序在每一代处理器上都能更快速地运行，他们必须提高自己程序的并行度。

摩尔定律的通俗版本（即用每一代新技术来提高性能）现在摆在了程序员的面前。

易犯错误 Amdahl心碎定律的牺牲品。

几乎每一位有实践经验的计算机架构师都知道Amdahl定律。尽管如此，我们几乎都曾经在测量某一功能的使用之前，花费大量力气来对其进行优化。只有在总加速比让人感到失望时，才会想起在花费如此之多的精力进行改进之前，应当先对其进行测量。

55

易犯错误 单点故障。

利用1.9.4节的Amdahl定律来计算可靠性改进，可以发现可靠性不再是链子中最薄弱的一环了。无论我们让电源变得多么可靠（就像例子中那样），磁盘子系统的可靠性也会因为仅有一个风扇而受到限制。通过观察Amdahl定律得出一个有关容错系统的经验法则，那就是确保所有组件都有冗余，使单一组件故障不会导致整个系统宕机。第6章说明软件层如何避免仓库级计算机内部出现单点故障。

谬论 能够提高性能的硬件改进也可以提高能耗效率，至少不会加大能耗。

Esmaeilzadeh等人[2011]在一个使用Turbo模式（1.5节）的2.67 GHz Intel Core i7上进行SPEC2006测试。当时钟频率增加到2.94 GHz（1.10倍）时，性能提高到原来的1.07倍，但消耗的功率超过原来的1.37倍，能耗超过原来的1.47倍。

谬论 基准测试永远有效。

有几个因素会影响到一个基准测试在预测实际性能方面的有效性，其中一些因素是随时间变化的。一个影响基准测试有效性的重要因素是它要能够对抗“基准测试工程”或“基准测试技巧”。一旦某项基准测试变为标准普及测试，人们感受到提高性能的巨大压力，于是就可能进

① La-Z-Boy是美国知名的家具公司，其生产的“懒汉椅”在美国家喻户晓。——编者注

行有针对性的优化，或者对基准测试的运行规则作出对自己有利的主动解读。一些在少量代码上花费大量时间的小型内核或程序尤其缺乏这种免疫力。

例如，最初的 SPEC89 基准测试套件中包括一个名为 matrix300 的小型内核，它由 8 个不同的 300×300 矩阵乘法组成（当然最初包含这个小型内核的出发点是好的）。在这个内核中，99%的执行时间用于运行一行代码（见 SPEC[1989]）。当一个 IBM 编译器针对这一内部循环进行优化后（采用一种名为**分块**的思想，在第 2 章和第 4 章中详细讨论），其性能提高到该编译器上一版本的 9 倍！这个基准测试程序测试的是编译器的调优情况，当然不能很好地表征整体性能，也不是这一具体优化的典型值。

经过很长一段时间之后，这些修改甚至会让一个精心选择的基准测试被迫弃用；gcc 是 SPEC89 中的长期幸运者。图 1-9 列出了各种 SPEC 版本中所有 70 个基准测试的状态。令人惊讶的是，在 SPEC2000（或更早版本）的所有程序中，大约 70%都会在新一版中被弃用。

谬论 *磁盘的额定平均无故障时间为 1 200 000 小时，差不多是 140 年，所以磁盘实际上永远不会发生故障。*

56

磁盘制造商现在采用的一些营销手段可能会误导用户。这样一个 MTTF 是如何计算得到的呢？在早期过程中，制造商将数千个磁盘放在一个房间里，运行几个月的时间，记下故障磁盘的数目。他们计算 MTTF 的方法是将所有这些磁盘的累积工作小时数除以发生故障的磁盘数目。

这里的问题是，这个数字远远超过了磁盘的寿命（人们通常认为一个磁盘的寿命为 5 年或 43 800 小时）。为了使这个巨大的 MTTF 数字还有点意义，磁盘制造商宣称这个模型适用于那些购买磁盘后每 5 年更换一次的用户（这里的 5 年就是磁盘的预期寿命）。这种声明是说，如果许多客户（和他们的曾孙）到下一个世纪都还坚持这种做法，那在发生故障之前，他们要更换 27 次磁盘，也就是大约 140 年的时间。

一个更有用的测量数字应当是故障磁盘的百分比。假定有 1000 个 MTTF 为 1 000 000 小时的磁盘，这些磁盘每天使用 24 小时。如果用一个具有相同可靠特性的新磁盘更换故障磁盘，那在一年中（8760 个小时）发生故障的磁盘数目为：

$$故障磁盘数=\frac{磁盘数\times时间}{MTTF}=\frac{1000个磁盘\times 8760小时/驱动器}{1\,000\,000小时/故障}=9$$

或者说，每年将有 0.9%的磁盘发生故障，或者说在 5 年的寿命期中有 4.4%的磁盘发生故障。

另外，这些很大的数字是在假定温度与振动范围有限的情况下得到的，如果超过这些范围，那就不一定会是什么样的结果了。对实际环境中磁盘驱动器的一次调查[Gray 和 van Ingen 2005]发现：每年有 3%～7%的磁盘驱动器发生故障，MTTF 大约为 125 000 到 300 000 小时。一个更大型的调查发现每年的磁盘故障率为 2%～10%[Pinheiro、Weber 和 Barroso 2007]。因此，实际 MTTF 大约要比制造商宣称的 MTTF 差 2～10 倍。

谬论 *峰值性能能够反映实际观测性能。*

峰值性能唯一适用的定义是："计算机肯定不会超越的性能水平"。图 1-11 给出了 4 个程序在 4 个多处理器上的运行性能占峰值性能的百分比。其变化范围为 5%～58%。由于这个间距如此之大，而且可能受到基准测试的显著影响，所以峰值性能对于预测实际观测性能一般没有什么用。

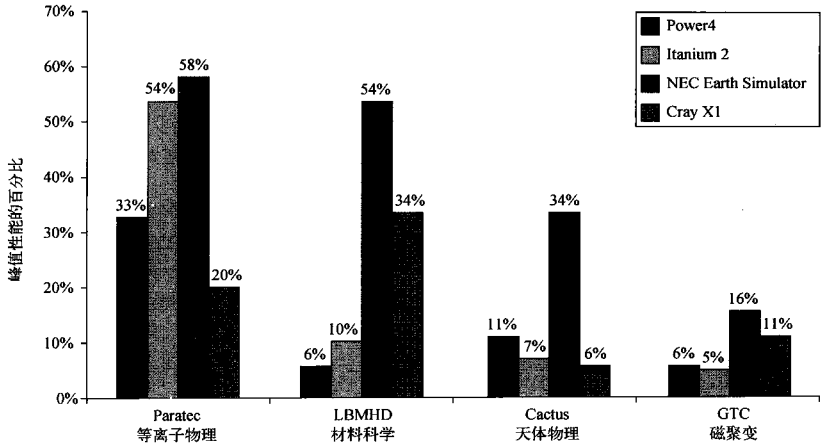


图 1-11 4 个程序在 4 个多处理器上的运行性能占峰值性能的百分比（峰值性能系在 64 个处理器获得）。 Earth Simultor 和 X1 为向量处理器（见第 4 章和附录 G）。它们不仅提供了较高的峰值性能比例，而且也具有最高的峰值性能和最低的时钟频率。除了 Paratec 程序之外，Power 4 和 Itanium 2 系统提供的性能占其峰值功能的 5%到 10%之间。数据来自 Oliker 等人[2004]

易犯错误 故障检测会降低可用性。

之所以会出现这一明显与事实相反的错误，是因为计算机硬件的状态中，有相当一部分并非总会对计算机的正常运行产生关键性影响。例如，如果分支预测器中发生错误，可能不会产生致命性的后果，而只是使性能受损。

在那些积极开发指令级并行的处理器中，并不是所有操作都需要程序的正确执行。Mukherjee 等人[2003]发现，对于在 Itanium 2 上运行的 SPEC2000 基准测试，只有不到 30%的操作可能处在关键路径上。

这一观察结果对程序也是成立的。如果程序中的一个寄存器“死亡”（也就是说，这个程序在再次读取该寄存器之前会先向其写入内容），那寄存器中发生错误就没有什么关系。如果在一个死亡寄存器中检查到晶体管错误就要终止程序，那才会不必要地降低可用性。 57

Sun 公司在 2000 年就犯了这样一个错误，它在 Sun E3000 至 Sun E10000 系统中使用了一个包含奇偶校验却无法纠错的 L2 缓存。他们用于构造这些缓存的 SRAM 有一些可用奇偶校验检测到的间歇性故障。如果缓存中的数据未被修改，则处理器直接从缓存中重新读取数据。由于设计人员没有用 ECC（纠错码）来保护缓存，所以操作系统别无选择，只能报告脏数据错误，并终止程序。现场工程师在实地查看时发现，90%以上的此种情况都不存在问题。

为了减少出现此类错误的频率，Sun 修改了 Solaris 操作系统，为它添加了一个主动将脏数据写到存储器的进程，从而“洗净”缓存。由于处理器芯片没有足够的管脚用于添加 ECC，所以对于脏数据的唯一硬件选项就是复制外部缓存，使用没有奇偶错误的副本来纠正这些错误。

这种易犯错误在于检测了错误却没有提供纠正错误的机制。这些工程师不大可能再设计另一个没有对外部缓存提供ECC保护的计算机了。

58

1.12 结语

这一章已经介绍了大量概念，并提供了一种量化框架，我们将在本书中对其进行扩展。从这一版开始，在讨论性能时将会增加能耗效率这一指标。

在第2章，我们将开始讨论存储器设计的所有重要内容。我们将研究各种技术，这些技术同心协力，让存储器看起来有无限大，同时还能尽可能保持快速。（附录B为没有太多经验和背景知识的读者提供了有关缓存的介绍性材料。）在后面各章中，我们将看到硬件与软件的结合已经成为高性能存储器系统的关键，就如同它们是高性能流水线的关键一样。这一章还将介绍虚拟机，这种保护技术的重要性正在与日俱增。

在第3章，我们将研究指令级并行（ILP），流水线是它的最简单、最常见形式。ILP的开发是构建高速单处理器的最重要技术之一。第3章首先对基础概念展开广泛讨论，为本章及第4章研究的大量思想奠定基础。第3章使用的例子跨度差不多有40年，来源范围从第一代超级计算机（IBM 360/91）到2011年市场上的最快速处理器。它强调了一种开发ILP的方法，称为**动态方法**或**运行时方法**。它还讨论了ILP思想的局限性，并介绍了多线程，在第4章和第5章将深入展开这一内容。附录C为那些没有太多流水线经验和背景知识的读者提供了有关流水线的介绍性材料。（我们认为这一内容可以帮助许多读者来复习流水线知识，包括我们编写的入门教材——《计算机组成与设计：硬件/软件接口》）。

第4章是这一版的新增内容，它解释了三种开发数据级并行的方法。其中最经典、最早的方法是向量体系结构，我们首先从这里入手，给出SIMD设计的基本原理。（附录G将更深入地讨论向量体系结构。）接下来解释当今大多数桌面微处理器都存在的SIMD指令集扩展。4.4节详细解释了现代图形处理器（GPU）的工作方式。大多数GPU说明文字都是从程序员的角度撰写的，通常隐藏了计算机的实际工作方式。这一部分从内部知情人的角度解释GPU，包括GPU术语与更传统的体系结构术语之间的对应关系。

第5章主要讨论使用多个处理器（或称多处理器）实现更高性能的问题。多重处理采用并行机制不是为了重叠各个指令的执行过程，而是允许同时在不同处理器上执行多个指令流。我们的重点是多处理器的主要形式——共享存储器多处理器，当然，我们也会介绍其他一些类型，并讨论在所有多处理器都会出现的主要问题。我们还是会研究各种技术，重点放在20世纪80年代到20世纪90年代首次提出的重要思想上。

59

第6章也是这一版新增加的内容。我们介绍了集群，然后深入讨论仓库级计算（WSC），它是由计算机架构师帮助设计的。WSC的设计人员是超级计算机先驱（比如Seymour Cray）的专业接班人，因为他们也正在设计超大型计算机。这些仓库级计算机中包含成千上万的服务器，容纳它们的设备和机房需要近2亿美元。前面各章对性价比和能耗效率的关注也适用于WSC，量化的决策方法也同样适用。

本书提供了丰富的线上材料（详细介绍请见“前言”部分），以便在降低成本的同时还能向读者介绍大量高深主题。表1-10列出了所有这些内容。以印刷方式出现在本书中的附录A、B、C可供许多读者回顾相关知识。

表1-10 附录清单

附 录	题 目
A	指令集基本原理
B	存储器层级结构回顾
C	流水线：基础概念与中级概念
D	存储系统
E	嵌入式系统
F	互连网络
G	深入讨论向量处理器
H	VLIW和EPIC的硬件与软件
I	大规模多处理器和科学应用
J	计算机算法
K	指令集体系结构调查
L	历史回顾与参考文献

在附录D中，我们离开以处理器为中心的视点，转而讨论存储系统中的问题。我们采用了一种类似的量化方法，但这种方法主要依靠对系统行为的观察，并使用端到端方法进行性能分析。它解决了如何主要使用低成本磁存储技术来实现高效数据存储与提取的重要问题。重点是研究磁盘存储系统对典型 I/O 密集型工作负载的性能，比如本章介绍的 OLTP 基准测试。我们将广泛研究 RAID 系统中的高级主题，这种系统使用冗余磁盘来同时获得高性能和高可用性。最后，这一章介绍了排队理论，为权衡利用率与延迟奠定基础。

附录E从嵌入式计算的角度来研究前面每一章、每个附录中介绍的思想。

附录F广泛讨论系统互连内容，包括实现计算机通信的广域网和系统域网络。

60

附录H回顾了 VLIW 硬件和软件，与本书上一版出版之前 EPIC 出现时相比，VLIW 的流行度已经有所降低了。

附录I描述了在高性能计算中使用的大规模多处理器。

附录J是唯一从第一版一直保留下来的附录，介绍了计算机算法。

附录K提供了一份关于指令体系结构的调查报告，包括 80x86、IBM 360 和 VAX，还有许多 RISC 体系结构，包括 ARM、MIPS、Power 和 SPARC。

下面介绍附录L。

1.13 历史回顾与参考文献

附录L从历史角度回顾了本书每一章中介绍的重要思想。这些历史回顾可以让我们通过一系列计算机或重大项目的介绍来了解一种思想的发展历程。如果你想研究一种思想或机器的最初发展，或者希望扩展阅读，可以参见每段历史介绍的末尾提供的参考文献。关于本章，可参见附录L.2节（The Early Development of Computers，计算机的早期发展），其中讨论了数字计算机及性能测量方法的早期发展。

在阅读这些历史资料时，你很快就会意识到，与许多其他工程领域相比，计算科学是如此“年轻”，它的重要优势之一就是许多先驱仍然健在——我们可以直接向他们了解历史！

案例研究与练习（Diana Franklin 设计）

案例研究 1：芯片制造成本

本案例研究说明的概念

- 制造成本
- 制造成品率
- 以冗余容忍缺陷

计算机芯片的价格中涉及许多因素。新的小型化技术大幅提升了芯片性能，降低了所需要的芯片面积。采用小型化技术，人们可以缩小芯片的面积或者在芯片上放置更多硬件，从而实现更多功能。在这个研究案例中，我们将研究包括制造技术、芯片面积和冗余在内的各种不同设计决策是如何影响芯片成本的，见表 1-11。

61

表1-11 几种现代处理器的制造成本因素

芯　片	晶片尺寸（mm^2）	估测缺陷率（每cm^2）	制程（nm）	晶体管数（百万个）
IBM Power5	389	0.30	130	276
Sun Niagara	380	0.75	90	279
AMD Opteron	199	0.75	90	233

1.1 [10/10] <1.6>表 1-11 给出了影响几种当前芯片成本的相关芯片统计数字。在下面几个练习中，我们将研究 IBM Power5 的不同设计决策所产生的影响。

a. [10] <1.6>IBM Power5 的成品率是多少？

b. [10] <1.6>为什么 IBM Power5 的缺陷率要低于 Niagara 和 Opteron？

1.2 [20/20/20/20] <1.6>建造一套新的制造设备需要 10 亿美元。我们将销售由这家工厂生成的一系列芯片，需要决定每种芯片的生产量。Woods 芯片的大小为 150 mm^2，每个无缺陷芯片的利润为 20 美元。Markon 芯片的大小为 250 mm^2，每个无缺陷芯片的利润为 25 美元。这套制造设备与 Power5 的制造设备相同。每个晶圆的直径为 300 mm。

a. [20] <1.6>如果制造 Woods 芯片，每个晶圆的利润为多少？

b. [20] <1.6>如果制造 Markon 芯片，每个晶圆的利润为多少？

c. [20] <1.6>我们应当用这套设备生产哪种芯片？

d. [20] <1.6>每种新 Power5 芯片的利润是多少？如果 Woods 芯片的月需求量为 50 000 个，Markon 芯片为 25 000 个，而这套设备一个月可以生产 150 个晶圆，如何分配这些晶圆？

1.3 [20/20] <1.6>AMD 公司的一位同事建议：由于成品率如此之低，所以如果在晶片上再放一个核心，只有那些两个处理器都失效的芯片才被扔掉，那就有可能降低芯片的制造成本。为了完成这个练习，我们将成品率看作是在给定缺陷率下，特定区域未发生缺陷的概率。分别基于每种 Opteron 核心计算概率（这种计算方法可能不完全准确，成品率公式是根据实验证据得出的，而不是根据芯片不同部分出现错误的概率，再经过数学计算得出的）。

a. [20] <1.6>在两个处理器核心中，缺陷核心数不超过一个的概率是多少？

b. [20] <1.6>如果每个旧芯片的成本为 20 美元，考虑到新的芯片面积和成品率，新芯片的成本应当是多少？

62

案例研究 2：计算机系统中的功耗

本案例研究说明的概念

- Amdahl 定律

- 冗余
- MTTF
- 功耗

现代系统中的功率取决于多种因素，包括芯片时钟频率、效率、磁盘驱动器速度、磁盘驱动器使用率和 DRAM。下面的练习研究不同设计决策和使用情景对功率的影响。

1.4 [20/10/20] <1.5>表 1-12 给出了几种计算机系统组件的功耗。在这个练习中，我们将研究硬盘驱动器是如何影响系统功耗的。

a. [20] <1.5>假定每个组件的负载最大，且电源效率为 80%，有一系统，采用 Intel Pentium 4 芯片，2 GB 240 管脚金士顿 DRAM 和一个 7200 rpm 的硬盘驱动器，问服务器电源必须向这个系统提供的功率为多少瓦？

b. [10] <1.5>如果这个 7200 rpm 磁盘驱动器的空闲时间大约占 60%，则它会消耗多少功率？

c. [20] <1.5>假定从一个 7200 rpm 磁盘驱动器读取数据的时间大约是 5400 rpm 磁盘读取时间的 75%，7200 rpm 磁盘的空闲时间占多大比例时，两个磁盘的平均功耗相等吗？

1.5 [10/10/20] <1.5>在为服务器场供电时，一个至关重要的因素就是冷却。如果不能有效地使计算机散热，风扇就会把热空气而不是冷空气吹回计算机。我们将研究各种不同设计决策如何影响一个系统的必要冷却方式，从而影响它的价格。请使用表 1-12 进行功率计算。

63

a. [10] <1.5>机架上安装一个冷却门的成本是 4000 美元，可以消散 14 KW（向室内散热；向室外散热需要增加成本）。如果服务器采用 Intel Pentium 4 处理器，1 GB 240 管脚 DRAM 和一个 7200 rpm 硬盘驱动器，那么一个冷却门可以冷却多少个服务器？

b. [10] <1.5>我们正在考虑为硬盘驱动器提供容错功能。RAID 1 将磁盘数目加倍（见第 6 章）。现在，一个只有一个冷却器的机架中可以放入多少个系统？

c. [20] <1.5>典型的服务器场每平方英尺可以消耗最多 200W 的热量。如果一个服务器机架需要 11 平方英尺（包括前后间隙），在一个机架内可以放入多少个第(a)步中提到的服务器？需要多少个冷却门？

表1-12 几种计算机组件的功耗

组件类型	产 品	性 能	功 率
处理器	Sun Niagara 8-core	1.2 GHz	72~79 W（峰值）
	Intel Pentium 4	2 GHz	48.9~66 W
DRAM	金士顿X64C3AD2 1 GB	184个管脚	3.7W
	金士顿D2N3 1 GB	240个管脚	2.3W
硬盘驱动器	DiamondMax 16	5400 rpm	读取/寻道时7.0 W，空闲时2.9 W
	DiamondMax 9	7200 rpm	读取/寻道时7.9 W，空闲时4.0 W

1.6 [讨论]<1.8>表 1-13 给出了几种基准测试对两个服务器的功率和性能进行对比的结果。两个服务器分别为 Sun Fire T2000（采用 Niagara）和 IBM x346（使用 Intel Xeon 处理器）。这一信息曾在 Sun Web 网站上发布。共发布了两条信息：两个基准测试的功率和速度。对于所示结果，Sun Fire T2000 显然要更好一些。还有其他哪些因素也很重要，会使一些人因为 IBM x346 在这些领域有出众表现而选择它？

表1-13 Sun功率/性能对比，由Sun选择性报告

	Sun Fire T2000	IBM x346
功率（瓦）	298	438
SPECjbb（操作数/秒）	63 378	39 985
功率（瓦）	330	438
SPECWeb（复合）	14 001	1348

1.7 [20/20/20/20] <1.6、1.9>公司的内部研究表明：一个单核系统就足以满足对处理能力的要求，而我们正在研究使用两个核心能否节省功率。

a. [20] <1.9>假定应用程序有 80%是可并行化的。可以将频率降低多少而仍能获得相同性能？

b. [20] <1.6>假定电压可能随频率线性下降。利用 1.5 节的公式，与单核系统相比，双核系统可能需要多少动态功率？

c. [20] <1.6、1.9>现在假定电压不会降至原电压的 25%以下。这一电压被称为电压下限，低于这一电压就会丢失状态。当并行化百分比为多少时，将使电压处于这一电压下限？

d. [20] <1.6、1.9>使用 1.5 节的公式，考虑到电压下降，双核系统与单核系统相比，需要多少动态功率？

64

练习

1.8 [10/15/15/10/10] <1.4、1.5>架构师面对的一个挑战是，今天拟定的设计方案可能需要几年的时间进行实施、验证和测试，然后才能上市。这就意味着架构师必须提前为几年之后的技术进步制定计划。有时，这是很难做到的。

a. [10] <1.4>根据摩尔定律观测到的器件发展趋势，到 2015 年，一个芯片上的晶体管数目应当是 2005 年的多少倍？

b. [15] <1.5>时钟频率的增加也一度反映了这一趋势。如果时钟频率仍以 20 世纪 90 年代的相同速度攀升，2015 年的时钟速率大约是多少？

c. [15] <1.5>以目前的增长速率，2015 年的时钟频率是多少？

d. [10] <1.4>是什么限制了时钟频率的增长速度？为了提升性能，架构师现在能用多出来的晶体管做些什么？

e. [10] <1.4>DRAM 容量的增长速度也已变缓。20 年来，DRAM 容量每年提高 60%。这一速率下降到每年 40%，现在的改进速率为每年 25%～40%。如果这一趋势继续下去，2020 年的 DRAM 容量增速大约是多少？

1.9 [10/10] <1.5>我们正在为一种实时应用设计系统，这种应用要求必须在指定期限之前完成计算。提前完成计算没有任何收益。我们发现，在最糟糕的情况下，这一系统执行必需代码的速度是最低要求速度的两倍。

a. [10] <1.5>如果以当前速度执行计算，并在完成任务后关闭系统，可以节省多少能量？

b. [10] <1.5>如果将电压和频率设置为现在的一半，可以节省多少能量？

1.10 [10/10/20/20] <1.5>诸如 Google 和 Yahoo!之类的服务器场都为当时的最高请求速率提供了足够的计算容量。假设这些服务器在大多数时间内仅以 60%的容量运行。进一步假设功率不会随负载线性改变，也就是说，当服务器以 60%的容量运行时，它们消耗的功率为最大功率的 90%。这些服务器可以关闭，但在负载更多时，需要的重新启动时间过长。有人提议采用一种新型系统，它能够快速重新启动，但处在这种“勉强生存”状态时需要消耗一定的功率，为最大值的 20%。

a. [10] <1.5>关闭 60%的服务器可以节省多少功率？

b. [10] <1.5>将 60%的服务器置于“勉强生存”状态，可以节省多少功率？

65

c. [20] <1.5>将电压降低 20%，频率降低 40%，可以节省多少功率？

d. [20] <1.5>将 30%的服务器置于“勉强生存”状态，30%的服务器关闭，可以节省多少功率？

1.11 [10/10/20] <1.7>可用性是服务器设计中的最重要考虑事项，紧随其后的是可扩展性和吞吐量。

a. [10] <1.7>有一个处理器，其 FIT 为 100。这个系统的平均无故障时间（MTTF）为多少？

b. [10] <1.7>如果需要 1 天的时间才能让这个系统再次正常运行，这个系统的可用性是多少？

c. [20] <1.7>假设政府为了降低成本，准备用廉价计算机构建一个超级计算机，而不是使用可靠但却昂贵的计算机。一个具有 1000 个处理器的系统，其 MTTF 为多少？（假设这些处理器一损俱损。）

1.12 [20/20/20] <1.1、1.2、1.7>在 Amazon 或 eBay 使用的服务器场中，一个故障不会导致整个系统崩溃，而是减少在任意时刻能够满足的请求数目。

a. [20] <1.7>如果一个公司有 10 000 台计算机，每台计算机的 MTTF 为 35 天，而且只有当 1/3 以上的计算机发生故障时才会经历灾难性故障，系统的 MTTF 为多少？

b. [20] <1.1、1.7>如果一台计算机的 MTTF 加倍，需要另加 1000 美元，这是不是一个很好的业务决策？证明你的结论。

c. [20] <1.2>表 1-2 给出了宕机的平均成本，假定在一年的所有时间内，该成本不变。但对于零售商来说，圣诞季节是盈利水平最高的（因此，如果因为宕机造成无法销售，损失也最大）。如果目录销售中心在第四季度的通信流量是其他任一季度的两倍，那第四季度每小时的平均宕机成本是多少？其他时间的宕机成本又是多少？

1.13 [10/20/20] <1.9>你的公司正要选择是购买 Opteron，还是 Itanium 2。你已经分析了公司的应用情况，在 60%的时间里运行类似于 wupwise 之类的应用程序，20%的时间里运行类似于 ammp 之类的应用程序，20%的时间里运行类似于 apsi 之类的应用程序。

a. [10] 如果仅依据 SPEC 总体性能进行选择，你选择哪一种？为什么？

b. [20] 对 Opteron 和 Itanium 2 来说，这种混合应用程序的加权平均执行时间比是多少？

c. [20] Opteron 相对于 Itanium 2 的加速比是多少？

1.14 [20/10/10/10/15] <1.9>在这个练习中，假定我们正在考虑通过添加向量硬件来提高一个机器的性能。当一个计算运行于向量硬件的向量模式时，其速度是正常执行模式的 10 倍。我们将使用向量模式时花费的时间百分比称为向量化百分比。向量将在第 4 章讨论，但回答下面的问题不需要知道有关其工作方式的任何信息！ 66

a. [20] <1.9>绘制一条曲线，以加速比为因变量，以向量模式下所执行计算的比例为自变量。将 y 轴标记为“净加速比”，x 轴标记为“向量化百分比”。

b. [10] <1.9>向量化百分比达到多少时，才能使加速比为 2？

c. [10] <1.9>如果已经使加速比为 2，在向量模式下的计算运行时间占多大百分比？

d. [10] <1.9>向量化百分比达到多少时，才能使加速比为向量模式所能实现的最大加速比的一半？

e. [15] <1.9>假定已经测得程序的向量化百分比为 70%。硬件设计组估计，通过大量追加投入，可以加快向量硬件的速度。你想知道编译器组是否也能提高向量化百分比。编译器团队需要实现多大的向量化百分比，才能在向量单元中获得另外 2 倍的加速（超过最初的 10 倍以上）？

1.15 [15/10] <1.9>假定我们对一台计算机进行了升级，使某种执行模式提升为原来的 10 倍。升级模式的使用时间占总时间的 50%，这一数值是在使用该升级模式时测得的执行时间百分比。回想一下，Amdahl 定律需要的是能改进但还没有改进的原执行时间比例。因此，在使用 Amdahl 定律计算加速比时，不能使用这个 50%的测量值。

a. [15] <1.9>从快速模式获得的加速比是多少？

b. [10] <1.9>转换为快速模式的原执行时间比例是多少？

1.16 [20/20/15] <1.9>在为了优化处理器的某一部分而进行改变时，经常会出现这样一种情况：加速某种类型的指令时，会降低其他某些指令的速度。例如，如果放入一个复杂的浮点单元，它要占用空间，为了容纳它，就得将某些东西移得远一些，这样就会要增加一些延迟周期才能到达被挪远的单元。基本的 Amdahl 定律公式没有考虑这种折中。

a. [20] <1.9>如果这个新的快速浮点单元使浮点运算平均提高到 2 倍，浮点运算占用的时间为原程序执行时间的 20%，那么总加速比为多少（忽略对所有其他指令的影响）？

b. [20] <1.9>现在假定浮点单元的加速会降低数据缓存访问的速度，减缓倍数为 1.5（或者说加速比为 2/3）。数据缓存访问时间为总执行时间的 10%。现在的总加速比为多少？ 67

c. [15] <1.9>在实现新的浮点运算之后，在浮点运算上花费的执行时间占多大比例？数据缓存访问又占多大比例？

1.17 [10/10/20/20] <1.10>公司刚刚购买了一个新的 Intel Core i5 双核处理器，你接到针对这一处理器来优化软件的任务。你将在这个双核处理器上运行两个应用程序，但它们的资源需求并不一样。第一个程序需要 80%的资源，另一个仅需要 20%的资源。假定对该程序的一部分进行并行化时，该部分的加速比为 2。

a. [10] <1.10>假定第一个应用程序的 40%可以并行化，那么在隔离运行时，通过这个应用程序可以实现多大的加速比？

b. [10] <1.10>假定第二个应用程序的 99%可以并行化，那么在隔离运行时，这个应用程序可以达到多大的加速比？

c. [20] <1.10>假定第一个应用程序的 40%可以并行化，如果对其实现并行化，系统总加速比为多少？

d. [20] <1.10>假定第二个应用程序的 99%可以并行化，如果对其实现并行化，系统总加速比为多少？

1.18 [10/20/20/20/25] <1.10>在实现一个应用程序的并行化时，理想加速比应当等于处理器的个数。但它要受到两个因素的限制：可并行化应用程序的百分比和通信成本。Amdahl 定律考虑了前者，但没有考虑后者。

a. [10] <1.10>如果应用程序的 80%可以并行化，N 个处理器的加速比为多少？（忽略通信成本。）

b. [20] <1.10>如果每增加一个处理器，通信开销为原执行时间的 0.5%，则 8 个处理器的加速比为多少？

c. [20] <1.10>如果处理器数目每增加一倍，通信开销增加原执行时间的 0.5%，则 8 个处理器的加速比为多少？

d. [20] <1.10>如果处理器数目每增加一倍，通信开销增加原执行时间的 0.5%，则 N 个处理器的加速比为多少？

e. [25] <1.10>写出求解这一问题的一般公式：如果一个应用程序中占原执行时间的 P%可以并行化，处理器数目每增加一倍，通信成本增加原执行时间的 5%，则达到最高加速比的处理器数目为多少？

68

2

存储器层次结构设计

理想情况下，我们希望拥有无限大的内存容量，这样就可以立刻访问任何一个特定的……机器字……但我们……不得不认识到有可能需要构建分层结构的存储器，每一层次容量都要大于前一层次，但其访问速度也要更慢一些。

A. W. Burks、H. H. Goldstine 及
J. von Neumann,
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)

69 ~ 71

2.1 引言

一些计算机先驱准确地预测到程序员肯定会希望拥有无限数量的快速存储器。满足这一愿望的一种经济型解决方案是**存储器层次结构**，这种方式利用了局域性原理，并在存储器技术的性能与成本之间进行了折中。第 1 章中介绍的**局域性原理**是指，大多数程序都不会均衡地访问所有代码或数据。局域性可以在时间域发生（**即时间局域性**），也可以在空间域发生（**即空间局域性**）。根据这一原理和“在给定实现技术和功率预算的情况下，硬件越小，速度可以越快”的准则，就产生了存储器层次结构，这些层次由不同速度、不同大小的存储器组成。图 2-1 显示了一个多级存储器层次结构，包括典型大小和访问速度。

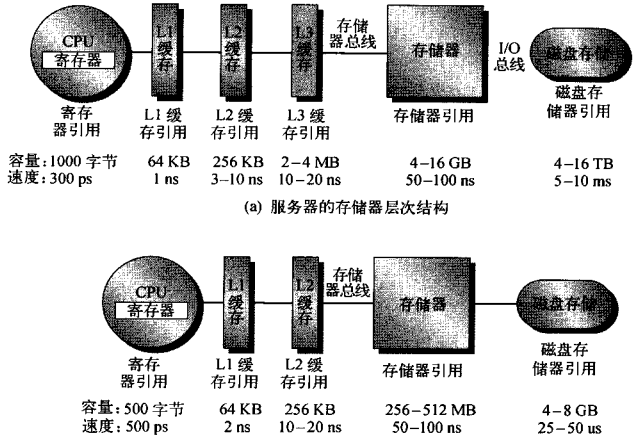


图 2-1 典型存储器层次结构中的级别，图中上部为服务器计算机中的层次结构，下部为 PMD（个人移动设备）中的层次结构。存储器级别离处理器越远，速度越慢、容量越大。注意，时间单位改变了 10^9 倍（从皮秒变为毫秒），容量单位改变了 10^{12} 倍（从字节到百万兆字节）。PMD 的时钟频率较低，缓存和主存储器较小。服务器与桌面计算机的主要区别是以磁盘存储作为层次结构中的最低层级，而 PMD 则采用以 EEPROM 技术制造的 Flash 作为最低层级

由于快速存储器非常昂贵，所以将存储器层次结构分为几个级别——越接近处理器，容量越小、速度越快、每字节的成本也越高。其目标是提供一种存储器系统，每字节的成本几乎与最便宜的存储器级别相同，速度几乎与最快速的级别相同。在大多数（但并非全部）情况下，低层级存储器中的数据是其上一级存储器中数据的超集。这一性质称为**包含性质**，层次结构的最低级别必须具备这一性质，如果这一最低级别是缓存，则由主存储器组成，如果是虚拟存储器，则由磁盘存储器组成。

72

随着处理器性能的提高，存储器层次结构的重要性也在不断增加。图 2-2 是单处理器性能

和主存储器访问次数的历史发展过程。处理器曲线显示了平均每秒发出存储器请求数的增加(即两次存储器引用之间延迟值的倒数)，而存储器曲线显示每秒 DRAM 访问数的增加（即 DRAM 访问延迟的倒数）。在单处理器中，由于峰值存储器访问速度要快于平均速度（也就是图中绘制的速度），所以其实际情况要稍差一些。

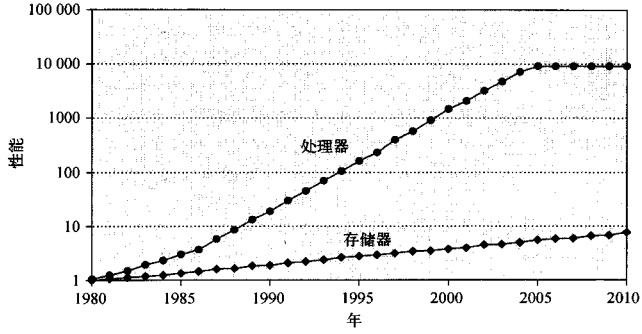


图 2-2 以 1980 年的性能为基准，处理器的存储器请求（对于单处理器或单核心）与 DRAM 访问延迟之间性能差距的时间变化曲线，该差距是通过测量存储器请求数和 DRAM 访问数目得出的。注意，为了记录处理器与 DRAM 性能差距的大小，纵轴必须采用对数刻度。存储器基准为 1980 年的 64 KB DRAM，延迟性能每年改进 1.07（见表 2-2）。处理器曲线假定 1986 年之前每年改进 1.25，2000 年之前每年改进 1.52，2000～2005 年每年改进 1.20，2005～2010 年处理器性能没有变化，见第 1 章的图 1-1

近来，高端处理器已经转向多核，与单核相比，进一步提高了带宽需求。事实上，总峰值带宽基本上随核心个数的增大而增大。现代高端处理器（比如 Intel Core i7）每个时钟周期可以由每个核心生成两次数据存储器引用，i7 有 4 个核心，时钟频率为 3.2 GHz，除了大约 128 亿次 128 位指令引用的峰值指令要求之外，每秒最多还可生成 256 亿次 64 位数据存储器引用；总峰值带宽为 409.6 GB/s！这一难以置信的高带宽是通过以下方法实现的：实现缓存的多端口和流水线；利用多级缓存，为每个核心使用独立的第一级缓存，有时也使用独立的第二级缓存；在第一级使用独立的指令与数据缓存。与其形成鲜明对比的是，DRAM 主存储器的峰值带宽只有它的 6%（25 GB/s）。

73

传统上，存储器层次结构的设计人员把重点放在优化存储器平均访问时间上，这一时间是由缓存访问时间、缺失率和缺失代价决定的。但最近，功率已经成为设计人员的主要考虑事项。在高端微处理器中，可能有 10 MB 或更多的片上缓存，大容量的第二或第三级缓存会消耗大量功率，既可能是在没有操作时作为泄露功率（称为**静态功率**），也可能是在执行读取或写入时的活动功率（称为**动态功率**），如 2.3 节所示。这一问题在 PMD 的处理器中甚至更为突出，它的 CPU 的主动性较低，功率预算可能降至 1/20～1/50。在此类情况下，缓存消耗的功率可能占总功耗的 25%～50%。因此，需要更多地考虑性能与功率之间的权衡，在本章将对这两个因素进行研究。

存储器层次结构基础：快速回顾

前面曾经提到，存储器性能和处理器性能之间的差距越来越大，这一点越来越重要，所以

存储器层次结构的基础知识已经出现在计算机体系结构本科课程中，甚至还出现在操作系统和编译器的相关课程中。因此，我们将首先快速回顾一下缓存及其操作。不过，本章的主要内容是介绍一些用来应对处理器-存储器性能差距的更高级创新技术。

如果在缓存中找不到某一个字，就必须从层次结构的一个较低层级（可能是另一个缓存，也可能是主存储器）中去提取这个字，并把它放在缓存中，然后才能继续。出于效率原因，会一次移动多个字，称为**块**（或**行**），这样做还有另外一个原因：由于空间局域性原理，很可能马上就会用到这些字。每个缓存块都包括一个**标记**，指明它与哪个存储器地址相对应。

在设计时需要作出一个非常重要的决策：哪些块（或行）可以放在缓存中。最常见的方案是**组相联**，其中组是指缓存中的一组块。一个块首先被映射到一个组上，然后可以将这个块放到这个组中的任意位置。要查找一个块，首先将这个块的地址映射到这个组，然后再搜索这个组（通常为并行搜索），以找到这个块。这个组是根据数据地址选择的：

（块地址）MOD（缓存中的组数）

如果组中有 n 个块，则缓存的布局被称为 n **路组相联**。组相联的端点有其自己的名字。**直接映射缓存**每组中只有一个块（所以块总是放在同一个位置），**全相联**缓存只有一个组（所以块可以放在任何地方）。

要缓存只读数据是一件很容易的事情，这是因为缓存和存储器中的副本是相同的。缓存写入难一些；比如，缓存和存储器中的副本怎样才能保持一致呢？共有两种主要策略。一种是**直写**（write-through）缓存，它会更新缓存中的项目，并直接写入主存储器，对其进行更新。另一种是**回写**（write-back）缓存，仅更新缓存中的副本。在马上要替换这个块时，再将它复制回存储器。这两种写入策略都使用了一种**写缓冲区**，将数据放入这个缓冲区之后，马上就可以进行缓存操作，不需要等待将数据写入存储器的全部延迟时间。

74

为了衡量不同缓存组织方式的优劣，可以采用一个名为**缺失率**的指标。**缺失率**是指那些未能找到预期目标的缓存访问所占的比例，即未找到目标的访问数目除以总访问数目。

为了深刻理解高缺失率的原因，从而更好地设计缓存，一种“3C”模式将所有这些缺失情景分为以下三个简单的类别。

- **强制**（Compulsory）：对一个数据块的第一次访问，这个块不可能在缓存中，所以必须将这个块调入缓存中。即使拥有无限大的缓存，也会发生强制缺失。
- **容量**（Capacity）：如果缓存不能包含程序运行期间所需要的全部块，就会因为有些块被放弃，之后再被调入，从而导致容量缺失（还有强制缺失）。
- **冲突**（Conflict）：如果块放置策略不是全相联的，如果多个块映射到一个块的组中，对不同块的访问混杂在一起，一个块可能会被放弃，之后再被调入，从而发生冲突缺失（还有强制缺失和容量缺失）。

表 B-4 和图 B-5 是根据 3C 模型对缓存缺失进行分解后，各种缓存缺失的相对频率。在第 3 章和第 5 章将会看到，多线程和多核都增加了缓存的复杂性，都加大了出现容量缺失的可能性，而且还因为缓存刷新增加了第 4 个 C———致性（Coherency）缺失，之所以进行缓存刷新是为了保持多处理器中多个缓存的一致性；我们将在第 5 章考虑这些问题。

要小心，缺失率可能会因为多个原因而产生误导。因此，一些设计人员喜欢测量**每条指令的缺失次数**，而不是每次存储器引用的缺失次数（缺失率）。这两者之间的关系如下：

$$\frac{\text{缺失数}}{\text{指令}}=\frac{\text{缺失率}\times\text{存储器访问}}{\text{指令数}}=\text{缺失率}\times\frac{\text{存储器访问}}{\text{指令}}$$

（为了避免出现小数，通常用每 1000 条指令的缺失次数来表示这一指标，使其变为整数形式。）

这两种度量指标的问题在于它们都没有考虑缺失成本的因素。一种更好的度量指标是**存储器平均访问时间**：

存储器平均访问时间=命中时间+缺失率×缺失代价

其中，**命中时间**是指在缓存中命中目标花费的时间，**缺失代价**是从内存中替代块的时间（即缺失成本）。存储器平均访问时间仍然是一个间接性能测量指标，尽管它比缺失率好一些，但不能代替执行时间。在第 3 章，我们将会看到推理处理器可以在缺失期间执行其他指令，从而降低实际缺失代价。多线程的使用（将在第 3 章介绍）也允许一个处理器容忍一些缺失，而不会被强制转入空闲状态。我们稍后将会研究，为了利用这些容忍延迟的技术，就需要一些缓存，以便在处理一次未完成缺失时为请求提供服务。 75

如果读者是新接触这一资料，或者说这一快速回顾浏览过快，可以参阅附录 B。它更深入地涵盖了同一介绍性内容，并包括一些真实计算机的缓存示例，还对其有效性进行了量化评估。

附录 B 的 B.3 节给出了 6 种基本的缓存优化方法，我们在此概括性地看一下。这个附录给出了这些优化方法收益的量化示例。我们还将对这些折中的功率因素进行简要评述。

(1) **增大块以降低缺失率**。这是降低缺失率的最简单方法，它利用了空间局域性，并增大了块的大小。使用较大的块可以减少强制缺失，但也增加了缺失代价。因为较大块减少了标记数目，所以它们可以略微降低静态功率。较大块还可能增大容量缺失或冲突缺失，特别是当缓存较小时尤为如此。选择合适的块大小是一项很复杂的权衡过程，具体取决于缓存的大小和缺失代价。

(2) **增大缓存以降低缺失率**。要减少容量缺失，一个显而易见的方法就是增大缓存容量。其缺点包括可能会延长较大缓存存储器的命中时间，增加成本和功率。较大的缓存会同时增大静态功率和动态功率。

(3) **提高相联程度以降低缺失率**。显然，提高相联程度可以减少冲突缺失。较大的相联程度是以延长命中时间为代价的。稍后将会看到，相联程度也会增大功耗。

(4) **采用多级缓存以降低缺失代价**。是加快缓存命中速度，以跟上处理器的高速时钟频率，还是加大缓存，以缩小处理器访问和主存储器访问之间的差距，这是一个艰难的决策。在原缓存和存储器之间加入另一级缓存可以降低这一决策的难度（见图 2-3）。第一级缓存可以非常小，足以跟上快速时钟频率，而第二级（或第三级）缓存可以非常大，足以收集容纳许多本来要对主存储器进行的访问。为了减少第二级缓存中的缺失，需要采用更大的块、更大的容量和更高的相联程度。与一级总缓存相比，多级缓存的功率效率更高。如果用 L1 和 L2 分别指代第一级和第二级缓存，可以将平均存储器访问时间重新定义为：

$$命中时间_{L1}+缺失率_{L1}\times（命中时间_{L2}+缺失率_{L2}\times缺失代价_{L2}）$$

(5) **为读取缺失指定高于写入操作的优先级，以降低缺失率**。写缓冲区是实现这一优化的好地方。因为写缓冲区拥有在读取缺失时所需位置的更新值，所以写缓冲区存在隐患，即通过存储器进行先写后读的隐患。一种解决方案就是在读取缺失时检查写缓冲区的内容。如果没有冲突，如果存储器系统可用，则在写入操作之前发送读取请求会降低缺失代价。大多数处理器为读取指定的优先级要高于写入操作。这种选择对功耗几乎没有什么影响。 76

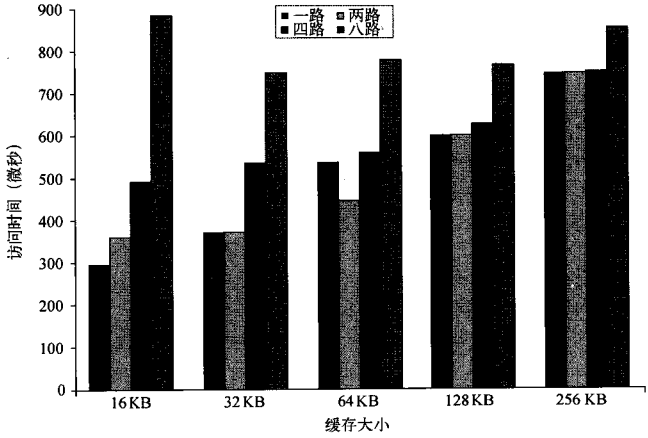


图 2-3 访问时间通常会随着缓存大小和相联程度的增大而增加。这些数据来自 Tarjan、Thoziyoor 和 Jouppi[2005]提出的 CACTI 模型 6.5。这些数据假定采用 40 nm 制程（介于 Intel 最快速 i7 和第二快速 i7 所用技术之间，与最快速的 ARM 嵌入处理器所用技术相同）、单一存储器组、块大小为 64 字节。由于对缓存布局所做的假定，以及在互连延迟（通常取决于正在访问的缓存块的大小）和标记检查与多工之间的复杂权衡，会得到一些有时看起来令人惊奇的结果，比如对于两路组相联的 64 KB 缓存，其访问时间会低于直接映射。与此类似，当缓存大小增大时，八路组相联产生的结果也会导致一些不同寻常的行为。由于这些观察结构都严重依赖于技术和具体设计假定，所以诸如 CACTI 之类的工具用于缩小权衡过程的搜索空间，而不是对权衡结果进行准确分析

(6) 在缓存索引期间避免地址转换，以缩短命中时间。缓存必须妥善应对从处理器虚拟地址到访问存储器的物理地址之间的转换。（虚拟存储器在 2.4 节和 B.4 节介绍。）一种常见的优化方法是使用页偏移地址（虚拟地址和物理地址中的相同部分）来索引缓存，如附录 B 所述。这种虚拟索引/物理标记方法增加了某些系统复杂度以及（或者）对 L1 缓存大小与结构的限制，但从关键路径消除转换旁视缓冲区（TLB）访问这一优点抵得过这些缺点。

77

注意，上述 6 种优化方法中，每一种都有自己潜在的弱点，可能会导致存储器平均访问时间的延长，而不是缩短。

本章后续部分假定读者熟悉上述材料及附录 B 中的详细内容。在“融会贯通”一节中，我们将研究 Intel Core i7 的存储器层次结构，Intel Core i7 是一种为高端服务器设计的微处理器，还将研究 Arm Cortex-A8 的存储器层次结构，Arm Cortex-A8 是一种为 PMD 设计的处理器，它是 Apple iPad 和几种高端智能手机所用处理器的基础。由于这些处理器是为不同计算机应用而设计的，所以每一类采用的方法都有明显的不同。与桌面计算机中使用的 Intel 处理器相比，服务器中使用的高端处理器拥有更多的核心和更大的缓存，但这些处理器的体系结构是类似的。差异性主要体现在性能和工作负载的性质上，桌面计算机在某一时刻只有一位用户，主要在操作系统的顶层运行一个程序，而服务器计算机可能会同时拥有数百位运行几十个应用程序的用户。由于这些工作负载的差异性，桌面计算机通常更多地考虑存储器层次结构带来的平均延迟，

而服务器计算机还要考虑存储器带宽。即使同为桌面计算机，也是品种繁多，既有低端的上网本（其处理器有所缩减，类似于高端 PMD 中使用的处理器），也有高端桌面计算机，其处理器包括多个核心，组成方式与低端服务器类似。

与之相对，PMD 不但只为一位用户提供服务，而且其操作系统通常要更小一些，通常很少采用多任务工作方式（同时运行几个应用程序），应用程序也要更简单一些。另外，PMD 通常采用闪存而不是磁盘，大多要同时考虑性能和能耗，能耗决定着电池寿命。

2.2 缓存性能的 10 种高级优化方法

上面的存储器平均访问时间公式提供了三种缓存优化度量：命中时间、缺失率和缺失代价。根据最近的发展趋势，我们向这个列表中添加了缓存带宽和功耗两个度量。根据这些度量，可以将我们研究的 10 种高级缓存优化方法分为以下 5 类。

(1) **缩短命中时间**。小而简单的第一级缓存和路预测。这两种技术通常还都能降低功耗。

(2) **增加缓存带宽**。流水化缓存、多组缓存和无阻塞缓存。这些技术对功耗具有不确定影响。 78

(3) **降低缺失代价**。关键字优化，合并写缓冲区。这两种优化方法对功率的影响很小。

(4) **降低缺失率**。编译器优化。显然，缩短编译时间肯定可以降低功耗。

(5) **通过并行降低缺失代价或缺失率**。硬件预取和编译器预取。这些优化方法通常会增加功耗，主要是因为提前取出了未用到的数据。

一般来说，在采用这些技术方法时，硬件复杂度会增加。另外，这些优化技术中有几种需要采用高级编译器技术。后面将在表 2-1 中总结这 10 种技术的实现复杂度和性能优势。对其中比较简单的优化方法仅作简单介绍，而对其他技术将给出更多描述。

2.2.1 第一种优化：小而简单的第一级缓存，用以缩短命中时间、降低功率

提高时钟频率和降低功率的双重压力都推动了对第一级缓存大小的限制。与此类似，使用较低级别的相联度，也可以缩短命中时间、降低功率，不过这种权衡要比限制大小涉及的权衡更复杂一些。

缓存命中过程中的关键计时路径由 3 个步骤组成：使用地址中的索引确定标记存储器的地址，将读取的标签值与地址进行比较。接下来，如果缓存为组相联缓存，则设置多路转换器以选择正确的数据项。直接映射的缓存可以将标记核对与数据传输重叠起来，有效缩短命中时间。此外，在采用低相联度时，由于减少了必须访问的缓存行，所以通常还可以降低功率。

尽管在各代新型微处理器中，片上缓存的总数已经大幅增加，但由于大容量 L1 缓存带来的时钟频率影响，所以 L1 缓存大小最近的涨幅很小，甚至根本没有增长。选择相联度时的另一个考虑因素是消除地址别名的可能性；我们稍后对此进行讨论。

使用 CAD 工具，可以在制造芯片之前判断各项选择对命中时间和功耗的影响。CACTI 程序可以在 CMOS 微处理器上估算各种缓存结构的访问时间和能耗，它的详细程度在 CAD 工具中排在前十名之内。对于一个给定的最小工艺尺寸，CACTI 估算在不同缓存大小、相联度、读/写端口数目和更复杂参数条件下的缓存命中时间。根据缓存大小的不同，对于模型建议的这些参数，直接映射的命中时间略快于两路组相联，两路组相联是四路的 1.2 倍，四路组相联是八路的 1.4 倍。当然，这些估计值受技术及缓存大小的影响。 79

例题　利用图 2-3 和附录 B 中表 B-4 中的数据，判断 32 KB 四路组相联 L1 缓存的存储器访问时间是否快于 32 KB 两路组相联 L1 缓存器。假定 L2 的缺失代价是快速 L1 缓存访问时间的 15 倍。忽略超出 L2 之外的缺失。哪种缓存的存储器平均缓存时间较短？

解答　设两路组相联缓存的访问时间为 1。则，对于两种缓存：

$$\text{存储器平均访问时间}_{\text{两路}}=\text{命中时间}+\text{缺失率}\times\text{缺失代价}$$
$$=1+0.038\times15=1.57$$

对于四路缓存，访问速度是它的 1.4 倍。缺失代价占用的时间为 15/1.4=10.1。为简单起见，设其为 10：

$$\text{存储器平均访问时间}_{\text{四路}}=\text{命中时间}_{\text{两路}}\times1.4+\text{缺失率}\times\text{缺失代价}$$
$$=1.4+0.037\times10=1.77$$

显然，采用较高的相联度看起来是一种糟糕的权衡选择；不过，由于现代处理器中的缓存访问通常都实现了流水化，所以很难评估对时钟周期时间的确实影响。

如图 2-4 所示，在选择缓存大小和相联度时，能耗也是一个考虑因素。在 128 KB 或 256 KB 缓存中，当从直接映射变到两路组相联时，高相联度的能量消耗从大于 2 变到可以忽略。

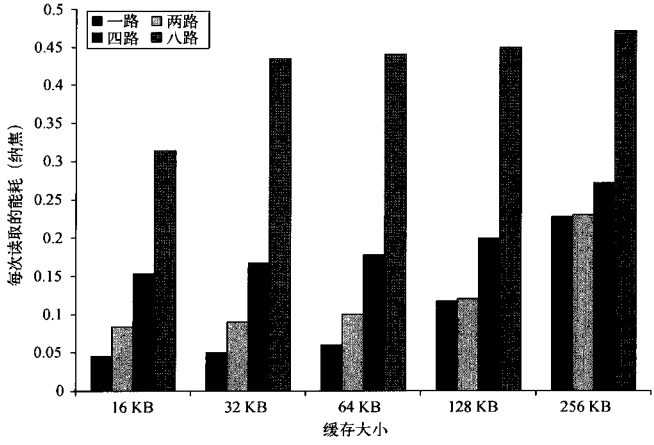


图 2-4　每次读取操作的能耗随缓存大小、相联度的增加而增加。和图 2-3 一样，使用 CACTI 对相同技术参数进行建模。八路组相联缓存的代价之所以很高，是因为并行读取 8 个标签及相应数据的成本造成的

在最近的设计中，有三种其他因素导致了在第一级缓存中使用较高的相联度。第一，许多处理器在访问缓存时至少需要两个时钟周期，因此命中时间较长可能不会产生太过严重的影响。第二，将 TLB 排除在关键路径之外（TLB 带来的延迟可能要大于高相联度导致的延迟），几乎所有 L1 缓存都应当是变址寻址的。这就将缓存的大小限制为页大小与相联度的乘积，这是因为

只有页内的位才能用于变址。在完成地址转换之前对缓存进行变址的问题还有其他一些解决方案，但提高相联度是最具吸引力的一种，它还有其他一些好处。第三，在引入多线程之后（参见第 3 章），冲突缺失会增加，从而使提高相联度更具吸引力。

80

2.2.2 第二种优化：采用路预测以缩短命中时间

这是另外一种可以减少冲突缺失，同时又能保持直接映射缓存命中速度的方法。在**路预测技术**中，缓存中另外保存了一些位，用于预测下一次缓存访问组中的路或块。这种预测意味着尽早设定多工选择器，以选择所需要的块，在与缓存数据读取并行的时钟周期内，只执行一次标签比较。如果缺失，则会在下一个时钟周期中查看其他块，以找出匹配项。

在一个缓存的每个块中都添加块预测位。根据这些位选定要在下一次缓存访问中优先尝试哪些块。如果预测正确，则缓存访问延迟就等于这一快速命中时间。如果预测错误，则尝试其他块，改变路预测器，延迟会增加一个时钟周期。模拟表明，对于一个两路组相联缓存，组预测准确度超过 90%；对于四路组相联缓存，超过 80%，对 I-缓存的准确度优于对 D-缓存的准确度。如果路预测能够至少快 10%（这是非常可能的），路预测方法可以缩短两路组相联缓存的存储器平均访问时间。路预测最早于 20 世纪 90 年代中期应用在 MIPS R10000 中。这在使用两路组相联缓存的处理器中很常见，在使用四路组相联缓存的 ARM Cortex-A8 中也采用了这一技术。对于非常快速的处理器，要实现一个周期的时延是非常具有挑战性的，而这对于降低路预测代价非常关键。

81

还有一种扩展形式的路预测，它使用路预测位来判断实际访问的缓存块，可以用来降低功耗（路预测位基本上就是附加地址位）；这种方法也可称为**路选择**，当路预测正确时，它可以节省功率，但在路预测错误时则会显著增加时间，这是因为需要重复进行访问，而不仅是重复标记匹配与选择过程。这种优化方法只有在低功率处理器中才可能有意义。Inoue、Ishihara 和 Murakami [1999]根据 SPEC95 基准测试进行估算，对于四路组相联缓存使用路选择方法，可以使 I-缓存的平均访问速度提高 1.04 倍，D-缓存提高 1.13 倍，与正常的四路组相联缓存相比，I-缓存的平均缓存功耗降为原来的 0.28，D-缓存降为 0.35。路选择方法的一个重要缺点就是它增大了实现缓存访问流水化的难度。

例题 假定在一个普通四路组相联实现中，D-缓存访问次数是 I-缓存访问次数的一半，I-缓存和 D-缓存分别占该处理器功耗的 25%～15%。根据上述研究的估计值，判断路选择方法是否提高了每瓦功耗的性能。

解答 对于 I-缓存，节省的功率为总功率的 $25 \times 0.28=0.07$，对于 D-缓存，为 $15 \times 0.35=0.05$，共节省 0.12。路预测版本需要的功率为标准四路缓存的 0.88。缓存访问时间的增加等于 I-缓存平均访问时间的增加量加上 D-缓存访问时间增加量的一半，即 $1.04+0.5 \times 0.13=1.11$ 倍。这一结果意味着路选择的性能是标准四路缓存的 0.90。因此，路选择方法略微提高了每焦功耗的性能，比值为 0.90/0.88=1.02。这一优化方法最适合用在看重功率胜于看重性能的情景。

2.2.3 第三种优化：实现缓存访问的流水化，以提高缓存带宽

这种优化方法就是实现缓存访问的流水化，使第一级缓存命中的实际延迟可以分散到多个

时钟周期，从而缩短时钟周期时间、提高带宽，但会减缓命中速度。例如，对于20世纪90年代中期的Intel Pentium处理器，指令缓存访问的流水线需要1个时钟周期，对于从20世纪90年代中期到2000年的Pentium Pro到Pentium III，需要2个时钟周期，对于2000年出现的Pentium4和现在的Intel Core i7，需要4个时钟周期。这一变化增加了流水线的段数，增加了预测错误分支的代价，延长了从发出载入指令到使用数据之间的时钟周期数（参见第3章），但的确更便于采用高相联度的缓存。

82

2.2.4 第四种优化：采用无阻塞缓存，以提高缓存带宽

对于允许乱序执行（在第3章讨论）的流水化计算机，其处理器不必因为一次数据缓存缺失而停顿。例如，在等待数据缓存返回缺失数据时，处理器可以继续从指令缓存中提取指令。**无阻塞缓存**（或称为**自由查询缓存**）允许数据缓存在一次缺失期间继续提供缓存命中，进一步加强了这种方案的潜在优势。这种“缺失时仍然命中”优化方法在缺失期间非常有用，它不会在此期间忽略处理器的请求，从而降低了实际缺失代价。还有一种精巧而复杂的选项：如果能够重叠多个缺失，那缓存就能进一步降低实际缺失代价，这种选项被称为“多次缺失时仍然命中”（hit under multiple miss）或者“缺失时缺失”（miss under miss）优化方法。只有当存储器系统可以为多次缺失提供服务时，第二种选项才有好处，大多数高性能处理器（比如Intel Core i7）通常都支持这两种优化方法，而低端处理器，比如ARM A8，仅在L2提供了有限的无阻塞支持。

为了研究无阻塞缓存在降低缓存缺失代价方面的有效性，Farkas和Jouppi [1994]做了一项研究：假定8 KB缓存的缺失代价为14个周期；在缺失情况下允许一次命中时，他们都观测到实际缺失代价的降低，对于SPECINT92基准测试降低20%，对于SPECFP92基准测试降低30%。

Li、Chen、Brockman和Jouppi[2011]最近对这一研究内容进行了更新，他们采用多级缓存，根据最近的技术情况对缺失代价作出了假设，并采用了规模更大、要求更严格的SPEC2006基准测试。这项研究采用一种基于Intel i7单核心的模型（见2.6节）运行SPEC2006基准测试。图2-5显示了一次缺失时允许1、2、64次命中时，数据缓存访问延迟的降低情况。自从研究早期就采用了大缓存容量、增加了L3缓存，所以采用“缺失时命中”的好处显得不够明显，SPECINT2006基准测试显示缓存延迟平均降低大约9%，而采用SPECFP2006基准测试结果大约为12.5%。

例题 对于浮点程序来说，主数据缓存的两路组相联和一次缺失时仍然命中，哪个更重要一些？对整数程序来说呢？假定32 KB数据缓存的平均缺失率如下：对于采用直接映射缓存的浮点程序为5.2%，对于采用两路组相联缓存的浮点程序为4.9%，采用直接映射缓存的整数程序为3.5%，对于采用两路组相联缓存的整数程序为3.2%。假定L2的缺失代价为10个周期，L2的缺失率和缺失代价相同。

解答 对于浮点程序，存储器平均停顿时间为

$$缺失率_{DM} \times 缺失代价 = 5.2\% \times 10 = 0.52$$

83

$$缺失率_{两路} \times 缺失代价 = 4.9\% \times 10 = 0.49$$

两路相联缓存的访问延迟（包括停顿）为0.49/0.52，或者为直接映射缓存的94%。图2-5的标题表明允许一次缺失时仍然命中，可以将浮点程序的平均数据缓存访问延迟缩短为一次阻塞缓存的87.5%。因此，对于浮点程序而言，支持一次缺失时仍然命中一次的直接映射数据缓存，在性能上要优于在缺失时阻塞的两路相联缓存。

对于整数程序，计算如下：

$$缺失率_{DM} \times 缺失代价=3.5\% \times 10=0.35$$

$$缺失率_{两路} \times 缺失代价=3.2\% \times 10=0.32$$

两种组相联缓存的数据缓存访问延迟为 0.32/0.35，或直接映射缓存的 91%，而在一次缺失时允许一次命中可以将访问延迟降低 9%，使两种选择的性能大体相当。

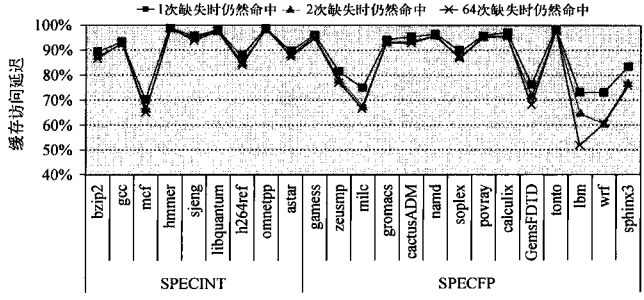


图 2-5 用 9 个 SPECINT（左）和 9 个 SPECFP（右）基准测试评估在 1、2 或 64 次缓存缺失时仍然允许命中情况下非阻塞缓存的效果。这个数据存储器系统以 Intel i7 建模，它包括 32 KB L1 缓存，访问延迟为 4 个周期。L2 缓存（与指令共享）为 256 KB，访问延迟为 10 个时钟周期。L3 为 2 MB，访问延迟为 35 个周期。所有访问都是八路组相联，大小为 64 字节块大小。在缺失时允许一次命中可以使整数基准测试的缺失代价降低 9%，浮点基准测试降低 12.5%。允许第二次命中会将这些结果提高 10% ~ 16%，得到 64 个没有多少额外改进的结果

对非阻塞缓存进行性能评估时，真正的难度在于一次缓存缺失不一定会使处理器停顿。在这种情况下，很难判断一次缺失造成的影响，因此也就难以计算存储器平均访问时间。实际缺失代价并不等于这些缺失时间之和，而是处理器的非重叠停顿时间。非阻塞缓存的优势评估非常复杂，因为它取决于存在多次缺失时的缺失代价、存储器引用模式以及处理器在有未处理缺失时能够执行多少条指令。

84

通常，对于一个能够在 L2 缓存中命中的 L1 数据缓存缺失，乱序处理器能够隐藏其大部分缺失代价，但不能隐藏更低层次缓存缺失的大部分代价。而在决定支持多少个未处理缺失时，需要考虑多种因素，如下所述。

- 缺失流中的时间与空间局域性，它决定了一次缺失是否会触发对低级缓存或对存储器的新访问操作。
- 对访问请求作出回应的存储器或缓存的带宽。
- 为了允许在最低级别的缓存中出现更多的未处理缺失（在这一级别的缺失时间是最长的），需要在较高级别上支持至少数目相等的缺失，这是因为这些缺失必须在最高级别缓存上启动。

- 存储器系统的延迟。

下面这个简化示例表明了这一关键思想。

例题 假定主存储器的访问时间为36ns，存储器系统的持续传输速率为16 GB/s。设块大小为64字节，如果在给定请求流的情况下能够保持峰值带宽，而且访问永远不会冲突，则需要支持的最大未完成缺失数目为多少？如果一次引用与前4次引用发生冲突的概率为50%，估计最大未完成引用数目。为简单起见，忽略缺失之间的时间。

解答 在第一种情况下，假定我们可以保持峰值带宽，存储器系统可以支持每秒$(16 \times 10)^9/64 = 250$百万次引用。由于每次引用耗时36 ns，因此可以支持$250 \times 10^6 \times 36 \times 10^{-9} = 9$次引用。如果发生冲突的概率大于0，由于我们不能在这些引用时开始工作，所以需要更多的未完成引用；存储器系统需要的独立引用数应当更多而不是更少！为了估计这一数目，可以假定有一半存储器引用不需要发送到存储器。这就意味着必须能够支持两倍数量的未完成引用，即18次。

Li、Chen、Brockman和Jouppi在研究中发现：对于在缺失时允许1次命中的情况，整数程序的CPI减少大约7%，而在允许64次命中时，减少大约12.7%。对于浮点程序，允许1次命中时减少12.7%，64次时减少17.8%。这些减少数值与图2-5中所示数据缓存访问延迟的减少值非常接近。

2.2.5 第五种优化：采用多种缓存以提高缓存带宽

我们可以将缓存划分为几个相互独立、支持同时访问的缓存组，而不是将它们看作一个整体。分组方式最初用于提高主存储器的性能，现在也用于DRAM芯片和缓存中。Arm Cortex-A8在其L2缓存中支持1至4个缓存组；Intel Core i7的L1中有4个组（每个时钟周期内可以支持2次存储器访问），L2有8个组。

85

显然，当访问请求很自然地分布在缓存组之间时，分组方式的效果最佳，所以将地址映射到缓存组的方式影响着存储器系统的行为。一种简单有效的映射方式是将缓存块地址按顺序分散在这些缓存组中，这种方式称为顺序交错。例如，如果有4个缓存组，0号缓存组中的所有缓存块地址对4求模后余0，1号缓存组中的所有缓存块地址对4求模后余1，以此类推。图2-6显示了这种交错方式。采用分组方式还可以降低缓存和DRAM中的功耗。

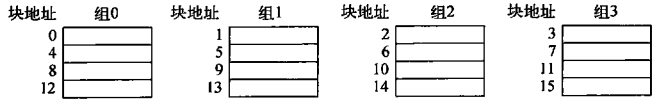


图2-6 使用块寻址的四路交错缓存组。假定每个块64个字节，这些地址需要分别乘以64才能实现字节寻址

2.2.6 第六种优化：关键字优先和提前重启动以降低缺失代价

这一技术的事实基础是人们观测到处理器在某一时刻通常仅需要缓存块的一个字。这一策

略显得“不够耐心”：无须等待完成整个块的载入过程，一旦载入被请求字之后，立即将其发出，然后就重启处理器。下面是两个特定策略。

- **关键字优先**：首先从存储器中请求缺失的字，在其到达缓存之后立即发给处理器；使处理器能够在载入块中其他字时继续执行。
- **提前重启动**：以正常顺序提取字，但只要块中的被请求字到达缓存，就立即将其发送给处理器，让处理器继续执行。

通常，只有在采用大型缓存块的设计中，这些技术才有用武之地，如果缓存块很小，它们带来的好处是很低的。注意，在载入某个块中的其余内容时，缓存通常可以继续满足对其他块的访问请求。

不过，根据空间局域性原理，下一次引用很可能就会指向这个块的其余内容。和非阻塞缓存一样，其缺失代价的计算也不是非常容易。在采用关键字优先策略时，如果存在第二次请求，实际缺失代价等于从本次引用开始到第二部分内容到达之前的非重叠时间。关键字优先策略和提前重启动策略的好处取决于块的大小以及在尚未获取某部分内容时又出现另一次访问的机率。

86

2.2.7 第七种优化：合并写缓冲区以降低缺失代价

因为所有存储内容都必须发送到层次结构的下一层级，所以直写缓存依赖于写缓冲区。即使是写回缓存，在替代一个块时也会使用一个简单的缓冲区。如果写缓冲区为空，则数据和整个地址被写到缓冲区中，从处理器的角度来看，写入操作已经完成；在写缓冲区准备将字写到存储器时，处理器继续自己的工作。如果缓冲区中包含其他经过修改的块，则检查它们的地址，看看新数据的地址是否与写缓冲区中有效项目的地址匹配。如果匹配，则将新数据与这个项目合并在一起。这种优化方法称为写合并。Intel Core i7 和其他许多处理器都采用了写合并方法。

如果缓冲区已满，而且没有匹配地址，则缓存（和处理器）必须等待，直到缓冲区中拥有空白项目为止。由于多字写入的速度通常快于每次只写入一个字的写入操作，所以这种优化方法可以更高效地使用存储器。Skadron 和 Clark [1997]发现：即使在一个合并四项的写缓冲区中，所生成的停顿也会导致 5% ~ 10%的性能损失。

这种优化方式还会减少因为写缓冲区已满而导致的停顿。图 2-7 显示了一个写缓冲区在采用和不采用写合并时的情况。假定这个写缓冲区中有四项，每一项有 4 个 64 位的字。在采用这一优化方法时，图中的 4 个字可以完全合并，放在写缓冲区的一个项目中，而在不采用这一优化方法时，对写缓冲区的连续地址执行 4 次存储操作，将整个缓冲区填满，每个项目中保存一个字。

注意，输入/输出设备寄存器经常被映射到物理地址空间。由于 I/O 寄存器是分享的，不能像存储器中的字数组那样操作，所以这些 I/O 地址不能允许写合并。例如，它们可能要求为每个 I/O 寄存器提供一个地址和一个数据字，而不能只提供一个地址进行多字写入。为了实现这些副作用，通常由缓存将这些页面进行标记，表明其需要采用非合并直写方式。

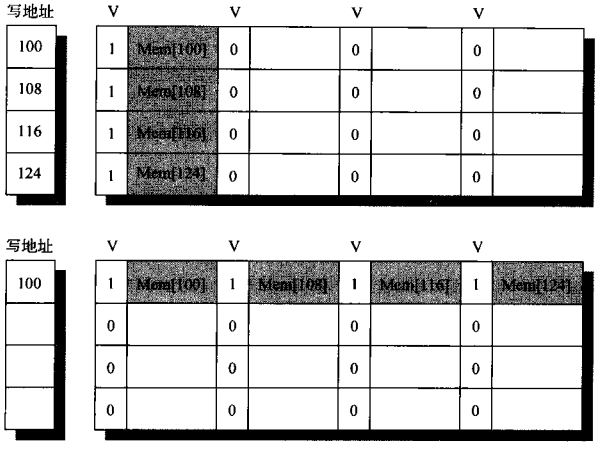


图 2-7 为了说明写合并过程，上面的写缓冲区未采用该技术，下面的写缓冲区采用了这一技术。在进行合并时，4 次写入内容被合并到一个缓冲区项目中；而未进行合并时，4 次写入操作就填满了整个缓冲区，每个项目的四分之三被浪费。这个缓冲区有四个项目，每一项保存 4 个 64 位字。每个项目的地址位于左侧，有效位（V）指明这个项目的下面 8 个连续字节是否被占用（未采用写合并时，图中上半部分右侧的字只会用于同时写多个字的指令。）

2.2.8 第八种优化：采用编译器优化以降低缺失率

前面介绍的技术都需要改变硬件。下面这种技术可以在不做任何硬件改变的情况下降低缺失率。

这种神奇的降低效果来自软件优化——硬件设计人员最喜爱的解决方案！处理器与主存储器之间的性能差距越拉越大，已经促使编译器编写人员深入研究存储器的层次结构，以了解能否通过编译时间优化来提高性能。同样，研究内容包括两个方面：指令缺失性能改进和数据缺失性能改进。下面给出的优化技术在很多现代编译器中均有应用。

87

1. 循环交换

一些程序中存在嵌套循环，它们会以非连续顺序访问存储器中的数据。只要交换一下这些循环的嵌套顺序，就可能使程序代码按照数据的存储顺序来访问它们。如果缓存中无法容纳这些数组，这一技术可以通过提高空间局域性来减少缺失；通过重新排序，可以使缓存块中的数据在被替换之前，得到最大限度的充分利用。例如，设 x 是一个大小为[5000,100]的两维数据，其分配方式使得 x[i,j]和 x[i,j+1]相邻（由于这个数组是按行进行排列的，所以我们说这种顺序是以行为主的），以下两段代码说明可以怎样来优化访问过程：

```
/*优化之前*/
for (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5000; i = i+1)
            x[i][j] = 2 * x[i][j];
```

88

```
/*优化之后*/
for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];
```

原代码以 100 个字的步幅跳跃式浏览存储器，而修改后的版本在访问了一个缓存块中的所有字之后才进入下一个块。这一优化方法提高了缓存性能，却没有影响到所执行的指令数目。

2. 分块

这一优化方法通过提高时间局域性来减少缺失。这一次还是要处理多个数组，有的数组是按行来访问的，有的是按列来访问的。由于在每个循环迭代中都用到了行与列，所以按行或按列来存储数组并不能解决问题（按行存储称为行主序，按列存储称为列主序）。这种正交访问方式意味着在进行循环交换之类的转换操作之后，仍然有很大的改进空间。

分块算法不是对一个数组的整行或整列进行操作，而是对其子矩阵（或称块）进行操作。其目的是在缓存中载入的数据被替换之前，在最大限度上利用它。下面这段执行矩阵乘法的示例可以帮助理解这一优化方法的动机：

```
/*优化之前*/
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;
              };
```

两个内层循环读取 Z 的所有 N×N 个元素，重复读取 y 中一行的同一组 N 个元素，再写入 X 的一行 N 个元素。图 2-8 是访问这三个数组的一个快照。深色阴影区域表示最近的访问，浅色阴影区域表示较早的访问，白色表示还没有进行访问。

容量缺失的数目显然取决于 N 和缓存的大小。如果它能容纳所有这 3 个 N×N 矩阵，只要没有缓存冲突，就一切正常。如果缓存可以容纳一个 N×N 矩阵和包含 N 个元素的一行，则至少 y 的第 i 行和数组 Z 可以停留在缓存中。如果缓存的容量更小，那可能对于 x 和 z 都会发生缺失。在最差情况下，进行 N^3 次操作可能需要从存储器中读取 $2N^3+N^2$ 个字。

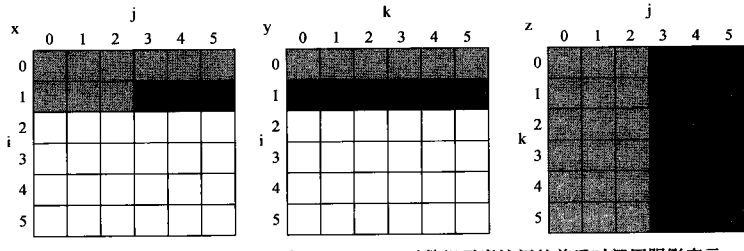


图 2-8 三个数组 x、y 和 z 的快照，其中 N=6，i=1。对数组元素访问的前后时间用阴影表示：白色表示还没有被访问过，浅色表示较早的访问，深色表示最近的访问。与图 2-9 相对照，为了计算 x 的新元素，会重复读取 y 和 z 的元素。在行或列的旁边显示了用于访问这些数组的变量 i、j 和 k

为了确保正在访问的元素能够放在缓存中，对原代码进行了修改，改为计算一个大小为 B×B 的子矩阵。两个内层循环现在以大小为 B 的步长进行计算，而不是以 x 和 z 的完整长度为步

长。B被称为分块因子。（假定x被初始化为0。）

```
/*优化之后*/
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,N); j = j+1)
          {r = 0;
           for (k = kk; k < min(kk+B,N); k = k + 1)
                r = r + y[i][k]*z[k][j];
           x[i][j] = x[i][j] + r;
          };
```

图 2-9 展示了使用分块方法对三个数组的访问。仅观察容量缺失，从存储器中访问的总字数为 $2N^3/B+N^2$。这一总数的改善因素大约为B。由于y获益于空间局域性，z获益于时间局域性，所以分层方法综合利用了空间局域性和时间局域性。

尽管我们的目标是降低缓存缺失，但分块方法也可用于帮助寄存器分配。通过设定一个较小的分块大小，使这个块能够保存在寄存器中，可以在最大程度上降低程序中的载入与存储数量。

在4.8节将会看到，对于以矩阵为主要数据结构的程序，要在基于缓存的处理器上获得出色性能，缓存分块方法是必不可少的。

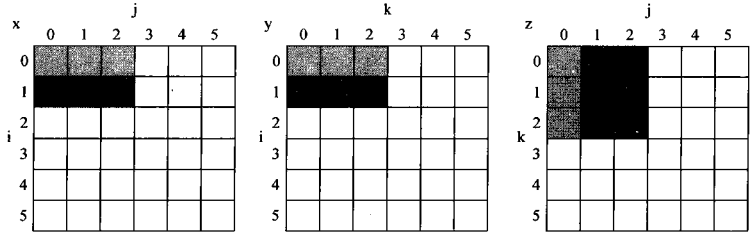


图 2-9 当B=3时，对数组x、y和z的访问历史。注意，与图2-8相比，访问的元素数目较少

2.2.9 第九种优化：对指令和数据进行硬件预取，以降低缺失代价或缺失率

通过执行过程与存储器访问的重叠，无阻塞缓存有效地降低缺失代价。而另一种方法是在处理器请求项目之前，预先提取它们。指令和数据都可以预先提取，既可以直接放在缓存中，也可以放在一个访问速度快于主存储器的外部缓冲区中。

指令预取经常在缓存外部的硬件中完成。通常，处理器在一次缺失时提取两个块：被请求块和下一个相邻块。被请求块放在它返回时的指令缓存中，预取块被放在指令流缓冲区中。如果被请求块当前存在于指令流缓冲区中，则原缓存请求被取消，从流缓冲区中读取这个块，并发出下一条预取请求。

类似方法可应用于数据访问[Jouppi 1990]。Palacharla和Kessler [1994]研究了一组科学程序，并考查了多个可能处理指令或数据的流缓冲区。他们发现，对于一个具有两个64 KB四路组相联缓存的处理器（一个用于缓存指令，另一个用于缓存数据），八个流缓冲区可以捕获其所有缺失的50%～70%。

Intel core i7支持利用硬件预先提取到L1和L2中，最常见的预取情况是访问下一行。一些

较早的 Intel 处理器使用更主动的硬件预取，但会导致某些应用程序的性能降低，一些高级用户会因此而关闭这一功能。

图 2-10 显示了在启用硬件预取时，部分 SPEC2000 程序的整体性能改进。注意，这一数字仅包含 12 个整数程序中的 2 个，而大多数 SPEC 浮点程序则都包含在内。

91

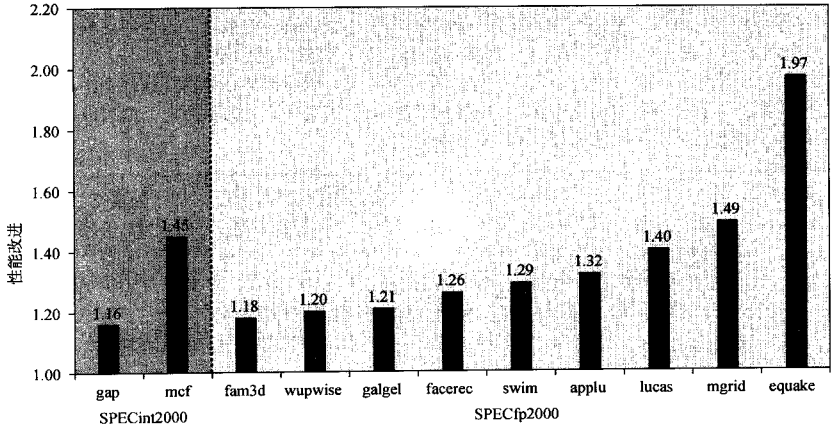


图 2-10 在 intel Pentium 4 上启用硬件预取之后，12 个 SPECint2000 基准测试中的 2 个测试、14 个 SPECfp2000 基准测试中的 9 个测试获得的加速比。图中仅给出从预取中获益最多的程序，对于图中未给出的 15 个 SPEC 基准测试，预取加速比低于 15%[Singhal 2004]

预取操作需要利用未被充分利用的存储器带宽，但如果它干扰了迫切需要的缺失内容，反而可能会实际降低性能。在编译器的帮助下，可以减少无用预取。当预取操作正常执行时，它对功率的影响可以忽略。如果未用到预取数据或者替换了有用数据，预取操作会对功率产生非常负面的影响。

2.2.10 第十种优化：用编译器控制预取，以降低缺失代价或缺失率

作为硬件预取的替代方法，可以在处理器需要某一数据之前，由编译器插入请求该数据的预取指令。共有以下两种预取。

- **寄存器预取**将数据值载入到一个寄存器中。
- **缓存预取**仅将数据载入到缓存中，而不是寄存器中。

这两种预取都可能是**故障性的**，也都可能是**非故障性的**；也就是说，其地址可能会导致虚拟地址错误异常和保护冲突异常，也可能不会导致。利用这一术语，正常的载入指令可能被看作是“故障性寄存器预取指令”。非故障性预取只是在通常导致异常时转换为空操作，而这正是我们想要的结果。

92

最有效的预取对程序来说是“语义透明的”：它不会改变寄存器和存储器的内容，**也不会**导致虚拟存储器错误。今天的大多数处理器都提供非故障性缓存预取。本节假定非故障性缓存预取，也称为**非绑定**预取。

只有当处理器在预取数据时能够继续工作的情况下，预取才有意义；也就是说，缓存在等待返回预取数据时不会停顿，而是继续提供指令和数据。可以想到，这些计算机的数据缓存通常是非阻塞性的。

与硬件控制的预取操作类似，这里的目标也是将执行过程与数据预取过程重叠起来。循环是重要的预取优化目标，因为它们本身很适于进行预取优化。如果缺失代价很小，编译器只需要将循环展开一两次，在执行时调度这些预取操作。如果缺失代价很大，它会使用软件流水线（参见附录H）或者将循环展开多次，以预先提取数据，供后续迭代使用。

不过，发出预取指令会导致指令开销，所以编译器必须非常小心地确保这些开销不会超过所得到的好处。如果程序能够将注意力放在那些可能导致缓存缺失的引用上，就可以避免不必要的预取操作，同时明显缩短存储器平均访问时间。

例题 对于以下代码，判断哪些访问可能导致数据缓存缺失。然后，插入预取指令，以减少缺失。最后，计算所执行的预取指令数目和通过预取避免的缺失数。假设有一个8 KB直接映射的数据缓存，块大小为16字节，它是一个执行写入分派的写回缓存。a和b是双精度浮点数组，所以它们的元素长8个字节。a有3行、100列，b有101行、3列。另外假定在程序开始时，这些数据没有在缓存中。

```
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];
```

解答 编译器首先判断哪些访问可能导致缓存缺失，否则可能会为那些能够命中的数据发出预取指令，白白浪费时间。a的元素是以它们在存储器中的存储顺序写入的，所以a可以受益于空间局域性：j的偶数值会缺失，奇数值会命中。由于a有3行、100列，所以对它的访问将会导致3×(100/2)=150次缺失。

数组b不会从空间局域性中获益，因为对它的访问不是按照存储顺序执行的。数组b可以从时间局域性中获得双重受益：每次对i进行迭代时会访问相同元素，每次对j进行迭代时使用的b元素值与上一次迭代相同。忽略可能存在的冲突缺失，由b导致的缺失在i=0时访问b[j+1][0]出现，还有就是在j=0时首次访问b[j][0]出现。当i=0时，j从0增至99，所以对b的访问将导致100+1=101次缺失。

因此，这次循环将会出现的数据缓存缺失大约包括a的150次和b的101次，也就是251次缺失。

为了简化优化过程，我们不用费心为循环中的第一次访问进行预取。这些内容可能已经放在缓存中了，或者我们需要为a或b的前几个元素承担缺失代价。在到达循环末尾时，预取操作会尝试提前提取超出a末端之外的内容（a[i][100]...a[i][106]）和b末端之外的内容（b[101][0]...b[107][0]），我们并不需要费心来禁用这些预取。如果这是一些故障性预取，那我们也许不能承担如此之大的开销。我们假定缺失代价非常大，需要至少提前（比如）7次迭代来开始预取。（换句话说，我们假定在第8次迭代之前进行预取不会得到任何好处。）以下代码中加下划线的部分，是为了添加预取优化而对前面代码所做的修改。

```
for (j = 0; j < 100; j = j+1) {
        prefetch(b[j+7][0]);
        /* b(j,0) for 7 iterations later */
        prefetch(a[0][j+7]);
        /* a(0,j) for 7 iterations later */
        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]);
                /* a(i,j) for +7 iterations */
                a[i][j] = b[j][0] * b[j+1][0];}
```

这一经过修订的代码预取 a[i][7]至 a[i][99]和 b[7][0]至 b[100][0]，将非预取缺失的数目减至：

- 第一次循环中访问元素 b[0][0]，b[1][0]，…，b[6][0]时的 7 次缺失
- 第一次循环中访问元素 a[0][0]，a[0][1]，…，a[0][6]时的 4 次缺失（[7/2]）（利用空间局域性将缺失数减少为每 16 字节缓存块一次缺失）
- 第二次循环中访问元素 a[1][0]，a[1][1]，…，a[1][6]的 4 次缺失（[7/2]）
- 第二次循环中访问元素 a[2][0]，a[2][1]，…，a[2][6]的 4 次缺失（[7/2]）

即总共 19 次非预取缺失。避免 232 次缓存缺失的成本是执行了 400 条预取指令，这很划算。

例题　计算上面例题中节省的时间。忽略指令缓存缺失，并假定数据缓存中没有冲突缺失或容量缺失。假定预取过程可以相互重叠，并能与缓存缺失重叠。因此可以用最高存储器带宽传输。下面是忽略缓存缺失的关键循环次数：原循环每次迭代需要 7 个时钟周期，第一次预取循环每次迭代需要 9 个时钟周期，第二次预取循环每次迭代需要 8 个周期（包含循环外部的开销）。一次缺失需要 100 个时钟周期。

94

解答　原来的双层嵌套循环执行 3 × 100=300 次。由于该循环每次迭代需要 7 个时钟周期，所以总共为 300 × 7 = 2100 时钟周期再加上缓存缺失。缓存缺失增加 251 × 100=25 100 个时钟周期，总共 27 200 个时钟周期。第一次预取循环迭代 100 次，每次迭代为 9 个时钟周期，所以总共 900 个时钟周期再加上缓存缺失。现在加上缓存缺失的 11 × 100=1100 个时钟周期，总共 2000 个时钟周期。第二个循环执行 2 × 100=200 次，每次迭代为 8 个时钟周期，所以需要 1600 个时钟周期，再加上缓存缺失 8 × 100=800 个时钟周期。总共 2400 个时钟周期。由上个例子可以知道，这一代码为了执行这两个循环，在 2000+2400=4400 个时钟周期内执行了 400 条预取指令。如果我们假定这些预取操作完全与其他执行过程相重叠，那么这一预取代码要快 27 200/4400=6.2 倍。

尽管数组优化理解起来比较容易，但现代程序更可能使用指针。Luk 和 Mowry [1999]已经证明，基于编译器的预取优化有时也可以扩展到指针。在 10 个使用递归数据结构的程序中，在访问一个节点时预取所有指针，可以使半数以上的程序性能提高 4% ~ 31%。而另一方面，其他程序的性能变化不超过原性能的 2%。问题症结包括两个方面，一是要预先提取的数据是否已经存在于缓存中，二是预取数据能否在需要它们时提前到达缓存。

许多处理器支持缓存预取指令，高端处理器（比如 Intel Core i7）还经常在硬件中完成某种类型的自动预取。

2.2.11　缓存优化小结

用于改进命中时间、带宽、缺失代价及缺失率等性能的技术通常会影响到存储器平均访问时间公式的其他部分，而且还会影响到存储器层次结构的复杂性。表 2-1 总结了这些技术，并估计了它们对复杂性的影响，其中“+”表明该技术会改进该项因素，“－”表示会损害该项因素，空白则表示没有影响。一般来说，没有哪种技术能够改进多项因素。

95

表 2-1　10 种高级缓存优化技术小结，给出了对缓存性能、功耗和复杂度的影响

技　术	命中时间	带　宽	缺失代价	缺失率	功　耗	硬件成本/复杂度	注　释
小而简单的缓存	+			－	+	0	很普通、应用广泛
路预测缓存	+				+	1	在Pentium 4中使用
流水化缓存访问	－	+				1	应用广泛
非阻塞缓存		+	+			3	应用广泛
分组缓存		+			+	1	在i7和Cortex-A8的L2中使用
关键字优先和提前重启动			+			2	应用广泛
合并写缓冲区			+			1	和直写法一起广泛应用
以编译器技术减少缓存缺失				+		0	软件是一个挑战，但许多编译器都能处理常见的线性代数计算
指令与数据的硬件预取			+	+	－	2（指令） 3（数据）	大多提供预取指令，现代高端处理器还会在硬件中实现自动预取
编译器控制的预取			+	+		3	需要非阻塞缓存，可能存在指令开销；在许多CPU中得到应用

* 尽管一种技术通常仅能改进一个因素，但如果能够尽早完成，预取可以减少缺失；如果不能尽早完成，则可以减少缺失代价。“+”表示该技术会改进该项因素，“－”表示会损害该项因素，空白则表示没有影响。复杂度的评估具有主观性，0表示最容易，3标志着一项挑战。

2.3　存储器技术与优化

……唯一使计算机站稳脚跟的发展成果就是可靠存储器的发明，也就是核心存储器……它的成本在合理范围内，它是可靠的，而且因为它是可靠的，所以能够在适当的时候扩大其容量。[第209页]

——Maurice Wilkes
Memoirs of a Computer Pioneer (1985)

主存储器在层次结构上位于缓存的下一层级。主存储器满足缓存的需求，并充当 I/O 接口，既是输入的目的地，又是输出的源头。对主存储器的性能度量同时强调延迟和带宽。传统上，主存储器延迟（它会影响缓存缺失代价）是缓存的主要考虑因素，而主存储器带宽则是微处理器和 I/O 的主要考虑因素。

96

尽管缓存可以从低延迟存储器中获益，但一般来说，通过新的组织方式来提高存储器带宽要比降低延迟更容易一些。由于多级缓存日益普及，而且采用较大的分块，使主存储器带宽对于缓存也非常重要。事实上，缓存设计人员增大块大小就是为了利用更高的存储器带宽。

前面几节描述了可以通过缓存优化来缩小处理器与 DRAM 之间的性能差距，但仅仅增大缓存或增加更多的缓存级别并不能消除这一差距。主存储器也需要进行革新。

在过去，对主存储器的革新就是改变构成主存储器的众多 DRAM 芯片（比如多个存储器组）的组织方式。在使用存储器组时，通过拓宽存储器或其总线，或者同时加宽两者，可以提供更大的带宽。但是，具有讽刺意味的是，随着单片存储器芯片容量的增加，在同样大小的存储器系统中，所需要的芯片越来越少，从而降低了存储器系统容量不变、带宽加大的可能性。

为使存储器系统跟上现代处理器的带宽需求，存储器革新开始在 DRAM 芯片自身内部展开。本节描述存储器芯片内部的技术及其具有创新性的内部组成。在描述这些技术与选项之前，我们首先来复习一下存储器的性能度量指标。

随着突发传输存储器的引入（这种存储器现在在 Flash 和 DRAM 中都得到广泛应用），存储器延迟主要由两部分组成——访问时间和周期时间。**访问时间**是发出读取请求到收到所需字之间的时间，**周期时间**是指对存储器发出两次不相关请求之间的最短时间。

自 1975 年以来，几乎所有计算机都采用 DRAM 作为主存储器，采用 SRAM 作为缓存，在处理器芯片中集成一到三级缓存，与 CPU 集成在一起。在 PMD 中，存储器技术经常要在功率和速度之间进行平衡，高端系统会使用快速、高带宽存储器技术。

2.3.1 SRAM技术

SRAM 的第一个字母表示静态（static）。DRAM 电路的动态本质要求在读取数据之后将其写回，因此在访问时间和周期时间之间存在差异，并需要进行刷新。SRAM 不需要刷新，所以存在时间与周期时间非常接近。SRAM 通常使用 6 个晶体管保存 1 位数据，以防止在读取信息时对其造成干扰。在待机模式中，SRAM 只需要很少的功率来维持电荷。

在早些时候，大多数桌面系统和服务器系统使用 SRAM 芯片作为其主、辅或第三缓存；今天，所有这三级缓存都被集成在处理器芯片上。目前最大的片上第三级缓存为 12 MB，而这样一种处理器的存储器系统可能拥有 4 ~ 16 GB 的 DRAM。大容量第三级片上缓存的访问时间通常是第二级缓存的 2 ~ 4 倍，而它仍然要比访问 DRAM 存储器快 3 ~ 5 倍。 97

2.3.2 DRAM技术

早期 DRAM 的容量增大时，由于封装上需要提供所有必要的地址线，所以封装成本较高。解决方案是对地址线进行复用，从而将地址管脚数砍去一半。图 2-11 给出基本的 DRAM 组成结构。先在**行选通**（RAS）期间发送一半地址。然后在**列选通**（CAS）期间发送另一半地址。**行选通**和**列选通**等名字源于芯片的内部结构，这些存储器的内部是一个按行和列寻址的长方形矩阵。

对 DRAM 的另一要求来自其第一个字母 D 表示的特性，即**动态**（dynamic）。为了在每个芯片中容纳更多的位，DRAM 仅使用一个晶体管来存储一位数据。信息的读取过程会破坏该信息，所以必须进行恢复。这是 DRAM 周期时间一般要长于访问时间的原因之一；近来，DRAM 已经引入了多个组，从而可以隐藏访问周期中的重写部分，见图 2-11。此外，为了防止在没有读写某一个位时丢失信息，必须定期“刷新”该位。幸运的是，只需对一行进行读取就可以同

时刷新该行。因此，存储器系统中的每个 DRAM 必须在某一特定时间窗口内（比如 8 ms）访问每一行。存储器控制器包括定期刷新 DRAM 的硬件。

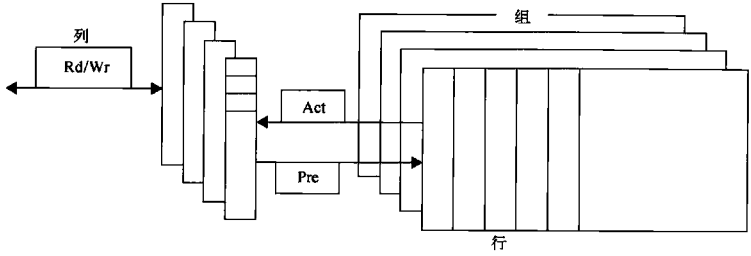


图 2-11 DRAM 中的内部组成结构。现代 DRAM 是以“组”为单位进行组织的，DDR3 通常有 4 组。每一组由一系列行构成。发送 PRE（precharge）命令会打开或关闭一个组。行地址随 Act（activate）命令发送，该命令会将一个行传送到缓冲区中。将行放入缓冲区后，就可以采用两种方式进行传送，一种是根据 DRAM 的宽度采用连续列地址传送（在 DDR3 中，这一宽度通常为 4 位、8 位、16 位），另一种是指定块传送方式，并给出起始地址。每个命令和块传送过程，都以一个时钟进行同步

这一要求意味着存储器系统偶尔会有不可用时间，因为它要发出一个信号，告诉每个芯片进行刷新。刷新时间通常是对 DRAM 中每一行进行完整存储器访问（RAS 和 CAS）的时间。由于 DRAM 中的存储器矩阵在概念上是方形的，所以一次刷新中的步骤通常是 DRAM 容量的平方根。DRAM 设计人员尽力将刷新时间保持在总时间的 5%以下。

98

到目前为止，我们对主存储器运作模式的描述就好像瑞士火车一样，完全根据时刻表持续不断地提供货物。而刷新操作与这一类比有些矛盾，因为某些访问操作花费的时间要比其他访问操作长得多。因此，刷新操作是存储器延迟发生变化的另一个原因，从而也会影响缓存缺失代价的变化。

Amdahl 提出一条经验规律：要保持系统的平衡，存储器容量应当随处理器的速度线性增长，所以一个运算速度为 1000 MIPS 的处理器应当拥有 1000 MB 的存储器。处理器设计人员依靠 DRAM 来满足这一要求。过去，他们可以指望存储器容量每三年翻两番，也就是年增长率为 55%。遗憾的是，DRAM 现在的性能增长速度非常慢。表 2-2 给出了行访问时间的性能改变，它与延迟有关，每年大约为 5%。CAS 或数据传输时间与带宽有关，其增长速度超过上述速度的两倍。

尽管我们前面讨论的是独立芯片，但 DRAM 通常是放在被称为**双列直插存储模块**（DIMM）的小电路板上出售的。DIMM 通常包括 4 ~ 16 个 DRAM，对于桌面系统和服务器系统，它们通常被组织为 8 字节宽度（+ECC）。

99

除了 DIMM 封装和用于提高数据传输速度的新接口之外（这些内容将在下面几个小节中讨论），DRAM 的最大改变是容量增长速度已经变缓。DRAM 在 20 年里都符合摩尔定律，每三年推出一代容量增长 4 倍的新芯片。从 1998 年开始，由于单比特 DRAM 所面对的制造难题，只能每两年推出一代容量加倍的新芯片。2006 年，增长速度进一步减缓，从 2006 年到 2010 年的 4 年时间里，容量只提高了 1 倍。

表 2-2 每一代 DRAM 的最快速、最慢速访问时间

生产年份	芯片大小	DRAM类型	行选通（RAS）		列选通（CAS）/数据传输时间（ns）	周期时间（ns）
			最慢DRAM (ns)	最快DRAM (ns)		
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

*（周期时间的定义参见73页。）行访问时间的性能改进为每年大约5%。1986年，由于从NMOS DRAM转为CMOS DRAM，所以列访问性能提高了2倍。20世纪90年代中期引入的各种突发传输模式以及20世纪90年代后期引入的SDRAM显著增加了数据块访问时间的计算复杂度。在本节后面讲到SDRAM访问时间和功率时将对此进行讨论。DDR4设计预计会在2012年的中后期完成。我们将在后面几页中讨论这些不同形式的DRAM。

2.3.3 提高DRAM芯片内部的存储器性能

晶体管的增长速度仍然符合摩尔定律，存储器与处理器之间的性能差距对存储器性能施加的压力也是日益增大，所以上一节介绍的各种思想已经开始寻求在DRAM芯片闯出一条道路。概括来说，有些创新技术已经提高了存储器的带宽，但有时是以增大延迟为代价的。本节将介绍一些充分利用DRAM自身特性的技术。

前面曾经提到，对DRAM的访问分为行访问和列访问两部分。DRAM必须在DRAM内部缓冲一行中的所有位，为列访问做好准备，一行的位数通常等于DRAM大小的平方根——比如，4 Mbit DRAM的每行有2 Kbit。随着DRAM容量的增大，添加了一些附加结构，也多了几种提高带宽的可能性。

第一，DRAM添加了定时信号，允许重复访问行缓冲区，从而节省了行访问时间。这种缓冲区的出现非常自然，因为每个数组会为每次访问操作缓冲1024～4094位信息。最初，在每次传输时都需要发送不同的列地址，从而在每组新的列地址之后都有一定的延迟时间。

DRAM原来有一个与存储器控制器相连的异步接口，所以每次传输都需要一定的开销，以完成与控制器的同步。第二，向DRAM接口添加一个时钟信号，使重复传输不需要承担这一开销。这种优化的名字就是同步DRAM（SDRAM）。SDRAM通常还有一个可编程寄存器，用于保存请求字节数，因此可以在几个周期内为每个请求发送许多字节。通常，将DRAM置于突发模式，不需要发送任何新地址就能进行8次或更多次的16位传输；这种模式支持关键字优先传输方式，是唯一能够达到表2-3所示峰值带宽的方法。

第三，随着存储器系统密度的增加，为了从存储器获得较宽的比特流，而又不必使存储器系统变得过大，人们拓展了DRAM的宽度。原来提供一种4位传输模式；2010年、DDR2和DDR3 DRAMS已经拥有宽达16位的总线。

第四种提高带宽的 DRAM 重要创新是在 DRAM 时钟信号的上升沿和下降沿都传输数据，从而使峰值数据传输率加倍。这种优化方法被称为**双倍数据率**（DDR）。

为了提供交错（interleaving）的一些优点，并帮助进行电源管理，SDRAM 还引入了**分组**，将一块 SDRAM 分为 2～8 个可以独立运行的块（在目前的 DDR3 DRAM 中为 8 块）。（我们已经见过分组在内部缓存中的应用，它们也经常用于大容量主存储器中。）在一个 DRAM 中创建多个组可以有效地添加另一个地址段，它现在由组号、行地址和列地址组成。在发送指定一个新组的地址时，这个组必须已经打开，会增加一些延迟。分组与行缓冲区的管理完全由现代存储器控制接口处理，这样，如果后续地址指定了一个开放组的相同行时，只需要发送列地址就能快速完成访问。

100

在将 DDR SDRAM 封装为 DIMM 形式时，会用 DIMM 峰值带宽进行标记，这种标记方法很容易引起混淆。比如，之所以得到 PC2100 这样一个 DIMM 名称，是源于 133 MHz × 2 × 8 字节=2100 MB/s。为了避免这种混淆，在对芯片本身进行标记时，给出的不是其时钟频率，而是**每秒比特数**，因此一个 133 MHz 的 DDR 芯片被称为 DDR266。表 2-3 给出了时钟频率、每芯片每秒钟的传送数目、芯片名称、DIMM 带宽和 DIMM 名称之间的关系。

表 2-3 2010 年 DDR DRAM 和 DIMM 的时钟频率、带宽和名称

标 准	时钟频率（MHz）	每秒传输个数（百万个）	DRAM名称	MB/s/DIMM	DIMM名称
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10 664	PC10700
DDR3	800	1600	DDR3-1600	12 800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17 056–25 600	PC25600

* 请注意各列之间的数字关系。第三列是第二列的 2 倍，第四列 DRAM 芯片名称中使用了第三列中的数字。第五列是第三列的 8 倍，DIMM 名称中使用了这一数字的四舍五入值。根据 DDR 标准，DDR 还以 4 个数字指明了时钟周期中的延迟，但本表没有给出这一信息。例如，DDR3-2000 CL 9 的延迟为 9-9-9-28。这是什么意思呢？对于一个周期为 1 ns 的时钟（时钟周期是传输速度的一半），这一数字意味着行至列地址为 9 ns（RAS 时间），列地址到数据为 9 ns（CAS 时间），最短读取时间为 28 ns。关闭该行将需要 9 ns 进行预充电，但只有在完成该行的读取时才会进行这一操作。在突发模式中，经过第一 RAS 时间与 CAS 时间之后，在每个时钟的上下沿都会进行传输。此外，在读取整行之前不需要预充电。DDR4 将于 2012 年投产，预计在 2014 年将达到 1600 MHz 的时钟频率，届时将会升级至 DDR5。在练习中将进一步探讨这些细节。

DDR 现在已经成为一个标准序列。DDR2 将电压由 2.5 伏降至 1.8 伏，从而降低了功率，并提供了更高的时钟频率：266 MHz、333 MHz 和 400 MHz。DDR3 将电压降至 1.5 伏，最高时钟频率为 800 MHz。计划于 2014 年投产的 DDR4 将电压降到 1 至 1.2 伏，预计最高时钟频率为 1600 MHz。DDR5 大约会在 2014 年或 2015 年跟进。（在下一节将会讨论，GDDR5 是一种图形 RAM，以 DDR3 DRAM 为基础。）

101

图形数据 RAM

GDRAM 或 GSDRAM（图形 DARM 或图形同步 DRAM）是一种特殊类型的 DRAM，它们以 SDRAM 设计为基础，但为满足图形处理器中的高带宽需求进行了特别定制。GDDR5 以 DDR3 为基础，较早的 GDDR 以 DDR2 为基础。GPU（见第 4 章）对每个 DRAM 芯片的带宽要求要

高于 CPU，因此，GDDR 有以下几点重要不同。

(1) GDDR 的接口更宽，为 32 位，而目前的设计为 4、8 或 16 位。

(2) GDDR 数据管脚上的时钟频率更高。为了在不招致信号发送问题的前提下提高传输速率，GDRAMS 通常直接与 GPU 相连，将它们焊接在电路板上，这一点与 DRAM 不同，它们通常放置在可扩展的 DIMM 阵列中。

这些特性综合在一起，使 GDDR 每块 DRAM 的带宽比 DDR3 DRAM 高 2～5 倍，可以更好地为 GPU 提供支持。由于 GPU 中存储器请求的局域性较低，所以突发模式对 GPU 一般没有太大用处，但保持多个存储器组的开放状态并合理安排其应用，可以提高有效带宽。

2.3.4 降低SDRAM中的功耗

动态存储器芯片中的功耗由静态（或待机）功率和读写期间消耗的动态功率构成，这两者取决于工作电压。在大多数高端 DDR3 SDRAM 中，工作电压已经降到 1.35～1.5 伏，与 DDR2 SDRAM 相比，显著降低了功率。分组的增加也降低了功率，这是因为每次仅读取并预充电一个分组中的行。

除了这些变化之外，所有最新 SDRAM 都支持一种省电模式，通知 DRAM 忽略时钟即可进入这一模式。省电模式会禁用 SDRAM，但内部自动刷新除外（如果没有自动刷新，当进入省电模式时间长于刷新时间后，将会导致存储器内容丢失）。图 2-12 给出了一个 2 Gbit DDR3 SDRAM 在三种情况下的功耗。从低功率模式返回正常模式所需的确切延迟时间取决于 SDRAM，但从自动刷新低功率模式返回正常状态的时间一般为 200 个时钟周期；在第一个命令之前复位模式寄存器可能需要延长一些时间。

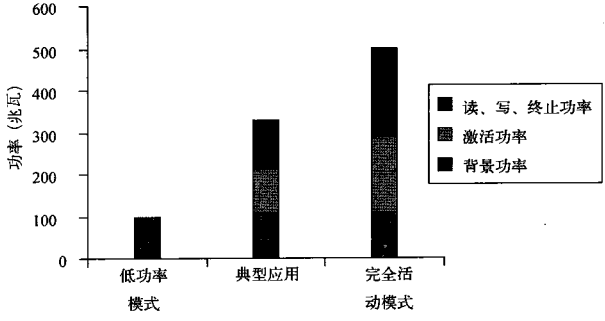


图 2-12 DDR3 SDRAM 在 3 种不同运行条件下的功耗：低功率（关闭）模式、典型系统模式。（在读取操作中，DRAM 有 30%的时间处于活动状态，在写入操作中有 15%的时间处于活动状态）和完全活动模式，在这种模式下，DRAM 在非预充电状态下持续读取或写入。读取和写入采用由 8 次传输组成的突发形式。这些数据系根据 Micron 1.5V 2 Gbit DDR3-1066 测得

2.3.5 闪存

闪存是一种 EEPROM（电可擦可编程只读存储器），它通常是只读的，但可以擦除。闪存的另一个重要特性是能在没有任何供电的情况下保存其内容。

闪存在PMD中用作备份存储，其工作方式与笔记本型计算机或服务器中的磁盘相同。此外，由于大多数PMD的DRAM数量有限，所以闪存在很大程度上还要充当存储器层次结构的一级，这种可能要比在桌面计算机或服务器中高出很多，因为后者的主存储器可能要大出10~100倍。

闪存使用的体系结构与标准DRAM有很大不同，性质也有所不同。最重要的区别在于以下几方面。

(1) 在改写内存之前，必须对其进行擦除（因此，“闪存”名字中的“闪”字就是指快速擦除的意思），（在高密度闪存中，称为NAND闪存，大多数计算机都采用这种闪存）它的擦除过程是按块进行的，而不是单独擦除各个字节或各个字。这意味着在需要向闪存中写入数据时，必须对整个块进行处理，或者全是新数据，或者将要写入的数据与块中的其他内容合并在一起。

(2) 闪存是静态的（也就是说，即使在没有供电的情况下，它也能保持其内容），在未进行读写时，只消耗非常低的一点功率（在待机模式下会低于一半，在完全非活动状态上可以为零）。

(3) 对任何一个块来说，闪存为其提供有限数目的写入周期，通常至少为100 000个。这样，系统可以确保写入块均匀分布在整个存储器中，从而在最大程度上延长闪存系统的寿命。

(4) 高密度闪存比SDRAM便宜，但比磁盘贵：闪存的价格大约是2美元/GB，SDRAM为20美元到40美元/GB，磁盘为0.09美元/GB。

(5) 闪存的速度比SDRAM慢得多，但比磁盘快得多。例如，从一个典型高密度闪存传送256字节数据大约需要6.5 μs（它使用的突发传送模式与SDRAM类似，但要慢一些）。从DDR SDRAM进行类似传输需要的时间大约长四分之一，而从磁盘上传输则大约慢1000倍。对于写入操作，这种区别更是大得多，SDRAM至少比闪存快10倍，也可能达到100倍，具体数值取决于环境。

103

高密度闪存在过去十年里的快速发展，已经使这一技术成为移动设备存储器层次结构中最具活力的组成部分，也成为磁盘的固态替代技术。随着DRAM密度增长速度的持续下降，闪存将在未来存储器系统中扮演越来越重要的角色，既可取代硬盘，也可作为DRAM与磁盘之间的中间存储。

2.3.6 提高存储器系统的可靠性

缓存和主存储器容量的增大也大幅提高了在制造过程期间和对存储器单元进行动态冲击时（主要来自宇宙射线）出现错误的概率。这些动态错误会改变存储器单元的内容，但不会改变电路，称之为**软错误**。所有DRAM、闪存和许多SRAM在制造时都留有备用行，这样可以容忍少量的制造缺陷，只需要通过编程方式用备用行替代缺陷行即可。除了必须在配置期间纠正的制造错误之外，还可能在运行时发生**硬错误**，它可能会永久改变一个或多个存储器单元的运行方式。

动态错误可以使用奇偶校验位检测，可以使用纠错码（ECC）检测和纠正。因为指令缓存是只读的，用奇偶校验位就足够了。在更大型的数据缓存和主存储器中，则使用ECC技术来检测和纠正错误。奇偶校验位只需要占用一个数据位就可以检测一系列数据位中的一个错误。由于无法使用奇偶校验位来检测多位错误，所以必须限制用奇偶校验位提供保护的位数。一个常用比值是每8个数据位使用一个奇偶校验位。如果采用ECC技术，在每64个数据位中，8位的开销成本可以检测两个错误，纠正一个错误。

在规模庞大的系统中，出现多个错误乃至单个存储器芯片完全失效的机率都大大增加。IBM引入了Chipkill来解决这一问题，许多大规模系统，比如IBM和SUN服务器和Google Clusters都使用这一技术。（Intel将其自己的版本命名为SDDC。）Chipkill在本质上类似于磁盘中使用RAID方法，它分散数据和ECC信息，在单个存储器芯片完全失效时，可以从其余存储器芯片中重构丢失数据。根据IBM的分析，假定有10 000个服务器（每个处理器有4 GB存储器），在三年的运行中出现不可恢复错误的数目如下所示。

- 仅采用奇偶校验位——大约90 000个，或者说每17分钟一个不可恢复（或未检测到）的故障。
- 仅采用ECC——大约3500个，或者说大约每7.5小时一个不可恢复（或未检测到）的故障。
- Chipkill——6个，或者说大约每2个月一个不可恢复（或未检测到）的故障。

104

另一种研究方法是在错误率与Chipkill保持一致的情况下求出其他两种方式能够保护的最大服务器数目（每个服务器拥有4 GB存储器）。采用奇偶校验位方法，即使是一个仅包括一个处理器的服务器，其不可恢复错误率也要高于由10 000个服务器组成、受Chipkill保护的系统。采用ECC方法，一个包含17个服务器的系统与包含10 000个服务器的Chipkill系统的故障率大体相同。对于仓库级计算机中的50 000～100 000个服务器来说，需要采用Chipkill方法（见6.8节）。

2.4 保护：虚拟存储器和虚拟机

> 虚拟机被用作真实机器的一种高效、独立副本。我们通过虚拟机监视器（VMM）的思想来解释这些概念……VMM有三个基本特性。第一，VMM为程序提供了一种与原机器基本相同的运行环境；第二，在这种环境中运行的程序最糟糕的情况也不过是速度略有降低；最后，VMM可以完全控制系统资源。
>
> ——Gerald Popek和Robert Goldberg
> "Formal requirements for virtualizable third generation architectures,"
> *Communications of the ACM*（1974年7月）

2011年，在信息技术面对的最令人困扰的挑战中，安全与保密占据了两个席位。人们不断看到有关电子窃案的报道，其中经常涉及大量信用卡号，人们普遍认为还有大量此类案件没有报道出来。因此，研究人员和业内人员都在探寻能够提高计算机系统安全性的新方法。信息保护并非仅限于硬件方面，我们认为真正的安全与保密需要在计算机体系结构和系统软件方面均有所创新。

本节首先回顾体系结构如何通过虚拟内存来保护进程，避免它们相互伤害。接下来介绍虚拟机增加的保护措施、虚拟机在体系结构方面的需求以及虚拟机的性能。在第6章将会看到，虚拟机是实现云计算的基础技术。

2.4.1 通过虚拟存储器提供保护

页式虚拟存储器（包括缓存页表项目的变换旁视缓冲区）是保护进程免受相互伤害的主要机制。附录B的B.4节、B.5节回顾了虚拟存储器的相关内容，详细介绍了80x86中通过分段、

分页提供的保护。这一小节仅作一个快速回顾；如果过于简短，请参考上述章节。

105 多道程序设计（multiprogramming，几个同时运行的程序共享一台计算机的资源）需要在各个程序之间提供保护和共享，从而产生了进程概念。打个比方，进程就是一个程序呼吸的空气、生存的空间——也就是一个正在运行的程序加上它持续运行所需要的全部状态。在任意时刻，必须能够从一个进程切换到另一个进程。这种交换被称为**进程切换**或**上下文切换**。

操作系统和体系结构联合起来就能使进程共享硬件而不会相互干扰。为此，在运行一个用户进程时，体系结构必须限制用户进程能够访问的资源，但要允许操作系统进程访问更多资源。体系结构至少要做到以下几点。

(1) 提供至少两种模式，指出正在运行的进程是用户进程还是操作系统进程。后者有时被称为**内核进程**或**管理进程**。

(2) 提供一部分处理器状态信息，用户进程可以使用但不能写入。这种状态信息包括用户/管理模式位、异常启用/禁用位和存储器保护位。之所以禁止用户写入这些状态信息，是因为如果用户可以授予自己管理权限、禁用异常或者改变存储器保护，那操作系统就不能控制用户进程了。

(3) 提供处理器借以从用户模式转为管理模式及反向转换的机制。前一种转换通常通过**系统调用**完成，使用一种特殊指令将控制传递到管理代码空间的一个专用位置。保存系统调用时刻的程序计数器，处理器转入管理模式。返回用户模式的过程类似于一个全程返回过程，恢复到先前的用户/管理模式。

(4) 提供限制存储器访问的机制，在上下文切换时不需要将一个进程切换到磁盘就能保护该进程的存储器状态。

附录 A 介绍了几种存储器保护机制，但到目前为止，最流行的机制还是添加对虚拟存储器各个页面的保护性限制。固定大小的页面（通常长 4 KB 或 8 KB）通过一个页表由虚拟地址空间映射到物理地址空间。这些保护性限制就包含在每个页表项中。保护性限制可以决定一个用户进程能否读取这个页面，一个用户进程能否写这个页面以及能否从这个页面执行代码。此外，如果一个进程没有包含在页表中，那它就既不能读取也不能写入一个页面。由于只有操作系统才能更新页表，所以分页机制提供了全面的访问保护。

分页虚拟存储器意味着每次存储器访问在逻辑上都要花费至少两倍的时间，一次存储器访问用以获取物理地址，第二次访问用于获取数据。这种成本可能太过高昂了。解决方案就是依靠局域性原理，如果这些访问具有局域性，那么访问操作的地址转换也肯定具有局域性。只要将这些地址转换放在一个特殊的缓存中，存储器访问就很少再需要第二次访问操作来转换地址了。这种特殊的地址转换缓存被称为**变换旁视缓冲区**（TLB）。

106

TLB 项目类似于缓存项目，其中的标记保存虚拟地址部分，数据部分保存物理页地址、保护字段、有效位，通常还有一个使用位和一个更改位（dirty bit）。操作系统在改变这些位时，改变页表中的值，然后使相应的 TLB 项失效。当这个项目重新载入到页表中时，TLB 即获得这些位的准确副本。

如果计算机严格遵守对页面的限制，将虚拟地址映射到物理地址，那看起来我们就万事大吉了。但报纸头条向人们描述的可是另一番景象。

我们之所以还有任务要做，是因为我们依赖于操作系统和硬件的准确性。今天的操作系统由数千万行代码组成。由于 Bug 的计算是以每千行代码中的个数为单位的，所以在一个生产操

作系统中可能会出现数千个 bug。操作系统中的缺陷产生了经常会被利用的系统漏洞。

与过去相比，现在不实施保护措施所导致的代价很可能要比过去高昂得多，这种可能性的存在再加上上述问题，已经促使人们寻找一种保护模型，其基本代码要比完整的操作系统小许多，比如虚拟机就是这样一种模型。

2.4.2 通过虚拟机提供保护

有一个与虚拟存储器相关而且几乎与它一样古老的概念，那就是**虚拟机**（VM）。它们最早是在 20 世纪 60 年代后期提出的，多年以来一直是大型机计算的重要组成部分。尽管它在 20 世代 80 年代和 90 年代的单用户计算机领域被广泛忽视，但近来再度得到广泛关注，原因如下：

- 隔离与安全在现代系统中的重要性提高；
- 标准操作系统的安全性和可靠性出现问题；
- 许多不相关用户（比如一个数据中心或云中的用户）共享同一计算机；
- 处理器原始速度的飞速增大，使虚拟机的开销更容易接受。

最广义的虚拟机定义基本上包括了所有提供标准软件接口的仿真方法，比如 Java VM。我们感兴趣的是那些在二进制指令集体系结构（ISA）级别提供完整系统级环境的虚拟机。最常见的情况是，VM 支持的 ISA 与底层硬件相同；但也有可能支持不同的 ISA，在 ISA 之间迁移时经常采用这种方法，这样，在能够迁移到新 ISA 之前，使软件能够继续在原 ISA 上使用。在我们重点关注的虚拟机中，所提供的 ISA 与其底层硬件相匹配。这种虚拟机称为（操作）**系统虚拟机**。IBM VM/370、VMware ESX Server 和 Xen 都属于此类虚拟机。它们让虚拟机用户感觉到自己拥有整个计算机，包括操作系统的副本在内。一台计算机可以运行多个虚拟机，可以支持多种不同操作系统（OS）。在传统平台上，一个操作系统“拥有”所有硬件资源，但在使用虚拟机时，多个操作系统一起共享硬件资源。

107

为虚拟机提供支持的软件称为**虚拟机监视器**（VMM）或管理程序，VMM 是虚拟机技术的核心。底层硬件平台称为**主机**，其资源在来宾 VM 之间共享。VMM 决定了如何将虚拟资源映射到物理资源：物理资源可以分时共享、划分，甚至可以在软件内模拟。VMM 要比传统操作系统小得多，VMM 的一个隔离部分大约只有 10 000 行代码。

一般来说，处理器虚拟化的成本取决于工作负载。用户级别的处理器操作密集型程序（比如 SPEC CPU 2006）的虚拟化开销为零，这是因为很少会调用操作系统，所有程序都以原速运行。与之相对的是 I/O 操作密集的工作负载，它们通常也会大量涉及操作系统，执行许多系统调用（以满足 I/O 需求）和可能导致高虚拟化开销的特权指令。这一开销的大小取决于必须由 VMM 模拟的指令数目和模拟这些指令的缓慢程度。因此，根据我们的假定，如果一个来宾 VM 运行的 ISA 与主机相同，则这个体系结构和 VMM 的目的就是直接在原始硬件上运行几乎所有指令。另一方面，如果涉及大量 I/O 操作的工作负载也是 I/O 密集型的，由于处理器经常要等待 I/O，所以处理器虚拟化的成本可以完全被较低的处理器利用率所隐藏。

尽管我们这里关心的是 VM 提供保护的功能，但 VM 还提供了其他两个具有重要商业价值的优点。

(1) 软件管理——VM 提供一种能够运行整个软件栈的抽象层，甚至包括诸如 DOS 之类的旧操作系统。一种典型部署是用一部分 VM 运行原有操作系统，大量 VM 运行当前稳定的操作

系统版本，而一少部分 VM 用于测试下一代 OS 版本。

(2) **硬件管理**——需要多个服务器的原因之一是希望每个应用程序都能在独立的计算机上与其兼容的操作系统一起运行，这种分离可以提高系统的可靠性。VM 允许这些分享软件栈独立运行，却共享硬件，从而减少了服务器的数量。还有一个例子，一些 VMM 允许将正在运行的 VM 迁移到不同计算机上，既可能是为了平衡负载，也可能是为了撤出发生故障的硬件。

这两个原因说明了云服务器（比如 Amazon 的云服务器）为什么要依赖于虚拟机的原因。

2.4.3 对虚拟机监视器的要求

一个 VM 监视器必须完成哪些任务？它向来宾软件提供一个软件接口，必须使不同来宾的状态相互隔离，还必须保护自己免受客户端软件的破坏（包括来宾操作系统）。定性需求包括：

108

- 来宾软件在 VM 上的运行情况应当与在原始硬件上完全相同，当然，与性能相关的行为或者因为多个 VM 共享固定资源所造成的局限性除外；
- 来宾软件应当不能直接修改实际系统资源的分配。

为了实现处理器的“虚拟化”，VMM 必须控制几乎所有操作——特权状态的访问、地址转换、I/O、异常和中断，即使目前正在运行的来宾 VM 和操作系统正在临时使用它们，也不应当影响到这些控制。

例如，在计时器中断时，VMM 将挂起当前正在运行的来宾 VM，保存其状态、处理中断、判断接下来运行哪个来宾 VM，然后载入其状态。依赖计时器中断的来宾 VM 都会有一个由 VMM 提供的虚拟计时器和仿真计时器。

为了进行管理，VMM 的管理权限必须高于来宾 VM，后者通常运行于用户模式；这样还能确保任何特权指令的执行都由 VMM 处理。系统虚拟机的基本需求几乎与上述分页虚拟存储器的要求相同。

- 至少两种处理器模式：系统模式和用户模式。
- 指令的一些特权子集只能在系统模式下使用，如果在用户模式下执行将会导致陷阱。所有系统资源都只能通过这些指令进行控制。

2.4.4 虚拟机（缺少）的指令集体系结构支持

如果在设计 ISA 期间已经为 VM 作了规划，那就可以比较轻松地减少 VMM 必须执行的指令数、缩短模拟这些指令所需要的时间。如果一种体系结构允许 VM 直接在硬件上运行，则为其冠以可虚拟化的头衔，IBM 370 体系结构很骄傲地拥有了这一头衔。

遗憾的是，由于为桌面系统和基于 PC 的服务器应用程序考虑 VM 只是最近的事情，所以大多数指令集在设计时都没有考虑虚拟化问题。80x86 和大多数 RISC 体系结构都属于此类。

由于 VMM 必须确保客户系统只能与虚拟资源进行交互，所以传统的来宾操作系统是作为一种用户模式程序在 VMM 的上层运行的。因此，如果一个来宾操作系统试图通过特权指令访问或修改与硬件相关的信息（比如，读取或写入页表指针），它会向 VMM 发出陷阱中断。VMM 随后可以对相应的实际资源实进行适当修改。

因此，如果任何以用户模式执行的指令试图读写此类敏感信息陷阱，VMM 可以截获它，根据来宾操作系统的需要，向其提供敏感信息的一个虚拟版本。 109

如果缺乏此类支持，则必须采取其他措施。VMM 必须采取特殊的防范措施，找出所有存在问题的指令，并确保来宾操作系统执行它们时能够正常运行，这样自然就会增加 VMM 的复杂度，降低 VM 的运行性能。

2.5 节和 2.7 节给出了 80x86 体系结构一些问题指令的具体示例。

2.4.5 虚拟机对虚拟存储器和I/O的影响

由于每个 VM 中的每个来宾操作系统都管理其自己的页表集，所以虚拟存储器的虚拟化就成为另一项挑战。为了实现这一功能，VMM 区分了**实际存储器**（real memory）和**物理存储器**的概念（这两个词经常看作是同义词），使实际存储器成为虚拟存储器与物理存储器之间的独立、中间级存储器。（有人用**虚拟存储器**、**物理存储器**和**机器存储器**来命名这三个层级。）来宾操作系统通过它的页表将虚拟存储器映射到实际存储器，VMM 页表将来宾的实际存储器映射到物理存储器。虚拟存储器体系结构可以通过页表指定，IBM VM/370 和 80x86 属于此类，也可以通过 TLB 结构指定，许多 RISC 体系结构属于此类。

VMM 没有再为所有存储器访问进行一级间接迂回，而是维护了一个**影子页表**，直接从来宾虚拟地址空间映射到硬件的物理地址空间。通过检测来宾页表的所有修改，VMM 就能保证硬件在转换地址时使用的影子页表项与来宾操作系统环境的页表项一一对应，只是用正确的物理页代替了来宾表中的实际页。因此，只要来宾试图修改它的页表，或者试图访问页表指针，VMM 都必须加以捕获。这一功能通常通过以下方法来实现：对来宾页表提供写保护，并捕获来宾操作系统对页表指针的所有访问尝试。前面曾经指出，如果对页表指针的访问属于特权操作，就会很自然地实现捕获。

IBM 370 体系结构在 20 世纪 70 年代添加了一个由 VMM 管理的迂回层，解决了页表问题。来宾操作系统和以前一样保存自己的页表，所以就不再需要影子页表。AMD 在本公司与 80x86 相对应的 Pacifica 版本中采用了一种类似方案。

在许多 RISC 计算机中，为了实现 TLB 的虚拟化，VMM 管理实际 TLB，并拥有每个来宾 VM 的 TLB 内容副本。为实现这一功能，所有访问 TLB 的功能都必须被捕获。具有进程 ID 标记的 TLB 可以将来自不同 VM 与 VMM 的项目混合在一起，所以不需要在切换 VM 时刷新 TLB。与此同时，VMM 在后台支持 VM 的虚拟进程 ID 与实际进程 ID 之间的映射。

体系结构中最后一个要虚拟化的部分是 I/O。到目前为止，这是系统虚拟化中最困难的一部分，原因在于连接到计算机的 I/O 设备数目在增加，而且这些 I/O 设备的类型也变得更加多样化。另外一个难题是在多个 VM 之间共享实际设备，还有一个难题是需要支持不同的设备驱动程序，在同一 VM 系统上支持不同来宾操作系统时尤为困难。为了仍然能够实现 VM，可以为 110
每个 VM 提供每种 I/O 设备驱动程序的一个通用版本，然后交由 VMM 来处理实际 I/O。

将虚拟 I/O 设备映射到物理 I/O 设备的方法取决于设备类型。例如，VMM 通常会对物理进行分区，为来宾 VM 创建虚拟磁盘，而 VMM 会维护虚拟磁道与扇区到物理磁盘与扇区的映射。网络接口通常会在非常短的时间片内在 VM 之间共享，VMM 的任务就是跟踪虚拟网络地址的消息，以确保来宾 VM 只收到发给自己的消息。

2.4.6 VMM实例：Xen虚拟机

在VM发展的早期，发现了许多效率低下的问题。例如，来宾操作系统管理自己从虚拟页到实际页的映射，但这种映射会被VMM忽略，到物理页的实际映射是由VMM执行的。换句话说，仅仅为了“取悦”来宾操作系统就浪费了大量精力。为了减少这些低效问题，VMM开发人员认为有必要让来宾操作系统知道它自己是在VM上运行的。例如，来宾操作系统可以假定实际存储器与它的虚拟存储器一样大，所以来宾操作系统不需要进行存储器管理。

为了简化虚拟化而允许对来宾操作系统进行微小修改的做法称为**泛虚拟化**（paravirtualization），开源Xen VMM是很好的一个例子。Amazon Web服务数据中心使用的就是Xen VMM，它为来宾操作系统提供了一个与物理硬件相似的虚拟机抽象，但去掉了许多很麻烦的部分。例如，为了避免刷新TLB，Xen将它自己映射到每个VM的高位64 MB地址空间。它允许来宾操作系统分配页，只要核实没有违犯保护约束条件即可。为了保护来宾操作系统免受VM中用户程序的破坏，Xen利用了80x86中可以使用的4种保护级别。Xen VMM以最高级权限运行（0级），来宾操作系统运行在下一级别（1级），应用程序以最低权限级别运行（3级）。大多数针对80x86的操作系统以0级或3级权限运行所有程序。

为了使各部分能够协调工作，Xen对来宾操作系统进行了修改，不再使用体系结构中容易产生问题的部分。例如，Linux到Xen的端口大约修改了3000行代码，大约占80x86专用代码的1%。但是这些修改不会影响来宾操作系统的应用二进制接口。

为了简化VM的I/O难题，Xen为每个硬件I/O设备指定了具有特权的虚拟机。这些特殊VM称为**驱动程序域**。（Xen将它的VM称为“域”）。驱动程序域运行物理设备驱动程序，但在向适当的驱动程序域发送中断之前，这些中断仍然由VMM处理。常规VM称为**来宾域**，运行简单的虚拟设备驱动程序，它们必须通过一个信道与驱动程序域中的物理设备驱动程序进行通信，以访问物理I/O硬件。数据通过页面再映射在来宾域和驱动程序域之间传送。

111

2.5 交叉问题：存储器层次结构的设计

这一节介绍三个将在其他章节讨论的主题，它们是存储器层次结构的基础。

2.5.1 保护和指令集体系结构

保护是由体系结构和操作系统协力完成的，但是当虚拟存储器变得更为普及时，体系结构必须修改现有指令集体系结构中一些不便使用的细节。例如，为了在IBM 370中支持虚拟存储器，架构师必须对成功的IBM 360指令集体系结构进行修改，而这一指令集是6年前刚刚发布的。为了适应虚拟机，今天也要进行一些类似调整。

例如，80x86指令POPF从存储器栈的顶端载入标志寄存器，其中一个标志是中断使能（IE）标志。在最近为了支持虚拟化而作出修改之前，以用户模式运行POPF指令（而不是采用陷阱中断捕获它）将会改变除IE之外的所有标志位。而在系统模式下运行时，的确会修改IE标志。由于来宾操作系统是以用户模式在VM中运行的，而它希望修改IE标志位，所以这就成为一个问题。为了支持虚拟化而对80x86体系结构的扩展，消除了这一问题。

在历史上，IBM大型机硬件和VMM通过以下3个步骤来提高虚拟机的性能。

(1) 降低处理器虚拟化的成本。

(2) 降低由于虚拟化而造成的中断开销成本。

(3) 将中断传送给正确的 VM，而不调用 VMM，以降低中断成本。

IBM 仍然是虚拟机技术的黄金标准。例如，一台 IBM 大型机在 2000 年运行数千个 Linux VM，而 Xen 在 2004 年仅运行 25 个 VM[Clark 等人 2004]。Intel 和 AMD 芯片集的最近版本添加了一些指令，用以支持 VM 中的设备，在较低层级屏蔽来自每个 VM 的中断，将中断发送到适当的 VM。

2.5.2 缓存数据的一致性

数据可以同时出现在存储器和缓存中。只要处理器是唯一修改或读取数据的组件，并且缓存存在于处理器和存储器之间，那处理器看到旧副本或者说过期副本的危险性就很低。后面将会看到，使用多个处理器和 I/O 设备增大了副本不一致及读取错误副本的机会。

处理器出现缓存一致性问题的频率与 I/O 不同。对 I/O 来说，出现多个数据副本是非常罕见的情况（应当尽可能避免这种情况的发生），而一个在多处理器器上运行的程序会希望在几个缓存中拥有同一数据的多个副本。多处理器程序的性能取决于系统共享数据的性能。 112

I/O 缓存一致性问题可以表述如下：I/O 发生在计算机中的什么地方——是在 I/O 设备与缓存之间，还是在 I/O 设备与主存储器之间？如果输入将数据放在缓存中，而且输出从缓存中读取数据，那 I/O 和处理器会看到相同数据。这种方法的难点在于它干扰了处理器，可能会导致处理器因为等待 I/O 而停顿。输入还可能会用某些不会马上用到的新数据来取代缓存中的某些信息，从而对缓存造成干扰。

在带有缓存的计算机中，I/O 系统的目标应当是防止出现数据过期问题，同时尽可能减少干扰。因此，许多系统喜欢直接对主存储器进行 I/O 操作，把主存储器当作一个 I/O 缓冲区。如果使用直写缓存，存储器中将拥有最新的信息副本，在输出时不存在过期数据问题。（这一好处也是处理器使用直写方式的一个原因。）遗憾的是，现在通常只会在第一级数据缓存中使用直写方式，由使用写回方式的 L2 缓存为其提供后盾。

输入操作还需要另外做点功课。软件解决方案是保证输入缓冲区的所有数据块都没有在缓存中。可以将包含缓冲区的页标记为不可缓存，操作系统总是可以向这样一个页面中输入数据。或者，可以由操作系统在输入之前从缓存刷新缓冲区地址。硬件解决方案则是在输入时检查 I/O 地址，查看它们是否在缓存中。如果在缓存中找到了 I/O 地址的匹配项，则使缓存项失效，以避免过期数据。所有这些方法也都能用于带有写回缓存的输出操作。

在多核处理器的发展过程中，处理器缓存一致性是一个关键问题，我们将在第 5 章对其进行深入研究。

2.6 融会贯通：ARM Cortex-A8 和 Intel Core i7 中的存储器层次结构

本节揭示 ARM Cortex-A8（下文称为 Cortex-A8）和 Intel Core i7（下文称为 i7）的存储器层次结构，并根据一组单线程基准测试展示其组件的性能。由于 Cortex-A8 的存储器系统更简单一些，所以首先来研究它；之后将更详细地研究 i7，具体介绍一个存储器引用。本节假定读者熟悉一种使用虚索引缓存的两级缓存层次结构的组织方式。这种存储器系统的基础知识在附

录B中进行了详细解释，如果读者不熟悉这种系统的组织方式，强烈建议复习附录B中的Opteron示例。读者一旦理解了Opteron的组织方式，就能很轻松地看懂Cortex-A8系统的简要解释了，它们是类似的。

113

2.6.1 ARM Cortex-A8

Cortex-A8是一种支持ARMv7指令集体系结构的可配置核心。它是作为一种IP（知识产权）核心交付的。在嵌入式、PMD和相关市场上，IP核心是主要的技术交付形式；利用这些IP核心已经生成了数十亿个ARM和MIPS处理器。注意，IP核心不同于Intel i7中的核心和AMD Athlon多核心。一个IP核心（它本身可能是多核心）就是为与其他逻辑集成而设计的（因此，它是一个芯片的核心），这些其他逻辑包括专用处理器（比如视频编解码器）、I/O接口和存储器接口，从而制造出一种专门针对特定应用进行优化的处理器。例如，在Apple iPad和几家制造商（包括摩托罗拉和三星）生成的智能手机中都使用了Cortex-A8 IP核心。尽管这些处理器核心基本上是一样的，但最后得到的芯片有许多区别。

整体来说，IP核心分为两类。硬核心针对特定半导体厂家进行优化，是一些具有外部接口的黑盒（不过这些接口仍然在片上）。硬核心通常仅允许对核心外面的逻辑进行参数设定，比如L2缓存大小，不能对IP核心本身进行修改。软核心的交付形式通常采用一个标准的逻辑元件库。软核心可以针对不同的半导体厂家进行编译，也可以进行修改，不过由于当今IP核心的复杂度，很难对其进行大幅修改。一般来说，硬核心的性能较高、晶片面积较小，而软核心则允许针对不同厂家进行调整，其修改更容易一些。

当时钟频率高达1 GHz时，Cortex-A8每个时钟周期可以发出两条指令。它可以支持一种两级缓存层次结构，第一级是一个缓存对（I和D），分别为16 KB或32 KB，其组织形式为四路组相联缓存，并使用路预测和随机替代。其目的是使缓存的访问延迟只有一个周期，使Cortex-A8将从载入到使用的延迟时间保持在一个周期，简化指令提取，在分支缺失导致预取了错误指令时，降低提取正确指令的代价。第二级缓存是可选的，如果存在这一级缓存，则采用八路组相联，容量可达128 KB～1 MB；它的组织形式分为1～4组，允许同时从存储器进行多次传输。一个64位至128位的外部总线用来处理存储器请求。第一级缓存为虚索引、物理标记，第二级缓存为物理索引与标记；这两级缓存的块大小都是64字节。当D缓存为32 KB且页大小为4 KB时，每个物理页可以映射到两个不同的缓存地址；在发生缺失时，通过硬件检测可以避免出现混淆现象，附录B中的B.3节对此进行说明。

存储器管理由TLB对处理（I和D），每个TLB与32个项目完全相关，页面大小可变（4 KB、16 KB、64 KB、1 MB和16 MB）；TLB中的替换用一种轮询算法完成。TLB缺失在硬件中处理，它会遍历存储器中的一个页表结构。图2-13显示如何使用32位虚拟地址来索引TLB和缓存，假定主缓存为32 KB，次级缓存为512 KB，页面大小为16 KB。

114

Cortex-A8存储器层次结构的性能

Cortex-A8的存储器层次结构使用32 KB主缓存和1 MB八路组相联L2缓存来模拟，用整数Minnespec基准测试进行测试（见KleinOsowski和Lilja [2002]）。Minnespec是一个基准测试集，由SPEC2000基准测试组成，但其输入不一样，将运行时间缩短了几个数量级。尽管使用较小规模的输入并不会改变指令混合比例，但它的确会影响缓存行为。例如，根据mcf的测试结果（它是存储器操作最密集的SPEC2000整数基准测试），当缓存为32 KB时，Minnerspec

115

的缺失率只有完整 SPEC 版本缺失率的 65%。当缓存为 1 MB 时，这种差距为 6 倍。根据许多其他基准测试，这些比值都与 mcf 的测试结果类似，但绝对缺失率要小得多。由于这一原因，不能将 Minniespec 基准测试与 SPEC2000 基准测试进行对比。不过这些数据对于研究 L1 和 L2 缺失对整体 CPI 的相对影响是很有用的，下一章就将进行这些研究。

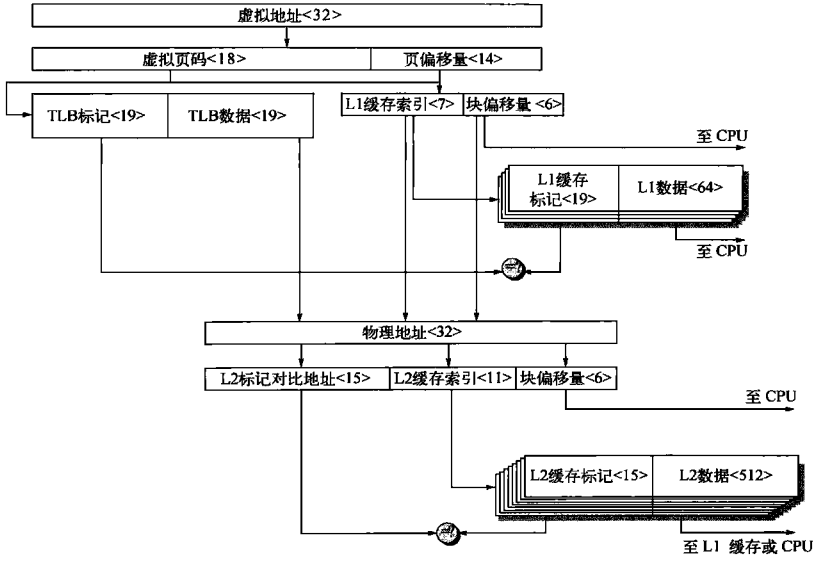


图 2-13 ARM Cortex-A8 数据缓存和数据 TLB 的虚拟地址、物理地址、索引、标记和数据块。由于指令与数据层次结构是对称的，所以这里只给出其中一个。TLB（指令或数据）与 32 个项目完全相关联。L1 缓存是四路组相联，块大小为 64 个字节，容量为 32 KB。L2 缓存是八路组相联，块大小为 64 个字节，容量为 1 MB。本图没有显示缓存和 TLB 的有效位和保护位，也没有使用可以指示 L1 缓存预测组的路预测位

即使仅对于 L1，这些基准测试（以及作为 Minniespec 基础的完全 SPEC2000 版本）的指令缓存缺失率也非常低：大多接近于零，都低于 1%。这种低缺失率的原因可能是因为 SPEC 程序在本质上是计算密集型的，而且四路组相联缓存消除了大多数冲突缺失。图 2-14 给出了数据缓存结果，这些结果中的 L1 和 L2 缺失率非常高。以 DDR SDRAM 为主存储器时，1 GHz Cortex-A8 的 L1 缺失代价为 11 个时钟周期，L2 缺失代价为 60 个时钟周期。通过这些缺失代价数据，图 2-15 给出了每次数据存取的平均代价。在下一章，我们将研究缓存缺失对整体 CPI 的影响。

116

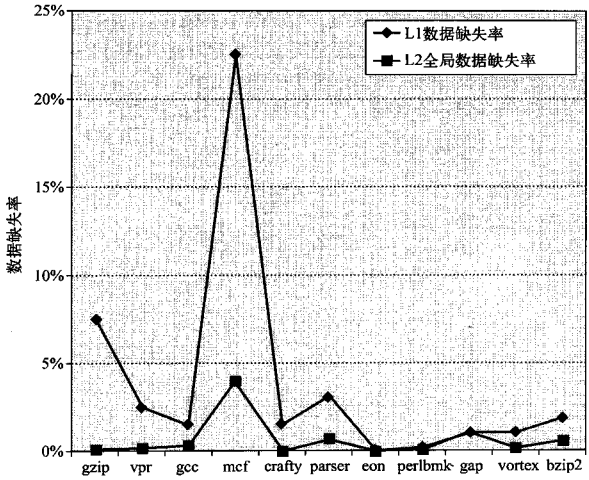


图 2-14 根据整数 Minnespec 基准测试的结果，对于采用 32 KB L1 的 ARM 的数据缺失率和 1 MB L2 的全局数据缺失率受到应用程序的影响很大。应用程序的存储器足印越大，L1 和 L2 的缺失率可能越高。注意，L2 缺失率为全局缺失率，它对所有引用进行计数，包括在 L1 中命中的情景。mcf 被称为缓存克星

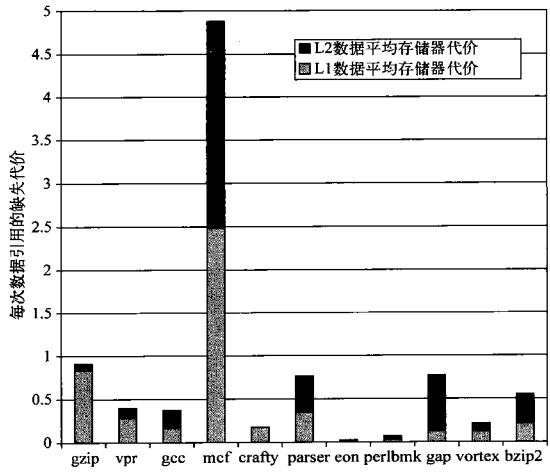


图 2-15 ARM 处理器在运行 Minniespec 时源于 L1 和 L2 的每次数据存储器引用的平均存储器访问代价。尽管 L1 的缺失率要高出许多，但 L2 的缺失代价（要高出 5 倍）意味着 L2 的缺失占据主导地位

2.6.2 Intel Core i7

i7 支持 x86-64 指令集体系结构，它是 80x86 体系结构的 64 位扩展。i7 是包含四个核心的乱序执行处理器。本章主要从单核心角度来研究存储器系统的设计与性能。多处理器设计的系统性能（包括 i7 多核）将在第 5 章详细研究。

i7 中的每个核心采用一种多次发送、动态调度、16 级流水线（将在第 3 章详细介绍），每个时钟周期可以执行多达 4 个 80x86 指令。i7 还使用一种名为“同时多线程”的技术（将在第 4 章介绍），每个处理器可以支持两个同时线程。2010 年，最快速 i7 的时钟频率为 3.3 GHz，指令的峰值执行速度为每秒 132 亿条指令，四核芯片超过每秒 500 万条指令。 117

i7 可以支持多达三个存储器通道，每个通道由独立的 DIMM 组构成，它们能够并行传输数据。i7 采用 DDR3-1066（DIMM PC8500），峰值存储器带宽超过 25 GB/s。

i7 使用 48 位虚拟地址和 36 位物理地址，物理存储器容量最大为 36 GB。存储器管理用一个两级 TLB 处理（见 B.4 节），表 2-4 中对此进行了总结。

表 2-4 i7 TLB 结构的特性，这种结构的第一级指令、数据 TLB 是分离的，第二级 TLB 合而为一

特　性	指令TLB	数据DLB	第二级TLB
大小	128	64	512
相联度	四路	四路	四路
替换	伪LRU	伪LRU	伪LRU
访问延迟	1个周期	1个周期	6个周期
缺失	7个周期	7个周期	访问页表需要数百个周期

* 第一级 TLB 支持标准 4 KB 页面大小，在 2MB～4MB 的大型页面上拥有有限数目的项目；在第二级 TLB 中仅支持 4 KB 页面。

表 2-5 总结了 i7 的三级缓存层次结构。第一级缓存为虚索引、物理标记（见 B.3 节），而 L2 和 L3 则采用物理索引。图 2-16 标有对存储器层次结构进行存取的步骤。首先，向指令缓存发送程序计数器。指令缓存索引为： 118

$$2^{索引}=\frac{缓存大小}{块大小\times组相联度}=\frac{32K}{64\times4}=128=2^7$$

也就是 7 位。指令地址的页帧（36=48－12 位）被发送给指令 TLB（第 1 步）。同时，来自虚拟地址的 7 位索引（再加上来自块偏移量的另外两位，用以选择适当的 16 字节，指令提取数）被发送给指令缓存（第 2 步）。注意，对于四路相联指令缓存，缓存地址需要 13 位：7 位用于索引缓存，再加上 64 字节块的 6 位块偏移量，但页大小为 4 KB=2^{12}，这意味着缓存索引的一位必须来自虚拟地址。使用 1 位虚拟地址意味着对应块实际上可能位于缓存中的两个不同位置，这是因为对应的物理地址在这一位置既可能为 0 也可能为 1。对指令来说，这样不会有什么问题，因为即使一条指令出现在缓存中的两个不同位置，这两个版本也必然是相同的。但如果允许对数据进行此类重复或别名设置，那在改变页映射时就必须对缓存进行检查，这一事件并非经常出现。注意，只要很简单地应用页着色（见 B.3 节）就能消除出现这种混淆的可能性。如果偶数地址的虚拟页被映射到偶数地址的物理页（奇数页也一样），那么因为虚拟页号和物理页中的低阶位都是相同的，所以就不可能发生这种混淆。

表 2-5 i7 中三级缓存层次结构的特性

特 性	L1	L2	L3
大小	32 KB I/32 KB D	256 KB	每个核心2 MB
相联度	四路 I/八路 D	八路	十六路
访问延迟	4个周期、流水化	10个周期	35个周期
替代方案	伪LRU	伪LRU	伪LRU，但采用一种有序选择算法

* 所有这三级缓存都采用写回方式，块大小都为 64 个字节。每个核心的 L1 和 L2 缓存分离，而 L3 缓存在一个芯片的所有核心之间共享，每个核心总共 2 MB。所有这三级缓存都是非阻塞的，允许存在多个未完成写入。为 L1 缓存使用一个合并写缓冲区，在要写入数据但 L1 中没有行时，用这个缓冲区保存数据。（即，一次 L1 写入缺失不会导致为其分配行。）L3 是 L1 和 L2 共有的；我们将在解释多核心缓存时更详细地研究这一属性。替换由伪 LRU 算法的一种变化方式完成。在 L3 中，被替换的块总是其访问位被关闭的具有最低编号的那一路。这种做法的随机性较弱，但易于计算。

为查找地址与有效页表项（PTE）之间的匹配项而访问指令 TLB（第 3 步和第 4 步）。除了转换地址之外，TLB 还会进行检查，以了解这一访问操作所需要的 PTE 是否会因为非法访问而产生异常。

指令 TLB 缺失首先进入 L2 TLB，它包含 512 个页大小为 4 KB 的 PTE，为四路组相联。从 L2 TLB 中载入 L1 TLB 需要两个时钟周期。如果 L2 TLB 缺失，则使用一种硬件算法遍历页表，并更新 TLB 项。在最糟糕情况下，这个页不在存储器中，操作系统从磁盘中获取该页。由于在页面错误期间可能执行数百万条指令，所以这时如果有另一进程正在等待，操作系统将转入该进程。否则，如果没有 TLB 异常，则继续访问指令缓存。

地址的索引字段被发送到指令缓存的所有 4 个组中（第 5 步）。指令缓存标记为 36–7 位（索引）–6 位（块偏移）=23 位。将 4 个标记及有效位与来自指令 TLB 中的物理页帧进行对比（第 6 步）。由于 i7 希望每个指令获取 16 个字节，所以使用 6 位块偏移量中的 2 位来选择适当的 16 个字节。因此，在向处理器发送 16 字节指令时使用了 7+2=9 位。L1 缓存实现了流水化，一次命中的延迟为 4 个时钟周期（第 7 步）。一次缺失将进入第二级缓存。

119
2
120

前面曾经提到，指令缓存采用虚寻址、物理标记。因为第二级缓存是物理寻址的，所以来自 TLB 的物理页地址包含页偏移量，构成一个能够访问 L2 缓存的地址。L2 索引为：

$$2^{索引}=\frac{缓存大小}{块大小\times组相联度}=\frac{256K}{64\times 8}=512=2^9$$

所以长 30 位的块地址（36 位物理地址–6 位块偏移）被分为一个 21 位的标记和一个 9 位的索引（第 8 步）。索引和标记再次被发送给统一 L2 缓存的所有 8 个组（第 9 步），同时对它们进行比较。如果有一个匹配且有效（第 10 步），则在开头的 10 周期延迟之后按顺序返回该块，返回速度为每个时钟周期 8 个字节。

如果 L2 缓存缺失，则访问 L3 缓存。对于一个四核 i7（它的 L3 为 8 MB），其索引大小为：

$$2^{索引}=\frac{缓存大小}{块大小\times组相联度}=\frac{8M}{64\times 16}=8192=2^{13}$$

这个长 13 位的地址（第 11 步）被发送给 L3 的所有 16 个组（第 12 步）。L3 标记的长度为 36–(13–6)=17 位，将其与来自 TLB 的物理地址进行对比（第 13 步）。如果发生命中，则在初始延迟之后以每个时钟周期 16 字节的速度返回这个块，并将它放在 L1 和 L3 中。如果 L3 缺失，则启动存储器访问。

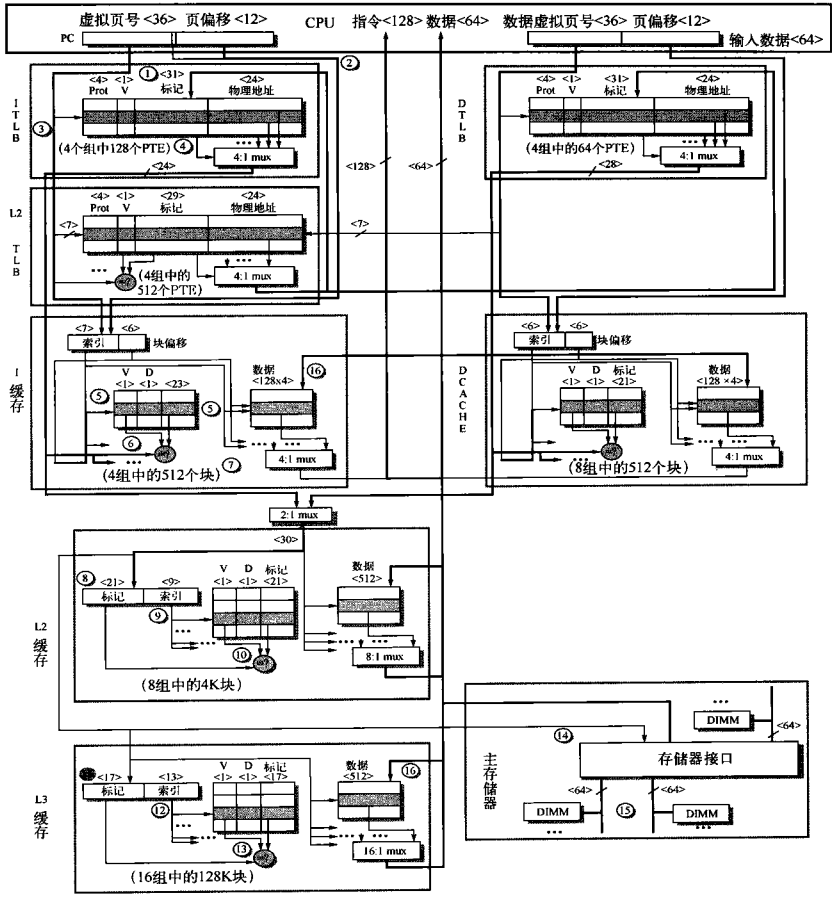


图 2-16 Intel i7 存储器层次结构及指令与数据访问步骤。我们只给出了数据读取步骤。写入步骤与其类似，因为它们也是以读取操作开始的（由于缓存采用写回方式）。由于 L1 缓存没有进行写入分配，所以只需要将数据放在写缓冲区中就可以处理缺失问题

如果在 L3 缓存中没有找到这个指令，片上存储器控制器必须从主存储器获取这个块。i7 有三个 64 位存储器通道，它们可以用作一个 192 位通道，这是因为只有一个存储器控制器，在两个通道上发送的是相同地址（第 14 步）。当两个通道具有相同的 DIMM 时，就可以进行宽通道传送。每个通道最多支持 4 个 DDR DIMM（第 15 步）。由于 L3 包含在内，所以在数据返回时，会将它们放在 L3 和 L1 中（第 16 步）。

在发生指令缺失时，由主存储器提供这一指令的总延迟包括用于判断发生了 L3 缺失的约 35 个处理器周期，再加上关键指令的 DRAM 延迟。对于一个单组 DDR1600 SDRAM 和 3.3 GHz CPU 来说，在接收到前 16 个字节之前的 DRAM 延迟为大约 35 ns 或 100 个时钟周期，所以总的缺失代价为 135 个时钟周期。存储器控制器以每个存储器时钟周期 16 个字节的速度填充 64 字节缓存块的剩余部分，这将另外花费 15 ns 或 45 个时钟周期。

由于第二级缓存是一个写回缓存，任何缺失都会导致将旧块写回存储器中。i7 有一个 10 项合并写缓冲区，当缓存的下一级未用于读取时写回脏缓存行。在发生任何缺失时都会查看此写缓冲区，看看该缓存地是否在这个缓冲区中；如果在，则从缓冲区中获取缺失内容。在 L1 和 L2 缓存之间使用了一个类似缓冲区。

如果初始指令是一个载入指令，则将数据地址发送给数据缓存和数据 TLB，与指令缓存访问非常类似，但有一个非常关键的区别。第一级数据缓存为八路组相联，也就是说索引是 6 位（指令缓存为 7 位），用于访问此缓存的地址与页偏移相同。因此，就不再需要担心数据缓存中的混淆问题。

121

假定这个指令是存储指令，而不是载入指令。在发出存储指令时，它会像载入指令一样进行数据缓存查询。发生缺失时，会将这个块放到写缓冲区中，这是因为 L1 缓存在发生写缺失时不会分配该块。在命中时，存储不会立即更新 L1（或 L2）缓存，而是要等到确认没有疑问时才会进行更新。在此期间，存储指令驻存在一个“载入-存储”队列中，这是处理器乱序控制机制的一个组成部分。

i7 还支持从层次结构的下一层级为 L1 和 L2 进行预取。在大多数情况下，预取行就是缓存中的下一个块。在仅为 L1 和 L2 预取时，就可以避免向存储器执行高成本的提取操作。

i7 存储器系统的性能

我们使用 SPECCPU2006 基准测试中的 19 个基准测试（12 个整型和 7 个浮点）来评估 i7 缓存结构的性能，这些基准测试在第 1 章进行了介绍。本节的数据由路易斯安那州大学的 Lu Peng 教授和 Ying Zhang 博士生收集。

我们首先来看 L1 缓存。这个 32 KB、4 路组相联指令缓存的指令缺失率非常低，最主要的原因是因为 i7 的指令预取十分有效。当然，由于 i7 不会为单个指令单元生成单独的请求，而是预取 16 字节的指令数据（通常介于 4 ~ 5 个指令之间），所以如何评估这一缺失率需要一点技巧。为了简单起见，如果我们就像处理单一指令引用那样研究指令缓存缺失率，那么 L1 指令缓存缺失率的变化范围为 0.1% ~ 1.8%，平均略高于 0.4%。这一比率与利用 SPECCPU2006 基准测试对指令缓存行为进行的其他研究一致，这些研究也显示指令缓存缺失率很低。

L1 数据缓存更有趣，对它的评估需要更强的技巧性，原因如下所述。

(1) 因为 L1 数据缓存不进行写入分派，所以写入操作可以命中，从来不会真正缺失，之所以这么说，是因为那些没有命中的写入操作会将其数据放在写缓冲区中，而不会记录为缺失。

(2) 因为推测有时可能会错误（请参阅第 3 章的详尽讨论），所以会有一些对 L1 数据缓存的引用，它们没有对应最终会完整执行的载入或存储操作。这样的缺失应当怎样处理呢？

(3) 最后，L1 数据缓存进行自动预取。发生缺失的预取是否应当计算在内？如果要计算在内，如何计算？

为了解决这些问题，在保持数据量合理的情况下，图 2-17 以两种方式显示了 L1 的数据缓存缺失：一种是相对于实际完成的载入指令数（通常称为“已完成”或“中途退出”），另一种是相对于从任意来源执行的 L1 数据缓存访问数。可以看到，在仅相对于已完成载入指令测试的

缺失率要高出 1.6 倍（平均 9.5%对 5.9%）。表 2-6 以表格形式显示了相同数据。 122

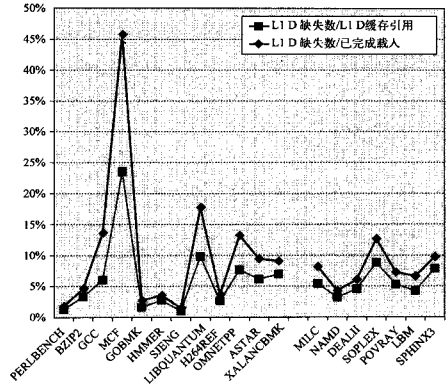


图 2-17 以两种方式给出了 17 个 SPECCPU2006 基准测试的 L1 数据缓存缺失率：相对于成功完成的实际载入指令和相对于对 L1 的所有引用，这些引用中还包括没有完成预取、未完成的推测载入、写入，它们会被记作引用，但没有生成缺失。与本节其余部分一样，这些数据也由来自路易斯安那州大学的 Lu Peng 教授和 Ying Zhang 博士生根据先前对 Intel Core Due 和其他处理器的研究进行收集。（见[Peng 等人 2008]）

表2-6 相对于已完成的全部载入操作数及全部引用数（包括预测及预取请求）给出的主要数据缓存缺失

基准测试	L1数据缺失/已完成载入	L1数据缺失/L1数据缓存引用
PERLBENCH	2%	1%
BZIP2	5%	3%
GCC	14%	6%
MCF	46%	24%
GOBMK	3%	2%
HMMER	4%	3%
SJENG	2%	1%
LIBQUANTUM	18%	10%
H264REF	4%	3%
OMNETPP	13%	8%
ASTAR	9%	6%
XALANCBMK	9%	7%
MILC	8%	5%
NAMD	4%	3%
DEALII	6%	5%
SOPLEX	13%	9%
POVRAY	7%	5%
LBM	7%	4%
SPHINX3	10%	8%

123

由于L1数据缓存缺失率达到5%~10%，有时还会更高一些，所以L2和L3缓存的重要性应当就非常明显了。图2-18给出了L2和L3缓存相对于L1引用的缺失率（表2-7以表格形式给出同一数据）。由于对存储器的一次缺失需要超过100个周期的成本，而且L2中的平均数据缺失率达4%，所以L3的重要性就不言而喻了。如果没有L3，并假定一半指令是载入或存储指令，L2缓存缺失会使CPI增加每条指令2个周期！作为对比，L3的平均数据缺失率为1%，仍然非常显眼，但只有L2缺失率的1/4，是L1缺失率的1/6。在下一章，我们将研究i7 CPI与缓存缺失之间的关系，以及其他流水线效果。

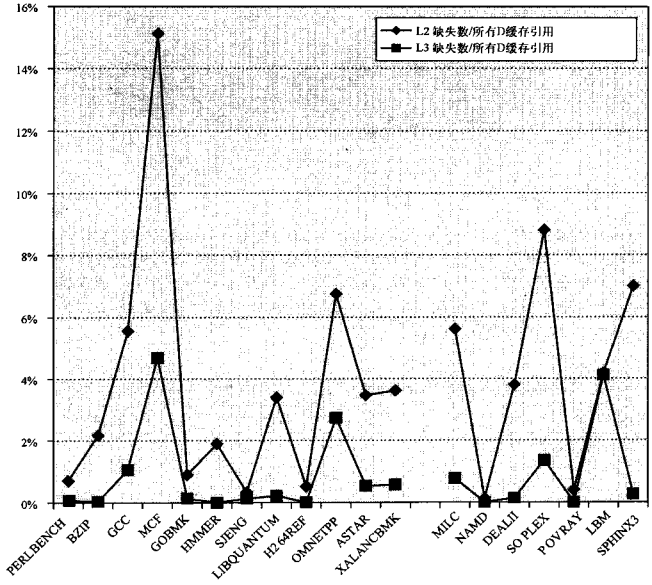


图2-18 相对于L1的所有引用给出17个SPECCPU2006基准测试的L2和L3数据缓存缺失，这些引用中还包括预取、未完成的预测载入和程序生成的载入和存储指令。和本节其余数据一样，这些数据也由来自路易斯安那州大学的Lu Peng教授和Ying Zhang博士生收集

124

表2-7 以表格形式给出相对于数据请求个数的L2和L3缺失率

	L2缺失数/所有数据缓存引用	L3缺失数/所有数据缓存引用
PERLBENCH	1%	0%
BZIP2	2%	0%
GCC	6%	1%
MCF	15%	5%
GOBMK	1%	0%
HMMER	2%	0%
SJENG	0%	0%

（续）

	L2缺失数/所有数据缓存引用	L3缺失数/所有数据缓存引用
LIBQUANTUM	3%	0%
H264REF	1%	0%
OMNETPP	7%	3%
ASTAR	3%	1%
XALANCBMK	4%	1%
MILC	6%	1%
NAMD	0%	0%
DEALII	4%	0%
SOPLEX	9%	1%
POVRAY	0%	0%
LBM	4%	4%
SPHINX3	7%	0%

2.7 谬论与易犯错误

作为计算机体系结构中最容易实现量化的部分，存储器层次结构看起来似乎不太容易产生谬论与错误。但事实并非如此，在编写这部分内容时，困扰我们的不是没有问题可讲，而是苦于篇幅所限！

谬论 由一个程序推测另一个程序的缓存性能。

图 2-19 显示在缓存大小变化时，由 SPEC2000 基准测试测得三个程序的指令缺失率和数据缺失率。根据程序的不同，对于一个容量为 4096 KB 的缓存，每千条指令的数据缺失率分别为 9、2 和 90，对于一个容量为 4 KB 的缓存，每千条指令的指令缺失分别为 55、19 和 0.0004。诸如数据库之类的商业程序甚至在大型第二级缓存中的缺失率也非常高，而对 SPEC 程序来说一般不是这种情况。图 2-18 提醒我们，测量结果的变化很大，从 mcf 和 sphnix3 可以看出，甚至关于整数和浮点密集型程序相对缺失率的预测也可能是错误的！ 125

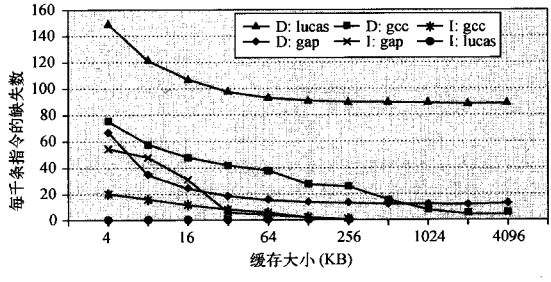


图 2-19 当容量大小从 4 KB 变化至 4096 KB 时，每 1000 条指令的指令缺失与数据缺失。gcc 的指令缺失比 lucas 大 30 000 ~ 40 000 倍，与之相反，lucas 的数据缺失比 gcc 大 2 至 60 倍。程序 gap、gcc 和 lucas 均来自 SPEC2000 基准测试套件

易犯错误 模拟足够多的指令以获取存储器层次结构的准确性能测量值。

这里实际上有三处陷阱。一是试图通过使用小型轨迹来预测大型缓存的性能。二是程序的局域特性在运行整个程序期间不是恒定的。三是程序的局域特性可能随输入的变化而变化。

图 2-20 显示为一个 SPEC2000 程序提供 5 个输入时，每千条指令的累积平均指令缺失。对于这些输入，前 19 亿条指令的平均存储器缺失率与执行其余指令时的平均缺失率有很大不同。

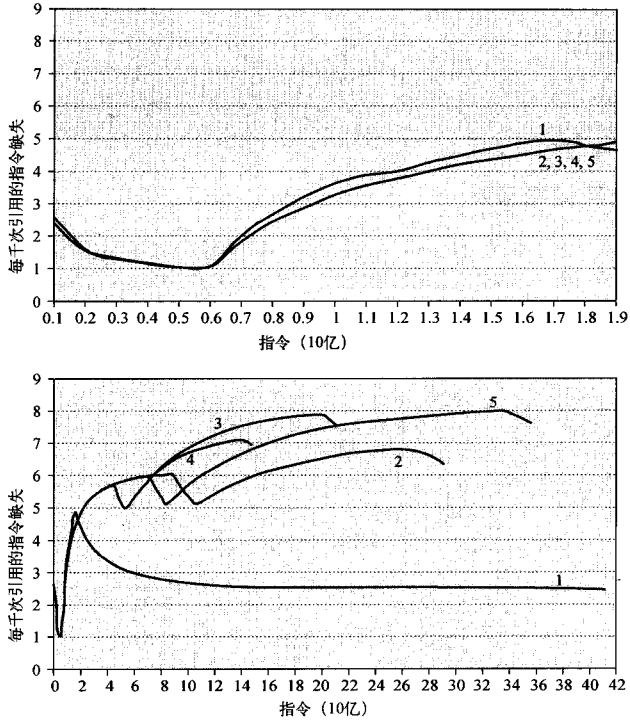


图 2-20 **为 SPEC2000 中的 perl 基准测试提供 5 种输入时，每 1000 次引用发生的指令缺失。**对于前 19 亿条指令，缺失的变化不大，5 种不同输入之间的区分也很小。运行到结束之后，将会看出在该程序的整个生存期内缺失率是如何变化的，以及它们与输入有什么样的关系。上图显示前 19 亿条指令的平均缺失率，对于全部 5 种输入，开始时每 1000 次引用的缺失大约为 2.5 个，结束时大约为 4.7 个。下图给出运行到结束后的平均缺失，根据输入的不同，它需要 160～410 亿条指令。在前 19 亿指令之后，根据输入的不同，每 1000 次引用的缺失从 2.4 变化到 7.9。这些仿真是对 Alpha 处理器实现的，它为指令和数据采用分离的 L1 缓存，每个缓存为两路 64 KB，采用 LRU 算法，共用一个 1 MB 的直接映射 L2 缓存

易犯错误 没有在基于缓存的系统中提供高存储器带宽。

缓存可以帮助缩短平均缓存存储器延迟，但可能不会提供应用程序进入主存储器所必需的高存储器带宽。架构师必须在这种缓存背后为这些应用程序设计一种高带宽存储器。我们将在第 4 章和第 5 章再次讨论这一易犯错误。

易犯错误 在一个指令集体系结构上实施虚拟机监视器，而这种体系结构的设计是不能虚拟化的。

20 世纪 70 年代和 80 年代的许多架构师都没有非常认真地确保所有读写硬件资源相关信息的指令都是特权指令。这一放任态度为所有这些体系结构上的 VMM 都带来了问题，其中就包括我们这里当作示例的 80x86。 126 ~ 127

表 2-8 描述了 18 种可能会为虚拟化带来问题的指令[Robin 和 Irvine 2000]。这些指令可以分为两大类：

- 以用户模式读取控制寄存器，表明来宾操作系统运行在虚拟机中（比如前面提到的 POPF）；
- 根据分段体系结构的要求提供检查保护，但假定操作系统是以最高权限级别运行。

虚拟存储器仍然富有挑战性。因为 80x86 TLB 和大多数 RISC 体系结构一样，不支持进程 ID 标记，所以 VMM 和来宾操作系统共享 TLB 的成本要更高一些；每次地址空间的变化通常都需要进行一次 TLB 刷新。

表 2-8 虚拟化时导致问题的 18 个 80x86 指令小结[Robin 和 Irvine 2000]

问题分类	存在问题的80x86指令
当运行于用户模式时，在没有采用陷阱中断的情况下访问敏感寄存器	存储全局描述符表寄存器（SGDT）
	存储局部描述符表寄存器（SLDT）
	存储中断描述符表寄存器（SIDT）
	存储机器状态字（SMSW）
	压入标志（PUSHF、PUSHFD）
	弹出标志（POPF、POPFD）
在以用户模式访问虚拟存储器机制时，指令未能进行80x86保护检查	从分段描述符载入存取权限（LAR）
	从分段描述符载入分段界限（LSL）
	验证分段描述符是否可读（VERR）
	验证分段描述符是否可写（VERW）
	弹至分段寄存器（POP CS、POP SS，…）
	压入分段寄存器（PUSH CS、PUSH SS，…）
	调用不同的权限级别（CALL）
	返回不同的权限级别（RET）
	跳至不同的权限级别（JMP）
	软件中断（INT）
	存储分段选择器寄存器（STR）
	移至/移出分段寄存器（MOVE）

* 上面一组中的前 5 个指令允许一个以用户模式运行的程序在没有产生陷阱中断的情况下读取控制寄存器，比如描述符表寄存器。弹出标志指令修改拥有敏感信息的控制寄存器，但在用户模式上会失败，不会给出提示消息。80x86 分段体系结构的保护检查在下面一组中，在读取一个控制寄存器时，这些指令都会隐式检查权限级别，这一检查将作为执行过程的组成部分。这一检查过程假定必须以最高权限级别运行，而来宾 VM 并非如此。只有 MOVE 到分段寄存器才会尝试修改控制状态，保护检查功能也会阻止它。 128

对80x86来说，I/O的虚拟化也是一项挑战，部分原因在于它既支持存储器映射I/O，也拥有独立的I/O指令，但更重要的原因在于PC拥有数目庞大、种类繁多的设备和设备驱动程序，需要VMM进行处理。第三方供应商提供他们自己的驱动程序，它们也许不能正确地虚拟化。传统VM实现提出的一种解决方案是将实际设备驱动程序直接载入到VMM中。

为了简化80x86上的VMM实现，ADM和Intel都对体系结构进行了扩展。Intel的VT-x为运行VM提供了一种新的运行模式，VM状态的体系结构定义、快速切换WM的指令，还有一大组参数，用于在必须调用VMM时选择环境。VT-x总共为80x86添加了11种新指令。AMD的安全虚拟机（SVM）提供了类似功能。

在打开支持VT-x的模式之后（通过VMXON指令），VT-x为来宾操作系统提供了4种权限级别，它们的优先级要低于原来的4个级别（解决了前面提到的POPF指令的一些问题）。在虚拟机控制状态（VMCS）中，VT-x捕获一个虚拟机的全部状态，然后提供了用于保存和恢复VMCS的原子指令。除了关键状态之外，VMCS包含一些配置信息，用于判断什么时候调用VMM，以及是什么导致了VMM的调用。为了减少必须调用VMM的次数，这种模式添加了一些敏感寄存器的影子版本，并添加了一些掩码，用于判断一个敏感寄存器的关键位是否会在捕获之前发生改变。为了减少虚拟化虚拟存储器的成本，AMD的SVM另外添加了一个间接层级，称为**嵌套页表**。有了它就不再需要影子页表了。

2.8 结语：展望

在过去三十年里，已经多次有人预测计算机的性能提升将会停止。这些预测都错了，其原因在于它们未经证实的假设依据都被后来的事实推翻。例如，由于未能预测到从离散组件到集成电路的转变，从而错误地预测到光速将会限制计算机的速度，预测值要比现在的实现际低几个数量级。我们对存储器墙（memory wall）的预测也可能是错误的，但它提醒我们必须开始换种考虑方式了。

——Wm. A. Wulf和Sally A. McKee
Hitting the Memory Wall: Implications of the Obvious
弗吉尼亚州大学计算机系（1994年12月）
这篇论文中提出了**存储器墙**一词。

129 使用存储器层次结构的可能性可回溯到20世纪40年代后期到50年代早期通用数字计算机的最早时期。20世纪60年代早期，在研究计算机中引入了虚拟存储器，70年代引入了IBM大型机。缓存大约在同一时间出现。随着时间的推移，这些基本概念已经进行了扩展和延伸，以帮助缩小主存储器与处理器在访问时间方面的差距，但基本概念没有根本性变化。

有一个趋势可能会导致存储器层次结构设计的巨大改变，那就是DRAM密度和访问时间的持续变缓。在过去十年里，人们已经观察到这两种趋势。尽管DRAM带宽有所提高，但访问时间的缩短速度要慢得多，部分原因是为了限制功耗，一直在降低电平。为了提高带宽，人们正在研究一种思路：在每一组存储器上重叠多个访问操作。这样就为增加分组数目提供了替代方案，同时还可以提供较高的带宽。传统的DRAM设计在每个单元中使用电容器，通常将它们放在一个深沟道中，这种设计在制造方面也有一些困难，从而减缓了DRAM密度的增长速度。在

本书英文版付印时，有一家制造商发布了一种不需要电容器的新 DRAM，这也可能为 DRAM 技术的持续发展提供了契机。

闪存不受 DRAM 发展的限制，由于在功率和密度方面的潜在优势，很可能会扮演更重要的角色。当然，在 PMD 中，闪存已经代替了磁盘驱动器，提供了许多桌面计算机不能提供的优势，比如“即时启动”。闪存相对于 DRAM 的潜在优势（不需要逐位晶体管来控制写操作）也正是它自己的“阿喀琉斯脚跟”（唯一致命弱点）。闪存必须采用批擦除重写周期（其速度相当缓慢）。因此，一些 PMD，比如 Apple iPad 将较小的 SDRAM 主存储器与闪存结合使用，既充当文件系统，也充当页存储系统，用于处理虚拟存储器。

此外，几种全新的存储器技术也正在研究之中，其中包括 MRAM，它采用磁技术存储数据，还有相变 RAM（称为 PCRAM、PCME 和 PRAM），它采用一种能够在非晶态和晶态之间进行变化的玻璃。这两种存储器都是永久性的，其密度有可能高于 DRAM。这些并不是什么新思路，磁阻存储器技术和相变存储器的出现已经有几十年了。任何一种技术都可以取代目前的闪存，而取代 DRAM 的难度要大得多。尽管 DRAM 的发展速度已经减缓，但由于无电容器单元的可能性及其他潜在改进，至少在接下来的十年中不会轻易抛弃 DRAM。

几年来，人们对存储器墙的到来进行了各种预测（见前面摘引的论文），声称这可能会导致处理器性能的降低。但是，多级缓存的扩展、更高级的填充与预取方案、编译器及程序员对局域特性重要程度的更深入理解、使用并行机制来克服延迟的影响，所有这些都阻挡了存储器墙的到来。在基于缓存的系统中仍然存在存储器延迟，会导致缺失的出现，而在有多处缺失尚未解决时采用乱序流水线，就可以利用指令级并行机制来消除这些延迟的影响。而多线程及更多线程级并行机制的引入则更进一步，提供了更多的并行机制，因此也就提供了更多克服延迟影响的机会。指令级、线程级并行机制的应用，是对抗现代多级缓存系统中各种存储器延迟的主要工具。 130

一个时不时就会冒出来的想法是使用由程序员控制的高速暂存存储器（scratchpad）或其他高速存储器，我们后面将会看到 GPU 中使用了这些存储器。这些想法之所以没有成为主流，有几个原因：第一，它们引入了具有不同行为特性的地址空间，打破了存储器模型。第二，高速暂存存储器的存储器转换不同于基于编译器或基于程序员的缓存优化方式（比如预取），它们必须完全处理从主存储器地址空间到高速暂存存储器地址空间的重新映射。这就增加了此类转换的难度，限制了它的适用范围。在 GPU 中（见第 4 章）大量使用了本地高速暂存存储器，而管理这些存储器的重担现在落在了程序员的肩上。

尽管人们在预测计算技术的未来时一定要非常小心，但历史已经证明，缓存技术是一种强有力的、高度可扩展的思想，它有可能让我们继续建造更快速的计算机，确保存储器层次结构能够提供系统正常运转所需要的指令和数据。

2.9 历史回顾与参考文献

附录 L.3 节研究了缓存、虚拟存储器和虚拟机的历史。IBM 在所有这三种技术的历史上都扮演着重要角色。这一节还包含了供扩展阅读的参考文献。

案例研究与练习（Norman P. Jouppi、Naveen Muralimanohar 和 Sheng Li 设计）

案例研究 1：通过高级技术优化缓存性能

本案例研究说明的概念

- 无阻塞缓存
- 缓存的编译器优化
- 软件和硬件预取
- 缓存性能对更多复杂处理器计算性能的影响

131

矩阵的转置就是交换它的行与列；说明如下：

$$\begin{bmatrix} A11 & A12 & A13 & A14 \\ A21 & A22 & A23 & A24 \\ A31 & A32 & A33 & A34 \\ A41 & A42 & A43 & A44 \end{bmatrix} \Rightarrow \begin{bmatrix} A11 & A21 & A31 & A41 \\ A12 & A22 & A32 & A42 \\ A13 & A23 & A33 & A43 \\ A14 & A24 & A34 & A44 \end{bmatrix}$$

下面是一段显示转置运算的简单 C 循环：

```
for (i = 0; i < 3; i++) {
 for (j = 0; j < 3; j++) {
 output[j][i] = input[i][j];
 }
}
```

假定输入与输出矩阵都以行主序存储（行主序意味着行索引的变化最快速）。假定我们正在一个处理器上执行 256×256 双精度转置，该处理器带有一个 16 KB 完全相联（不用担心缓存冲突）、最近应用最少（LRU）替换的 L1 数据缓存，块大小为 64 字节。假定 L1 缓存缺失或预取需要 16 个周期，而且总会在 L2 缓存中命中，L2 缓存每两个处理器周期可以处理一个请求。假定数据存在于 L1 缓存时，上述内层循环的每次迭代需要 4 个周期。假定该缓存对写缺失采用写入分派（write-allocate）、写时取（fetch-on-write）策略。尽管不太现实，但我们假定写回脏缓存块需要 0 个周期。

2.1 [10/15/15/12/20] <2.2>对于上面给出的简单实现，这种执行顺序对输入矩阵来说不是理想顺序，但如果进行循环交换优化会为输出矩阵生成一种非理想状态。由于循环交换不足以提高其性能，因此必然会被阻塞。

a. [10] <2.2>为发挥分块执行的优势，缓存的最小容量应当为多少？

b. [15] <2.2>与以上最小容量的缓存相比，分块和非分块版本的相对缺失数如何？

c. [15] <2.2>编写代码，执行 $B \times B$ 块的转置，其中 B 为块大小参数。

132

d. [12] <2.2>为了保持缓存性能的一致性，使其不受两个数组在存储器中的位置影响，L1 缓存需要的最小相联度为多少？

e. [20] <2.2>在一台计算机上尝试阻塞式与非阻塞式 256×256 矩阵转置。根据你对计算机存储系统的了解，结果与预期的吻合程度如何？如果可能，请解释它们之间的差异性。

2.2 [10] <2.2>假定你正在为上述非阻塞矩阵转置代码设计硬件预取器。最简单的硬件预取器仅在发生缺失之后预取连续缓存块。更复杂的“非单位步幅”硬件预取器可以分析一个缺失引用流，检测并预取超过非单位步幅的块。与之相对，软件预取可以像判断单位步幅一样，轻松地判断非单位步幅。假定预取内容直接写入缓存，所以不存在“污染”（改写在预取数据之前必须使用的数据）。对于给定非单位步幅预取器，为了获得最佳性能，在内层循环的稳定状态中，在给定时刻必须有多少个等待完成的预取操作？

2.3 [15/20] <2.2>在采用软件预取时，非常重要的是要保证两点，一方面要及时进行预取以供使用，另一方面要在最大程度上减少待处理预取的数目，使之不会超出微体系结构的能力范围，并将缓存污染降至最低。由于不同处理器的性能和局限性也都不同，从而使情况变得复杂。

a. [15] <2.2>采用软件预取，生成矩阵转置的一个阻塞版本。

b. [20] <2.2>估计和对比阻塞与非阻塞转置代码在有无软件预取情况下的性能。

案例研究 2：融会贯通：高度并行存储器系统

本案例研究说明的概念

- 交叉问题：存储器层次结构的设计

代码清单 2-1 中的程序可用于评估一个存储器系统的行为。其关键在于拥有精确的定时，使程序能够穿过存储器，调用层次结构的不同层级。代码清单 2-1 给出了用 C 语言编写的代码。第一部分是一个过程，它使用一种标准实用程序获取用户 CPU 时间的准确测量值；为了能够在某些系统上运行，可能需要对这一过程进行修改。第二部分是一个嵌套循环，用于以不同的步幅和缓存大小来读写存储器。为了获得准确的缓存定时，这一代码要重复许多次。第三部分仅确定嵌套循环开销的时间，以便能够从总测量时间中扣取该时间，以了解访问时间的长短。其结果以.csv 文件格式输出，便于导入电子表格中。根据我们要回答的问题以及被测系统上存储器的大小，可能需要修改 CACHE_MAX。以单用户模式运行此程序，或者至少在没有其他活动应用程序的情况下运行此程序，可以获得更为一致的结果。代码清单 2-1 中的代码源自加州大学伯克利分校 Andrea Dusseau 编写的一段程序，在 Saavedra-Barrera [1992]有其详尽说明。原来的程序在现代计算机上运行时会有许多问题，为了克服这些问题并使其能够在 Microsoft Visual C++环境下运行，已经对原程序进行了一些修改。该程序可以从 www.hpl.hp.com/research/cacti/aca_ch2_cs2.c 下载。

代码清单2-1 用于评估存储器系统的C程序

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time in seconds */
    __time64_t ltime;
    _time64( &ltime );
    return (double) ltime;
}
int label(int i) {/* generate text labels */
    if (i<1e3) printf("%1dB,",i);
    else if (i<1e6) printf("%1dK,",i/1024);
    else if (i<1e9) printf("%1dM,",i/1048576);
    else printf("%1dG,",i/1073741824);
    return 0;
}
int _tmain(int argc, _TCHAR* argv[]) {
int register nextstep, i, index, stride;
int csize;
double steps, tsteps;
double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

/* Initialize output */
printf(",");
for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
    label(stride*sizeof(int));
printf("\n");

/* Main loop for each configuration */
for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
    label(csize*sizeof(int)); /* print cache size this loop */
    for (stride=1; stride <= csize/2; stride=stride*2) {

        /* Lay out path of memory references in array */
        for (index=0; index < csize; index=index+stride)
            x[index] = index + stride; /* pointer to next */
        x[index-stride] = 0; /* loop back to beginning */

        /* Wait for timer to roll over */
        lastsec = get_seconds();
         sec0 = get_seconds(); while (sec0 == lastsec);

        /* Walk through path in array for twenty seconds */
```

```
        /* This gives 5% accuracy with second resolution */
        steps = 0.0; /* number of steps taken */
        nextstep = 0; /* start at beginning of path */
        sec0 = get_seconds(); /* start timer */
           { /* repeat until collect 20 seconds */
              (i=stride;i!=0;i=i-1) { /* keep samples same */
                 nextstep = 0;
                 do nextstep = x[nextstep]; /* dependency */
                 while (nextstep != 0);
           }
           steps = steps + 1.0; /* count loop iterations */
           sec1 = get_seconds(); /* end timer */
        } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
        sec = sec1 - sec0;

        /* Repeat empty loop to loop subtract overhead */
        tsteps = 0.0; /* used to match no. while iterations */
        sec0 = get_seconds(); /* start timer */
           { /* repeat until same no. iterations as above */
              (i=stride;i!=0;i=i-1) { /* keep samples same */
                 index = 0;
                 do index = index + stride;
                 while (index < csize);
           }
           tsteps = tsteps + 1.0;
           sec1 = get_seconds(); /* - overhead */
        } while (tsteps<steps); /* until = no. iterations */
        sec = sec - (sec1 - sec0);
        loadtime = (sec*1e9)/(steps*csize);
        /* write out results in .csv format for Excel */
        printf("%4.1f,", (loadtime<0.1) ? 0.1 : loadtime);
      }; /* end of inner for loop */
      printf("\n");
    }; /* end of outer for loop */
    return 0;
}
```

133 ~ 134

上述程序假定程序地址与物理地址一致，在少数使用虚拟寻址缓存的计算机上，比如 Alpha 21264，这一假设是成立的。一般情况下，在计算机重新启动后的很短时间内，虚拟地址会与物理地址保持一致，所以为了在结果中获得平滑曲线，可能需要重启计算机。为了回答以下问题，假定存储器层次结构中所有组件的大小都是 2 的幂。假定在第二级缓存中（如果存在二级缓存的话），页的大小要远大于块的大小，第二级缓存块的大小大于或等于第一级缓存中的块大小。在图 2-21 中绘出了该程序的一个输出示例；图例中列出了所使用数组的大小。

2.4 [12/12/12/10/12] <2.6>使用图 2-21 中的程序结果示例回答以下问题。

a. [12] <2.6>第二级缓存的总大小和块大小为多少？

135

b. [12] <2.6>第二级缓存的缺失代价为多少？

c. [12] <2.6>第二级缓存的相联度为多少？

d. [10] <2.6>主存储器的大小是多少？

e. [12] <2.6>如果页大小为 4 KB，而分页时间为多少？

2.5 [12/15/15/20] <2.6>根据需要修改代码清单 2-1 中的代码，以测量以下特性。以 y 轴为经过时间，x 轴为存储器步幅，绘制试验结果曲线。两个轴均采用对象刻度，为每种缓存大小绘制一条曲线。

a. [12] <2.6>系统页大小为多少？

b. [15] <2.6>转换旁视缓冲区（TLB）中有多少项？

c. [15] <2.6>TLB 的缺失代价为多少？

d. [20] <2.6>TLB 的相联度为多少？

2.6 [20/20] <2.6>在多处理器存储系统中，单个处理器也许不能填满存储器层次结构的较低层级，但多个一同工作的处理器也许能够填满。修改代码清单 2-1 中的代码，同时运行多个副本。你能否作出以下判断。

a. [20] <2.6>你的计算机系统中实际有多少个处理器？多少系统处理器只是额外的多线程上下文？

b. [20] <2.6>你的系统有多少存储器控制器？

2.7 [20] <2.6>你能否想出一种方法，使用程序来测试指令缓存的某些特性？提示：编译器可以由一段代码生成大量不太容易看透的指令。尝试使用你所用指令集体系结构（ISA）中一些长度已知的简单算法指令。

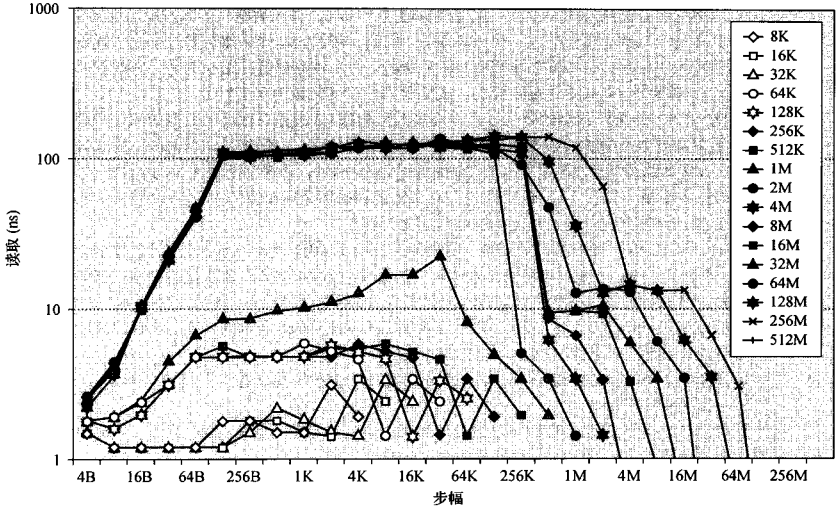


图 2-21 代码清单 2-1 中程序的输出举例

练习

2.8 [12/12/15] <2.2>以下问题利用 CACTI 研究小而简单的缓存产生的影响，假定采用 65 nm（0.065 μm）工艺。（CACTI 可以从网上获取：http://quid.hpl.hp.com:9081/cacti/）。

a. [12] <2.2>对比块大小为 64 字节的 64 KB 缓存与单组存储器的访问时间。与直接映射的组织方式相比，两路与四路组相联缓存的相对访问时间为多少？

b. [12] <2.2>对比块大小为 64 字节的四路组相联缓存与单组存储器的访问时间。与 16 KB 缓存相比，32 KB 与 64 KB 缓存的相对访问时间是多少？

c. [15] <2.2>对于 64 KB 缓存和特定工作负载，每条指令的缺失数据如下：直接映射为 0.00664、两路组相联为 0.00366、四路组相联为 0.00987、八路组相联缓存为 0.000266，求具有最短平均存储器访问时间的缓存相联度（介于 1 至 8 之间）。从整体来看，每条指令有 0.3 次数据引用。假定缓存缺失在所有模型中均耗费 10 ns。为了以周期为单位计算命中时间，假定使用 CACTI 输出周期时间，它对应于在流水线中没有气泡时，缓存的最高工作频率。 136

2.9 [12/15/15/10] <2.2>你正在研究路预测 L1 缓存可能带来的好处。假定 64 KB 四路组联单组 L1 数据缓存是某个系统中的周期时间限制器。作为一种替代缓存组织方式，你正在考虑一种路预测缓存，其模型为一个预测准确度为 80%的 64 KB 直接映射缓存。除非另行指出，否则假定，一次预测错误的路访问在缓存中命中需要多消耗一个周期。假定缺失率和缺失代价如 2.8 题(c)部分所示。

a. [12] <2.2>与路预测缓存相比，当前缓存的平均存储器访问时间是多少（用周期表示）？

b. [15] <2.2>如果所有其他组件都是最短的路预测缓存周期时间工作（包括主存储器在内），使用路预测缓存会产生什么样的性能影响？

c. [15] <2.2>路预测缓存通常仅用于为指令队列或缓冲区提供内容的指令缓存。设想一下你希望

为数据缓存尝试使用路预测。假定预测精度为80%，在正确路预测时发出后续操作（例如，其他指令的数据缓存访问、相关操作）。因此，路预测错误必然需要进行流水线刷新和重新执行陷阱中断，这些操作将需要 15 个周期。在采用数据缓存路预测时，每条载入指令的平均存储器访问时间变化是正面的，还是负面的？变化量为多少？

d. [10] <2.2>作为路预测的替代方式，许多大型相联 L2 缓存对标记与数据访问进行序列化，从而只需要激活必需的数据集数组。这样可以节省功率，但增加了访问时间。为 0.065μm 工艺 1 MB 四路组相联缓存使用 CACTI 的详尽 Web 接口，该缓存的块大小为 64 字节、144 位读出，1 个组、只有 1 个读/写端口，30 位标记和采用全局布线的 ITRS-HP 技术。实现标记与数据访问序列化与并行访问的访问时间比为多少？

2.10 [10/12] <2.2>有一种新的微处理器，需要研究分组 L1 数据缓存与流水化 L1 数据缓存的相对性能。假定有一种 64 KB 两路组相联缓存，其块大小为 64 字节。流水化缓存由三级流水构成，类似于 Alpha 21264 数据缓存的容量。分组实现方式由两个 32 KB 两路组相联组成。使用 CACTI，并假定采用 65 nm（0.065 μm）工艺，回答以下问题。Web 版本的周期时间输出表明缓存可以在什么样的频率下正常工作，不会在流水线中产生气泡。

a. [10] <2.2>该缓存的周期时间与其访问时间相比为多少？该缓存将占用多少个流水级（精确到小数点后两位）？

137

b. [12] <2.2>对比流水线设计与分组设计的每次访问的面积及总动态读取能耗。说明哪种设计占用的面积较少，哪种需要的功率较多，解释其原因。

2.11 [12/15] <2.2>考虑在 L2 缓存缺失时使用关键字优先和提前重启动。假定 L2 缓存的容量为 1 MB、块大小为 64 字节、填充路径宽 16 字节。假定能够以每 4 个处理器周期 16 个字节的速度写入 L2，从存储器控制器接收前 16 个字节块的时间为 120 个周期，每从主存储器接收另外 16 个字节的块需要 16 个周期，也可以直接将数据传送给 L2 缓存的读取端口。忽略向 L2 缓存发送缺失请求及向 L1 缓存传送被请求数据的周期数。

a. [12] <2.2>在使用、不使用关键字优先和提前重启动时，为 L2 缓存缺失提供服务分别需要多少个周期？

b. [15] <2.2>你是否认为关键字优先和提前重启动对于 L1 缓存或 L2 缓存更重要一些，哪些因素影响着它们的相对重要性？

2.12 [12/12] <2.2>在直写 L1 缓存与写回 L2 缓存之间设计一个写缓冲区。L2 缓存写数据总线的宽度为 16 B，可以每 4 个处理器周期向一个独立缓存地址执行一次写操作。

a. [12] <2.2>每个写缓冲区项目应当为多少字节？

b. [15] <2.2>如果所有其他指令可以与存储指令并行发射，块存在于 L2 缓存中，在通过执行 64 位存储指令将存储器置零时，使用一个合并写缓冲区来代替非合并缓冲区，在稳定状态下可以得到什么样的加速比？

c. [15] <2.2>对于采用阻塞缓存与非阻塞缓存的系统，可能出现的 L1 缺失对于所需写缓冲区项目的个数有什么样的影响？

2.13 [10/10/10] <2.3>考虑一个桌面系统，它的处理器连接到一个采用纠错码（ECC）的 2 GB DRAM。假定只有一个宽度为 72 位的存储器通道，其中 64 位用于数据，8 位用于 ECC。

a. [10] <2.3>如果使用 1 GB DRAM 芯片，DIMM 上有多少个 DRAM 芯片，如果仅有一个 DRAM 连接到每个 DIMM 数据管脚，每个 DRAM 必须拥有多少数据 I/O？

b. [10] <2.3>为了支持 32 B L2 缓存块，突发（burst）长度需要为多少？

c. [10] <2.3>计算为了从某个活动页读取内容，DDR2-667 和 DDR2-533 DIMM 的峰值带宽为多少？不计 ECC 开销。

2.14 [10/10] <2.3>图 2-22 给出一个 DDR2 SDRAM 时序图示例。tRCD 是激活一组中的一行所需要的时间，列地址选通（CAS）延迟（CL）是从一行中读出一列所需要的周期数。假定此 RAM 是

在具有 ECC 的标准 DDR2 DIMM 上，拥有 72 个数据行。另外假定突发长度为 8，它从 DIMM 读出 8 位，也就是总共 64 B。假定 tRCD = CAS (或 CL) * clock_frequency、clock_frequency = transfers_per_second/2。在发生缓存缺失时，通过第 1 级、第 2 级并返回的片上延迟时间为 20 ns，不包括 DRAM 访问时间。 138

a. [10] <2.3>对于 DDR2-667 1 GB CL = 5 DIMM，从出现激活命令一直到从 DRAM 请求的最后一个数据位由有效变为无效，一共需要多少时间？假定对于每个请求，我们会自动预取同一页中的另一个相邻缓存线。

b. [10] <2.3>在使用 DDR2-667 DIMM 时，如果一次读取操作需要激活分组，那与读取已打开页面相比，其相对延迟是多少？（包括在处理器内部处理该缺失所需要的时间。）

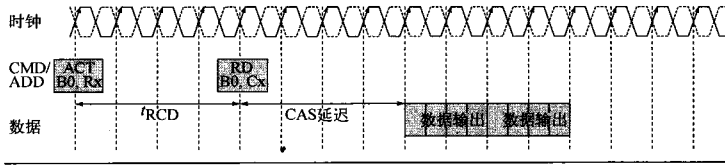


图 2-22 DDR2 SDRAM 时序图

2.15 [15] <2.3>假定 DDR2-667 2 GB DIMM（CL = 5）的价格为 130 美元，DDR2-533 2 GB DIMM（CL = 4）的价格为 100 美元。假定在一个系统中使用两个 DIMM，系统的其余组件需要 800 美元。对于一个工作负载，第 1000 条指令出现 3.33 次 L2 缺失，并假定所有 DRAM 读取操作中有 80% 需要激活，考虑系统使用 DDR2-667 和 DDR2-533 DIMM 时的性能。假定在某一时刻只有一个 L2 缺失等待处理，循序（in-order）核心的 CPI 为 1.5，不包括 L2 缓存缺失存储器访问时间，则整个系统的性价比如何？

2.16 [12] <2.3>你正在准备一台服务器，它采用八核 3 GHz CMP，在执行某一工作负载时的总 CPI 为 2.0（假定 L2 缓存缺失填充没有延迟）。L2 缓存行的大小为 32 字节。假定该系统采用 DDR2-667 DIMM。如果有时需要的带宽为平均带宽的 2 倍，那么应当提供多少个独立存储器通道才能使系统不受存储器带宽的限制？该工作负载平均第 1000 条指令导致 6.67 次 L2 缺失。

2.17 [12/12] <2.3>DRAM 功率中有很大一部分是因为页面激活消耗的（超过三分之一）（请参阅 http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf和www.micron.com/systemcalc）。假定你正在构建一个拥有 2 GB 存储器的系统，或者使用 8 组 2 GB x 8 DDR2 DRAM，或者使用 8 组 1 GB x8 DRAM，这两者的速度相同。使用的页大小都是 1 KB，最后一级缓存行大小为 64 字节。假定 DRAM 在未激活之前处于预充电待机状态，消耗的功率可以忽略。假定从待机状态到活动状态的过渡时间不是很长。 139

a. [12] <2.3>预计哪种类型的 DRAM 提供的系统性能较高？解释原因。

b. [12] <2.3>由 1 GB x8 DDR2 DRAM 组成的 2 GB DIMM 与容量相同但由 1 GB x4 DDR2 DRAM 组成的 DIMM 相比，在功率方面有何差异？

2.18 [20/15/12] <2.3>为了从一个典型 DRAM 访问数据，必须首先激活适当的行。假定这一操作会将大小为 8 KB 的整个页面发送到行缓冲区，然后从行缓冲区中选择一个特定列。如果对 DRAM 的后续访问目标也是同一页，就可以略过激活步骤；如果不是，就必须关闭当前页，对位行进行预充电，以准备下一次激活。另一种常用 DRAM 策略是在访问结束之后立即主动关闭一个页，并对位行进行预充电。假定对 DRAM 的每次读取或写入都是采用 64 字节的大小，发送 512 位的 DDR 总线延迟（图 2-21 中的数据输出）为 Tddr。

a. [20] <2.3>假定采用 DDR2-667，如果它需要 5 个周期进行预充电、5 个周期进行激活、4 个周期读取列，为了获得最短访问时间，如何根据行缓冲区命中率（r）选择策略？假定对 DRAM 的每次访问之间都有足够的时间，用以完成新的随机访问。

b. [15] <2.3>如果在对 DRAM 的所有访问中有 10%是一个接一个地发生，或者没有任何时间间隙地连续发生，应当如何改变自己的决定？

c. [12] <2.3>使用上面计算的行缓冲区命中率，计算在采用两种策略时，每次访问的平均 DRAM 能耗差别。假定预充电需要 2 nJ，激活需要 4 nJ，从行缓冲区进行读写需要 100 pJ/位。

2.19 [15] <2.3>每当计算机空闲时，既可以将其置于待机状态（DRAM 仍然处于活动状态），也可以让它休眠。为了使其进入休眠状态，假定必须仅将 DRAM 的内容复制到永久性介质中，比如闪存中。如果将大小为 64 字节的缓存行读写至闪存需要 2.56 μJ，读写至 DRAM 需要 0.5 nJ，如果 8 GB DRAM 的空闲功耗为 1.6 W，那么一个系统空闲多长时间后才能从休眠中获益？假定主存储器的容量为 8 GB。

2.20 [10/10/10/10/10] <2.4>虚拟机（VM）具有向计算机系统添加大量有益功能的潜力，比如降低总拥有成本（TCO）或提高可用性。能否使用 VM 来提供以下功能？如果可以，如何提升？

a. [10] <2.4>使用开发计算机测试应用程序在生产环境中的性能？

140

b. [10] <2.4>在发生灾难或故障时快速部署应用程序？

c. [10] <2.4>在 I/O 操作密集的应用程序中获得更高性能？

d. [10] <2.4>实现不同应用程序之间的故障隔离，提高服务的可用性？

e. [10] <2.4>在系统上执行软件维护，同时不对正在运行的应用程序造成严重干扰？

2.21 [10/10/12/12] <2.4>许多事件都可能导致虚拟机的性能下降，比如执行特权指令、TLB 缺失、陷阱和 I/O。这些事件通常是在系统模式中处理的。因此，为了评估应用程序在 VM 中运行时的减缓程度，可以计算该应用程序在系统模式下的运行时间占用户模式下执行时间的百分比。例如，某个程序有 10%的执行过程是在系统模式下完成，当它在 VM 中运行时，速度可能会降低 60%。表 2-9 是在 Itanium 系统上使用 Xen 虚拟机，在三种不同情况下运行 LMbench 时，各种系统调用的早期性能，这三种不同情况分别为：无虚拟执行、纯虚拟化和部分虚拟化，时间的测量单位为微秒（感谢新南威尔士大学的 Matthew Chapman）。

a. [10] <2.4>预计哪种类型的程序在 VM 下运行时速度减缓较少？

b. [10] <2.4>如果速度减缓与系统时间成线性关系，给定上述减缓数据，如果一个程序有 20%的执行是在系统时间内完成，那它会减缓多少？

c. [12] <2.4>在纯虚拟化和半虚拟化条件下，上表中系统调用速度减缓的中间值为多少？

d. [12] <2.4>上表中哪些函数的速度减缓最严重？可能是因为什么原因？

表2-9 在无虚拟化、纯虚拟化和半虚拟化条件下各种系统调用的早期性能

基准测试	无虚拟化	纯虚拟化	半虚拟化
Null call	0.04	0.96	0.50
Null I/O	0.27	6.32	2.91
Stat	1.10	10.69	4.14
Open/close	1.99	20.43	7.71
Install sighandler	0.33	7.34	2.89
Handle signal	1.69	19.26	2.36
Fork	56.00	513.00	164.00
Exec	316.00	2084.00	578.00
Fork + exec sh	1451.00	7790.00	2360.00

141

2.22 [12] <2.4>Popek 和 Goldberg 在给出虚拟机定义时指出，性能是唯一能够将虚拟机与真实计算机区分开来的指标。在这个问题中，我们将利用这一定义来查明是在一个处理器上以无虚拟化方式运行，还是运行在虚拟机上。Intel VT-x 技术为使用虚拟机实际提供了第二组权限级别。假定采用 VT-x 技术，如果一个虚拟机要运行在另一个虚拟机的顶层，它必须做些什么？

2.23 [20/25] <2.4>随着 x86 体系结构开始支持虚拟化，虚拟机得以快速发展，成为主流。比较 Intel Vt-x 和 AMD 的 AMD-V 虚拟化技术（AMD-V 的相关信息可以参见 http://sites.amd.com/us/business/it-solutions/virtualization/Pages/resources.aspx。）

a. [20] <2.4>对于内存印记较大、存储器操作密集的应用程序，哪一种技术的性能较高？

b. [25] <2.4>有关 AMD 对虚拟化 I/O 的 IOMMU 支持信息，可以在 http://developer.amd. com/documentation/articles/pages/892006101.aspx 找到。虚拟化技术和输入/输出存储器管理单元（IOMMU）为提高虚拟化 I/O 性能做了哪些工作？

2.24 [30] <2.2、2.3>由于在循序超标量处理器和具有预测功能的超长指令字（VLIW）处理器上都可以有效地开发指令级并行，所以构建乱序（OOO）超标量处理器的一个重要原因就是能够容忍由于缓存缺失导致的不可预测延迟。因此，我们可以把硬件支持的 OOO 发射看作存储器系统的一部分！看一下图 2-23 中 Alpha 21264 的平面布置图，找出整数和浮点发射队列及映射器与缓存相比的相对面积。队列调度要发射的指令，映射器重命名寄存器标识符。因此，为了支持 OOO 发射，有必要增加这些内容。21264 的芯片上只有 L1 数据与指令缓存，它们都是 64 KB 两路组相联。使用一种 OOO 超标量模拟器（比如 SimpleScalar，www.cs.wisc.edu/~mscalar/simplescalar.html）执行存储器操作密集的基准测试，如果在循序超标量处理器中，发射队列和映射器的区域被用于附加的 L1 数据缓存区域，而不是 21264 模型中的 OOO 发射。确保计算机的其他方面尽可能相似，使比较公平合理。忽略任何由于大型缓存造成访问时间或周期时间的增长，以及大型数据缓存对芯片平面布置图的影响。（注意，这种对比不是完全公平的，因为编译器没有为循序处理器调度这些代码。）

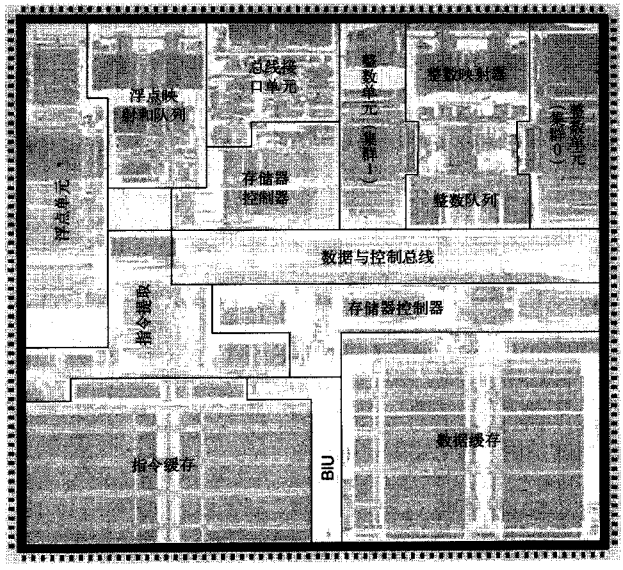
2.25 [20/20/20] <2.6>Intel 性能分析器 VTune 可用于对缓存行为进行许多种测量。VTune 在 Windows 和 Linux 平台上的免费评估版本都可以从 http://software.intel.com/enus/articles/intel-vtune-amplifier-xe/ 下载。它修改了案例研究 2 中使用的程序（aca_ch2_cs2.c），使其能够在 Microsoft Visual C++上与现成的 VTune 一起工作。这个程序可以从 www.hpl.hp.com/research/cacti/aca_ch2_cs2_vtune.c 下载。为了在性能分析期间扣除初始化与循环开销，已经插入了一些特殊的 VTune 函数。在这个程序的 README 部分给出了详细的 VTune 安装指令。这个程序为每种配置循环 20 秒。在下面的试验中，可以求出数据规模对缓存和整体处理器性能的影响。在 Intel 处理器上的 VTune 中运行该程序，输入数据大小分别为 8 KB、128 KB、4 MB 和 32 MB，步幅保持为 64 个字节（也就是 Intel i7 处理器上的一个缓存行）。收集有关整体性能、L1 数据缓存、L2 和 L3 缓存性能的统计数字。

142

a. [20] <2.6>对于每一种数据集大小和处理器模型与速度，列出 L1 数据缓存、L2 和 L3 缓存中每 1000 条指令的缺失数。根据结果，你可以对处理器上的 L1 数据缓存、L2 和 L3 缓存大小得到什么结论？请解释你的结论。

b. [20] <2.6>对于每种数据集大小和处理器模型与速度，列出每个时钟周期执行的指令数（IPC）。根据结果，可以对处理器上的 L1、L2 和 L3 缺失代价得出什么结论？请解释你的结论。

c. [20] <2.6>在 Intel OOO 处理器上的 VTune 中运行该程序，输入数据集大小为 8 KB 和 128 KB。对于两种配置，列出每 1000 条指令的 L1 数据缓存和 L2 缓存缺失数，并给出 CPI。对于高性能 OOO 处理器上内存延迟隐藏技术的有效性，可以说些什么？提示：需要求出处理器的 L1 数据缓存缺失代价。对于最新的 Intel i7 处理器，大约为 11 个时钟周期。



143

图 2-23 Alpha 21264 的平面布置图[Kessler 1999]

3 指令级并行及其开发

“谁是第一？”
“美国。”
“谁是第二？”
“没有第二。”

这是数年一度的“美洲杯”帆船赛后两位观众之间的对话。John Cocke 受到它的启发将 IBM 的研究处理器命名为“美国”。这个处理器是 RS/6000 系列的前身，是第一个超标量微处理器。

146 ~ 147

3.1 指令级并行：概念与挑战

大约1985年之后的所有处理器都使用流水线来重叠指令的执行过程，以提高性能。由于指令可以并行执行，所以指令之间可能实现的这种重叠称为**指令级并行**（ILP）。在本章和附录H中，我们将研究一系列通过提高指令并行度来扩展基本流水线概念的技术。

与附录C中有关流水线的基础材料相比，本章内容要深入得多。如果读者还不是特别熟悉附录C中的思想，应当在复习这个附录之后再开始探索本章内容。

本章首先研究数据和控制冒险带来的局限性，然后再转而讨论如何提高编译器和处理器对并行的开发能力。在这几节中介绍了大量概念，本章和下一章都以这些概念为基础。尽管在理解本章中一些比较基础的材料时并不需要前两节的全部思想，但这一基础材料对于本章后面各节非常重要。

ILP大体有两种不同开发方法：(1)依靠硬件来帮助动态发现和开发并行，(2)依靠软件技术在编译时静态发现并行。使用基于硬件动态方法的处理器，包括Intel Core系列，在桌面和服务器市场上占据主导地位。在个人移动设备市场，提高能耗效率通常是一个关键目标，所以设计人员开发较低级别的指令级并行。因此，2011年，PMD市场的大多数处理器都采用静态方法，我们将会看到ARM Cortex-A8中即是如此；不过，未来的处理器（比如新的ARM Cortex-A9）将采用动态方法。从20世纪80年代开始到最近的Intel Itanium系列，人们已经无数次尝试基于编译器的积极方法。尽管付出了无数努力，但除了非常有限的科学应用领域之外，这些方法都没有获得成功。

过去几年里，在主要基于某种方法进行设计时，也会采用大量原本为另一方法开发的技术。本章介绍这些基本概念和这两种方法，还讨论ILP方法的一些局限性，正是因为这些局限性而直接导致了向多核的转移。深入了解这些局限性对于平衡ILP与线程级并行的应用仍然非常重要。

在这一节，我们将讨论程序和处理器的一些特性，它们限制了可在指令间开发的并行数量，还将介绍程序结构与硬件结构之间的一些重要映射，要知道某一程序特性是否会对性能造成限制以及在什么条件下会造成限制，上述映射是非常关键的。

一个流水化处理器的CPI（每条指令占用的周期数）值等于基本CPI与因为各种停顿而耗费的全部周期之和：

148

流水线CPU=理想流水线CPI+结构化停顿+数据冒险停顿+控制停顿

理想流水线CPI可以用来度量能够实现的最佳性能。通过缩短上式右侧各项，可以降低总流水线CPI，也就是提高IPC（每个时钟周期执行的指令数）。利用上面的公式，我们可以说明一项技术能够缩小总CPI的哪一部分，以此来刻画各种技术的特征。表3-1显示了将在本章和附录H中研究的技术，还有一些主题将在附录C中的介绍性材料中介绍。在本章将会看到，我们介绍用来降低理想流水线CPI的技术会证明应对冒险的重要性。

表3-1 一并列出了附录C、第3章和附录H中研究的主要技术，同时给出了这些技术会分别影响CPI公式的哪一部分

技　术	降低CPI的哪一组成部分	章　节
转发和旁路	潜在的数据冒险停顿	C.2
延迟分支和简单分支调度	控制冒险停顿	C.2

（续）

技　术	降低CPI的哪一组成部分	章　节
基本编译器流水线调度	数据冒险停顿	C.2、3.2
基本动态调度（记分板）	由真相关引起的数据冒险停顿	C.7
循环展开	控制冒险停顿	3.2
分支预测	控制停顿	3.3
采用重命名的动态调度	由数据冒险、输出相关和反相关引起的停顿	3.4
硬件推测	数据冒险和控制冒险停顿	3.6
动态存储器消除二义	涉及存储器的数据冒险停顿	3.6
每个周期发出多条指令	理想CPI	3.7、3.8
编译器相关性分析、软件流水线、踪迹调试	理想CPI、数据冒险停顿	H.2、H.3
硬件支持编译器推测	理想CPI、数据冒险停顿、分支冒险停顿	H.4、H.5

3.1.1 什么是指令级并行

这一章的所有技术都是开发指令间的并行。**基本块**（一段顺序执行代码，除入口外没有其他转入分支，除出口外没有其他转出分支）中可以利用的并行数非常有限。对于典型的 MIPS 程序，平均动态分支频率通常介于 15%到 25%之间，也就是说在一对分支之间会执行 3～6 条指令。由于这些指令可能相互依赖，所以在基本块中可以开发的重叠数量可能要少于基本块的平均大小。为了真正地提高性能，我们必须跨越多个基本块开发 ILP。

提高 ILP 的最简单、最常见方法是在循环的各次迭代之间开发并行。这种并行经常被称作**循环级并行**。下面是一个简单的循环示例，它对两个分别有 1000 个元素的数组求和，完全可以并行实现：

```
for (i=0; i<=999; i=i+1)
      x[i] = x[i] + y[i];
```

这个循环的每一次迭代都可以与任意其他迭代重叠，当然，在每次循环迭代中进行重叠的机会不大，甚至没有这种机会。

我们将研究大量将这种循环级并行转换为指令级并行的技术。这些技术的工作方式基本上都是采用编译器静态展开循环（下一节介绍这种方法）或者利用硬件动态展开循环（3.5 节、3.6 节介绍这种方法）。

149

开发循环级并行的一种重要替代方法是使用向量处理器和图形处理器（GPU）中的 SIMD，这两种处理器都将在第 4 章介绍。SIMD 指令在开发数据级并行时，并行处理少量到中等数量的数据项（通常为 2～8 项）。而向量指令在开发数据级并行时，则通过使用并行执行单元和深流水线，并行处理许多数据项。例如，上述代码序列的简单形式在每次迭代中需要 7 条指令（2 次载入、1 次求和、1 次存储、2 次地址更新和 1 次分支），总共 7000 条指令，而在每条指令可以处理 4 个数据项的某种 SIMD 体系结构中，只需要四分之一的指令就能完成任务。在一些向量处理器中，这个序列可能只需要 4 条指令：2 条指令用于从存储器中载入向量 x 和 y，1 条指令用于对两个向量求和，还有 1 条指令用于将结果向量存回存储器。当然，这些指令可以是流水化的，其延迟相对较长，但可以对这些延迟进行重叠。

3.1.2 数据相关与冒险

要确定一个程序中可以存在多少并行以及如何开发并行，判断指令之间的相互依赖性是至

关重要的。具体来说，为了开发指令级并行，我们必须判断哪些指令可以并行执行。如果两条指令是**并行**的，只要流水线有足够资源（因而也就不存在任何结构性冒险），就可以在一个任意深度的流水线中同时执行它们，不会导致任何停顿。如果两条指令是相关的，它们就不是并行的，尽管它们通常可以部分重叠，但必须按顺序执行。这两种情景的关键在于判断一条指令是否依赖于另一指令。

1. 数据相关

共有3种不同类型的相关：**数据相关**（也称为真数据相关）、**名称相关**和**控制相关**。如果以下任一条件成立，则说指令 j 数据相关于指令 i：

- 指令 i 生成的结果可能会被指令 j 用到；
- 指令 j 数据相关于指令 k，指令 k 数据相关于指令 i。

150

第二个条件就是说：如果两条指令之间存在第一类型的相关链，那么这两条指令也是相关的。这种相关链可以很长，贯穿整个程序。注意，单条指令内部的相关性（比如 ADDD R1，R1，R1）不认为是相关。

例如，考虑以下 MIPS 代码序列，它用寄存器 F2 中的一个标量来递增存储器中的一个值向量（从 0(R1)开始，最后一个元素是 8(R2)）。（为简单起见，本章中的所有示例都忽略延迟分支的影响。）

```
Loop:   L.D    F0,0(R1)    ;F0=数组元素
        ADD.D  F4,F0,F2    ;加上 F2 中的标量
        S.D    F4,0(R1)    ;存储结果
        DADDUI R1,R1,#-8   ;使指针递减 8 个字节
        BNE    R1,R2,LOOP  ;R1!=R2 时跳转
```

这一代码序列中的数据相关涉及两个浮点数据：

```
Loop:   L.D    F0,0(R1)    ;F0=数组元素
        ADD.D  F4,F0,F2    ;加上 F2 中的标量
        S.D    F4,0(R1)    ;存储结果
```

和整型数据：

```
DADDIU  R1,R1,#-8   ;使指针递减 8 个字节
                    ;(每个 DW)
BNE     R1,R2,Loop  ;R1!=R2 时跳转
```

在以上两个相关序列中，如箭头所示，每条指令都依赖于上一条指令。这段代码及以下示例中给出的箭头表示为了正确执行必须保持的顺序。位于箭头起始处的指令必须位于箭头所指指令之前。

如果两条指令是数据相关的，那它们必须按顺序执行，不能同时执行或不能完全重叠执行。这种相关意味着两条指令之间可能存在由一个或多个数据冒险构成的链。（关于数据冒险的简单描述，请参阅附录C，其中用几页的内容给出了数据冒险的详细定义。）同时执行这些指令会导致一个具有流水线互锁（而且流水线深度大于指令间距离，以周期为单位）的处理器检测冒险和停顿，从而降低或消除重叠。在依靠编译器调度、没有互锁的处理器中，编译器在调度相关指令时不能使它们完全重叠，这样会使程序无法正常执行。指令序列中存在数据相关，反映出据以生成该指令序列的源代码中存在数据相关。原数据相关的影响一定会保留下来。

151

相关是**程序**的一种属性。某种给定相关是否会导致检测到实际冒险，这一冒险又是否会实际导致停顿，这都属于**流水线结构**的性质。这一区别对于理解如何开发指令级并行至关重要。

数据相关传递了三点信息：(1)冒险的可能性；(2)计算结果必须遵循的顺序；(3)可开发并行

度的上限。这些限制将在3.10节和附录H中进行更详尽的研究。

由于数据相关可能会限制我们能够开发的指令级并行数目，所以本章的一个重点就是如何克服这些局限性。可以采用两种不同方法来克服相关性：(1)保护相关性但避免冒险；(2)通过转换代码来消除相关性。对代码进行调度是在不修改相关性的情况下避免冒险的主要方法，这种调度既可以由编译器完成，也可以由硬件完成。

数据值既可以通过寄存器也可以通过存储器位置在指令之间传送。当数据传送在寄存器中发生时，由于指令中的寄存器名字是固定的，所以相关性的检测很简单，当然，如果存在分支干扰以及为了保持正确性而迫使编译器或硬件变得保守，那可能会变得复杂一些。

当数据在存储器位置之间流动时，由于两个看起来不同的地址可能引用同一位置，所以其相关性更难以检测，比如100(R4)和20(R6)可能是同一个存储器地址。此外，载入指令或存储指令的实际地址可能会在每次执行时发生变化（所以20(R4)和20(R4)可能是不一样的），这使相关性的检测进一步复杂化。

本章研究采用硬件来检测那些涉及存储器位置的数据相关，但我们将会看到，这些技术也有局限性。用于检测这些相关的编译器技术是揭示循环级别并行的关键所在。

2. 名称相关

第二种相关称为**名称相关**。当两条指令使用相同的寄存器或存储器位置（称为**名称**），但与该名称相关的指令之间并没有数据流动时，就会发生名称相关。在指令 i 和指令 j（按照程序顺序，指令 i 位于指令 j 之前）之间存在两种类型的名称相关。

(1) 当指令 j 对指令 i 读取的寄存器或存储器位置执行写操作时就会在指令 i 和指令 j 之间发生**反相关**。为了确保 i 能读取到正确取值，必须保持原来的顺序。在前面的例子中，S.D和DADDIU之间存在关于寄存器R1的反相关。

(2) 当指令 i 和指令 j 对同一个寄存器或存储器位置执行写操作时，发生**输出相关**。为了确保最后写入的值与指令 j 相对应，必须保持指令之间的排序。 152

由于没有在指令之间传递值，所以反相关和输出相关都是名称相关，与真数据相关相对。因为名称相关不是真正的相关，因此，如果改变这些指令中使用的名称（寄存器号或存储器位置），使这些指令不再冲突，那名称相关中涉及的指令就可以同时执行，或者重新排序。

对于寄存器操作数，这一重命名操作更容易实现，这种操作称作**寄存器重命名**。寄存器重命名既可以由编译器静态完成，也可以由硬件动态完成。在介绍因分支导致的相关性之前，先让我们来看看相关与流水线数据冒险之间的关系。

3. 数据冒险

只要指令间存在名称相关或数据相关，而且它们非常接近，足以使执行期间的重叠改变对相关操作数的访问顺序，那就会存在冒险。由于存在相关，必须保持**程序顺序**，也就是由原来的源程序决定的指令执行顺序。软、硬件技术的目的都是尽量开发并行方式，**仅在程序顺序会影响程序输出时才保持程序顺序**。检测和避免冒险可以确保不会打乱必要的程序顺序。

根据指令中读、写访问的顺序，可以将数据冒险分为三类，附录C中简要介绍了数据冒险。根据惯例，一般按照流水线必须保持的程序顺序为这些冒险命名。考虑两条指令 i 和 j，其中 i 根据程序顺序排在 j 的前面。可能出现的数据冒险为：

- RAW（写后读）——j 试图在 i 写入一个源位置之前读取它，所以 j 会错误地获得旧值。这一冒险是最常见的类型，与真数据相关相对应。为了确保 j 会收到来自 i 的值，必须保持程序顺序。

- WAW（写后写）——j试图在i写一个操作数之前写该操作数。这些写操作最终将以错误顺序执行，最后留在目标位置的是由i写入的值，而不是由j写入的值。这种冒险与输出相关相对应。只有允许在多个流水级进行写操作的流水线中，或者在前一指令停顿时允许后一指令继续执行的流水线中，才会存在WAW冒险。
- WAR（读后写）——j尝试在i读取一个目标位置之前写入该位置，所以i会错误地获取新值。这一冒险源于反相关（或名称相关）。在大多数静态发射流水线中（即使是较深的流水线或者浮点流水线），由于所有读取操作都较早进行（在附录C流水线的ID中），所有写操作都要晚一些（附录C流水线中的WB中），所以不会发生WAR冒险。如果有一些指令在指令流水线中提前写出结果，而其他指令在流水线的后期读取一个源位置，或者在对指令进行重新排序时，就会发生WAR冒险，在本章后面将对此进行讨论。

153

注意，RAR（读后读）情况不是冒险。

3.1.3 控制相关

最后一种相关是**控制相关**。控制相关决定了指令i相对于分支指令的顺序，使指令i按正确程序顺序执行，而且只会在应当执行时执行。除了程序中第一基本块中的指令之外，其他所有指令都与某组分支存在控制相关，一般来说，为了保持程序顺序，必须保留这些控制相关。控制相关的最简单示例之一就是分支中if语句的then部分中的语句。例如，在以下代码段中：

```
if p1 {
      S1;
};
if p2 {
      S2;
}
```

S1与p1控制相关，S2与p2控制相关，但与p1没有控制相关。

一般来说，控制相关会施加下述两条约束条件。

(1) 如果一条指令与一个分支控制相关，那就不能把这个指令移到这个分支之前，使它的执行不再受控于这个分支。例如，不能把if语句then部分中的一条指令拿出来，移到这个if语句的前面。

(2) 如果一条指令与一个分支没有控制相关，那就不能把这个指令移到这个分支之后，使其执行受控于这个分支。例如，不能将if之前的一个语句移到它的then部分。

当处理器保持严格的程序顺序时，确保了控制相关也不会破坏。但是，在不影响程序正确性的情况下，我们可能希望执行一些还不应当执行的指令，从而会违犯控制相关。因此，控制相关并不是一个必须保持的关键特性。有两个特性对程序正确性是至关重要的，即**异常行为**和**数据流**，通常保持数据相关与控制相关也就保护了这两种特性。

保护异步行为意味着对指令执行顺序的任何改变都不能改变程序中激发异常的方式。通常会放松这一约束条件，要求改变指令的执行顺序时不得导致程序中生成任何新异常。下面的简单示例说明维护控制相关和数据相关是如何防止出现这类情景的。考虑以下代码序列：

154

```
        DADDU   R2,R3,R4
        BEQZ    R2,L1
        LW      R1,0(R2)
L1:
```

在这个例子中，可以很容易地看出如果不维护涉及R2的数据相关，就会改变程序的结果。还有

一个事实没有那么明显：如果我们忽略控制相关，将载入指令移到分支之前，这个载入指令可能会导致存储器保护异常。注意，**没有数据相关**禁止交换 BEQZ 和 LW；这只是控制相关。要允许调整这些指令的顺序（而且仍然保持数据相关），我们可能希望在执行这一分支操作时忽略此异常。在 3.6 节，我们将研究一种可以解决这一异常问题的硬件技术——**推测**。附录 H 研究用于支持推测的软件技术。

通过维护数据相关和控制相关来保护的第二个特性是数据流。**数据流**是指数据值在生成结果成和使用结果的指令之间进行的实际流动。分支允许一条给定指令从多个不同地方获取源数据，从而使数据流变为动态的。换种说法，由于一条指令可能会与之前的多条指令存在数据相关性，所以仅保持数据相关是不够的。一条指令的数据值究竟由之前哪条指令提供，是由程序顺序决定的。而程序顺序是通过维护控制相关来保证的。

例如，考虑以下代码段：

```
        DADDU   R1,R2,R3
        BEQZ    R4,L
        DSUBU   R1,R5,R6
L:      ...
        OR      R7,R1,R8
```

在这个例子中，OR 指令使用的 R1 值取决于是否进行了分支转移。单靠数据相关不足以保证正确性。OR 指令数据相关于 DADDU 和 DSUBU 指令，但仅保持这一顺序并不足以保证能够正确执行。

在执行这些指令时，还必须保持数据流：如果没有进行分支转移，那么由 DSUBU 计算的 R1 值应当由 OR 使用；如果进行了分支转移，由 DADDU 计算的 R1 值则应当由 OR 使用。通过保持分支中 OR 的控制相关，就能防止非法修改数据流。出于类似原因，DSUBU 指令也不能移到分支之前。推测不但可以帮助解决异常问题，还能在仍然保持数据流的同时降低控制相关的影响，在 3.6 节将会对此进行讨论。 155

有些情况下，我们可以断定违犯控制相关并不会影响异常行为或数据流。考虑以下代码序列：

```
        DADDU   R1,R2,R3
        BEQZ    R12,skip
        DSUBU   R4,R5,R6
        DADDU   R5,R4,R9
skip:   OR      R7,R8,R9
```

假定我们知道 DSUBU 指令的目标寄存器（R4）在标有 skip 的指令之后不再使用。（一个值是否会被后续指令使用，这一特性被称为**活性**。）如果 R4 不会再被使用，由于它在 skip 之后的代码部分变为**死亡**（不再具备活性），那么就在这个分支之前改变 R4 的值并不会影响数据流。因此，如果 R4 已经死亡，而且现有 DSUBU 指令不会生成异常（处理器会从某些指令处重启同一过程，这些指令除外），那就可以把 DSUBU 指令移到分支之前，数据流不会受这一改变的影响。

如果进行了分支转移，将会执行 DSUBU 指令，之后不再有用，但这样仍然不会影响程序结果。由于编译器在对分支结果进行猜测，所以这种类型的代码调度也是一种推测形式，通常称为**软件推测**；在这个例子中，编译器推测通常不会进行分支转移。附录 H 讨论了一些更加雄心勃勃的编译器推测机制。在我们说到“推测”时，通常可以清楚地知道是在说硬件机制，还是软件机制；如果不够明确，最好使用“硬件推测”或“软件推测”加以区分。

对导致控制停顿的控制冒险进行检测，可以保持控制相关。控制停顿可以通过各种软硬件技术加以消除或减少，3.3 节将研究这些技术。

3.2 揭示 ILP 的基本编译器技术

这一节研究一些简单的编译器技术，可以用来提高处理器开发 ILP 的能力。这些技术对于使用静态发射或静态调度的处理器非常重要。有了这一编译器技术，稍后将会研究那些采用静态发射的处理器的设计与性能。附录 H 将研究一些更高级的编译器和相关硬件方案，其设计目的就是使处理器能够开发更多的指令级并行。

156

3.2.1 基本流水线调度和循环展开

为使流水线保持满载，必须找出可以在流水线中重叠的不相关指令序列，充分开发指令并行。为了避免流水线停顿，必须将相关指令与源指令的执行隔开一定的时间周期，这一间隔应当等于源指令的流水线延迟。编译器执行这种调度的能力既依赖于程序中可用 ILP 的数目，也依赖于流水线中功能单元的延迟。表 3-2 给出了在本章采用的 FP 单元延迟，如果偶尔采用不同延迟，会另行明确说明。假定采用一个标准的 5 级整数流水线，所以分支的延迟为一个时钟周期。假定这些功能单元被完全流水化或复制（复制次数与流水线深度相同），所以在每个时钟周期可以发射任何一个类型的指令，不存在结构性冒险。

表 3-2 本章使用的 FP 运算延迟

生成结果的指令	使用结果的指令	延迟（以时钟周期为单位）
FP ALU 运算	另一个FP ALU运算	3
FP ALU 运算	存储双精度值	2
载入双精度值	FP ALU运算	1
载入双精度值	存储双精度值	0

* 最后一列是为了避免停顿而需要插入的时钟周期数。这些数字与我们在 FP 单元上看到的平均延迟类似。由于可以旁路载入指令的结果，不会使存储指令停顿，所以浮点载入指令对载入指令的延迟为 0。我们还假定整数载入延迟为 1，整数 ALU 操作延迟为 0。

在这一小节，我们将研究编译器如何通过转换循环来提高可用 ILP 的数目。下面这个例子，一方面用于说明一种非常重要的技巧，另一方面还用于推荐人们采用附录 H 中描述的一种功能更强大的程序转换。我们的讨论将就以下代码段展开，它将对一个标量和一个向量求和：

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

注意，这个循环的每个迭代体都是相互独立的，从而可以看出这个循环是并行的。在附录 H 中给出这一概念的正式定义，并说明如何在编译时判断循环迭代是独立的。首先来看这个循环的性能，说明如何利用并行来提高一个 MIPS 流水线的性能（采用以上所示的延迟值）。

第一步是将以上代码段转换为 MIPS 汇编语言。在以下代码段中，R1 最初是数组元素的最高地址，F2 包含标量值 s。寄存器 R2 的值预先计算得出，使 8(R2)成为最后一个进行运算的元素的地址。

157

这段简单的 MIPS 代码应当类似如下所示（未针对流水线进行调度）：

```
Loop:   L.D     F0,0(R1)      ;F0=数组元素
        ADD.D   F4,F0,F2      ;加上 F2 中的标量
        S.D     F4,0(R1)      ;存储结果
        DADDUI  R1,R1,#-8     ;使指针递减 8 个字节
                              ;(每个 DW)
        BNE     R1,R2,Loop    ;R1!=R2 时跳转
```

首先来看看在针对 MIPS 的简单流水线上调度这个循环时的执行情况（延迟如表 3-2 所示）。

例题 写出在进行调度与不进行调度的情况下，这个循环在 MIPS 上的执行过程，包括所有停顿或空闲时钟周期。调度时要考虑浮点运算产生的延迟，但忽略延迟分支。

解答 在不进行任何调度时，循环的执行过程如下，共花费 9 个周期：

```
                                        发射的时钟周期
Loop:   L.D       F0,0(R1)              1
        停顿                            2
        ADD.D     F4,F0,F2              3
        停顿                            4
        停顿                            5
        S.D       F4,0(R1)              6
        DADDUI    R1,R1,#-8             7
        停顿                            8
        BNE       R1,R2,Loop            9
```

我们可以调度这个循环，使其只有 2 次停顿，将花费时间缩短至 7 个周期：

```
Loop:   L.D       F0,0(R1)
        DADDUI    R1,R1,#-8
        ADD.D     F4,F0,F2
        停顿
        停顿
        S.D       F4,8(R1)
        BNE       R1,R2,Loop
```

ADD.D 之后的停顿是供 S.D.使用的。

在上面这个例子中，每 7 个时钟周期完成一次循环迭代，并存回一个数组元素，但对数组元素进行的实际运算仅占用这 7 个时钟周期中的 3 个（载入、求和与存储）。其余 4 个时钟周期包括循环开销（DADDUI 和 BNE）和 2 次停顿。为了消除这 4 个时钟周期，需要使循环体中的运算指令数多于开销指令数。 158

要提高运算指令相对于分支和开销指令的数目，一种简单的方案是**循环展开**。展开就是将循环体复制多次，调整循环的终止代码。

循环展开还可用于提高调度效率。由于它消除了分支，因此可以将来自不同迭代的指令放在一起调度。在这个例子中，我们通过在循环体内创建更多的独立指令来消除数据使用停顿。如果在展开循环时只是简单地复制这些指令，最后使用的都是同一组寄存器，所以可能会妨碍对循环的有效调度。因此，我们希望为每次迭代使用不同寄存器，这就需要增加寄存器的数目。

例题 展开以上循环，使其包含循环体的 4 个副本，假定 R1-R2（即数组的大小）最初是 32 的倍数，也就是说循环迭代的数目是 4 的倍数。消除任何明显的冗余计算，不要重复使用任何寄存器。

解答 合并 DADDUI 指令，删除在展开期间重复的非必需 BNE 运算，得到的结果如下。注意，现在必须对 R2 进行置位，使 32(R2)成为后 4 个元素的起始地址。

```
Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)        ;删除 DADDUI 和 BNE
        L.D     F6,-8(R1)
        ADD.D   F8,F6,F2
        S.D     F8,-8(R1)       ;删除 DADDUI 和 BNE
        L.D     F10,-16(R1)
        ADD.D   F12,F10,F2
        S.D     F12,-16(R1)     ;删除 DADDUI 和 BNE
        L.D     F14,-24(R1)
        ADD.D   F16,F14,F2
        S.D     F16,-24(R1)
        DADDUI  R1,R1,#-32
        BNE     R1,R2,Loop
```

我们省掉了三次分支转移和对 R1 的三次递减。并对载入和存储指令的地址进行了补偿，以允许合并针对 R1 的 DADDUI 指令。这一优化看起来似乎微不足道，但实际并非如此；它需要进行符号替换和化简。符号替换和化简将重新整理表达式，以合并其中的常量，比如表达式$((i+1)+1)$可以重写为$(i+(1+1))$，然后化简为$(i+2)$。这种优化方式消除了相关计算，在附录 H 中将会看到它们的更一般形式。

159

如果没有调度，展开循环中的每个操作后面都会跟有一个相关操作，从而导致停顿。这个循环将运行 27 个时钟周期（每个 LD 有 1 次停顿，每个 ADDD 2、DADDUI 1，加上 14 个指令发射周期），或者说在 4 个元素的每个元素上平均花费 6.75 个时钟周期，但通过调度可以显著提高其性能。循环展开通常是在编译过程的早期完成的，所以优化器可以发现并消除冗余运算。

在实际程序中，人们通常不知道循环的上限。假定此上限为 n，我们希望展开循环，制作循环体的 k 个副本。我们生成的是一对连续循环，而不是单个展开后的循环。第一个循环执行$(n \bmod k)$次，其主体就是原来的循环。第二个循环是由外层循环包围的展开循环，迭代(n/k)次。（在第 4 章将会看到，这一技巧类似于一种在向量处理器的编译器中使用的**条带挖掘**技巧。）当 n 值较大时，大多数执行时间都花费在未展开的循环体上。

在前面的例子中，通过展开消除了开销指令，尽管这样会显著增大代码规模，但却可以提高这一循环的性能。如果针对先前介绍的流水线来调度展开后的循环，它的执行情况又会如何呢？

例题 针对具有表 3-2 所示延迟的流水线，调度前面例子中展开后的循环，写出其执行情况。

解答

```
Loop:   L.D     F0,0(R1)
        L.D     F6,-8(R1)
        L.D     F10,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F4,F0,F2
        ADD.D   F8,F6,F2
        ADD.D   F12,F10,F2
        ADD.D   F16,F14,F2
        S.D     F4,0(R1)
        S.D     F8,-8(R1)
        DADDUI  R1,R1,#-32
        S.D     F12,16(R1)
        S.D     F16,8(R1)
        BNE     R1,R2,Loop
```

展开后循环的执行时间已经缩减到总共 14 个时钟周期，或者说每个元素需要 3.5 个时钟周期，而在未进行任何展开或调度之前为每个元素 9 个时钟周期，进行调度但未展开时为 7 个周期。

对展开循环进行调度所获得的收益甚至还会大于对原循环进行调度的收益。之所以会这样，是因为展开后的循环暴露了更多可以进行调度的计算，从而可以将停顿时间减至最低；上述代码中就没有任何停顿。要以这种方式调度循环，必须意识到载入指令和存储指令是不相关的，可以交换位置。 160

3.2.2 循环展开与调度小结

在本章和附录 H 中，我们将会研究各种可以利用指令级并行的硬件与软件技术，以充分发挥处理器中各功能单元的潜力。大多数此类技术的关键在于判断何时能够改变指令顺序以及如何改变。我们的例子中进行了许多此类改变，对于人类来说，可以很容易地判断出是可以进行此类改变的。而在实践中，这一过程必须采用系统方式，由编译器或硬件来完成。为了获得最终展开后的代码，必须进行如下决策和变换。

- 确认循环迭代不相关（循环维护代码除外），判定展开循环是有用的。
- 使用不同寄存器，以避免由于不同运算使用相同寄存器而施加的非必要约束（比如，名称相关）。
- 去除多余的测试和分支指令，并调整循环终止与迭代代码。
- 观察不同迭代中的载入与存储指令互不相关，判定展开后的循环中的载入和存储指令可以交换位置。这一变换需要分析存储器地址，查明它们没有引用同一地址。
- 对代码进行调度，保留任何必要相关，以得到与原代码相同的结果。

要进行所有这些变换，最关键的要求就是要理解指令之间的相关依赖关系，而且要知道在这些给定关系下如何改变指令或调整指令的顺序。

有 3 种不同的效果会限制循环展开带来的好处：(1)每次展开操作分摊的开销数目降低，(2)代码规模限制，(3)编译器限制。我们首先考虑循环开销问题。如果将循环展开 4 次，所生成的指令并行足以使循环调度消除所有停顿周期。事实上，在 14 个时钟周期中，只有 2 个周期是循环开销：维护索引值的 DADDUI 和终止循环的 BNE。如果将循环展开 8 次，这一开销将从每次原迭代的 1/2 周期降低到 1/4。

对展开的第二个限制是所生成代码规模的增长。对于较大规模的循环，如果代码规模的增长会导致指令缓存缺失率的上升，那就要特别加以关注。

还有一个因素通常要比代码大小更为重要，那就是由于大量进行展开和调试而造成寄存器数量不足。因为在大段代码中进行指令调度而产生的这一副作用被称为**寄存器紧缺**。它的出现是因为在为了提高 ILP 而调试代码时，导致存活值的数目增加。在大量进行指令调度之后，可能无法将所有存活值都分配到寄存器中。尽管转换后的代码在理论上可以提高运行速度，但由于它会造成寄存器短缺，所以可能会损失部分乃至全部收益。在没有展开循环时，分支就足以限制调度的大量使用，所以寄存器紧缺几乎不会成为问题。但是，循环展开与大量调度的综合应用却可能导致这一问题。在多发射处理器中需要公开更多可以重叠执行的独立指令序列，上述问题可能变得尤其困难。一般来说，高级复杂转换的应用已经导致现代编译器的复杂度大幅增加，而在生成具体代码之前，很难衡量这种应用带来的潜在改进。 161

直行代码段可以进行有效的调度，而循环展开是一种增大直行代码段规模的简单有效方法。这种转换在各种处理器上都非常有用，从我们前面已经研究过的简单流水线，到多发射超标量，再到本章后面要研究的 VLIW。

3.3 用高级分支预测降低分支成本

由于需要通过分支冒险和停顿来实施控制相关，所以分支会伤害流水线性能。循环展开是降低分支冒险的一种方法，我们还可以通过预测分支的行为方式来降低分支的性能损失。在附录 C 中，我们将研究一些简单的分支预测器，它们既可能依赖于编译时信息，也可能依赖于在隔离状态下观测到的分支动态行为。随着所用指令数目的增大，更准确的分支预测也变得越来越重要。在本节，我们将研究一些提高动态预测准确度的技术。

相关分支预测

2 位预测器方案仅使用单个分支的最近行为来预测该分支的未来行为。如果我们同时还能查看其他分支的最近行为，而不是仅查看要预测的分支，那就有可能提高预测准确度。考虑 eqntott 基准测试中的一小段代码，这个基准测试是早期 SPEC 基准测试套件的一个成员，用来显示特别糟糕的分支预测行为：

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

162

下面是通常为这一代码段生成的 MIPS 代码，假定 aa 和 bb 分别被赋值给 R1 和 R2：

```
        DADDIU    R3,R1,#-2
        BNEZ      R3,L1          ;branch b1   (aa!=2)
        DADD      R1,R0,R0       ;aa=0
L1:     DADDIU    R3,R2,#-2
        BNEZ      R3,L2          ;branch b2   (bb!=2)
        DADD      R2,R0,R0       ;bb=0
L2:     DSUBU     R3,R1,R2       ;R3=aa-bb
        BEQZ      R3,L3          ;branch b3   (aa==bb)
```

我们将这些分支标记为 b1、b2 和 b3。从中可以看出很重要的一点：分支 b3 的行为与分支 b1 和 b2 的行为有关。显然，如果分支 b1 和 b2 都未执行转移（即其条件均为真，且 aa 和 bb 均被赋值为 0），aa 和 bb 明显相等，所以会进行 b3 分支转移。如果预测器仅利用一个分支的行为来预测分支结果，那显然不会捕获到这一行为。

利用其他分支行为来进行预测的分支预测器称为**相关预测或两级预测器**。现有相关预测器增加最近分支的行为信息，来决定如何预测一个给定分支。例如，(1,2)预测器在预测一个特定分支时，利用最近一个分支的行为从一对 2 位分支预测器中进行选择。在一般情况下，(m,n)预测器利用最近 m 个分支的行为从 2^m 个分支预测器中进行选择，其中每个预测器都是单个分支的 n 位预测器。这种相关分支预测器的吸引力在于它的预测率可以高于 2 位方案，而需要添加的硬件很少。

硬件的这种简易性源自一个简单的观察事实：最近 m 个分支的全局历史可以记录在 m 位移位的寄存器中，其中每一位记录是否执行了该分支转移。将分支地址的低位与 m 位全局历史串联在一起，就可以对分支预测缓冲区进行寻址。例如，在一个共有 64 项的(2,2)缓冲区中，分支的低 4 位地址（字地址）和 2 个全局位（表示最近执行的两个分支的行为）构成一个 6 位索引，可用来对 64 个计数器进行寻址。

与标准的 2 位方案相比，相关分支预测器的性能可以提高多少呢？为了进行公平对比，进行比较的预测器必须使用相同数目的状态位。一个(m,n)预测器的位数为：

$$2^m \times n \times \text{由分支地址选中的预测项数目}$$

163

没有全局历史的 2 位预测器就是(0,2)预测器。

例题 在具有 4K 项的(0,2)分支预测器中有多少位？在具有同样位数的(2,2)预测器中有多少项？

解答 具有 4K 项的预测器拥有：

$$2^0 \times 2 \times 4\text{K}=8\text{K 位}$$

在预测缓冲区中共有 8K 位的(2,2)预测器中有多少由分支选中的项呢？我们知道：

$$2^2 \times 2 \times \text{由分支选中的预测项数}=8\text{K}$$

因此，由分支选中的预测项数=1K。

图 3-1 对比了前面具有 4K 项的(0,2)预测器与具有 1K 项的(2,2)预测器的错误预测率。可以看出，这种相关预测器的性能不但优于具有相同状态位数的简单 2 位预测器，还经常会优于具有无限项数的 2 位预测器。

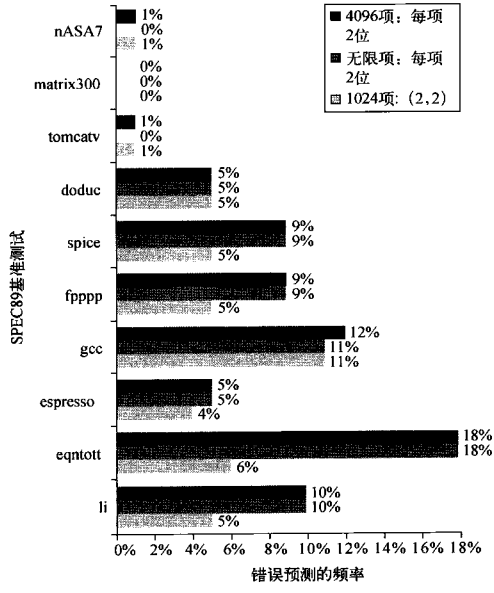


图 3-1 2 位预测器的对比。第一个是 4096 位的非相关预测器，然后是具有无限数目的非相关 2 位预测器，和一个具有 2 位全局历史和总共 1024 项的 2 位预测器。尽管这些数据是从 SPEC 的较早版本获取的，但最近 SPEC 基准测试的数据也显示了类似的准确度差异

3.3.1 竞赛预测器：局部预测器与全局预测器的自适应联合

采用相关分支预测器主是要因为观察到：仅使用局部信息的标准 2 位预测器无法预测某些重要分支，而通过增加全局信息就可以提升性能。**竞赛预测器**更进一步，采用了多个预测器，通常是一个基于全局信息的预测器和一个基于局部信息的预测器，用选择器将它们结合起来。竞赛预测器既可以以中等规模的预测位（8K ~ 32K 位）实现更好的预测准确度，还可以更有效地利用超大量预测位。现有竞赛预测器为每个分支使用一个 2 位饱和预测器，根据哪种预测器（局部、全局，甚至包括两者的组合方式）在最近的预测中最为有效，从两个不同预测器中进行选择。在简单的 2 位预测器中，饱和计数器要在两次预测错误之后才会改变优选预测器的身份。

竞赛预测器的优势在于它能够为特定分支选择正确的预测器，这一点对于整数基准测试尤为重要。对于 SPEC 整数基准测试，典型竞赛预测器为其选择全局预测器的概率大约为 40%，而对于 SPEC FP 基准测试，则少于 15%。除了在竞赛预测器方面走在前列的 Alpha 处理器之外，最近的 AMD 处理器，包括 Opteron 和 Phenom，都已经采用了竞赛类型的预测器。

164 图 3-2 以 SPEC89 为基准测试，研究三种不同预测器（一个局部 2 位预测器、一个相关预测器和竞赛预测器）在不同位数时的性能。和我们前面曾经看到的一样，局部预测器的预测性能改进不会超出一定的范围。相关预测器的改进非常突出，竞赛预测器的性能稍好一些。对于 SPEC 的最新版本，其结果是类似的，但需要采用稍大一些的预测器规模才能达到渐趋一致的行为。

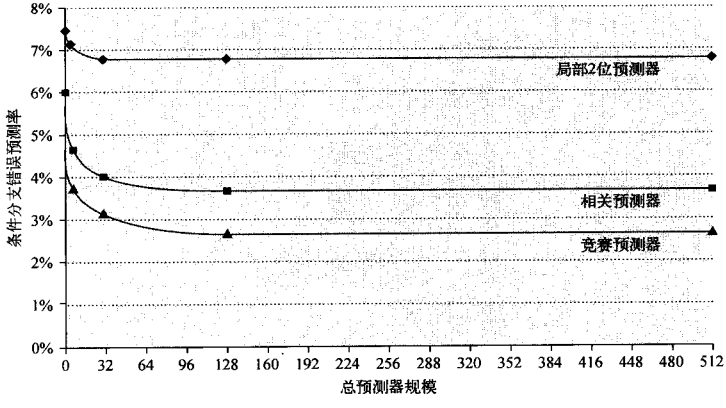


图 3-2 在总位数升高时，3 种不同预测器运行 SPEC89 的错误预测率。这些预测器包括：一个局部 2 位预测器；一个相关预测器；针对图中每一点的全局和局部信息采用了结构优化；一个竞赛预测器。尽管这些数据是从 SPEC 的较早版本获取的，但最近 SPEC 基准测试的数据也显示了类似行为，当预测器规模稍大时，可能趋于一个渐近值

局部预测器包括一个两级预测器。顶级是一个局部历史表，包括 1024 个 10 位项；每个 10 位项对应这一项最近 10 个分支输出。即，如果这个分支被选中 10 次或更多次，那么这个局部

历史表中的相应项都是 1。如果这个分支被交替选中和未选中，那么历史项中则包括交替的 0 和 1。这个 10 位历史信息最多可以发现和预测 10 个分支。局部历史表的选定项用来对一个拥有 1K 项的表进行索引，这个表由 3 位饱和计数器组成，可以提供局部预测。这种组合共使用 29K 位，可以提高分支预测的准确度。 165

3.3.2 Intel Core i7 分支预测器

关于 Core i7 的分支预测器，Intel 只发布了非常有限的信息，这种预测器是以 Core Duo 芯片中使用的较早预测器为基础。i7 使用了一个两级预测器，它的第一级预测器较小，设计用来满足周期约束条件：每个时钟周期预测一个分支，还有一个较大的二级预测器作为备份。每个预测器都组合了 3 个不同预测器：(1)简单的两位预测器，在附录 C 中介绍（上述竞赛预测器中使用了这种预测器）；(2)全局历史预测器，类似于我们刚刚看到的预测器；(3)循环退出预测器。对于一个被检测为循环分支的分支，循环退出预测器使用计数器来预测被选中分支的确切数目（也就是循环迭代的数目）。对于每个分支，通过跟踪每种预测的准确度从 3 个预测器中选择最佳预测，就像一个竞赛预测器。除了这一多级主预测器之外，还有一个独立单元为间接分支预测目标地址，还使用了栈来预测返回地址。 166

和在其他情景中一样，推测在评估预测器方面也导致了一些难题，这是因为对一个分支的错误预测很容易导致提取和错误预测另一个分支。为了使事情变得简单一些，我们看一下错误预测数占成功完成分支数（这些分支不是推测错误导致的结果）的百分比。图 3-3 显示了 19 个 SPECCPU 2006 基准测试的数据。这些基准测试明显大于 SPEC89 或 SPEC2000，其结果是：即使更精心地组合应用预测器，其错误预测率也只是稍大于图 3-2 中所示数据。由于分支预测错误会导致推测效率低下，所以它会导致一些无效工作，在本章后面将会了解到这一点。

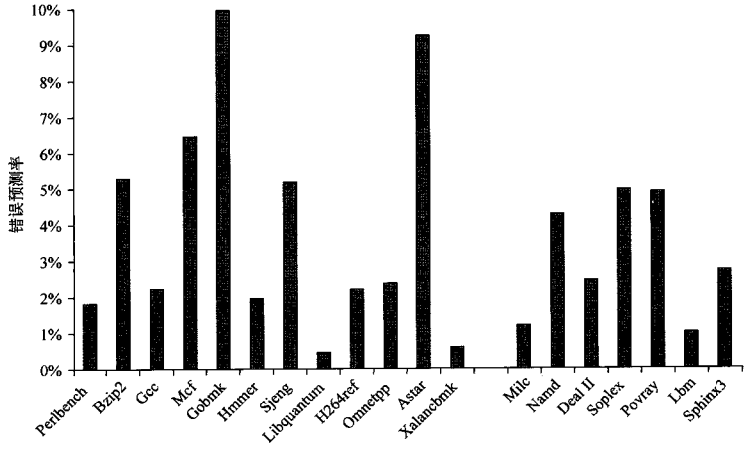


图 3-3 19 个 SPECCPU2006 基准测试错误预测数占成功完成分支数的比例，整数基准测试的平均值稍高于 FP（4%对 3%）。更重要的是，某些基准测试的错误率高出很多

3.4 用动态调度克服数据冒险

除非是流水线中的已有指令与要提取的指令之间存在数据相关，而且无法通过旁路或转发来隐藏这一数据相关，否则，简单的静态调度流水线就会提取一条指令并发射出去。（转发逻辑可以减少实际流水线延迟，所以某些特定的相关不会导致冒险。）如果存在不能隐藏的数据相关，那些冒险检测软件会从使用该结果的指令开始，将流水线置于停顿状态。在清除这一相关之前，不会提取和发射新的指令。

167

本节将研究**动态调度**，在这种调度方式中，硬件会重新安排指令的执行顺序以减少停顿，并同时保持数据流和异常行为。动态调度有几个优点。第一，它允许针对一种流水线编译的代码在不同流水线上高效执行，不需要在使用不同微体系结构时重新进行编译，并拥有多个二进制文件。在当今的计算环境中，大多数软件都来自第三方，而且是以二进制文件形式分发的，这一优势尤其明显。第二，在某些情况下，在编译代码时还不能知道相关性，利用动态调度可以处理某些此类情况；比如，这些相关可能涉及存储器引用或者与数据有关的分支，或者，它们可能源自使用动态链接或动态分发的现代编程环境。第三，也可能是最重要的一个优点，它允许处理器容忍一些预料之外的延迟，比如缓存缺失，它可以在等待解决缺失问题时执行其他代码。在3.6节，我们将研究以动态调度为基础的硬件推测，这一技术还有更多性能方面的优势。我们将会看到，动态调度的好处是以硬件复杂度的显著提高为代价的。

尽管动态调度的处理器不能改变数据流，但它会在存在相关性时尽力避免停顿。相反，由编译器调度的静态流水线（在3.2节介绍）也会尽力将停顿时间降至最低，具体方法是隔离相关指令，使它们不会导致冒险。当然，对于那些本来准备在采用动态调度流水线的处理器上运行的代码，也可以使用编译器流水线调度。

3.4.1 动态调度：思想

简单流水线技术的一个主要限制是它们使用循序指令发射与执行：指令按程序顺序发射，如果一条指令停顿在流水线中，后续指令都不能继续进行。因此，如果流水线中两条相距很近的指令存在相关性，就会导致冒险和停顿。如果存在多个功能单元，这些单元也可能处于空闲状态。如果指令 j 依赖于长时间运行的指令 i（当前正在流水线中执行），那么 j 之后的所有指令都必须停顿，直到 i 完成、j 可以执行为止。例如，考虑以下代码：

```
DIV.D     F0,F2,F4
ADD.D     F10,F0,F8
SUB.D     F12,F8,F14
```

由于 ADD.D 对 DIV.D 的相关性会导致流水线停顿，所以 SUB.D 指令不能执行；但是，SUB.D 与流水线中的任何指令都没有数据相关性。这一冒险会对性能造成限制，如果不需要以程序顺序来执行指令，就可以消除这一限制。

168

在经典的五级流水线中，会在指令译码（ID）期间检查结构冒险和数据冒险：当一个指令可以无冒险执行时，知道所有数据冒险都已经解决，从ID将其发射出去。

为了能够开始执行上面例子中的 SUB.D，必须将发射过程分为两个部分：检查所有结构冒险和等待数据冒险的消失。因此，我们仍然使用循序指令发射（即，按程序顺序发射指令），但我们希望一条指令能够在其数据操作数可用时立即开始执行。这样一种流水线实际是**乱序执行**，

也就意味着**乱序完成**。

乱序执行就可能导致 WAR 和 WAW 冒险，在这个五级整数流水线及其循序浮点流水线的逻辑扩展中不存在这些冒险。考虑以下 MIPS 浮点代码序列：

```
DIV.D    F0,F2,F4
ADD.D    F6,F0,F8
SUB.D    F8,F10,F14
MUL.D    F6,F10,F8
```

在 ADD.D 和 SUB.D 之间存在反相关，如果流水线在 ADD.D 之前执行 SUB.D（ADD.D 在等待 DIV.D），将会违犯反相关，产生 WAR 冒险。与此类似，为了避免违犯输出相关，比如由 MUL.D 写入 F6，就必须处理 WAW 冒险。后面将会看到，利用寄存器重命名可以避免这些冒险。

乱序完成还会使异常处理变得复杂。采用乱序完成的动态调度必须保持异常行为，使那些在严格按照程序顺序执行程序时会发生的异常仍然会**实际发生**，也不会发生其他异常。动态调度处理器会推迟发布相关异常的发布，一直等到处理器知道该指令就是接下来要完成的指令为止，通过这一方式来保持异常行为。

尽管异常行为必须保持，但动态调度处理器可能生成一些**非精确**异常。如果在发生异常时，处理器的状态与严格按照程序顺序执行指令时的状态不完全一致，那就说这一异常是**非精确的**。非精确异常可以因为以下两种可能性而发生。

(1) 流水线在执行导致异常的指令时，可能**已经完成**了按照程序顺序排在这一指令**之后**的指令。

(2) 流水线在执行导致异常的指令时，可能**还没有完成**按照程序顺序排在这一指令**之前**的指令。

非精确异常增大了在异常之后重新开始执行的难度。我们在这一节不会解决这些问题，而是讨论一种解决方案，能够在具有推测功能的处理器环境中（见 3.6 节）提供精确异常。对于浮点异常，已经采用了其他解决方案，见附录 J 中的讨论。 169

为了能够进行乱序执行，我们将五级简单流水线的 ID 流水级大体分为以下两个阶段。

(1) **发射**——译码指令，检查结构性冒险。

(2) **读操作数**——一直等到没有数据冒险，然后读取操作数。

指令提取阶段位于发射阶段之前，既可以把指令放到指令寄存器中，也可能放到一个待完成指令队列中；然后从寄存器或队列发射这些指令。执行阶段跟在读操作数阶段之后，这一点和五级流水线中一样。执行过程可能需要多个周期，具体数目取决于所执行的操作。

我们区分一个指令**开始执行**和**完成执行**的时刻，在这两个时刻之间，指令**处于执行过程中**。我们的流水线允许同时执行多条指令，没有这一功能，就会失去动态调度的主要优势。要同时执行多条执行，需要有多个功能单元、流水化功能单元，或者同时需要这两者。由于这两种功能（流水化功能单元和多个功能单元）在流水线控制方面大体相当，所以我们假定处理器拥有多个功能单元。

在动态调度流水线中，所有指令都循序经历发射阶段（循序发射）；但是，它们可能在第二阶段（读操作数阶段）停顿或者相互旁路，从而进行乱序执行状态。**记分板**技术允许具有足够资源和没有数据相关时乱序执行指令，它的名字源于 CDC 6600 记分板，CDC 6600 记分板开发了这一功能。这里重点介绍一种名为 **Tomasulo 算法**的更高级技术。它们之间的主要区别在于 Tomasulo 算法通过对寄存器进行有效动态重命名来处理反相关和输出相关。此外，还可以对

Tomasulo算法进行扩展，用来处理推测，这种技术通过预测一个分支的输出、执行预测目标地址的指令、当预测错误时采取纠正措施，从而降低控制相关的影响。使用记分板可能足以支持诸如ARM A8之类的简单两发射超标量处理器，而诸如四发射Intel i7之类更具主动性的处理器则受益于乱序执行的使用。

3.4.2 使用Tomasulo算法进行动态调度

IBM 360/91浮点单元使用一种支持乱序执行的高级方案。这一方案由Robert Tomasulo发明，它会跟踪指令的操作数何时可用，将RAW冒险降至最低，并在硬件中引入寄存器重命名功能，将WAW和WAR冒险降至最低。在现代处理器中存在这一方案的许多变体，但跟踪指令相关以允许在操作数可用时立即执行指令、重命名寄存器以避免WAR和WAW冒险，这些核心概念仍然是它们的共同特征。

170

IBM的目标是从指令集出发、从为整个350计算机系列设计的编译器出发来实现高浮点性能，而不是通过采用专门为高端处理器设计的编译器来实现。360体系结构只有4个双精度浮点寄存器，它限制了编译器调度的有效性；这一事实是开发Tomasulo方法的另一个动机。此外，IBM 360/91的内存访问时间和浮点延迟都很长，Tomasulo算法就是设计用来克服这些问题的。在本节的最后，我们将会看到Tomasulo算法还支持重叠执行一个循环的多个迭代。

我们将在MIPS指令集上下文中解释这一算法，重点放在浮点单元和载入-存储单元。MIPS与360之间的主要区别是后者的体系结构中存在寄存器-存储器指令。由于Tomasulo算法使用一个载入功能单元，所以添加寄存器-存储器寻址模式并不需要进行大量修改。IBM 360/91还有一点不同，它拥有的是流水化功能单元，而不是多个功能单元，但我们在描述该算法时仍然假定它有多个功能单元。它只是对功能单元进行流水化的概念扩展。

我们将会看到，如果仅在操作数可用时才执行指令，就可以避免RAW冒险，而这正是一些简单记分板方法提供的功能。WAR和WAW冒险（源于名称相关）可以通过**寄存器重命名**来消除。对所有目标寄存器（包括较早指令正在进行读取或写入的寄存器）进行重命名，使乱序写入不会影响到任何依赖某一操作数较早值的指令，从而消除WAR和WAW冒险。

为了更好地理解寄存器重命名如何消除WAR和WAW冒险，考虑以下可能出现WAR和WAW冒险的代码序列示例：

```
DIV.D    F0,F2,F4
ADD.D    F6,F0,F8
S.D      F6,0(R1)
SUB.D    F8,F10,F14
MUL.D    F6,F10,F8
```

以上代码共有两处反相关：ADD.D与SUB.D之间，S.D和MUL.D之间。在ADD.D和MUL.D之间还有一处输出相关，从而一共可能存在3处冒险：ADD.D使用F8和SUB.D使用F6时的WAR冒险，以及因为ADD.D可能在MUL.D之后完成所造成的WAW冒险。还有3个真正的数据相关：DIV.D和ADD.D之间、SUB.D和MUL.D之间、ADD.D和S.D之间。

171

这3个名称相关都可以通过寄存器重命名来消除。为简便起见，假定存在两个临时寄存器：S和T。利用S和T，可以对这一序列进行改写，使其没有任何相关，如下所示：

```
DIV.D     F0,F2,F4
ADD.D     S,F0,F8
S.D       S,0(R1)
SUB.D     T,F10,F14
MUL.D     F6,F10,T
```

此外，对 F8 的任何后续使用都必须用寄存器 T 来代替。在这个代码段中，可以由编译器静态完成这一重命名过程。要在后续代码中找出所有使用 F8 的地方，需要采用高级编译器分析或硬件支持，这是因为上述代码段与后面使用 F8 的位置之间可能存在插入分支。我们将会看到，Tomasulo 算法可以处理跨越分支的重命名问题。

在 Tomasulo 方案中，寄存器重命名功能由**保留站**提供，保留站会为等待发射的指令缓冲操作数。其基本思想是：保留站在一个操作数可用时马上提取并缓冲它，这样就不再需要从寄存器中获取该操作数。此外，等待执行的指令会指定保留站，为自己提供输入。最后，在对寄存器连续进行写入操作并且重叠执行时，只会实际使用最后一个操作更新寄存器。在发射指令时，会将待用操作数的寄存器说明符更名，改为保留站的名字，这就实现了寄存器重命名功能。

由于保留站的数目可能多于实际寄存器，所以这一技术甚至可以消除因为名称相关而导致的冒险，这类冒险是编译器所无法消除的。在研究 Tomasulo 方案的各个部分时，我们将再次讨论寄存器重命名这一主题，了解重命名究竟如何实现以及它如何消除 WAR 和 WAW 冒险。

使用保留站，而不使用集中式寄存器堆，可以导致另外两个重要特性。第一，冒险检测和执行控制是分布式的：每个功能单元保留站中保存的信息决定了一条指令什么时候可以开始在该单元中执行。第二，结果将直接从缓冲它们的保留站中传递给功能单元，而不需要经过寄存器。这一旁路是使用公共结果总线完成的，它允许同时载入所有等待一个操作数的单元（在 360/91 中，这种总线被称为**公共数据总线**，或 CDB）。在具有多个执行单元并且每个时钟周期发射多条指令的流水线中，将需要不止一条总线。

图 3-4 给出了基于 Tomasulo 算法的处理器的基本结构，其中包括浮点单元和载入/存储单元；所有执行控制表均未显示。每个保留站保存一条已经被发射、正在功能单元等待执行的指令，如果已经计算出这一指令的操作数值，则保留这些操作数值，如果还没有计算出，则保留提供这些操作数值的保留站名称。 172

载入缓冲区和存储缓冲区保存来自和进入存储器的数据或地址，其行为方式基本与保留站相同，所以我们仅在必要时才区分它们。浮点寄存器通过一对总线连接到功能单元，由一根总线连接到存储缓冲区。来自功能单元和来自存储器的所有结果都通过公共数据总线发送，它会通向除载入缓冲区之外的所有地方。所有保留站都有标记字段，供流水线控制使用。

在详细描述保留站和此算法之前，让我们看看一条指令所经历的步骤。尽管每条指令现在可能需要任意数目的时钟周期，但一共只有以下 3 个步骤。 173

(1) **发射**——从指令队列的头部获取下一条指令，指令队列按 FIFO 顺序维护，以确保能够保持数据流的正确性。如果有一个匹配保留站为空，则将这条指令发送到这个站中，如果操作数值当前已经存在于寄存器，也一并发送到站中。如果没有空保留站，则存在结构性冒险，该指令会停顿，直到有保留站或缓冲区被释放为止。如果操作数不在寄存器中，则一直跟踪将生成这些操作数的功能单元。这一步骤将对寄存器进行重命名，消除 WAR 和 WAW 冒险。（在动态调度处理器中，这一阶段有时被称为**分派**。）

(2) **执行**——如果还有一个或多个操作数不可用，则在等待计算的同时监视公共数据总线。当一个操作数变为可用时，就将它放到任何一个正在等待它的保留站中。当所有操作数都可用时，则可以在相应功能单元中执行运算。通过延迟指令执行，直到操作数可用为止，可以避免RAW冒险。(一些动态调度处理器将这一步骤称为“发射”，但我们使用“执行”一词，在第一个动态调度处理器CDC 6600中使用的就是这个名字。)

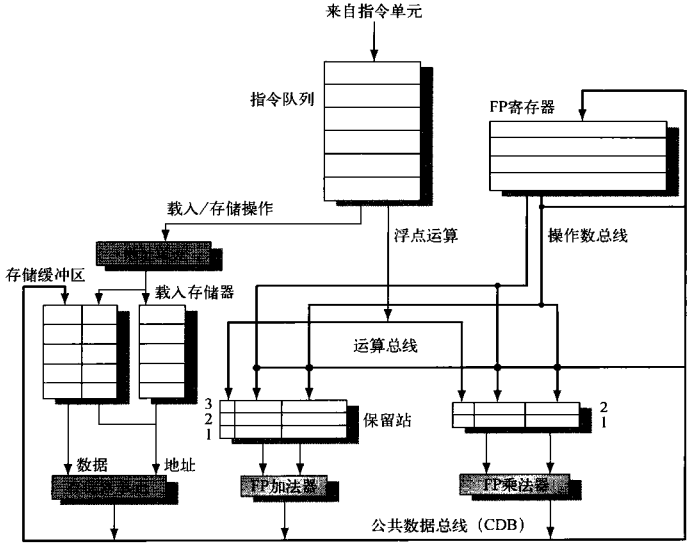


图 3-4 使用 Tomasulo 算法的 MIPS 浮点单元的基本结构。指令由指令单元发送给指令队列，再按先入先出（FIFO）顺序从指令队列中发射出去。保留站包括运算和实际操作数，还有用于检测和解决冒险的信息。载入缓冲区有3项功能：(1)保存有效地址的各个部分，直到计算完成；(2)跟踪正在等待存储器的未完成载入过程；(3)保存正在等待CDB的已完成载入过程的结果。与此类似，存储器缓冲区也有3项功能：(1)保存有效地址的各个部分，直到计算完成；(2)对于尚未完成、正在等待待存储数据值的存储过程，存储其目标存储器地址；(3)保存要存储的地址和数据值，直到存储器单元可用为止。来自FP单元或载入单元的所有结果都被放在CDB中，它会通向FP寄存器堆以及保留站和存储器缓冲区。FP加法器实现加法和减法，FP乘法器完成乘法和除法

注意，在同一时钟周期，同一功能单元可能会有几条指令同时变为就绪状态。尽管独立功能单元可以在同一时钟周期执行不同指令，如果单个功能单元有多条指令准备就绪，那这个单元就必须从这些指令中进行选择。对于浮点保留站，可以任意作出这一选择；但是载入和存储指令可能要更复杂一些。

载入和存储指令的执行过程需要两个步骤。第一步是在基址寄存器可用时计算有效地址，然后将有效地址放在载入缓冲区或存储缓冲区中。载入缓冲区中的载入指令在存储器单元可用时立

即执行。存储缓冲区中的存储指令等待要存储的值，然后将其发送给存储器单元。通过有效地址的计算，载入和存储指令保持程序顺序，稍后将会看到，这样有助于通过存储器来避免冒险。

为了保持异常行为，对于任何一条指令，必须要等到根据程序顺序排在这条指令之前的所有分支全部完成之后，才能执行该指令。这一限制保证了在执行期间导致异常的指令实际上已经执行。在使用分支预测的处理器中（就和所有动态调度处理器一样），这意味着处理器在允许分支之后的指令开始执行之前，必须知道分支预测是正确的。如果处理器记录了异常的发生，但没有实际触发，则可以开始执行一条指令，在进入写结果阶段之前没有停顿。

后面可以看到，推测提供了一种更灵活、更完整的异常处理方法，所以我会将推后进行这一改进，并说明推测是如何解决这一问题的。 174

(3) **写结果**——在计算出结果之后，将其写到 CDB 上，再从 CDB 传送给寄存器和任意等待这一结果的保留站（包括存储缓冲区）。存储指令一直缓存在存储缓冲区中，直到待存储值和存储地址可用为止，然后在有空闲存储器单元时，立即写入结果。

保留站、寄存器堆和载入/存储缓冲区都采用了可以检测和消除冒险的数据结构，根据对象的不同，这些数据结构中的信息也稍有不同。这些标签实际上就是用于重命名的虚拟寄存器扩展集的名字。在这里的例子中，标签字段包含 4 个数位，用来表示 5 个保留站之一或 5 个载入缓冲区之一。后面将会看到，这相当于设定了 10 个可以指定为结果寄存器的寄存器（而 360 体系结构中包含 4 个双精度寄存器）。在拥有更多真正寄存器的处理器中，我们可能希望重命名能够提供更多的虚拟寄存器。标签字段指出哪个保留站中包含的指令将会生成作为源操作数的结果。

在指令被发射出去并开始等待源操作数之后，将使用一个保留站编号来引用该操作数，这个保留站中保存着将对寄存器进行写操作的指令。如果使用一个未用作保留站编号的值来引用该操作数（比如 0），则表明该操作数已经在寄存器中准备就绪。由于保留站的数目多于实际寄存器数目，所以使用保留站编号对结果进行重命名，就可以避免 WAW 和 WAR 冒险。在 Tomasulo 方案中，保留站被用作扩展虚拟寄存器，而其他方法可能使用拥有更多寄存器的寄存器集，也可能使用诸如重排序缓冲区这样的结构，在 3.6 节将会进行讨论。

在 Tomasulo 方案以及后面将会介绍的支持推测的方法中，结果都是在受保留站监视的总线（CDB）上广播。采用公用结果总线，再由保留站从总线中提取结果，共同实现了静态调度流水线中使用的转发和旁路机制。但在这一做法中，动态调度方案会在源与结果之间引入一个时钟周期的延迟，这是因为要等到“写结果”阶段才能让结果与其应用匹配起来。因此，在动态调度流水线中，在生成结果的指令与使用结果的指令之间至少要比生成该结果的功能单元的延迟长一个时钟周期。

一定别忘了，Tomasulo 方案中的标签引用的是将会生成结果的缓冲区或单元；当一条指令发射到保留站之后，寄存器名称将会丢弃。（这是 Tomasulo 方案与记分板之间的一个关键区别：在记分板中，操作数保存在寄存器中，只有生成结果的指令已经完成、使用结果的指令做好执行准备之后才会读取操作数。）

每个保留站有以下 7 个字段。 175

- Op——对源操作数 S1 和 S2 执行的运算。
- Qj、Qk——将生成相应源操作数的保留站；当取值为 0 时，表明已经可以在 Vj 或 Vk 中获得源操作数，或者不需要源操作数。

- Vj、Vk——源操作数的值。注意，对于每个操作数，V字段和Q字段中只有一个是有效的。对于载入指令，Vk字段用于保存偏移量字段。
- A——用于保存为载入或存储指令计算存储器地址的信息。在开始时，指令的立即数字段存储在这里；在计算地址之后，有效地址存储在这里。
- Busy——指明这个保留站及其相关功能单元正被占用。

寄存器堆有一个字段Qi。

- Qi——一个运算的结果应当存储在这个寄存器中，则Qi是包含此运算的保留站的编号。如果Qi的值为空（或0），则当前没有活动指令正在计算以此寄存器为目的地的结果，也就是说这个值就是寄存器的内容。

载入缓冲区和存储缓冲区各有一个字段A，一旦完成了第一个执行步骤，这个字段中就包含了有效地址的结果。

在下一节，我们将首先看一些示例，说明这些机制是如何工作的，然后再详细研究具体算法。

3.5 动态调度：示例和算法

在详细研究Tomasulo算法之前，让我们看几个示例，这些示例有助于说明这种算法的工作方式。

例题 对于以下代码序列，写出在仅完成了第一条载入指令并已将其结果写到CDB总线时的信息表：

```
1.    L.D      F6,32(R2)
2.    L.D      F2,44(R3)
3.    MUL.D    F0,F2,F4
4.    SUB.D    F8,F2,F6
5.    DIV.D    F10,F0,F6
6.    ADD.D    F6,F8,F2
```

解答 表3-3用3个表显示了其结果。Add、Mult和Load之后附加的数字表示保留站的标签——Add1是第一加法单元计算结果的标签。此外，我们还给出了一个指令状态表。之所以列出这个表是为了帮助读者理解这一算法；它不是硬件的实际组成部分，而是由保留站来保存每个已发射运算的状态。

176

表3-3 当所有指令都已经被发射，但只有第一条载入指令已经完成而且已将其结果写到CDB时的保留站与寄存器标签。

指令		指令状态		
		发射	执行	写结果
L.D	F6,32(R2)	✓	✓	✓
L.D	F2,44(R3)	✓	✓	
MUL.D	F0,F2,F4	✓		
SUB.D	F8,F2,F6	✓		
DIV.D	F10,F0,F6	✓		
ADD.D	F6,F8,F2	✓		

（续）

保留站								
名　称	繁忙	Op	Vj	Vk	Qj	Qk	A	
Load1	否							
Load2	是			Load				44+Regs[R3]
Add1	是			SUB		Mem[32+ Regs[R2]]	Load2	
Add2	是			ADD			Add1	Load2
Add3	否							
Mult1	是			MUL		Regs[F4]	Load2	
Mult2	是			DIV		Mem[32+ Regs[R2]]	Mult1	

寄存器状态									
字段	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

* 第二条载入指令已经完成有效地址的计算，但还在等待存储器单元。我们用数组 Regs[]引用寄存器堆，用数组 Mem[]引用存储器。记住，在任何时刻，操作数由 Q 字段或 V 字段指定。注意，ADD.D 指令已经发射（它在 WB 阶段有一个 WAR 冒险），可能在 DIV.D 开始之前完成。

与先前的较简单方案相比，Tomasulo 方案有两点优势：(1)冒险检测逻辑的分布；(2)消除了可能产生 WAW 和 WAR 冒险的停顿。

177

第一个优势源于分布式保留站和 CDB 的使用。如果多条指令正在等待同一个结果，而每条指令的其他操作数均已准备就绪，那么在 CDB 上广播这一结果就可以同时释放这些指令。如果使用集中式的寄存器堆，这些单元必须在寄存器总线可用时从寄存器中读取自己的结果。

第二个优势（消除 WAW 和 WAR 冒险）的实现是利用保留站来重命名寄存器，并在操作数可用时立即将其存储在保留站中。

例如，尽管存在涉及 F6 的 WAR 冒险，但表 3-3 中的代码序列发射了 DIV.D 和 ADD.D。这一冒险通过两种方法之一消除。第一种方法：如果为 DIV.D 提供操作数的指令已经完成，则 Vk 中会存储这个结果，使 DIV.D 不需要 ADD.D 就能执行（表中所示的就是这种情况）。另一方面，如果 L.D 还没有完成，则 Qk 将指向 Load1 保留站，DIV.D 指令不再依赖于 ADD.D。因此，在任一情况下，ADD.D 都可以发射并开始执行。在用到 DIV.D 的结果时，都会指向保留站，使 ADD.D 能够完成，并将其值存储在寄存器中，不会影响到 DIV.D。

稍后将会看到一个消除 WAW 冒险的例子。但先来看看前面的示例是如何继续执行的。在这个例子以及本章后面的例子中，假定有如下延迟值：载入指令为 1 个时钟周期，相加指令为 2 个时钟周期，乘法指令为 6 个时钟周期，除法指令为 12 个时钟周期。

例题　对于上例中的同一代码段，给出当 MUL.D 做好写出结果准备时的状态表。

解答　其结果如表 3-4 中的 3 个表格所示。注意，在复制 DIV.D 的操作数时 ADD.D 已经完成，所以克服了 WAR 冒险问题。注意，既然 F6 的载入操作被延迟，在执行对 F6 的加法操作时也不会触发 WAW 冒险。

表3-4 只有乘法与除法指令还没有完成

指令		指令状态		
		发射	执行	写结果
L.D	F6,32(R2)	✓	✓	✓
L.D	F2,44(R3)	✓	✓	✓
MUL.D	F0,F2,F4	✓	✓	
SUB.D	F8,F2,F6	✓	✓	✓
DIV.D	F10,F0,F6	✓		
ADD.D	F6,F8,F2	✓	✓	✓

保留站

名称	繁忙	Op	Vj	Vk	Qj	Qk	A
Load1	否						
Load2	否						
Add1	否						
Add2	否						
Add3	否						
Mult1	是	MUL	Mem[44 + Regs[R3]]	Regs[F4]			
Mult2	是	DIV		Mem[32 + Regs[R2]]	Mult1		

寄存器状态

字段	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1					Mult2			

3.5.1 Tomasulo 算法：细节

表 3-5 给出了每条指令都必须经历的检查和步骤。前面曾经提到，载入指令和存储指令在进入独立载入或存储缓冲区之前，要经过一个进行有效地址计算的功能单元。载入指令会进入第二执行步骤，以访问存储器，然后进入“写结果”阶段，将来自存储器的值写入寄存器堆以及（或者）任何正在等待的保留站。存储指令在写结果阶段完成其执行，将结果写到存储器中。注意，无论目标是寄存器还是存储器，所有写入操作都在“写结果”阶段发生。这一限制简化了 Tomasulo 算法，是其扩展到支持推测功能的关键（在 3.6 节将讨论这一扩展）。

178

表 3-5 算法步骤以及每个步骤需要的内容

指令状态	等待条件	操作或记录工作

发射
FP操作
	站r空	if (RegisterStat[rs].Qi|0)
		{RS[r].Qj ← RegisterStat[rs].Qi}
		else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};
		if (RegisterStat[rt].Qi|0)
		{RS[r].Qk ← RegisterStat[rt].Qi
		else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
		RS[r].Busy ← yes; RegisterStat[rd].Q ← r;

载入或存储	缓冲区r空	if (RegisterStat[rs].Qi|0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;
仅载入		RegisterStat[rt].Q1 ← r;
仅存储		if (RegisterStat[rt].Qi|0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
执行 FP操作	(RS[r].Qj = 0) 和 (RS[r].Qk = 0)	计算结果：操作数在Vj和Vk中
载入/存储步骤1	RS[r].Qj = 0 或 r 是载入-存储序列的头载入步骤1完成	RS[r].A ← RS[r].Vj + RS[r].A;
载入步骤2		从Mem[RS[r].A] 读取
写结果 FP操作或载入	r处的执行完成且CDB可用	∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;
存储	r处的执行完成且r&RS[r]Qk = 0	Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;

* 对于发射指令，rd 是目的地、rs 和 rt 是源寄存器编号、imm 是符号扩展立即数字段，r 是为指令指定的保留站或缓冲区。RS 是保留站数据结构。FP 单元或载入单元返回的值称为 result。RegisterStat 是寄存器状态数据结构（不是寄存器堆，寄存器堆应当是 Regs[]）。当发射指令，目标寄存器的 Qi 字段被设置为向其发射该指令的缓冲区或保留站编号。如果操作数已经存在于寄存器中，就将它们存储在 V 字段中。否则，设置 Q 字段，指出将生成源操作数值的保留站。指令将一直在保留站中等待，直到它的两个操作数都可用为止，当 Q 字段中的取值为 0 时即表示这一状态。当指令已被发射，或者当这一指令所依赖的指令已经完成并写回结果时，这些 Q 字段被设置为 0。当一条指令执行完毕，并且 CDB 可用时，它就可以进行写回操作。任何一个缓冲区、寄存器和保留站，只要其 Qj 或 Qk 值与完成该指令的保留站相同，都会由 CDB 更新其取值，并标记 Q 字段，表明已经接收到这些值。因此，CDB 可以在一个时钟周期中向许多目标广播其结果，如果正在等待这一结果的指令已经有了其他操作数，那就都可以在下一个时钟周期开始执行了。载入指令要经历两个执行步骤，存储指令在写结果阶段稍有不同，它们必须在这一阶段等待要存储的值。记住，为了保持异常行为，如果按照排在程序顺序前面的分支还没有完成，就不应允许执行后面的指令。由于在发射阶段之后不再保持任何有关程序顺序的概念，因此，为了实施这一限制，在流水线中还有未完成的分支时，通常不允许任何指令离开发射步骤。在 3.6 节中，我们将看到推测支持是如何消除这一限制的。

3.5.2 Tomasulo算法：基于循环的示例

为了理解通过寄存器的动态重命名来消除 WAW 和 WAR 冒险的强大威力，我们需要看一个循环。考虑下面的简单序列，将一个数组的元素乘以 F2 中的标量：

```
Loop:   L.D     F0,0(R1)
        MUL.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDIU  R1,R1,-8
        BNE     R1,R2,Loop; branches if R1|R2
```

如果我们预测会执行这些分支转移，那使用保留站可以同时执行这个循环的多条指令。不需要修改代码就能实现这一好处——实际上，这个循环是由硬件使用保留站动态展开的，这些保留站经过重命名获得，充当附加寄存器。

假定已经发射了该循环两个连续迭代中的所有指令，但一个浮点载入/存储指令或运算也没有完成。表 3-6 显示了在此时刻的保留站、寄存器状态表和载入缓冲区与存储缓冲区。（整数 ALU 运算被忽略，假定预测选中该分支。）一旦系统达到这一状态，如果乘法运算可以在 4 个时钟周期内完成，则流水线中可以保持该循环的两个副本，CPI 接近 1.0。在达到稳定状态之前，还需要处理其他一些迭代，延迟为 6 个时钟周期。这需要有更多的保留站来保存正在运行的指令。在本章后面将会看到，在采用多指令发射对 Tomasulo 方法进行扩展时，它可以保持每个时钟周期处理一条以上指令的速度。

表 3-6 还没有指令完成时，循环的两个活动迭代

指令		指令状态			
		来自迭代	发射	执行	写结果
L.D	F0,0(R1)	1	✓	✓	
MUL.D	F4,F0,F2	1	✓		
S.D	F4,0(R1)	1	✓		
L.D	F0,0(R1)	2	✓	✓	
MUL.D	F4,F0,F2	2	✓		
S.D	F4,0(R1)	2	✓		

保留站							
名称	繁忙	Op	Vj	Vk	Qj	Qk	A
Load1	是	Load					Regs[R1] + 0
Load2	是	Load					Regs[R1] - 8
Add1	否						
Add2	否						
Add3	否						
Mult1	是	MUL		Regs[F2]	Load1		
Mult2	是	MUL		Regs[F2]	Load2		
Store1	是	Store	Regs[R1]			Mult1	
Store2	是	Store	Regs[R1] - 8			Mult2	

寄存器状态									
字段	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Load2		Mult2						

181

* 乘法器保留站中的项目指出尚未完成的载入指令是操作数来源。存储保留站指出乘法运算的目标位置是待存储值的来源。

只要载入指令和存储指令访问的是不同地址，就可以放心地乱序执行它们。如果载入指令和存储指令访问相同地址，则会出现以下两种情况之一：

- 根据程序顺序，载入指令位于存储指令之前，交换它们会导致 WAR 冒险；
- 根据程序顺序，存储指令位于载入指令之前，交换它们会导致 RAW 冒险。

依此类似，交换两个访问同一地址的存储指令会导致 WAW 冒险。

因此，为了判断在给定时刻是否可以执行一条载入指令，处理器可以检查：根据程序顺序排在该载入指令之前的任何未完成存储指令是否与该载入指令共享相同数据存储器地址。对于存储指令也是如此，如果按照程序顺序排在它前面的载入指令或存储指令与它访问的存储器地

址相同，那它必须等到所有这些指令都执行完毕之后才能开始执行。在 3.9 节将考虑一种去除这一限制的方法。

为了检测此类冒险，处理器必须计算出与任何先前存储器运算有关的数据存储器地址。为了保证处理器拥有所有此类地址，一种简单但不一定最优的方法是按照程序顺序来执行有效地址计算。（实际只需要保持存储及其他存储器引用之间的相对顺序，也就是说，可以随意调整载入指令的顺序。）

首先来考虑载入指令的情况。如果按程序顺序执行有效地址计算，那么当一条载入指令完成有效地址计算时，就可以通过查看所有活动存储缓冲区的 A 字段来确定是否存在地址冲突。如果载入地址与存储缓冲区中任何活动项目的地址匹配，则在发生冲突的存储指令完成之前，不要将载入指令发送到载入缓冲区。（有些实施方式将要存储的值直接传送给载入指令，减少了因为这一 RAW 冒险造成的延迟。）

存储指令的工作方式类似，只是因为发生冲突的存储指令不能调整相对于载入或存储指令的顺序，所以处理器必须在载入与存储两个缓冲区中检查是否存在冲突。

如果能够准确预测分支（这是在上一节解决的问题），动态调度流水线可以提供非常高的性能。这种方法的主要缺点在于 Tomasulo 方案的复杂性，它需要大量硬件。具体来说，每个保留站都必须包含一个必须高速运转的相关缓冲区，还有复杂的控制逻辑。它的性能还可能受到单个 CDB 的限制。尽管可以增加更多 CDB，但每个 CDB 都必须与每个保留站进行交互，必须在每个保留站为每个 CDB 配备相关标签匹配硬件。

在 Tomasulo 方案中，可以组合使用两种不同技术：对体系结构寄存器重命名，提供更大的寄存器集合；缓冲来自寄存器堆的源操作数。源操作数缓冲消除了当操作数在寄存器中可用时出现的 WAR 冒险。后面将会看到，通过对寄存器重命名，再结合对结果的缓存，直到对寄存器早期数据的引用全部结束，这样也有可能消除 WAR 冒险。在我们讨论硬件推测时将会用到这一方法。 182

在 360/91 之后的许多年，Tomasulo 方案一直没有得到应用，但从 20 世纪 90 年代开始在多发射处理器中采用，原因有如下几个。

(1) 尽管 Tomasulo 算法是在缓存出现之前设计的，但缓存的出现以及其固有的不可预测的延迟，已经成为使用动态调度的主要动力之一。乱序执行可以让处理器在等待解决缓存缺失的同时继续执行指令，从而消除了全部或部分缓存缺失代价。

(2) 随着处理器的发射功能变得越来越强大，设计人员越来越关注难以调度的代码（比如，大多数非数值代码）的性能，所以诸如寄存器重命名、动态调度和推测等技术变得越来越重要。

(3) 无需编译器针对特定流水线结构来编译代码，Tomasulo 算法就能实现高性能，在盒装光盘套装、批量销售软件的时代，这是一个非常富有价值的性质。

3.6 基于硬件的推测

当我们尝试开发更多指令级并行时，控制相关的维护就会成为一项不断加重的负担。分支预测减少了由于分支导致的直接停顿，但对于每个时钟周期要执行多条指令的处理器来说，仅靠正确地预测分支可能不足以生成期望数量的指令级并行。宽发射处理器可能需要每个时钟周期执行一个分支才能维持最高性能。因此，要开发更多并行，需要我们克服控制相关的局限性。

通过预测分支的输出，然后在假定猜测正确的前提下执行程序，可以克服控制相关问题。这种机制对采用动态调度的分支预测进行了一种虽细微但很重要的扩展。具体来说，通过推测，我们提取、发射和**执行**指令，就好像分支预测总是正确的；而动态调度只是提取和发射这些指令。当然，我们需要一些机制来处理推测错误的情景。附录H讨论了各种由编译器支持推测的机制。这一节研究**硬件推测**，它延伸了动态调度的思想。

基于硬件的推测结合了3种关键思想：(1)用动态分支预测选择要执行哪些指令；(2)利用推测，可以在解决控制相关问题之前执行指令（能够撤消错误推测序列的影响）；(3)进行动态调度，以应对基本模块不同组合方式的调度。（与之相对，没有推测的动态调度需要先解析分支才能实际执行后期基本模块的操作，因为只能部分重叠基本模块。）

183

基于硬件的推测根据预测的数据值流来选择何时执行指令。这种执行程序的方法实际上是一种**数据流执行**：操作数一旦可用就立即执行运算。

为了扩展Tomasulo算法，使其支持推测，我们必须将指令结果的旁路（以推测方式执行指令时需要这一操作）从一条指令的实际完成操作中分离出来。进行这种分离之后，就可以允许执行一条指令，并将其结果旁路给其他指令，但不允许这条指令执行任何不能撤消的更新操作，直到确认这条指令不再具有不确定性为止。

使用旁路值类似于执行一次推测寄存器读取操作，因为在提供源寄存器值的指令不再具有不确定性之前，我们无法知道它是否提供了正确值。当一个指令不再具有不确定性时，允许它更新寄存器堆或存储器；我们将指令执行序列中的这个附加步骤称为**指令提交**。

实现推测之后的关键思想在于允许指令乱序执行，但强制它们**循序**提交，以防止在指令提交之前采取任何不可挽回的动作（比如更新状态或激发异常）。因此，当我们添加推测时，需要将完成执行的过程与指令提交区分开来，这是因为指令执行完毕的时间可能远远早于它们做好提交准备的时间。在指令执行序列中添加这一提交阶段需要增加一组硬件缓冲区，用来保存已经完成执行但还没有提交的指令结果。这一硬件缓冲区称为**重排序缓冲区**，也可用于在可被推测的指令之间传送结果。

重排序缓冲区（ROB）像Tomasulo算法通过保留站扩展寄存器集一样，提供了附加寄存器。ROB会在一定时间内保存指令的结果，这段时间从完成该指令的相关运算算起，到该指令提交完毕为止。因此，ROB是指令的操作数来源，就像Tomasulo算法中的保留站提供操作数一样。两者之间的关键区别在于：在Tomasulo算法中，一旦一条指令写出其结果之后，任何后续发射的指令都会在寄存器堆中找到该结果。而在采用推测时，寄存器堆要等到指令提交之后才会更新（我们非常确定该指令会被执行）；因此，ROB是在指令执行完毕到指令提交这段时间内提供操作数。ROB类似于Tomasulo算法中的存储器缓冲区，为简单起见，我们将存储器缓冲区的功能集成到ROB中。

ROB中的每个项目都包含4个字段：指令类型、目的地字段、值字段和就绪字段。指令类型字段指定这个指令是分支（没有目的地结果）、存储指令（含有存储器地址目的地），还是寄存器操作（ALU运算或载入指令，它含有寄存器目的地）。目的地字段提供了应当向其中写入指令结果的寄存器编号（对于载入指令和ALU运算）或存储器地址（对于存储指令）。值字段用于在提交指令之前保存指令结果值。我们稍后将会看到ROB项目的一个例子。最后一个就绪字段指出指令已经完成执行，结果值准备就绪。

184

图3-5给出包含ROB的处理器的硬件结构。ROB包含存储缓冲区。存储指令仍然分两步执

行，但第二步是由指令提交来执行的。尽管保留站的重命名功能由 ROB 代替，但在发射运算之后仍然需要一个空间来缓冲它们（以及操作数），直到它们开始执行为止。这一功能仍然由保留站提供。由于每条指令在提交之前都在 ROB 拥有一个位置，所以我们使用 ROB 项目编号而不是保留站编号来标记结果。这种标记方式要求必须在保留站中跟踪为一条指令分配的 ROB。在本节后面，我们将研究一种替代实施方式，它使用额外的寄存器进行重命名，使用一个队列来替代 ROB，用于决定什么时候可以提交指令。

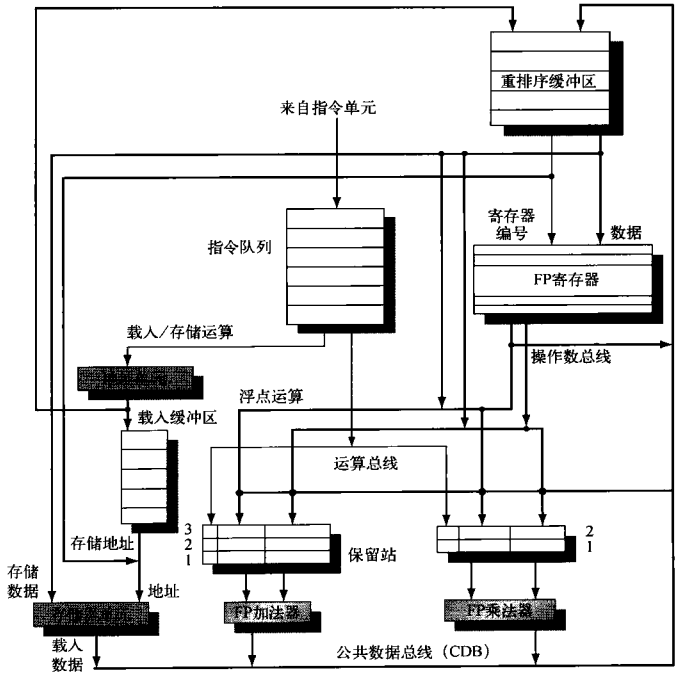


图 3-5 使用 Tomasulo 算法的 FP 单元的基本结构，为处理推测而进行了扩展。将此图与实施 Tomasulo 算法的图 3-4 对比，主要变化是添加了 ROB，去除了存储器缓冲区，后者的功能被集成到 ROB 中。如果拓宽 CDB，以允许每个时钟周期完成多条指令，则可以将这一机制扩展为支持多发射方案

在指令执行时涉及以下 4 个步骤。

(1) **发射**——从指令队列获得一条指令。如果存在空保留站而且 ROB 中有空插槽，则发射该指令；如果寄存器或 ROB 中已经含有这些操作数，则将其发送到保留站。更新控制项，指明这些缓冲区正在使用中。为该结果分配的 ROB 项目编号也被发送到保留站，以便在将结果放在 CDB 上时，可以使用这个编号来标记结果。如果所有保留站都被占满或者 ROB 被占满，则指令发射过程停顿，直到这两者都有可用项目为止。

(2) **执行**——如果还有一个或多个操作数不可用，则在等待计算寄存器的同时监视 CDB。这一步骤检查 RAW 冒险。当保留站中拥有这两个操作数时，执行该运算。指令在这一阶段可能占用多个时钟周期，载入操作在这一阶段仍然需要两个步骤。此时执行存储指令只是为了计算有效地址，所以在这一阶段只需要有基址寄存器可用即可。

(3) **写结果**——当结果可用时，将它写在 CDB 上（还有在发射指令时发送的 ROB 标签），并从 CDB 写到 ROB 并写到任何等待这一结果的保留站。将保留站标记为可用。对于存储指令需要执行一些特殊操作。如果要存储的值已经准备就绪，则将它写到 ROB 项目的 Value 字段，以备存储。如果要存储的值还不可用，CDB 必须进行监视，直到该数值被广播时再更新该存储指令 ROB 项目的 Value 字段。为简单起见，我们假定这一过程在存储操作的写结果阶段进行；稍后将会讨论如何放松这一要求。

(4) **提交**——这是完成指令的最后一个阶段，在此之后将仅留下它的结果。（一些处理器将这一提交阶段称为“完成”或“毕业”。）根据要提交的指令是预测错误的分支、存储指令，或是任意其他指令（正常提交），在提交时共有 3 种不同的操作序列。当一个指令到达 ROB 的头部而且其结果出现在缓冲区中，则进行正常提交；此时，处理器用结果更新其寄存器，并从 ROB 清除该指令。提交存储指令与正常提交类似，但更新的是存储器而不是结果寄存器。当预测错误的分支到达 ROB 的头部时，它指出推测是错误的。ROB 被刷新，执行过程从该分支的后续正常指令处重新开始。如果对该分支的预测正确，则该分支完成提交。

185 ~ 186

指令一旦提交完毕，它在 ROB 的相应项将被收回，寄存器或存储器目的地被更新，不再需要 ROB 项。如果 ROB 填满，那么只需要停止发射指令，直到有空闲项目为止。下面，我们研究一下这一机制如何处理前面为 Tomasulo 算法所举的示例。

例题 假定浮点功能单元的延迟与前面示例中相同：加法为 2 个时钟周期、乘法为 6 个时钟周期、除法为 12 个时钟周期。使用下面的代码段（也就是前面用于生成表 3-4 的代码段），写出当 MUL.D 做好提交准备时的状态表。

```
L.D      F6,32(R2)
L.D      F2,44(R3)
MUL.D    F0,F2,F4
SUB.D    F8,F2,F6
DIV.D    F10,F0,F6
ADD.D    F6,F8,F2
```

解答 表 3-7 用 3 个表给出了结果。注意，尽管 SUB.D 指令已经完成执行，但它不会在 MUL.D 提交之前提交。保留站和寄存器状态字段中的基本信息与 Tomasulo 算法中相同（见 3.5 节中关于这些字段的描述）。区别在于，Qj 和 Qk 字段以及寄存器状态字段中的保留站编号被 ROB 项目编号代替，我们已经将 Dest 字段加到保留站中。Dest 字段指定一个 ROB 项目，也就是这个保留站项目所生成结果的目的地。

表3-7 当MUL.D准备好提交时，尽管其他几个指令已经完成执行过程，但只有两个L.D指令已经提交

重排序缓冲区					
项目	繁忙	指令	状态	目的地	值
1	否	L.D F6,32(R2)	提交	F6	Mem[32 + Regs[R2]]
2	否	L.D F2,44(R3)	提交	F2	Mem[44 + Regs[R3]]
3	是	MUL.D F0,F2,F4	写结果	F0	#2 × Regs[F4]
4	是	SUB.D F8,F2,F6	写结果	F8	#2 - #1
5	是	DIV.D F10,F0,F6	执行	F10	
6	是	ADD.D F6,F8,F2	写结果	F6	#4 + #2

保留站							
名称	繁忙	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	否							
Load2	否							
Add1	否							
Add2	否							
Add3	否							
Mult1	否	MUL.D	Mem[44 + Regs[R3]]	Regs[F4]			#3	
Mult2	是	DIV.D		Mem[32 + Regs[R2]]	#3		#5	

FP寄存器状态										
字段	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
重排序#	3						6		4	5
繁忙	是	否	否	否	否	否	是	…	是	是

* MUL.D位于ROB的头部，此处两个L.D指令只是为了便于理解。尽管SUB.D和ADD.D指令的结果已经可用，而且可以用作其他指令的数据源，但它们在MUL.D指令提交之前不会提交。DIV.D正在执行过程中，但由于它的延迟要比MUL.D长，所有不会独自完成。“值”列表示所保存的值；#X格式表示ROB项目X的值字段。重排序缓冲区1和2实际上已经完成，但为了提供更多信息，也一并列在表中。我们没有给出载入/存储队列的项目，这些项目是按顺序保存的。

上面的例子说明了采用推测的处理器与采用动态调度的处理器之间的关键区别。对比表3-7与表3-4中的内容，后者显示的是同一代码序列在采用Tomasulo算法的处理器上的执行情况。关键区别在于：在上面的例子中，MUL.D是排在最前面的未完成指令，它之后的所有指令都不允许完成。而在表3-4中，SUB.D和ADD.D指令也已经完成。

这一区别意味着具有ROB的处理器可以在维持精确中断模式的同时动态执行代码。例如，如果MUL.D指令导致一个中断，我们只需要等待它到达ROB的头部并生成该中断，刷新ROB中的任意其他未完成指令。由于指令提交是按顺序进行的，所以这样会生成一个精确异常。

而在使用Tomasulo算法的例子中，SUB.D和ADD.D指令都可以在MUL.D激发异常之前完成。结果就是寄存器F8和F6（SUB.D和ADD.D指令的目的地）可能被改写，中断可能不准确。 187

一些用户和架构师认为不准确的浮点异常在高性能处理器中是可接受的，因为程序可能会终止；关于这一主题的深入讨论请参阅附录J。而其他类型的异常，比如页面错误，由于程序必须在处理此类异常之后透明地恢复执行，所以很难容忍这些异常出现不准确情况。

189 在循序提交指令时使用ROB，除了支持推测执行之外，还可以提供准确的异常，如下例所示。

例题 考虑前面Tomasulo算法使用的示例，表3-6显示了其执行情况：

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,#-8
        BNE      R1,R2,Loop     ;branches if R1≠R2
```

假定这个循环中所有指令已经发射了两次。还假定第一次迭代的L.D和MUL.D指令已经提交，并且所有其他指令都已经完成执行。正常情况下，存储指令将在ROB中等待有效地址操作数（本例中为R1）和值（本例中为F4）。由于我们只考虑浮点流水线，所以假定存储指令的有效地址在发射该指令时计算。

解答 表3-8用两个表给出了结果。

表3-8 尽管所有其他指令已经完成执行过程，但只有L.D和MUL.D指令已经提交。因此，没有保留站处于繁忙状态，所以图中没有示出

	重排序缓冲区					
项目	繁忙	指令		状态	目的地	值
1	否	L.D	F0,0(R1)	提交	F0	Mem[0+Regs[R1]]
2	否	MUL.D	F4,F0,F2	提交	F4	#1 × Regs[F2]
3	是	S.D	F4,0(R1)	写结果	0 + Regs[R1]	#2
4	是	DADDIU	R1,R1,#-8	写结果	R1	Regs[R1] - 8
5	是	BNE	R1,R2,Loop	写结果		
6	是	L.D	F0,0(R1)	写结果	F0	Mem[#4]
7	是	MUL.D	F4,F0,F2	写结果	F4	#6 × Regs[F2]
8	是	S.D	F4,0(R1)	写结果	0 + #4	#7
9	是	DADDIU	R1,R1,#-8	写结果	R1	#4 - 8
10	是	BNE	R1,R2,Loop	写结果		

	FP寄存器状态								
字段	F0	F1	F2	F3	F4	F5	F6	F7	F8
重排序#	6				7				
繁忙	是	否	否	否	是	否	否	...	否

* 剩下的指令将会尽可能快速地提交。前两个重排序缓冲区为空，但为了完整性也一并示出

189

由于在提交指令之前，寄存器值和存贮器值都没有实际写入，所以在发射分支预测错误时，处理器可以很轻松地撤销其推测操作。假定在表3-8中第一次没有选中分支BNE。当该分支之前的指令到达ROB的头部之后，直接提交即可；当分支到达缓冲区的头部时，将会清除缓冲区，处理器开始从其他路径提取指令。

在实践中，进行推测的处理器会在错误预测一个分支后尽早恢复。将预测错误的分支之后的所有ROB项目清空，使该分支之前的ROB项目继续执行，并在后续的正确分支处重新开始提取指令，从而完成恢复操作。在推测处理器中，由于错误预测的影响更大一些，所以性能对

分支预测也更敏感。因此，分支处理的各个方面（预测准确度、预测错误的检测延迟、预测错误的恢复时间）都变得更为重要。

在处理异常时，要等到做好提交准备时才会识别异常。如果推测的指令产生异常，则将异常记录在 ROB 中。如果出现分支预测错误，而且指令还没有执行，则在清除 ROB 时将异常连同指令一直刷新。如果指令到达 ROB 的头部，我们就知道它不再具有不确定性，应当激发该异常。我们还可以在异常出现之后、所有先前指令都已处理完毕的情况下立即处理异常，但异常要比分支预测错误的处理更难一些，而且由于异常的发生概率要更低一些，所以其重要性也要低一些。

表 3-9 给出了一条指令的执行步骤，以及为了继续执行这些步骤和要采取的动作而必须满足的条件。我们给出了到提交时才解决预测错误分支时的情景。尽管推测似乎只是对动态调度添加了非常简单的一点儿内容，但通过对比表 3-9 和表 3-5 中 Tomasulo 算法的相应内容，可能看出推测大大增加了控制复杂度。此外，还要记住分支预测也要更复杂一些。

在推测处理器处理存储指令时与 Tomasulo 算法中有一点非常重要的不同。在 Tomasulo 算法中，一条存储指令可以在到达“写结果”阶段（确保已经计算出有效地址）且待存储值可用时更新存储器。在推测处理器中，只有当存储指令到达 ROB 的头部时才能更新存储器。这一区别可以保证当指令不再具有不确定性时才会更新存储器。

表 3-9 大幅简化了存储指令，在实践中不需要这一简化。表 3-9 需要存储指令在写结果阶段等待寄存器源操作数，它的值就是要存储的内容；随后将这个值从该存储指令的保留站的 Vk 字段移到该存储指令 ROB 项目的“值”字段。但在现实中，待存储值只需要在提交存储指令之前到达即可，可以直接将源指令放到存储指令的 ROB 项目中。其实现方法为：用硬件来跟踪要存储的源值什么时候在该存储指令的 ROB 项目中准备就绪，并在每次完成指令时搜索 ROB，查看相关存储指令。 190

表 3-9 算法步骤及每一步骤需要满足的条件

状　态	等待条件	操作或记录工作

发射所有指令		`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/`
		`  {h ← RegisterStat[rs].Reorder;`
		`  if (ROB[h].Ready)/* 已完成指令 */`
		`  {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;}`
		`  else {RS[r].Qj ← h;} /* 等待指令 */`
		`} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;};`
		`RS[r].Busy ← yes; RS[r].Dest ← b;`
		`ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;`

浮点运算与存储	保留站(r)和 ROB(b)都可用	`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/`
		`  {h ←RegisterStat[rt].Reorder;`
		`  if (ROB[h].Ready)/* 已完成指令 */`
		`  {RS[r].Vk ←ROB[h].Value; RS[r].Qk ←0;}`
		`  else {RS[r].Qk ←h;} /* 等待指令 */`
		`} else {RS[r].Vk ←Regs[rt]; RS[r].Qk ←0;};`

浮点运算		`RegisterStat[rd].Reorder ←b; RegisterStat[rd].Busy ←yes;` `ROB[b].Dest ←rd;`
载入		`RS[r].A ← imm; RegisterStat[rt].Reorder ← b;` `RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;`
存储		`RS[r].A ← imm;`

（续）

状　态	等待条件	操作或记录工作
执行浮点运算	(RS[r].Qj == 0) 和 (RS[r].Qk == 0)	计算结果——操作数位于 Vj 和 Vk 中
载入步骤1	(RS[r].Qj == 0)，而且队列中没有更早的存储指令	RS[r].A ← RS[r].Vj + RS[r].A;
载入步骤2	载入步骤1完成，ROB 中所有先前存储指令都有不同地址	读取 Mem[RS[r].A]
存储	(RS[r].Qj == 0)且存储指令位于队列头部	ROB[h].Address ← RS[r].Vj + RS[r].A;
写所有非存储指令的结果	r 中的指令执行完毕，且 CDB 可用	b ← RS[r].Dest; RS[r].Busy ← no; ∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes;
存储	r 中的指令执行完毕，且(RS[r].Qk == 0)	ROB[h].Value ←RS[r].Vk;

提交	指令位于 ROB 头部（项目 h）且 ROB[h].ready ==yes	d ←ROB[h].Dest; /* 寄存器目的地，如果存在的话*/
		if (ROB[h].Instruction==Branch)
		{if (branch is mispredicted)
		{clear ROB[h], RegisterStat; fetch branch dest;};}
		else if (ROB[h].Instruction==Store)
		{Mem[ROB[h].Destination] ←ROB[h].Value;}
		else /* 将结果放在寄存器目的地 */
		{Regs[d] ←ROB[h].Value;};
		ROB[h].Busy ←no; /* 释放 ROB 项目*/
		/* 如果没有其他指令正在写目标寄存器，则释放该寄存器 */
		if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ←no;};

* 对于发射的指令，rd 为目的地、rs 和 rt 为源、r 为分配的保留站、b 是分配的 ROB 项目、h 是 ROB 的头项目。RS 是保留站数据结构。保留站返回的值被称为 result。RegisterStat 是寄存器数据结构，Regs 表示实际寄存器，ROB 是重排序缓冲区数据结构。

这一补充并不复杂，但添加之后有两个效果：需要向 ROB 中添加一个字段，表 3-9 尽管已经采用了小字体，但仍然会变得更长！尽管表 3-9 进行了简化，但在本示例中，我们将允许该存储指令跳过写结果阶段，只需要在准备提交前得到要保存的值即可。

和 Tomasulo 算法一样，我们必须避免存储器冒险。用推测可以消除存储器中的 WAW 和 WAR 冒险，这是因为存储器更新是循序进行的，当存储指令位于 ROB 头部时，先前不可能再有尚未完成的载入或存储指令。通过以下两点限制来解决存储器中的 RAW 冒险。

(1) 如果一条存储指令占用的活动 ROB 项目的“目的地”字段与一条载入指令的 A 字段取值匹配，则不允许该载入指令开始执行第二步骤。

(2) 在计算一条载入指令的有效地址时，保持相对于所有先前存储指令的程序顺序。

这两条限制条件共同保证了：对于任何一条载入指令，如果它要访问由先前存储指令写入的存储器位置，在这条存储指令写入该数据之前，该载入指令不能执行存储器访问。在发生此类 RAW 冒险时，一些推测处理器会直接将来自存储指令的值旁路给载入指令。另一种方法是采用值预测方式预测可能出现的冲突；我们将在 3.9 节考虑这一方法。

尽管这里对推测执行的解释主要是针对浮点运算的，但这些技术可以很容易地扩展到整

数寄存器和功能单元。事实上，推测在整数程序中可能更有用一些，因为这些程序中的代码可能更难预测一些。此外，只要允许在每个周期内发射和提交多条指令，就可以将这些技术扩展到能够在多发射处理器中工作。事实上，在这些处理器中，一些实用技术可能会在编译器的支持下在基本模块中开发足够的指令级并行，所以对这些处理器来说，推测技术可能是最有意义的。

3.7 以多发射和静态调度来开发 ILP

前面几节介绍的技术可以用来消除数据与控制停顿，使用 CPI 到达理想值 1。为了进一步提高性能，我们希望将 CPI 降低至小于 1，但如果每个时钟周期仅发射一条指令，那 CPI 是不可能降低到小于 1 的。 191 ~ 192

多发射处理器的目标（将在下面几节中讨论）就是允许在一个时钟周期中发射多条指令。多发射处理器主要有以下 3 类。

(1) 静态调度超标量处理器。

(2) VLIW（超长指令字）处理器。

(3) 动态调度超标量处理器。

两种超标量处理器每个时钟发射不同数目的指令，如果它们采用静态调度则采用循序执行，如果采用动态调度则采用乱序执行。

与之相对，VLIW 处理器每个时钟周期发射固定数目的指令，这些指令可以设置为两种格式之一：一种格式是一个长指令；另一种是一个固定的指令包，指令之间具有一定的并行度，由指令显式表示出来。VLIW 处理器由编译器进行静态调度。Intel 和 HP 在创建 IA-64 体系结构时（具体描述见附录 H），它们还将这种体系结构命名为 EPIC（显式并行指令计算机）。

尽管静态调度超标量处理器在每个周期内发射的指令数是可变的，而不是固定的，但它们在概念上实际与 VLIW 更接近一些，这是因为这两种方法都依靠编译器为处理器调度代码。由于静态调度超标量的收益会随着发射宽度的增长而逐渐减少，所以静态调度超标量主要用于发射宽度较窄的情况，通常仅有两条指令。超过这一宽度之后，大多数设计人员选择实现 VLIW 或动态调度超标量。由于两者的硬件要求和所需要的编译器技术是类似的，所以这一节将主要介绍 VLIW。深入理解这一节的内容之后，可以很轻松地将相关道理扩展到静态调度超标量。

表 3-10 总结了多发射的基本方法和它们的突出特征，并给出了使用每一方法的处理器。

基本 VLIW 方法

VLIW 使用多个独立功能单元。VLIW 没有尝试向这些单元发射多条独立指令，而是将多个操作包装在一个非常长的指令中，或者要求发射包中的指令满足同样的约束条件。由于这两种方法之间没有本质性的区别，所以假定将多个操作放在一条指令中，原始 VLIW 方法即是如此。

由于 VLIW 的收益会随着最大发射率的增长而增长，所以我们主要关注宽发射处理器。实际上，对于简单的两发射处理器，超标量的开销可能是最低的。许多设计人员可能会说：四-发射处理器的开销是可控的，但在本章后面将会看到，开销的增长是限制宽发射处理器的主要因素。 193

表 3-10 多发射处理器中使用的 5 种主要方法以及区分它们的主要特性

常见名称	发射结构	冒险检测	调度	突出特征	示例
超标量（静态）	动态	硬件	静态	循序执行	大多属于嵌入式领域：MIPS和ARM，包括ARM Cortex-A8
超标量（动态）	动态	硬件	动态	一些乱序执行，但没有推测	目前还没有
超标量（推测）	动态	硬件	带有推测的动态	具有推测的乱序执行	Intel Core i3、i5、i7，AMD Phenom，IBM Power 7
VLIW/LIW	静态	以软件为主	静态	所有冒险由编译器判断和指出（经常是隐式的）	大多数示例属于信号处理领域，比如TI C6x
EPIC	以静态为主	以软件为主	大多为静态	所有冒险由编译器隐式判断和指出	Itanium

* 本章主要讨论硬件操作密集的技术，它们都采用某种超标量形式。附录 H 主要介绍基于编译器的方法。EPIC 方法（在 IA-64 体系结构中实施）扩展了早期 VLIW 方法的主要概念，将静态与动态方法结合在一起。

我们考虑一个 VLIW 处理器，在上面运行包含 5 种运算的指令，这 5 种运算是：一个整数运算（也可以是一个分支）、两个浮点运算和两个存储器引用。这些指令可能拥有与每个功能单元相对应的一组字段，每个单元可能为 16 ~ 24 位，得到的指令长度介于 80 ~ 120 位之间。作为对比，Intel Itanium 1 和 2 的每个指令包中包含 6 个运算（也就是说，它们允许同时发射两个 3 指令包，如附录 H 所述）。

为使功能单元保持繁忙状态，代码序列必须具有足够的并行度，以填充可用操作插槽。这种并行是通过展开循环和调度单个更大型循环体中的代码而展现的。如果展开过程会生成直行代码，则可以使用**局部调度**技术，它可以对单个基本模块进行操作。如果并行的发现与开发需要在分支之间调度代码，那就必须使用可能更为复杂的**全局调度**算法。全局调度算法不仅在结构上更为复杂，而且由于在分支之间移动代码的成本很高，所以它们还必须进行非常复杂的优化权衡。

在附录 H 中，我们将讨论**跟踪调度**，它是专门为 VLIW 开发的全局调度技术之一；我们还将研究可以消除条件分支的特殊硬件支持，扩展了局部调度的用途，提高了全局调度的性能。

194

而现在，我们将依靠循环展开来生成一个长的直行代码序列，所以可以使用局部调度来构建 VLIW 指令，并集中研究这些处理器的运行情况。

例题 假定有一个 VLIW，它可以在每个时钟周期中发射两个存储器引用、两个浮点运算和整数运算或分支。写出针对这样一个处理器展开循环 `x[i] = x[i] + s`（3.2.1 节例题中的 MIPS 代码）的版本。可进行任意次展开，以消除所有停顿。忽略延迟分支。

解答 表 3-11 给出了这一代码。该循环被展开后，形成循环体的 7 个副本，消除了所有停顿（即，全空发射周期），运行 9 个时钟周期。这一代码的运行速度为 9 个周期生成 7 个结果，也就是每个结果需要 1.29 个周期，与 3.2 节使用非展开调度代码的两发射超标量相比，速度差不多是它的两倍。

表 3-11 占用内层循环并替代未展开序列的 VLIW 指令

存储器引用1	存储器引用2	浮点运算1	浮点运算2	整数运算/分支
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

* 在假定没有分支延迟的情况下，这一代码需要 9 个周期；通常，分支延迟也需要进行调度。其发射速率为 9 个时钟周期发射 23 个运算，或者说每个周期 2.5 个运算。其效率为大约 60%（也就是包含操作的可用插槽比例）。为了实现这一发射速率，需要更多寄存器，远远超过 MIPS 通常在处理这一循环时使用的寄存器数目。上面的 VLIW 代码序列需要至少 8 个浮点寄存器，而在基本 MIPS 处理器上，同一代码序列可以仅使用 2 个浮点寄存器，在使用未展开调度时，也只需要使用 5 个。

原始 VLIW 模块中既存在一些技术问题，也存在一些逻辑问题，从而降低了其效率。技术问题包括代码大小的增大和锁步（clockstep）操作的局限性。有两个不同因素共同造成 VLIW 代码大小的增大。第一，要在直行代码段中生成足够操作，需要大量展开循环（如前面的示例所示），从而增大了代码大小。第二，只要指令未被填满，那些没有用到的功能单元就会在指令编码时变为多余的位。在附录 H 中，我们研究软件调度方法，比如软件流水线，它们可以在没有明显增大代码规模的情况下获得循环展开的好处。 195

为了应对这种代码大小的增长，有时会使用智能编码。比如，一条指令中可能只有一个很大的立即数字段供所有功能单元使用。另一种技术是在主存储器中压缩指令，然后在到达缓存或进行译码时再展开它们。在附录 H 中，我们将介绍其他一些技术，并证明 IA-64 中存在的代码大幅扩展现象。

早期的 VLIW 是锁步工作的，根本就没有冒险检测硬件。在这种结构中，由于所有功能单元都必须保持同步，所以任意功能单元流水线中的停顿都必然导致整个处理器停顿。尽管一个编译器也许能够调度起决定作用的功能单元，以防止停顿，但要想预测哪些数据访问会遭遇缓存停顿，并对它们进行调度，那是非常困难的。因此，应当对缓存进行分块，并能导致所有功能单元停顿。由于发射速度和内存引用数目都变得很大，所以这一同步限制变得不可接受。在最近的处理器中，这些功能单元以更独立的方式工作，在发射时利用编译器来避免冒险，而在发射指令之后，可以通过硬件检测来进行非同步执行。

二进制代码兼容性也是 VLIW 的主要逻辑问题。在严格的 VLIW 方法中，代码序列既利用指令集定义，又要利用具体的流水线结构，包括功能单元及其延迟。因此，当功能单元数目和单元延迟不同时，就需要不同的代码版本。与超标量设计相比，由于这一要求而更难以在先后实施版本之间或者具有不同发射带宽的实施方式之间移植代码。当然，要想通过新的超标量设计而提高性能，可能需要重新编译。不过，能够运行旧版本二进制文件是超标量方法的一个实际优势。

EPIC 方法（IA-64 体系结构是它的一个主要示例）解决了早期 VLIW 设计中遇到的许多问题，包括扩展到更积极主动的软件推测与方法，在保证二进制兼容性的前提下克服硬件依赖的局限性。

所有多发射处理器都要面对的重要挑战是尝试开发大量 ILP。这种并行是通过展开浮点程序中的简单循环而实现的，而原来的循环很可能可以在向量处理器上高效地运行（向量处理器在下一章介绍）。对于这类应用程序，目前还不清楚多发射处理器是否优于向量处理器；其成本是类似的，向量处理器的速度可能与多发射处理器相同，或者还会更快一些。多发射处理器相对于向量处理器的潜在优势在于它们能够从结构化程度较低的代码中提取某些并行，以及能够很轻松地缓存所有形式的数据。因为这些原因，多发射方法已经成为利用指令级并行的主要方法，而向量主要作为这些处理器的扩展。

196

3.8 以动态调度、多发射和推测来开发 ILP

到目前为止，我们已经看到了动态调度、多发射和推测等各种机制是如何单独工作的。本节，我们将这三种技术结合在一起，得到一种非常类似于现代微处理器的微体系结构。为简单起见，我们只考虑每个时钟周期发射两条指令的发射速率，但其概念与每个时钟周期发射三条或更多条指令的现代处理器没有什么不同。

假定我们希望扩展 Tomasulo 算法，以支持具有分离整数、载入/存储和浮点单元（包括浮点乘和浮点加）的多发射超标量流水线，每个单元都可以在每个时钟周期启动一个操作。我们不希望向保留站乱序发射指令，这样可能会违犯程序语义。为了获得动态调度的全部好处，允许流水线在一个时钟周期内发射两条指令的任意组合，通过调度硬件向整数和浮点单元实际分配运算。由于整数指令和浮点指令的交互非常关键，所以还会扩展 Tomasulo 方案，以处理整数和浮点功能单元与寄存器，还能整合推测执行功能。如图 3-6 所示，其基本组织结构类似于每个时钟周期发射一条指令、具有推测功能的处理器的组织结构，不过必须改进其发射和完成逻辑，以允许每个时钟周期处理多条指令。

在动态调度处理器中（无论有无推测功能），每个时钟周期发射多条指令都非常复杂，原因很简单，这些指令之间可能存在相关性。因此，必须为这些并行指令更新控制表；否则，这些控制表中可能会出现错误，或者会丢失相关性。

在动态调度处理器中，已经采用两种方法在每个时钟周期内发射多条指令，这两种方法都基于这样一个事实：要在每个时钟周期中发射多条指令，其关键在于保留站的分配和流水线控制表的更新。一种方法是在一个时钟周期的一半时间内运行这一步骤，从而可以在一个时钟周期内运行 2 条指令；遗憾的是，很难将这一方法扩展为每个时钟周期处理 4 条指令。

第二种方法是构建必要的逻辑，一次处理两条或更多条指令，包括指令之间可能存在的相关性。可以在每个时钟周期发射四条或更多条指令的现代超标量处理器可能采用这两种方法：都采用流水线方式并拓宽了发射逻辑。一个重要的事实是：仅靠流水线无法解决这一问题。使指令发射占用多个时钟周期时，由于每个时钟周期都会发射新指令，所以必须能够分配保留站，并更新流水线表，使下一个时钟周期进行的相关指令发射能够利用更新后的信息。

在动态调度超标量中，这一发射步骤是最基本的瓶颈之一。为了说明这一过程的复杂性，表 3-12 给出了一种情景下的发射逻辑：在发射载入命令之后执行一个相关浮点运算。这个逻辑的基础是表 3-9，但它仅代表一种情景。在现代超标量中，对于所有可以在同一时钟周期内发射的相关指令的可能组合，都必须加以考虑。这种组合数与一个周期内可发射指令数的平方成正比，所以在尝试突破每时钟周期执行 4 条指令的速度时，发射步骤可能会成为一个瓶颈。

197

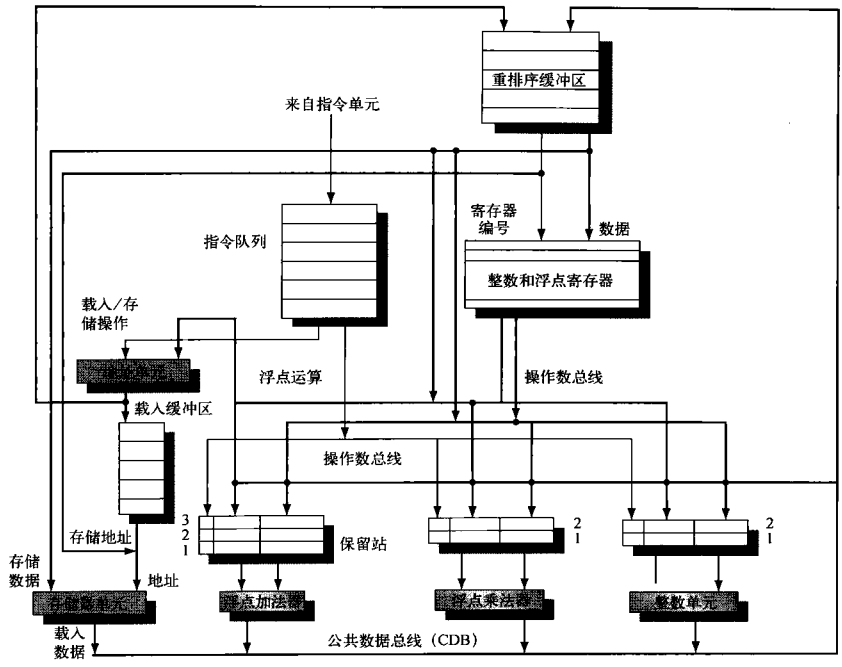


图 3-6 具有推测功能的多发射处理器的基本组成。在本例中，这种组成结构允许同时发射浮点乘法、浮点加法、整数运算和载入/存储指令（假定每个功能单元每个时钟周期发射一条指令）。注意，为了支持多发射，必须拓宽几条数据路径：CDB、操作数总线，还是非常关键的指令发射逻辑，本图中没有显示指令发射逻辑。正如正文中的讨论，最后一项是个难题

表 3-12 一对相关指令（称为指令 1 和指令 2）的发射步骤，其中指令 1 为浮点载入，指令 2 为浮点运算，它的第一个操作数是载入指令的结果，r1 和 r2 是为这些指令指定的保留站，b1 和 b2 是指定的重排序缓冲区项目

操作与记录	注　释

if (RegisterStat[rs1].Busy) /*正在执行的指令写 rs */
{h ← RegisterStat[rs1].Reorder;
if (ROB[h].Ready) /* 指令已完成*/
{RS[r1].Vj ← ROB[h].Value; RS[r1].Qj ← 0;}
else {RS[r1].Qj ← h;} /* 等待指令 */
} else {RS[r1].Vj ← Regs[rs]; RS[r1].Qj ← 0;};
RS[r1].Busy ← yes; RS[r1].Dest ← b1;
ROB[b1].Instruction ← Load; ROB[b1].Dest ← rd1;
ROB[b1].Ready ← no;
RS[r].A ← imm1; RegisterStat[rt1].Reorder ← b1;
RegisterStat[rt1].Busy ← yes; ROB[b1].Dest ← rt1;
	更新载入指令的保留表，载入指令只有一个源操作数。由于这是发射包中的第一条指令，所以看起来与载入指令的正常执行没有什么不同

（续）

操作与记录	注　释
RS[r2].Qj ← b1;} /* 等待载入指令 */	由于我们知道浮点运算的第一个操作数来自载入指令，所以这一步只是更新保留站，使其指向该载入操作。注意，在执行过程中必须分析这种相关，在发射步骤期间必须分配ROB项目，以正确地更新保留站

if (RegisterStat[rt2].Busy) /*正在执行的指令写 rt */
{h ← RegisterStat[rt2].Reorder;
if (ROB[h].Ready) /* 指令已完成*/
{RS[r2].Vk ← ROB[h].Value; RS[r2].Qk ← 0;}
else {RS[r2].Qk ← h;} /* 等待指令 */
} else {RS[r2].Vk ← Regs[rt2]; RS[r2].Qk ← 0;};
RegisterStat[rd2].Reorder ← b2;
RegisterStat[rd2].Busy ← yes;
ROB[b2].Dest ← rd2;
	由于假定浮点运算的第二个操作数来自前面的发射包，所以这一步骤看起来和单发射情景中一样。当然，如果这条指令依赖于同一发射包中的某些内容，那就需要使用指定的保留缓冲区更新这些表

RS[r2].Busy ← yes; RS[r2].Dest ← b2; ROB[b2].Instruction ← FP operation; ROB[b2].Dest ← rd2; ROB[b2].Ready ← no;	这一部分只是为浮点运算更新这些表，它与载入操作无关。当然，如果这个发射包中的其他指令依赖于浮点运算（比如在四发射超标量中的情景），所以这一指令会影响到对这些指令保留表的更新

* 对于发射指令，rd1 和 rd2 是目的地，rs1、rs2 和 rt2 是源（载入指令仅有一个源），r1 和 r2 是分配的保留站，b1 和 b2 是指定的 ROB 项目。RS 是保留站数据结构。RegisterStat 是寄存器数据结构，Regs 表示实际寄存器，ROB 是重排序缓冲区数据结构。注意，这一逻辑的正常运行需要指定重排序缓冲区项目，还有，别忘了所有这些更新是在单个时钟周期内并行完成的，不是顺序执行！

我们可以推广表 3-12 的细节，以描述在动态调度超标量中更新发射逻辑和保留表的基本策略，可以在一个时钟周期内发射多达 n 条指令，如下所示。

198

(1) 为可能在下一个发射包中发射的每条指令指定保留站和重排序缓冲区。这一指定过程可以在知道指令类型之前完成，只需要使用 n 个可用重排序缓冲区项依次为发射包中的指令预先分配重排序缓冲区项目，并确保有足够的保留站可用于发射整个包（无论包中包含多少指令）即可。通过限制一个给定类别的指令数目（比如，一个浮点运算、一个整数运算、一个载入指令、一个存储指令），就可以预告分配必要的保留站。如果没有足够的保留站可用，（比如，当程序中接下来的几条指令都是同一种指令类型时），则将这个包分解，仅根据原始程序顺序，发射其中一部分指令。包中的其余指令可以放在下一个发射包中。

199

(2) 分析发射包中指令之间的所有相关。

(3) 如果包中的一条指令依赖于包中的先前指令，则使用指定的重排序缓冲区编号来更新相关指令的保留表。否则，使用已有保留表和重排序缓冲区信息更新所发射指令的保留表项。

当然，由于所有这些都要在一个时钟周期中并行完成，所以使上述操作变得非常复杂。

在流水线的后端，必须能够在一个时钟周期内完成和提交多条指令。由于可以在同一时钟周期中实际提交的多条指令必须已经解决了相关性问题，所以这些步骤要比发射问题稍容易一些。后面将会看到，设计人员已经指出如何应对这一复杂性：在 3.13 节研究的 Intel i7 使用的方案基本上就是我们前面描述的推测多发射方案，包括大量保留站、重排序缓冲区、载入与存储缓冲区，后者也可用于处理非阻塞缓存缺失。

从性能的角度来看，我们可以用一个示例来说明这些概念是如何结合在一起的。

例题 考虑以下循环在两发射处理器上的执行情况，它会使整数数组的所有元素递增，一次没有推测，一次进行推测：

```
Loop:   LD      R2,0(R1)        ;R2=array element
        DADDIU  R2,R2,#1        ;increment R2
        SD      R2,0(R1)        ;store result
        DADDIU  R1,R1,#8        ;increment pointer
        BNE     R2,R3,LOOP      ;branch if not last element
```

假定有独立的整数功能单元用于有效地址计算、ALU运算和分支条件求值。给出这个循环在两种处理器上前3次迭代的控制表。假定可以在每个时钟周期内提交2条任意类型的指令。

解答 表3-13和表3-14给出了一个两发射动态调度处理器在有、无推测情况下的性能。在本例中，分支是一个关键的性能限制因素，推测会很有帮助。推测处理器中的第三分支在时间周期13中执行，而在非推测流水线中是在时钟周期19中执行。由于非推测流水线上的完成速率很快就会落在发射速率的后面，所以在再发射几个迭代之后，非推测流水线将会停顿。如果允许载入指令在决定分支之前完成有效地址计算，就可以提高非推测处理器的性能，但除非允许推测存储器访问，否则这一改进只会在每次迭代中获得一个时钟周期。 200

表3-13 在没有推测的情况下，双发射流水线版本中发射、执行和写结果的时机

迭代编号	指令		发射指令的时钟周期编号	执行指令的时钟周期编号	访问存储器的时钟周期编号	写CDB的时钟周期编号	注释
1	LD	R2,0(R1)	1	2	3	4	第一次发射
1	DADDIU	R2,R2,#1	1	5		6	等待LW
1	SD	R2,0(R1)	2	3	7		等待DADDIU
1	DADDIU	R1,R1,#8	2	3		4	直接执行
1	BNE	R2,R3,LOOP	3	7			等待DADDIU
2	LD	R2,0(R1)	4	8	9	10	等待BNE
2	DADDIU	R2,R2,#1	4	11		12	等待LW
2	SD	R2,0(R1)	5	9	13		等待DADDIU
2	DADDIU	R1,R1,#8	5	8		9	等待BNE
2	BNE	R2,R3,LOOP	6	13			等待DADDIU
3	LD	R2,0(R1)	7	14	15	16	等待BNE
3	DADDIU	R2,R2,#1	7	17		18	等待LW
3	SD	R2,0(R1)	8	15	19		等待DADDIU
3	DADDIU	R1,R1,#8	8	14		15	等待BNE
3	BNE	R2,R3,LOOP	9	19			等待DADDIU

* 注意，跟在BNE后面的LD不能提前开始执行，因为它必须等待分支结果的判断。这种类型的程序（带有不能提前解决的数据相关分支）展示了推测的威力。将用于地址计算、ALU运算和分支条件求值的功能单元分离开来，就可以在同一周期中执行多条指令。表3-14显示的是这个例子带有推测功能的版本。

表 3-14 在带有推测的情况下，双发射流水线版本中发射、执行和写结果的时机

迭代编号	指令	发射指令的时钟周期编号	执行指令的时钟周期编号	访问存储器的时钟周期编号	写CDB的时钟周期编号	提交指令的时钟周期编号	注释
1	LD R2,0(R1)	1	2	3	4	5	第一次发射
1	DADDIU R2,R2,#1	1	5		6	7	等待LW
1	SD R2,0(R1)	2	3			7	等待DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	循序提交
1	BNE R2,R3,LOOP	3	7			8	等待DADDIU
2	LD R2,0(R1)	4	5	6	7	9	没有执行延迟
2	DADDIU R2,R2,#1	4	8		9	10	等待LW
2	SD R2,0(R1)	5	6			10	等待DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	循序提交
2	BNE R2,R3,LOOP	6	10			11	等待DADDIU
3	LD R2,0(R1)	7	8	9	10	12	尽可能最早
3	DADDIU R2,R2,#1	7	11		12	13	等待LW
3	SD R2,0(R1)	8	9			13	等待DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	提前执行
3	BEN R2,R3, loop	9	13			14	等待DADDIU

* 注意，跟在BNE后面的LD可以提前开始执行，因为它是推测性的。

这个例子清楚地表明，推测方法在存在数据相关分支时可以带来一些好处，而在没有这种分支时会限制性能。但是，这种好处依赖于分支预测的准确性。错误预测不会提高性能，事实上，它通常会有损于性能，而且后面将会看到，它会极大地降低能耗效率。

201

3.9 用于指令传送和推测的高级技术

在高性能流水线中，特别是在多发射流水线中，仅仅很好地预测分支还不够；实际上还得能够提交高带宽的指令流。在最近的多发射处理器中，所谓高带宽发射流意味着每个时钟周期要提交4~8条指令。我们首先研究提高指令提交带宽的方法，然后再转而研究在实现高级推测技术中的一组关键问题，包括寄存器重命名的应用与重排序缓冲区、推测的积极性和一种称为值预测的技术，它尝试预测计算的结果，可以进一步增强ILP。

3.9.1 提高指令提取带宽

多发射处理器需要每个时钟周期提取的平均指令数目至少等于平均吞吐量。当然，提取这些指令需要有足够宽的路径能够连向指令缓存，但最重要的部分还是分支的处理。在本节，我们将研究两种处理分支的方法，然后讨论现在处理器如何将指令预测和预取功能结合在一起。

202

1. 分支目标缓冲区

为了减少这个简单的五级流水线以及更深流水线的分支代价，必须知道尚未译码的指令是不是分支，如果是分支，则需要知道下一个程序计数器（PC）应当是什么。如果这条指令是一个分支，而且知道下一个 PC 应当是什么，那就可以将分支代价降为零。分支预测缓存中存储着一条分支之后下一条指令的预测地址，这一缓存被称为**分支目标缓冲区**或**分支目标缓存**。图 3-7 给出了一个分支目标缓冲区。

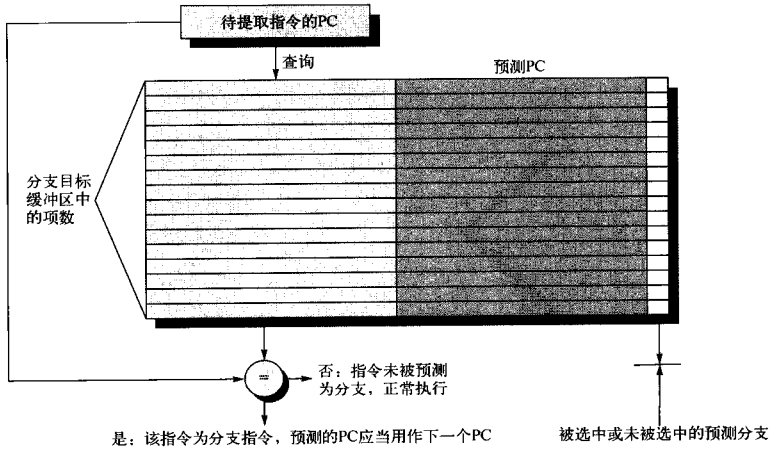


图 3-7　分支目标缓冲区。将所提取指令的 PC 与第一列中存储的一组指令地址进行匹配，这组地址代表的是已知分支的地址。如果 PC 与其中一项匹配，则所提取的指令为被选中的分支，第二个字段——预测的 PC 包含了对该分支之后下一个 PC 的预测值。会立即在该地址处开始提取指令。第三个字段是可选字段，可用于附加的预测状态位

由于分支目标缓冲区中预测下一条指令地址，并在对该指令译码之前把它发送出去，所以必须知道所提取的指令是否被预测为一条选中分支指令。如果所提供指令的 PC 与预测缓冲区中的一个地址匹配，则将相应的预测 PC 用作下一个 PC。这种分支目标缓冲区的硬件基本上与缓存硬件相同。 203

如果在分支目标缓冲区中找到一个匹配项，则立即在所预测的 PC 处开始提取指令。注意，与分支预测目标不同的是，由于在知道一条指令是否为分支之前就要将预测到的 PC 发送出去，所以预测项必须与这一指令匹配。如果处理器没有查看这一项是否与这个 PC 匹配，那么就会为不是分支的指令发送错误的 PC，导致性能恶化。我们只需要在分支目标缓冲区中存储预测选中的分支，这是因为未被选中的分支应当直接提取下一条顺序指令，就好像它不是分支指令一样。

图 3-8 显示了在为简单的五级流水线使用分支目标缓冲区时的步骤。从这个图中可以看出，如果在缓冲区中找到了分支预测项，而且预测正确，那就没有分支延迟。否则，至少存在两个时钟周期的代价。我们在重写缓冲区项目时通常会暂停指令提取，所以要处理错误预测与缺失是一个不小的难题。因此，我们希望快速完成这一过程，将代价降至最低。 204

为了评估一个分支目标缓冲区的工作情况，必须首先判断所有可能情景中的代价。表 3-15 给出了一个简单五级流水线的相关信息。

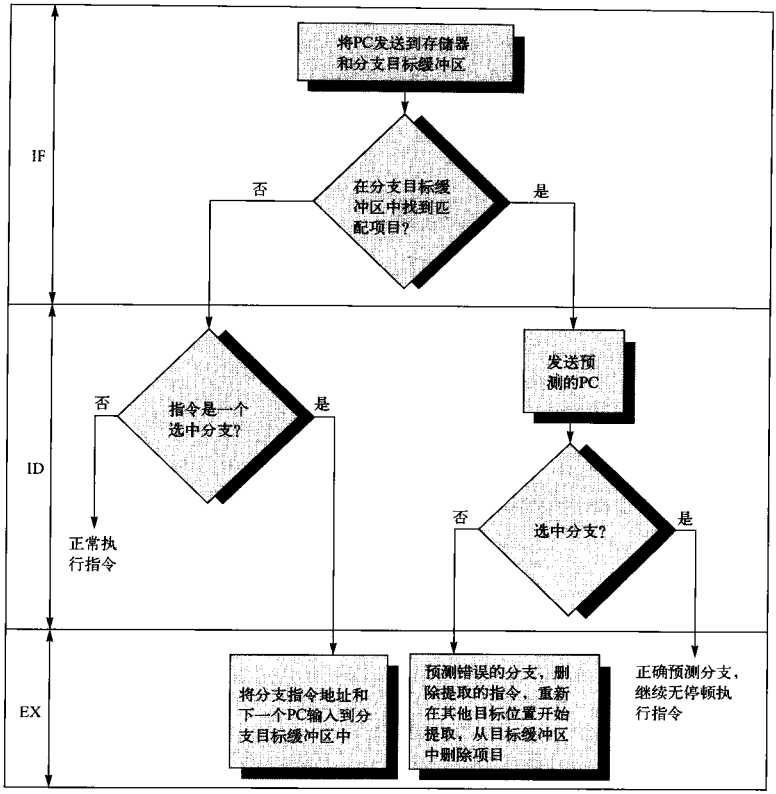


图 3-8 在使用分支目标缓冲区处理指令时涉及的步骤

表 3-15 一个分支是否在缓冲区中以及它实际完成何种任务，所有这些可能组合的代价，假定仅在缓冲区中存在选中分支

缓冲区中的指令	预　测	实际分支	代价周期数
是	选中	选中	0
是	选中	未选中	2
否		选中	2
否		未选中	0

* 如果一切都预测正确，而且在目标缓存中找到该分支，那就没有分支代价。如果分支预测错误，那代价就等于使用正常信息更新缓冲区的一个时钟周期（在此期间不能提取指令），在需要时，还有一个时钟周期用于重新开始为该分支提取下一个正确指令。如果这个分支没有找到，或未被选中，那代价就是两个周期，在此期间会更新缓冲区。

例题 假定各个错误预测的代价周期如表 3-15 所示，判断一个分支目标缓冲区的总体分支代价。关于预测准确率和命中率作以下假设：

- 预测准确率为 90%（对于缓冲区中的指令）；
- 缓冲区中的命中率为 90%（对于预测选中的分支）。

解答 通过研究两个事件的概率来计算代价，一个事件是预测分支将被选中但最后未被选中，另一个事件是分支被选中，但未在缓冲区中找到。这两个事件的代价都是两个周期。

概率（分支在缓冲区中，但未被选中）=缓冲区命中率 × 错误预测比例
=90% × 10% = 0.09

概率（分支不在缓冲区中，但被实际选中）=10%

分支代价=(0.09 + 0.10) × 2 ·

分支代价=0.38

这一代价略低于延迟分支的分支代价，我们在附录 C 中计算了后者，大约为每个分支 0.5 个时钟周期。记住，当流水线长度增加从而导致分支延迟增加时，通过动态分支预测得到的性能改善也会随之增加；此外，使用更准确的预测器也会获得更大的性能优势。现代高性能处理器的分支错误预测代价大约为 15 个时钟周期量级，显然，准确预测是非常关键的！

205

分支目标缓冲区的一种变体是存储一个或多个**目标指令**，用于作为预测**目标地址**的补充或替代。这一变体有两个潜在好处。第一，它允许分支目标缓冲区访问花费的时间长于连续两次指令提取之间的时间，从而可能允许采用更大型的分支目标缓冲区。第二，通过缓存实际目标指令可以让我们执行一种称为**分支折合**（branch folding）的优化方法。分支折合可用于实现 0 时钟周期的无条件分支，有时可以实现 0 时钟周期的条件分支。

考虑一个分支目标缓冲区，它缓冲来自预测路径的指令，可以用无条件分支的地址来访问。无条件分支的唯一作用就是改变 PC。因此，当分支目标缓冲区发出命中信号并指出该分支是无条件分支时，流水线只需要将分支目标缓冲区中的指令代替从缓存中的返回的指令（它是一个无条件分支）。如果处理器在每个周期发射多条指令，那么缓冲区需要提供多条指令，以获得最大好处。在一些情况下，有可能消除条件分支的成本。

2. 返回地址预测器

当我们试图提高推测的机会和准确性时，就面临着预测间接跳转的挑战，也就是说跳转到那些在运行时变化的目标地址。尽管高级语言程序会为间接过程调用、选择或 case 语句、FORTRAN 计算的 goto 语句等生成此类跳转，但大多数间接跳转都源于过程的返回操作。例如，对于 SPEC95 基准测试，过程返回操作平均占全部分支的 15%以上，占到间接跳转的绝大部分。对于诸如 C++ 和 Java 之类的面向对象的语言，过程返回操作甚至还要频繁一些。因此，将重点放在过程返回操作上似乎是恰当的。

尽管过程返回操作可以用分支目标缓冲区预测，但如果从多个地方调用这个进程，而且来自一个地方的多个调用在时间方面比较分散，那这种预测方法的准确性会很低。例如，在 SPEC CPU95 中，一个主动分支预测器对于此类返回分支所能达到的准确率不足 60%。为了解决这一问题，一些设计使用了一个小型的返回地址缓冲区，它的工作方式相当于一个栈。这种结构缓存最近的返回地址：在调用时将返回地址压入栈中，在返回时弹出一个地址。如果缓存足够大

（也就是与最大调用深度相等），它就能准确地预测过程返回操作。图 3-9 给出了这样的返回缓冲区在进行许多 SPEC CPU95 基准测试时的性能，缓冲区的元素数目为 0～16 个。在 3.10 节研究 ILP 时将使用一个类似的返回预测器。Intel Core 处理器和 AMD Phenom 处理器都有返回地址预测器。

206

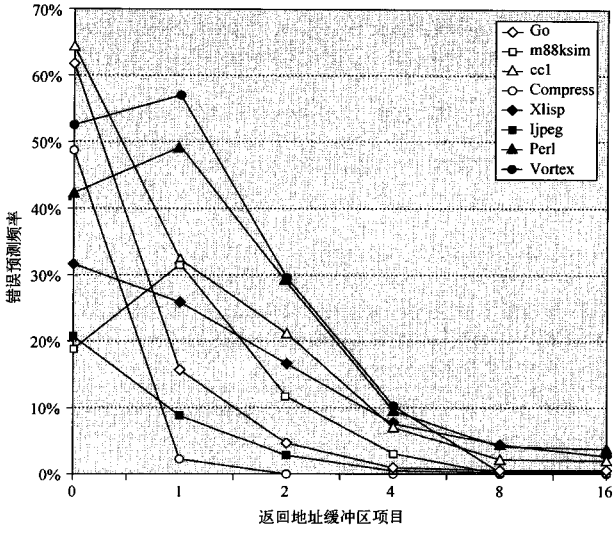


图 3-9 作为栈运行的返回地址缓冲区在进行大量 SPEC CPU95 基准测试时的预测准确率。这一准确率是正确预测返回地址所占的比例。若缓冲区中有 0 项，意味着使用标准分支预测。由于调用深度通常不是很大（当然也有一些例外），所以中等大小的缓冲区就可以取得很好的效果。这些数据来自 Skadron 等人[1999]，使用一种改进机制来防止缓存返回地址出现错误

3. 集成指令提取单元

为了满足多发射处理器的要求，近来的许多设计人员选择实现一个集成指令提取单元，作为独立的自主单元，为流水线的其余部分提供指令。实际上，这是因为他们意识到：由于多发射的复杂性，不能再将取指过程视为简单的单一流水级。

最近的设计已经开始使用集成了多种功能的集成指令提取单元，包括以下这些功能。

(1) 集成分支预测——分支预测器变为指令提取单元的组成部分，它持续预测分支，以驱动提取流水线。

207

(2) 指令预取——为了在每个时钟周期内提交多条指令，指令提取单元可能需要提前提取指令。这一单元自主管理指令的预取（见第 2 章对这一技术的讨论），把它与分支预测结合在一起。

(3) 指令存储器访问与缓存——在每个时钟周期提取多条指令时会遇到不同的复杂性，包括：提取多条指令可能需要访问多个缓存行，这是一个难题。指令提取单元封装了这一复杂性，尝试使用预取来隐藏跨缓存模块的成本。指令提取单元还可以提供缓存功能，大体充当一个按需单元，用于根据需要向发射级提供相应数量的指令。

几乎所有高端处理器现在都使用了一个独立的指令提取单元，通过一个包含未完成指令的缓冲区与流水线的其余部分连接在一起。

3.9.2 推测：实现问题与扩展

本节，我们将研究涉及推测设计权衡的 4 个问题，首先从寄存器重命名开始，这一方法经常被用来替代重排序缓冲区。然后讨论控制流推测的一个重要扩展：一种被称为**值推测**的思想。

1. 推测支持：寄存器重命名与重排序缓冲区

ROB（重排序缓冲区）的一种替代方法是明确使用更大的物理寄存器集，并与寄存器重命名方法结合在一起。这一方法以 Tomasulo 算法中使用的重命名为基础，并对其进行了扩展。在 Tomasulo 算法中，在执行过程的任意时刻，**体系结构可见的寄存器**（R0，…，R31 和 F0，…，F31）都包含在寄存器集和保留站的某一组合中。在添加了推测功能之后，寄存器值还会临时保存在 ROB 中。在任一情况下，如果处理器在一段时间内没有发射新指令，所有现有指令都会提交，寄存器值将出现在寄存器堆中，寄存器堆直接与在体系结构中可见的寄存器相对应。

在寄存器重命名方法中，使用物理寄存器的一个扩展集来保存体系结构可见寄存器和临时值。因此，扩展后的寄存器取代了 ROB 和保留站的大多数功能；只需要一个队列来确保循序完成指令。在指令发射期间，一种重命名过程会将体系结构寄存器的名称映射到扩展寄存器集中的物理寄存器编号，为目的地分配一个新的未使用寄存器。WAR 和 WAR 冒险通过目标寄存器的重命名来避免，在指令提交之前，保存指令目的地的物理寄存器不会成为体系结构寄存器，所以也解决了推测恢复问题。重命名映射是一种简单的数据结构，它提供当前与某个指定体系结构寄存器相对应的寄存器的物理寄存器编号，在 Tomasulo 算法中，这一功能由寄存器状态表完成。在提交指令时，重命名表被永久更新，用于指示一个物理寄存器与实际体系结构寄存器相对应，从而有效地完成对处理器状态的更新。尽管在采用寄存器重命名时并不需要 ROB，但硬件仍然必须在一个类似于队列的结构中跟踪信息，并严格按照顺序来更新重命名表。

208

与 ROB 方法相比，重命名方法的一个优点是简化了指令提交过程，它只需要两个简单操作：(1) 记录体系结构寄存器编号与物理寄存器编号之间的映射不再是推测结果；(2) 释放所有用于保存体系结构寄存器“旧”值的物理寄存器。在采用保留站的设计中，当使用一个保留站的指令完成执行后，该保留站会被释放，与一个 ROB 项目对应的指令提交之后，该 ROB 项目也被释放。

在采用寄存器重命名时，撤消寄存器分配的工作要更复杂一些，这是因为在释放物理寄存器之前，必须知道它不再与体系结构寄存器相对应，而且对该物理寄存器的所有使用都已完成。物理寄存器与体系结构寄存器相对应，直到该体系结构寄存器被改写为止，此时将使重命名表指向其他位置。也就是说，如果没有重命名项指向一个特定的物理寄存器，那它就不再对应于体系结构寄存器。但是，对该物理寄存器的使用可能仍未结束。处理器可以通过查看功能单元队列中所有指令的源寄存器说明符。如果一个给定物理寄存器没有显示为源寄存器，而且它也没有被指定为体系结构寄存器，那就可以收回该寄存器，重新进行分配。

或者，处理器也可以一直等待，直到对同一体系结构寄存器执行写入操作的另一指令提交为止。此时，对旧值的使用可能都已经完成了。尽管这种方法对物理寄存器的绑定时间可能要稍长于必要时间，但它的实现非常容易，在最近的超标量中得到了应用。

读者可能会问到的一个问题是：如果寄存器一直都在变化，那要如何知道哪些寄存器是体系结构寄存器呢？在程序运行的大多数时间内，这是无所谓的。当然，在某些情况下，另一个

进程（比如操作系统）必须知道特定的体系寄存器的内容到底在什么位置。为了理解如何提供这一功能，假定处理器在一段时间内没有发射指令。流水线中的所有指令最终都会提交，体系结构可见寄存器与物理寄存器之间的映射变得稳定。这时，物理寄存器的子集包含体系结构可见寄存器，任何未与体系结构寄存器关联的物理寄存器的都不再需要。从而可以很轻松地将体系结构寄存器移到物理寄存器的一个固定子集中，从而可以将这些值发送给另一进程。

209

寄存器重命名和重排序缓冲区都继续在高端处理器中使用，这些高端处理器现在能够同时运行40或50条指令（包括在缓存中等待的载入指令和存储指令）。无论是使用重命名还是重排序缓冲区，动态调度超标量的关键复杂性瓶颈仍然在于所发射的指令包中包含相关性的情景。具体来说，在发射一个发射包中的相关指令时，必须使用它们所依赖指令的指定虚拟寄存器。在采用寄存器重命名发射指令时，所部署的策略可以类似于采用重排序缓冲区（见3.8节）进行多发射时使用的策略，如下所示。

(1) 发射逻辑预先为整个发射包保留足够的物理寄存器（比如，当每个指令最多有一个寄存器结果时，为四指令包预留4个寄存器）。

(2) 发射逻辑判断包中存在什么样的相关。如果包中不存在相关，则使用寄存器重命名结构来判断哪个物理寄存器保存着（或将会保存）指令所依赖的结果。如果包中指令都不依赖于先前发射包中的结果，寄存器重命名表中将拥有正确的寄存器编号。

(3) 如果一条指令依赖于在该发射包中排在前列的某条指令，那么将使用在其中存放结果的预留物理寄存器来为发射指令更新信息。

注意，就像在重排序缓冲区中一样，发射逻辑必须在一个周期内判断包内相关性，并更新重命名表，而且和前面一样，当每个时钟处理大量指令时，这种做法的复杂度就会成为发射宽度中的一个主要限制。

2. 推测的代价

推测的重要优势之一是能够尽早发现那些本来会使流水线停顿的事件，比如缓存缺失。但是，这种潜在优势也带来一个潜在的重大不利条件。推测不是免费的，它需要时间和能量，错误预测的恢复过程还会进一步降低性能。此外，为了从推测中获益，需要支持更高的指令执行速率，为此，处理器必须拥有更多的资源，而这些资源又会占用硅面积、消耗功率。最后，如果推测导致异常事件的发生，比如缓存缺失或转变旁视缓冲区（TLB）缺失，而没有推测时本来不会发生这种事件，那推测造成重大性能损失的可能性就会增大。

为了在最大程度上保持优势、减少不利因素，大多数具有推测的流水线都仅允许以推测模式处理低成本的异常事件（比如，第一级缓存缺失）。如果发生成本高昂的异常事件，比如第二级缓存缺失或TLB缺失，处理器将会一直等待，等引发该事件的指令不再具有推测性时再来处理这一事件。尽管这样会使一些程序的性能稍有降低，但对于其他一些程序，尤其是会频繁出现此类事件且分支预测效果不佳的程序，可以避免性能大幅降低。

210

在20世纪90年代，推测的潜在不利因素还不是特别明显。随着处理器的发展，推测的实际成本变得越来越明显，宽发射和推测的局限性也变得更为突出。稍后我们会再次讨论这一主题。

3. 多分支预测

在本章已经研究过的示例中，在必须推测一个分支之前，已经有可能解决另一个分支。有三种情景可以通过同时推测多个分支获益：(1) 分支出现频率非常高；(2) 分支高度汇集；(3) 功能单元中的延迟很长。在前两种情况下，要实现高性能可能意味着对多个分支进行推

测，每个时钟周期甚至可能处理一条以上的指令。数据库程序和其他结构化程度较低的整数计算经常呈现这些特性，使多个分支的预测变得非常重要。同样，功能单元中的延迟很长时，也会增加对多个分支进行推测的重要性，用于避免因为流水线延迟过长而造成的停顿。

对多个分支进行推测会使推测恢复过程变得稍微复杂，但在其他方面比较简单。到 2011 年，还没有处理器能够将推测与每个时钟周期内处理多条分支完全结合在一起，从性能与复杂性、功率的对比来看，这样做的成本可能有些过高了。

4. 推测与能耗效率的挑战

推测对能耗效率有什么影响呢？乍看起来，有人可能会说推测的使用总是降低能耗效率，因为只要推测错误，就会以下面两种方式消耗更多的性能。

(1) 对某些指令进行了推测，但却不需要它们的结果，这些指令会为处理器生成多余工作，浪费能量。

(2) 撤销推测，恢复处理器的状态，以便在适当的地址处继续执行，这些操作都会多消耗一部分能量，在没有推测时是不需要消耗这部分能量的。

当然，推测的确会增大功率消耗，如果我们能够控制推测，那就有可能对成本进行测量（至少可以测量动态功率成本）。但是，如果推测过程缩短的执行时间多于它增加的平均功耗，那消耗的总能量仍然可能减少。

211

因此，为了了解推测对能耗效率的影响，我们需要研究推测生成非必要任务的频繁程度。如果会执行非常大量的非必要指令，那推测就不大可能大幅缩短运行时间！图 3-10 给出了由于错误预测而执行的指令比例。可以看到，在科技代码中，这一比例很小，而在整数代码中则很高（平均为大约 30%）。因此，对于整数应用程序来说，推测的能耗效率不会很高。设计人员可以避免推测，尝试减少错误预测，或者考虑采用新方法，比如仅对那些已经知道可预测性很强的分支进行推测。

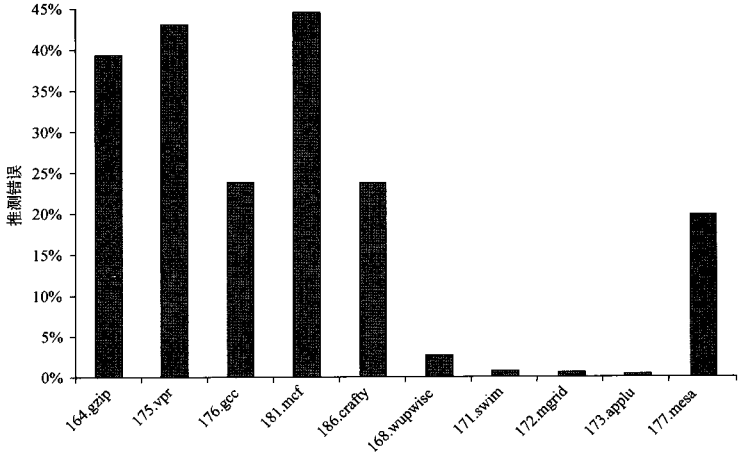


图 3-10 整数程序因为错误预测而执行的指令比例（前 5 个）通常要远大于浮点程序的这一比例（后 5 个）

5. 值预测

一种提高程序中可用 ILP 数目的技术是值预测。值预测尝试预测一条指令可能生成的值。显然，由于大多数指令在每次执行时都生成一个不同值（至少是一组取值中的一个不同值），所以值预测的成功率可能非常有限。但是，也有一些特定的指令，可以很轻松地预测它们的结果值，比如从常数池中进行载入的载入指令，或者载入一个不经常变化的取值。此外，如果一条指令只是从一组很少的取值中选择一个，那就有可能结合其他程序行为来预测结果值。

212

如果值预测能够显著提高可用 ILP 的数量，那它就是有用的。当某个值被用作一串相关运算的源数据时（比如一个载入操作），这种可能性就很大。因为值预测是用来提高推测能力的，而且错误预测会有不利的性能影响，所以预测的准确性非常关键。

尽管在过去十年里，许多研究人员都致力于值预测的研究，但其成果一直缺乏足够的吸引力，未能在实际处理器中得到应用。不过有一种比较简单的与值预测相关的较早思想已经得到了应用，那就是地址别名预测。地址别名预测是一种非常简单的技术，用来预测两个存储指令或者一个载入指令与一个存储指令是否引用同一存储器地址。如果这样两条指令没有引用同一地址，那就可以放心地交换它们的顺序。否则，就必须等待，直到知道这些指令访问的存储器地址为止。因为我们不需要实际预测地址值，只需要知道这些值是否冲突即可，所以这种预测更稳定，也更简单。这种有所限制的地址值预测形式已经在几种处理器中得到应用，可能会在将来普遍采用。

3.10 ILP 局限性的研究

在 20 世纪 60 年代出现第一批流水化处理时，就开始通过开发 ILP 来提高性能。在 20 世纪 80 年代和 90 年代，这些技术成为快速提高性能的关键。如果我们希望性能提升速度能够长期快于基础集成电路的增长速度，有一个问题是非常关键的：存在多少 ILP。要开发更多 ILP 都需要些什么？在短期范围内，这一关键问题对计算机设计人员和编译器编写人员都非常关键。这一节的数据还为我们提供了一种方法，用来验证本章所讨论思想的价值，这些思想包括存储器消歧、寄存器重命名和推测。

在这一节，我们将回顾围绕这些问题所做的一部分研究（基于 Wall 在 1993 年所做的研究）。所有这些针对可用并行进行的研究都会作出一组假定，然后求出在这些假定条件下有多少并行可用。这里给出的数据是由一项假设条件最少的研究得出的；事实上，最终的硬件模型可能是无法实现的。不过，所有这些研究都假定采用一种特定级别的编译器技术，尽管这些研究使用了过于强劲的硬件，但其中一些假定还是可能会影响到结果。

我们将会看到，对于一些成本合理的硬件模型来说，过多采用推测的成本可能会得不偿失：在功率与硅面积使用方面的效率过于低下。尽管许多研究团体和主流处理器制造商都认为大量开发 ILP 是有利的，最初不愿意接受这种可能性，但到了 2005 年，他们不得不改变自己的观点。

213

3.10.1 硬件模型

为了了解 ILP 可能有哪些局限性，我们首先需要定义一种理想处理器。在理想处理器中，去除了对 ILP 的所有约束条件。在这样一个处理器中，对 ILP 仅有的一些限制是由通过寄存器或存储器的实际数据流带来的。

对理想或完美处理器作出以下假设。

(1) **无限寄存器重命名**——有无限个虚拟寄存器可供使用，因此避免了所有 WAW 和 WAR 冒险，可以有无数条指令同时执行。

(2) **完美分支预测**——分支预测非常完美。所有条件分支都被准确预测。

(3) **完美跳转预测**——所有跳转（包括用于返回和计算跳转的跳转寄存器）都被完美预测。与完美分支预测相结合，这相当于拥有了一种处理器，它可以进行完美预测，并拥有一个极大的可执行指令缓冲区。

(4) **完美存储器地址别名分析**——所有存储器地址都已确切知道，如果载入指令与存储指令的地址不同，可以将载入指令移到存储指令之前。注意，这就实现了完美的地址别名分析。

(5) **完美缓存**——所有存储器访问都占用一个时钟周期。在实践中，超标量处理器通常会使用大量 ILP，隐藏了缓存缺失，使这些结果变得非常乐观。

假设(2)与假设(3)消除了所有控制相关。而假设(1)和假设(4)消除了除真数据相关之外的所有数据相关。这 4 条假设结合在起来，就意味着：对于程序执行中的任何一条指令，在它所依赖的先前指令执行完毕之后，可以将该指令调度到紧随其后的时钟周期上。根据这些假设，甚至有可能将程序中最后一条动态执行的指令调度到最前面的时钟周期上！因此，这一组假设同时包含了控制推测与地址推测，在实现时把它们当作是完美的。

我们首先研究一个可以同时发射无数条指令的处理器，能够提前看到运算过程中的任意指令。在我们研究的所有处理器型号中，对于一个时钟周期能够执行哪些类型的指令没有限制。在可以发射无限条指令时，这意味着在一个时钟周期中可能有无数条载入指令或存储指令。此外，所有功能单元的延迟都假定为一个时钟周期，所以任何相关指令序列都可以在连续周期上发射。如果延迟长于一个周期，尽管不会降低任意时刻正在执行的指令数目，但可能会降低每个周期发射的指令数目。（任意时刻正在执行的指令通常被称为是在 in flight。）

当然，这种理想处理器很可能是无法实现的。例如，IBM Power7（见 Wendell 等人[2010]）是目前所发布的最高级超标量处理器。Power7 一个时钟周期最多可以发射 6 条指令，最多在 12 个执行单元的 8 个单元上开始执行（其中只有两个是载入/存储单元），支持大型重命名寄存器集（允许数百条指令同时执行），使用大型主动分支预测器，采用动态存储器消歧。Power7 继续向前发展，使用更多的线程级并行，增大所支持的同时多线程（SMT）数目（达到每个核心 4 个线程），将每个芯片的核心数目增大到 8 个。在研究了完美处理器的可用并行之后，我们将研究在近期可能设计出的处理器中能够实现什么功能。 214

为了测量可用并行，可以使用标准 MIPS 优化编译器来编译和优化一组程序。对这些程序交付运行，生成一个指令与数据引用踪迹。然后尽早调度踪迹中的所有指令，仅受数据相关的限制。由于使用了踪迹，所以很容易实现完美分支预测和完美别名分析。利用这些机制，可以大大提前对这些指令的调度，在没有数据相关的大量指令之间进行移动，由于可以完美预测分支，所以这些指令也包括分支指令在内。

图 3-11 显示了 6 个 SPEC92 基准测试的平均可用并行度数目。本节中，始终以平均指令发射率为指标来测试并行。注意，所有指令的延迟为一个时钟周期；延迟较长时会减少每个时钟的平均指令数。这些基准测试中有 3 个（fpppp、doduc 和 tomcatv）是浮点操作密集的基准测试，另外 3 个为整数程序。浮点基准测试中有两个（fpppp 和 tomcatv）有大量并行，可供向量计算机或多处理器开发（不过，由于已经对代码进行了一些人工转换，所以 fpppp 中的结构十分杂乱）。doduc 程序拥有大量并行，但这些并行不会在简单的并行循环中出现，这一点与 fpppp 和 tomcatv 中不同。程序 li 是一个拥有许多短相关的 LISP 解释程序。

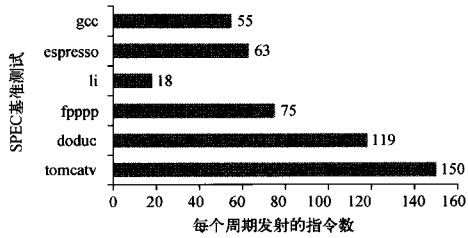


图 3-11 一个完美处理器中运行 6 个 SPEC92 基准测试时的可用 ILP。前 3 个程序是整数程序，后 3 个是浮点程序。浮点程序中循环数量很多，有大量循环级并行

215

3.10.2 可实现处理器上ILP的局限性

本节中，我们将研究一些处理器的性能，这些处理器的硬件支持水平非常强大，不低于 2011 年所能达到的水平，或者是考虑到过去十年发生的事件与教训，可能在近期达到的水平。具体来说，我们将假定以下固定属性。

(1) 每个时钟周期最多发射 64 条指令、没有发射限制，或者是 2011 年最宽处理器总发射宽度的 10 倍以上。后面将会讨论，超大发射宽度对时钟频率、逻辑复杂度和功率产生的实际影响可能才是对 ILP 开发的最重要限制。

(2) 一个竞赛预测器，拥有 1000 项和一个 16 项返回预测器。这个预测器可以与 2011 年的最佳预测器相媲美；这个预测器不是主要瓶颈。

(3) 对于动态完成的存储器引用能够完全消除歧义——这项要求是比较高的，但当窗口较小时（因此，发射速度和载入/存储缓冲区也较小），或者通过地址别名预测，还是有可能做到的。

(4) 具有 64 个附加整数寄存器和 64 个浮点寄存器的寄存器重命名，这一数目要略小于 2011 年的最强劲处理器。Intel Core i7 的重排序缓冲区中有 128 项，不过它们没有划分为整数与浮点寄存器，而 IBM Power7 差不多有 200 个寄存器。注意，我们假定流水线延迟为 1 个周期，它显著降低了对重排序缓冲区项的需要。Power7 和 i7 延迟都不低于 10 个时钟周期。

图 3-12 给出了这一配置在窗口大小变化时的结果。这一配置比任何已有实现方式都要复杂和昂贵，特别是在指令发射数方面，它要比 2011 年任意处理器上的最大可用发射数大 10 倍以上。不过，它对未来实施方式所能生成的内容给出了一个非常有用的范围。由于另一个原因，这些图形中的数据可能是非常乐观的。在 64 条指令中没有发射限制：它们可能都是存储器引用。在不远的将来，甚至没有人会在处理器中设计这一功能。遗憾的是，用合理的发射限制来界定处理器的性能是十分困难的，这不仅是因为存在各种各样的可能性，而且发射限制的存在需要用准确的指令调度器来评估并行，在研究具有大量发射指令的处理器时，其成本会非常昂贵。

另外还请记住，在解读这些结果时，没有考虑缓存缺失和长于 1 个时钟周期的延迟，这两个因素都可能产生严重影响！

216

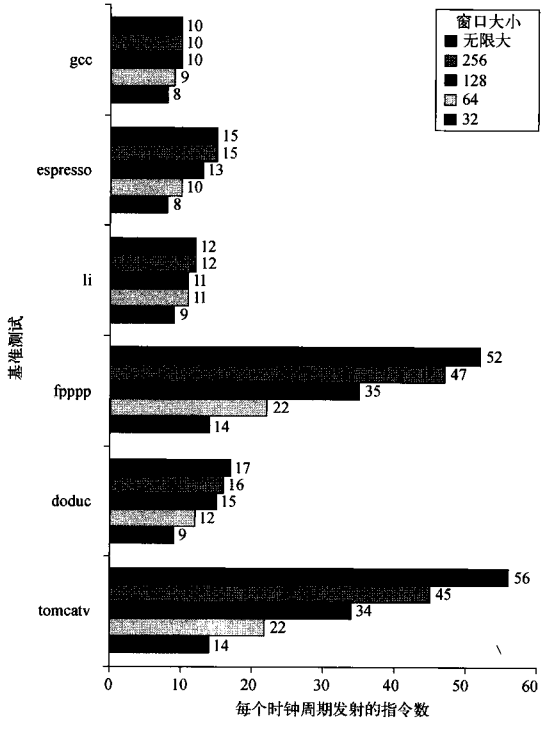


图 3-12 对于各种整数与浮点程序，每个时钟周期发射 64 个任意指令时，可用并行数随窗口大小的变化情况。尽管重命名寄存器的数目少于窗口大小，但所有操作的延迟为一个时钟周期、重命名寄存器的数目等于发射宽度，这一事实使处理器能够在整个窗口中开发并行。在实际实现中，必须平衡窗口大小和重命名寄存器的数目，以防止这些因素中的某一个过度限制发射速率

图 3-12 中最令人吃惊的观测结果是：考虑到以下所列的实际处理器约束条件，窗口大小对整数程序的影响不像对浮点程序那么严重。这一结果指向了这两种程序之间的关键区别。两个浮点程序中能够利用循环级并行，这意味着可以开发的 ILP 数目较高，而对于整数程序，其他因素（比如分支预测、寄存器重命名、并行较少，等等）都是重要的限制情况。从万维网和云计算于 20 世纪 90 年代中期开始爆发式发展以来，人们增加了对整数性能的重视，所以这一观察结果非常关键。事实上，在过去 10 年里，大多数市场增长（事务处理、Web 服务器等）都依赖于整数性能，而不是浮点性能。在下一节将会看到，对于 2011 年的真实处理器，实际性能要远低于图 3-12 显示的数据。 217

由于在实际硬件设计中，提高指令速率具有一定的难度，所以设计人员面临着一项挑战：决定如何更好地利用集成电路上有限的可用资源。是采用具有更大缓存和更高时钟速率的较简单处理器，还是将重点放在具有较慢时钟和较小缓存的指令级并行上。这是最重要的设计权衡

之一。下面的例子演示了这些挑战，在第 4 章，我们将会看到一种以 GPU 形式开发细粒度并行的替代方法。

例题　考虑以下 3 种假设的非典型处理器，我们将在上面运行 SPEC gcc 基准测试。

(1) 一个简单的 MIPS 双发射静态流水线，其时钟速率为 4 GHz，所实现的流水线 CPI 为 0.8。这一处理器的缓存系统每条指令发生 0.005 次缺失。

(2) 一个双发射 MIPS 处理器的深度流水线版本，缓存稍小一些，时钟速率为 5 GHz。处理器的流水线 CPI 为 1.0，缓存较小，平均每条指令生成 0.0055 次缺失。

(3) 一个推测超标量处理器，具有一个 64 项窗口。它能实现的发射率为这一窗口大小理想发射率的一半。（使用图 3-12 中的数据）这一处理器的缓存最小，每条指令产生 0.01 次缺失，但通过动态调度可以隐藏每次缺失 25%的缺失代价。这个处理器的时钟为 2.5 GHz。

假定主存储器时间（这一时间决定了缺失代价）为 50 ns。判断这 3 种处理器的相对性能。

解答　首先，我们使用缺失代价和缺失率信息计算每一种配置中缓存缺失对 CPI 的影响。计算公式如下：

$$\text{缓存 CPI}=\text{每条指令的缺失数}\times\text{缺失代价}$$

我们需要为每个系统计算缺失代价：

$$\text{缺失代价}=\frac{\text{存储器访问时间}}{\text{时钟周期}}$$

218

这 3 种处理器的时钟周期时间分别为 250 ps、200 ps 和 400 ps。因此，缺失代价是：

$$\text{缺失代价}_1=\frac{50\ \text{ns}}{250\ \text{ps}}=200\text{周期}$$

$$\text{缺失代价}_2=\frac{50\ \text{ns}}{200\ \text{ps}}=250\text{周期}$$

$$\text{缺失代价}_3=\frac{0.75\times 50\ \text{ns}}{400\ \text{ps}}=94\text{周期}$$

为每一缓存应用此公式：

$$\text{缓存 CPI}_1=0.005\times 200=1.0$$

$$\text{缓存 CPI}_2=0.0055\times 250=1.4$$

$$\text{缓存 CPI}_3=0.01\times 94=0.94$$

除了处理器 3 之外，我们知道了其他处理器的流水线 CPI 影响；它的流水线 CPI 给出如下：

$$\text{流水线 CPI}_3=\frac{1}{\text{发射速率}}=\frac{1}{9\times 0.5}=\frac{1}{4.5}=0.22$$

现在可以通过添加流水线和缓存 CPI 因素来求出每个处理器的 CPI：

$$\text{CPI}_1=0.8+1.0=1.8$$

$$\text{CPI}_2=1.0+1.4=2.4$$

$$\text{CPI}_3=0.22+0.94=1.16$$

由于这是同一体系结构，所以可以对比指令执行速度（单位为百万条指令秒 MIPS），

以确定相对性能：

$$指令执行速度=\frac{CR}{CPI}$$

$$指令执行速度_1=\frac{4000MHz}{1.8}=2222\ MIPS$$

$$指令执行速度_2=\frac{5000MHz}{2.4}=2083\ MIPS$$

$$指令执行速度_3=\frac{2500MHz}{1.16}=2155\ MIPS$$

在这个例子中，简单的双发射静态超标量看起来是最好的。在实践中，性能取决于CPI和时钟频率两项假设。

3.10.3 超越本研究的局限

和所有极限研究一样，我们在本节分析的研究内容也有其自己的局限性。我们将这些局限性分为两类：即使在完美推测处理器中也会存在的局限性；在一或多种现实模型中存在的局限性。当然，第一类中的所有局限性也适用于第二类。适用于完美模型的最重要局限性包括以下几个。 219

(1) **访问存储器的WAW和WAR冒险**——这一研究通过寄存器重命名消除了WAW和WAR冒险，但却没有消除存储器使用中的冒险。尽管乍看起来，此类情况很少会出现（特别是WAW冒险），但它们的确会因为栈帧分配而出现。某种被调用过程重复利用栈中上一过程使用的存储器位置，这样可能会导致WAW和WAR冒险，造成不必要的限制。Austin和Sohi在1992年研究了这一问题。

(2) **不必要的相关**——在有无数个寄存器时，就可以消除真寄存器数据相关之外的所有其他数据相关。但是，由于递归或代码生成约定而造成的相关仍然会引入一些不必要的真数据相关。其中一个例子就是在一个简单的 for 循环中对于控制变量的相关。由于控制变量在每次循环迭代时都会递增，所以循环中包含至少一个相关。在附录H中可以看到，通过展开循环和进行积极的代数优化可以消除此类相关计算。Wall的研究中包含了数量有限的一些此类优化，但更积极地应用这些优化方式有可能增加ILP的数目。此外，特定的代码生成约定也会引入一些不必要的相关，特别是在使用返回地址窗口和栈指针寄存器时（它会在调用/返回序列中递增和递减）。Wall消除了返回地址寄存器的影响，但在链接约定中使用规模指针可能会导致“不必要的”相关。Postiff等人[1999]研究了消除这一约束条件的益处。

(3) **克服数据流限制**——如果值预测的精度很高，那就可能克服数据流限制。但到目前为止，在关于这一主题的 100 多篇论文中，还没有一篇能够使用一种现实预测方案来显著提高ILP。显然，完美的数据值预测可以得到高效的无限并行，因为每个指令的每个值都可能提前预测得出。

对于不够完美的处理器，已经提出了几种可以发现更多 ILP 的思想。其中一个例子是沿多条路径进行推测。Lam和Wilson [1992]讨论了这一思想，在本节介绍的研究内容中进行了探讨。通过在多条路径上进行推测，可以降低错误恢复的成本、发现更多的并行。由于所需硬件资源呈指数增长，所以只有对有限个分支评估这一方案才有意义。Wall [1993]提供了用于在最多 8 路分支上进行双向推测所需要的数据。考虑到双向推测的成本，而且已经知道其

余一个分支方向会被丢弃（在多条路径上执行这样一个过程会增加无数计算的数量），因此所有商用设计都没有在双向路径上进行太多投入，而将更多的硬件用于在正确路径上进行更准确的推测。

220

这一节介绍的所有限制都不是根本性的限制，因为克服它们并不需要改变什么物理定律，理解这一点非常重要。它们只是在实践中遇到的一些限制，意味着在开发更多 ILP 的过程中存在一些难以逾越的障碍。这些限制（无论是窗口大小、别名检测，还是分支预测）都是设计人员和研究人员要应对的挑战。

在 21 世纪的前 5 年中，为突破这些限制所做的许多尝试都遭到了失败。一些技术取得了一点点改进，但通常会大幅提高复杂度、增加时钟周期、使功率的增加不合比例。总而言之，设计人员发现，在尝试开发更多 ILP 时，其效率非常低下。在本章的结语部分会再次讨论这一主题。

3.11 交叉问题：ILP 方法与存储器系统

3.11.1 硬件推测与软件推测

本章介绍的这些硬件密集的推测方法与附录 H 中的软件方法为开发 ILP 提供了不同选择。下面列出这些方法的一些权衡与局限性。

- 为了能够大范围进行推测，必须能够消除存储器引用的歧义。对于包含指针的整数程序，很难在编译时实现这一功能。在基于硬件的方案中，存储器地址的动态运行时消歧是使用前面介绍的 Tomasulo 算法的技术完成的。这一消歧功能可以在运行时把载入指令移到存储指令之后。对推测存储器引用的支持可以帮助克服编译器的保守性，但如果使用这些方法时不够仔细，那恢复机制的开销可能会大于它们所能带来的收益。
- 当控制流不可预测时，当基于硬件的分支预测优于在编译时完成的软件分支预测时，基于硬件的推测效果较佳。这些特性对于许多整数程序都是成立的。例如，一个好的静态预测器，对 4 个主要整数 SPEC92 程序的错误预测率大约为 16%，而硬件预测器的错误预测率低于 10%。因为当预测不正确时，推测错误的指令可能会拖慢计算速度，所以这一差别非常明显。因为这一差别而导致了一个结果：即使是静态调度的处理器中通常也会包含动态分支预测器。
- 221 即使对于被推测的指令，基于硬件的推测也能保持完全精确的异常模型。最近的基于软件的方法也添加了这一特殊支持，同样可以做到这一点。
- 基于硬件的推测不需要补充或记录代码，而那些雄心勃勃的软件推测机制则需要这一条件。
- 基于编译器的方法能够深入了解代码序列，从中获益，从而在代码调度方面要优于纯硬件驱动的方法。
- 对于一种体系结构的不同实现方式，采用动态调度的硬件推测不需要采用不同代码序列就能实现好的性能。尽管这一收益很难量化，但从长期来看，这一收益可能是最重要的。有意思的是，它曾经是设计 IBM 360/91 的动机之一。另一方面，最近的显式并行体系结构（比如 IA-64）已经增加一定的灵活性，可以减少代码序列中固有的硬件相关。

在硬件中支持推测的主要缺点是需要更多、更复杂的硬件资源。必须针对两个方面对这一硬件成本进行评估，一是与软件方法中编译器的复杂性相对比，一是与依赖此类编译器的处理器简化程度相对比。

一些设计人员已经尝试将动态方法和基于编译器的方法结合起来，以期达到两种方法的最佳效果。这种组合方式可以产生一些有趣但不够明确的交互。例如，如果将条件移动与寄存器重命名结合起来，就会出现一种微妙的副作用。由于之前已经在指令流水线中更改了目标寄存器的名称，所以一个被撤消的条件移动操作仍然会向目标寄存器中复制一个值。这种微妙的交互使设计与验证过程都变得非常复杂，还可能会降低性能。

迄今为止，在采用软件方法支持 ILP 和推测的计算机中，Intel Itanium 处理器是最强大的。它没有像设计人员所希望的那样提供硬件支持，对于通用、非科学代码尤为如此。意识到 3.10 节讨论的困难之后，设计人员开发 ILP 的热情已经减退，因此，在大多数体系结构最终采用的硬件方案中，发射速率为每个时钟周期发射 3 ~ 4 条指令。

3.11.2 推测执行与存储器系统

在一个支持推测执行或条件指令的处理器中，自然可能生成一些在没有推测执行时就不会用到的无效地址。如果激发了保护异常，那这不仅是一种错误行为，而且推测执行的收益还可能会被错误异常的开销抵消。因此，存储器系统必须识别推测执行的指令和条件执行的指令，并抑制相应的异常。

此类指令可能会导致缓存因缺失而停顿，这些不必要的停顿仍然可能超过推测带来的收益，出于和前面类似的原因，我们不能允许这种现象的发生。因此，这些处理器必须与非阻塞缓存相匹配。 222

在实际中，由于 L2 缺失的代价非常大，所以编译器通常都仅对 L1 缺失进行推测。图 2-5 表明：对于一些表现优异的科学程序，编译器可以承受多个未能命中的 L2 缺失，从而有效地降低 L2 缺失代价。同样，为使这一方法奏效，缓存背后的存储器系统必须能够满足编译器在存储器同时访问数量方面的要求。

3.12 多线程：开发线程级并行提高单处理器吞吐量

我们在这一节讨论的主题——多线程，是一个真正的交叉主题，它同流水线与超标量相关、与图形处理器（第 4 章）相关，还与多处理器（第 5 章）有关。我们将在这里介绍这一主题，并开发多线程的用途，利用多线程来隐藏流水线和存储器延迟，从而提高单处理器的吞吐量。在下一章，我们将会看到多线程是如何在 GPU 中提供相同好处的，最后，第 5 章将研究多线程与多处理的组合使用。由于多线程是向硬件展示更多并行的主要技术，所以这些主题相互紧密地交织在一起。严格来说，多线程使用线程级并行，因此也正是第 5 章的主题，但考虑到它在改进流水线应用中以及 GPU 中扮演的角色，都促使我们在这里介绍这一概念。

尽管使用 ILP 来提高性能具有很大的优势：它对编程人员是适度透明的，但我们已经看到，在某些应用程序中，ILP 可能受到很大的限制或者难以开发。具体来说，当指令发射率处于合理范围时，那些到达存储器或片外缓存的缓存缺失不太可能通过可用 ILP 来隐藏。当然，当处理器停顿下来等待缓存缺失时，功能单元的利用率会激剧下降。

由于人们在试图通过更多的 ILP 来应对很长的存储器停顿时，效果非常有限，所以很自然

地会问，一个应用程序中是否有其他形式的并行可用来隐藏存储器延迟呢？例如，在联机事务处理系统中，根据请求的多个查询和更新操作之间就存在着自然并行。当然，许多科学应用程序中也包含自然并行，这是因为它们经常是对自有并行特性的三维结构进行建模，可以使用相互分离的线程来开发这种结构。即使是使用基于 Windows 的现代操作系统的桌面应用程序，也经常会同时运行多个活动应用程序，提供了并行源。

多线程技术支持多个线程以重叠方式共享单个处理器的功能单元。而与之相对的是，开发**线程级并行**（TLP）的更一般方法是使用多处理器，它同时并行运行多个独立线程。但是，多线程不会像多处理器那样复制整个处理器。而是在一组线程之间共享处理器核心的大多数功能，仅复制私有状态，比如寄存器和程序计数器。在第 5 章将会看到，许多最新处理器既在一个芯片上集成多个处理器核心，又在每个核心中提供多线程。

223

要复制一个处理器核心中每个线程的状态，就要为每个线程创建独立的寄存器堆、独立的 PC 和独立的页表。存储器本身可以通过虚拟存储器机制共享，这些机制已经支持多重编程了。此外，硬件必须支持对不同线程进行较快速的修改；具体来说，线程切换的效率应当远远高于进程切换，后者通常需要数百个到数千个处理器周期。当然，为使多线程硬件实现性能改进，一个程序中必须包含能够以并发形式执行的多个线程（有时说这种应用程序是多线程的）。这些线程既可由编译器识别（通常是来自具有并行结构的语言），也可能由程序员识别。

实现多线程的硬件方法主要有 3 种。**细粒度多线程**每个时钟周期在线程之间进行一次切换，使多个线程的指令执行过程交织在一起。这种交织通常是以轮询方式完成的，当时发生停顿的所有进程都会被跳过。在细粒度多线程情况下，当一个线程停顿时，哪怕这一停顿只有几个周期，也可以执行其他线程中的指令，所以这种多线程的一个重要好处是它能够隐藏因为长、短停顿而导致的吞吐量损失。细粒度多线程的一个主要不足是它会减缓个体线程的执行速度，因为一个做好执行准备、没有停顿的线程可能会被其他线程的执行延迟。它用单个进程的性能损失（这里所说的性能是以延迟来衡量的）来换取多线程吞吐量的一致性。我们稍后将会研究 Sun Niagara 处理器，它采用了简单的细粒度多线程，Nvidia GPU 也是如此，我们将在下一章介绍。

粗粒度多线程的设计目的就是用来作为细粒度的替代选项。粗粒度多线程仅在发生成本较高的停顿时才切换线程，比如第 2 级或第 3 级缓存缺失时。通过这一变化，只有当一个线程遇到成本高昂的停顿时才会发射其他线程的指令，这样就不再严格要求线程切换操作必须是无成本的，同时也大大降低了减缓任一线程执行速度的可能性。

不过，粗粒度多线程也有一个严重不足：克服吞吐量损失的能力非常有限，特别是由于较短停顿导致的损失。这一局限性源于粗粒度多线程的流水线启动成本。由于采用粗粒度多线程的 CPU 仅发现来自单个线程的指令，所以当流水线发生停顿时，在新线程开始执行之前会出现“气泡”。由于这一启动开销，粗粒度多线程更多是用来应对那些成本超高的停顿，当发生此类停顿时，重新填充流水线的时间与停顿时间相比可以忽略不计。已经有几个研究项目对粗粒度多线程进行了探索，但现在的主流处理器中都还没有使用这一技术。

最常见的多线程实施方式称为**同时多线程**（SMT）。同时多线程是细粒度多线程的一种变体，它是在多发射、动态调度处理器的顶层实现细粒度多线程时自然出现的。和其他形式的多线程一样，SMT 利用线程级并行来隐藏处理器中的长延迟事件，从而提高功能单元的利用率。SMT 的关键在于认识到通过寄存器重命名和动态调度可以执行来自独立线程的多个指令，而不用考虑这些指令之间的相关性；这些相关性留给动态调度功能来处理。

224

图 3-13 从概念上给出了处理器在以下不同配置中开发超标量源的能力差异：

- 不支持多线程的超标量；
- 支持粗粒度多线程的超标量；
- 支持细粒度多线程的超标量；
- 支持同时多线程的超标量。

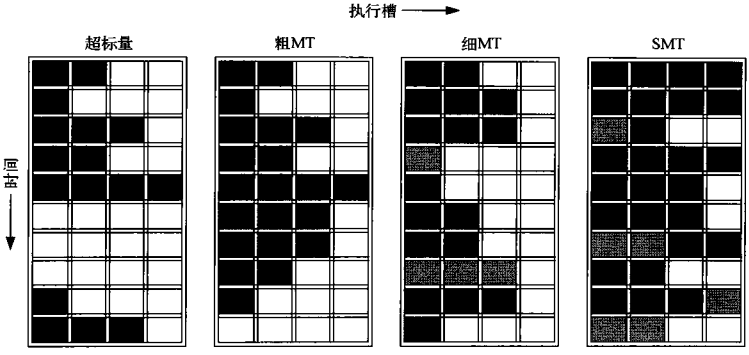


图 3-13 4 种不同方法在应用一个超标量处理器功能单元执行槽的表现状况。水平维度表示每个时钟周期中的指令执行能力。垂直维度代表时间周期序列。空白框（白色）表示该时钟周期的相应执行槽未被使用。灰色和黑色的阴影框对应多线程处理器中的 4 个不同线程。黑色框表示在不支持多线程的超标量中被占用的发射槽。Sun T1 和 T2（aka Niagara）处理器是细粒度多线程处理器，而 Intel Core i7 和 IBM Power7 处理器使用 SMT。T2 有 8 个线程、Power7 有 4 个、Intel i7 有 2 个。在所有现有 SMT 中，每次只发射一个线程中的指令。SMT 的区别在于：在后面决定执行哪条指令时不需要考虑相互之间的影响，可以在同一个时钟周期执行来自几个不同指令的操作

在不支持多线程的超标量中，由于缺乏 ILP（包括用于隐藏存储器延迟的 ILP），所以发射槽的使用非常有限。由于 L2 和 L3 缓存缺失的长度原因，处理器在大多数时间内可能保持空闲。 225

在粗粒度多线程超标量中，通过切换到另一个利用处理器资源的线程来部分隐藏长时间的停顿。这一切换减少了完全空闲的时钟周期数。但是，在粗粒度多线程处理器中，仅当存在停顿时才会进行线程切换。由于新线程有一个启动时间，所以仍然可能存在一些完全空闲的周期。

在细粒度情况中，线程的交织可以消除全空槽。此外，由于每个时钟周期都会改变发射线程，所以可以隐藏较长延迟的操作。由于指令发射和执行联系在一起，所以线程所能发射的指令数目仅限于准备就绪的指令。当发射宽度较窄时，没有什么问题（某个时钟周期要么被占用，要么不被占用），这就是细粒度多线程在单发射处理器中能够完美运行的原因，SMT 没有什么意义。事实上，在 Sun T2 中，每个时钟周期会有两次发射，但它们来自不同线程。这样就不再需要实施复杂的动态调度方法，而是依靠更多线程来隐藏延迟。

如果在多发射动态调度处理器的顶层实现细粒度线程，所得到的结果就是 SMT。在所有现

有 SMT 实现方式中，尽管来自不同线程的指令可以在同一时钟周期内开始执行，但所有发射都来自一个线程，使用动态调度硬件来决定哪些指令已经准备就绪。尽管图 3-13 极大地简化了这些处理器的实际操作，但它仍然能够说明在发射宽度较宽的动态调度处理器中，一般多线程和 SMT 的潜在性能优势。

同时多线程的出现是因为深刻地认识到：动态调度处理器已经拥有支持这一方案所需要的大量硬件方案，包括大量虚拟寄存器。通过为每个线程添加专用的重命名表、保持独立的 PC、支持提交来自多个不同线程的指令，也可以在乱序处理器上实现多线程。

3.12.1 细粒度多线程在Sun T1 上的效果

在这一节，我们利用 Sun T1 处理器来研究多线程隐藏延迟的能力。T1 是 Sun 公司在 2005 年发布的一种细粒度多线程多核微处理器。人们之所以特别关注 T1，是因为它几乎将全部重心都放在开发线程级并行（TLP）上，而不是用于开发指令级并行（ILP）。T1 不再强调对 ILP 的调度（就在此前不久，刚刚发布了更积极的 ILP 处理器），回归一种简单的流水线策略，着力开发 TLP，使用多核和多线程技术来提高吞吐量。

每个 T1 处理器包含 8 个处理器核心，每个核心支持 4 个线程。每个处理器核心包括一个简单的六级、单发射流水线（类似附录 C 中介绍的五级 RISC 流水线，还添加了用于线程切换的流水级）。T1 使用细粒度多线程（但不是 SMT），每个时钟周期切换到一个新的线程，在调度过程中，会跳过那些因为流水线延迟或缓存缺失而处于等待状态的空闲线程。只有在所有 4 个线程全部空闲或停顿时，处理器才会空闲。载入指令和分支指令都会导致一个长度为 3 个时钟周期的延迟，这一延迟只能由其他线程来隐藏。由于浮点性能不是 T1 的重点，所以它只提供了一组浮点功能单元，由所有 8 个核心共享。表 3-16 总结了 T1 处理器的各个特性。

226

表3-16 T1处理器小结

特　性	Sun T1
多处理器和多线程支持	每个芯片8个核心，每个核心4个线程。细粒度线程调度。8个核心共用一个浮点单元。仅支持片上多处理
流水线结构	简单的循序六级流水线，载入指令与分支指令的延迟为3个时钟周期
L1缓存	16 KB指令，8 KB数据。块大小为64个字节。在没有争用时，L2缺失为23个时钟周期
L2缓存	4个独立的L2缓存，每个大小为750 KB，与一个存储器组相联。块大小为64个字节。在没有争用时，主存储器缺失为110个时钟周期
初步实现	90 nm工艺，最大时钟速率为1.2 GHz，功率79 W，3亿个晶体，晶片大小379 mm^2

T1 多线程单核性能

T1 把自己的重点放在 TLP 上，既在各个核心上实施了多线程，又在单个晶片上使用了多个简单核心。在这一节，我们将研究 T1 通过细粒度多线程提升单核心性能的效果。在第 5 章，我们将研究将多线程与多核心结合在一起的效果。

为了研究 T1 的性能，我们使用 3 种面向服务器的基准测试：TPC-C、SPECJBB（SPEC Java 业务基准测试）和 SPECWeb99。由于多个线程会增大单个处理器对存储器的需求，所以可能会使存储器系统过载，从而降低多线程的潜在收益。图 3-14 是 TPC-C 基准测试中，在每个核心执行 1 个线程和每个核心执行 4 个线程时缺失率及所观测到的缺失延迟的相对增长情况。由于存储器系统的争用增加，缺失率和缺失延迟都会增大。缺失延迟的增幅较小，表示存储器系统仍然有未用空间。

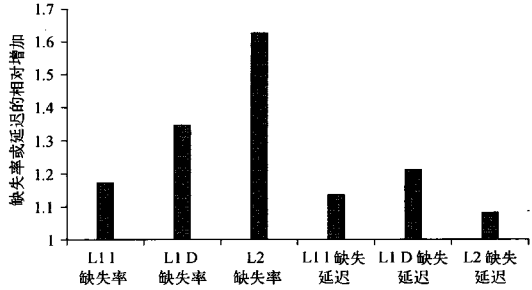


图 3-14 在 TPC-C 基准测试中，每个核心执行 1 个线程和每个核心执行 4 个线程时缺失率与缺失延迟的相对增长情况。这些延迟是指在一次缺失之后返回所请求数据的实际时间。在 4 线程情况中，其他线程的执行可能会隐藏这一延迟的大部分时间

通过研究一个一般线程的行为，可以了解线程之间进行交互以及它们使核心保持繁忙状态的能力。图 3-15 给出了 3 种时钟周期所占的百分比：一种是有线程正在执行，一种是有线程准备就绪，但尚未执行，还有一种是线程没有准备就绪。注意，没有准备就绪并不意味着拥有该线程的核心处于停顿状态，只有当所有 4 个线程都未准备就绪时，该核心才会停顿。

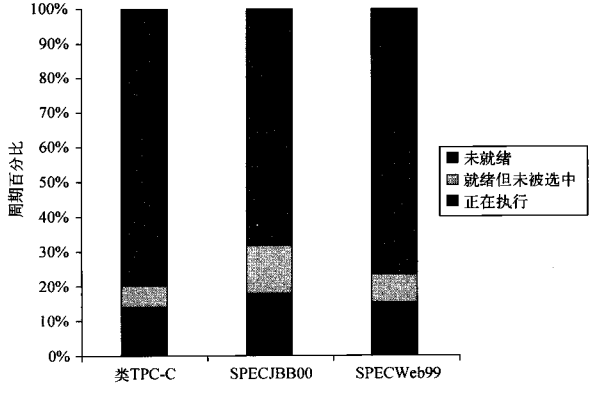


图 3-15 一般线程的状态细分。“正在执行”是指该线程在该时钟周期内发射了一条指令。“就绪但未被选中”是指它可以发射指令，但另一线程已被选中；“未就绪”是指该线程正在等待一个事件（例如，一次流水线延迟或缓存缺失）的完成

线程未准备就绪可能是因为缓存缺失、流水线延迟（由于一些延迟较长的指令导致，比如分支、载入、浮点或整数乘/除）以及各种影响较小的因素。图 3-16 显示了这些不同原因的发生频率。在 50%～70%的时间里，线程未准备就绪是由缓存原因造成的，L1 指令缺失、L1 数据缺失和 L2 缺失造成的影响大致相同。在 SPECJBB 中，由于流水线造成的潜在延迟（称为“流水线延迟”）是最严重的，可能是由于它的分支频率较高所致。

227

228

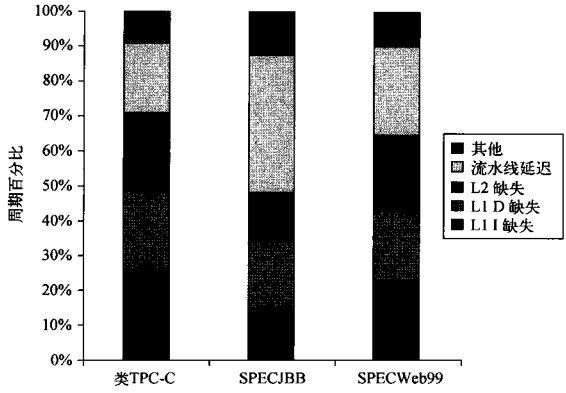


图 3-16 线程的未就绪原因细分。“其他”类别的原因是变化的。在 TPC-C 中，存储器缓冲区满是最主要的因素；在 SPEC-JBB 中，原子指令是最主要的因素；而在 SPECWeb99 中，这两个因素都有影响

表 3-17 显示了每个线程的 CPI 和每个核心的 CPI。由于 T1 是一种细粒度多线程处理器，每个核心有 4 个线程，当有足够的并行时，每个线程的理想有效 CPI 为 4，这意味着每个线程占用每 4 个时钟周期中的 1 个周期。每个核心的理想 CPI 为 1。2005 年，在积极采用 ILP 的核心上运行这些基准测试的 IPC 就已经可以达到 T1 核心上观测到的类似数据。但是，与 2005 年更积极的 ILP 核心相比，T1 核心的大小非常合适，这就是为什么 T1 拥有 8 个核心，而同一时期的其他处理器只提供了 2～4 个核心。因此，在 2005 年发布 Sun T1 处理器时，它在处理 TLP 密集、存储器性能需求较高的的整数应用程序和事务处理工作负载时，具有最佳性能。

表3-17 8核心T1处理器中每个线程的CPI、每个核心的CPI、有效8核心CPI和有效IPC（CPI的倒数）

基准测试	每个线程的CPI	每个核心的CPI
TPC-C	7.2	1.80
SPECJBB	5.6	1.40
SPECWeb99	6.6	1.65

229

3.12.2 同时多线程在超标量处理器上的效果

一个关键问题是：通过实施 SMT 可以使性能提高多少？在 2000～2001 年研究这一问题时，研究人员认为动态超标量会在接下来的 5 年里大幅改进，每个时钟周期可以支持 6～8 次发射，处理器会支持推测动态调度、许多并行载入和存储操作、大容量主缓存、4～8 种上下文，可以同时发射和完成来自不同上下文的指令。到目前为止，还没有处理器能够接近这一水平。

因此，那些认为多编程工作负载可以使性能提高 2～3 倍的模拟研究结果是不现实的。实际中，现有的 SMT 实现只能提供 2～4 个带有取值的上下文，只能发射来自一个上下文的指令，每个时钟周期最多发射 4 条指令。其结果就是：由 SMT 获得的收益是非常有限的。

例如，在 Pentium 4 Extreme 中（在 HP-Compaq 服务器中实现）使用 SMT 时，运行 SPECintRate 基准测试时可以使性能提高 1.01，运行 SPECfpRate 基准测试时可以使性能提高 1.07。Tuck 和 Tullsen [2003]报告：在进行 SPLASH 并行基准测试时，他们发现单核心多线程加速比的范围为

1.02 ~ 1.67，平均加速比大约为 1.22。

由于 Esmaeilzadeh 等人[2011]最近提供了大量包含丰富信息的测量数据，所以我们可以使用一组多线程应用程序来研究在单个 i7 核心中使用 SMT 时获得的性能与能耗收益。我们使用的基准测试包括一组并行科学应用程序和一组来自 DaCapo 和 SPEC Java 套件的多线程 Java 程序，在表 3-18 中对其进行了总结。Intel i7 支持拥有两个线程的 SMT。图 3-17 给出了 SMT 分别为关闭与开启状态时，在 i7 的一个核心上运行这些基准测试的性能比与能耗效率比。（我们绘制了能耗效率比曲线，能耗效率是能耗的倒数，所以和加速比一样，这个比值越大越好。）

表 3-18　此处用于研究多线程、第 5 章用于以 i7 研究多处理的并行基准测试

blackscholes	用Black Scholes PDE为一组期权定价
bodytrack	跟踪一个无标记的人体
canneal	用感知缓存模拟退火法将一个芯片的路由成本降至最低
facesim	模拟人脸的动作，用于可视目的
ferret	搜索引擎，查找一组与查询图像类似的图像
fluidanimate	用SPH算法模拟流体动画的物理规则，在动画中应用
raytrace	使用物理模拟实现可视化
streamcluster	为最佳数据点群集计算近似值
swaptions	用Heath–Jarrow–Morto框架为一组互换期权定价
vips	对某个图像应用一系列变换
x264	MPG-4 AVC/H.264视频编码器
eclipse	集成开发环境
lusearch	文本搜索工具
sunflow	相片真实渲染系统
tomcat	tomcat servlet容器
tradebeans	Tradebeans Daytrader基准测试
xalan	用于转换XML文档的XSLT处理器
pjbb2005	SPEC JBB2005的版本（但固定的是问题规模而不是时间长短）

* 上半部分包含由 Biena 等人[2008]收集的 PARSEC 基准测试。PARSEC 基准测试是用来代表那些适于多核处理器的计算密集、并行应用程序。下半部分包含来自 DaCapo 集合的多线程 Java 基准测试（见 BlackBurn 等人[2006]）和 SPEC 的 pjbb2005。所有这些基准测试都包含一些并行；DaCapo 中的 Java 基准测试和 SPEC Java 工作负载使用多个线程，但真正的并行很少，甚至没有，因此这里没有使用。除了这里及第 5 章中给出的测量结果，请参阅 Esmaeilzadeh 等人[2011]的文献，了解有关这些基准测试特性的更多信息。

尽管这两个 Java 基准测试的性能增益较小，但其加速比的调和均值为 1.28。在采用多线程时，两个基准测试（pjbb2005 和 tradebeans）的并行非常有限。之所以包含这些基准测试是因为它们是典型的多线程基准测试，可以在一个指望提高性能的 SMT 处理器上运行，它们提升效果非常有限。PARSEC 基准测试的加速比要比全套 Java 基准测试好一些（调和均值为 1.31）。如果省略 tradebeans 和 pjbb2005，Java 工作负载的实际加速比（1.39）要比 PARSEC 基准测试好得多。（在图 3-18 的图题中讨论了使用调和均值来汇总结果的含义。）

能耗由加速比与功耗的增加值共同决定。对于 Java 基准测试，SMT 的能耗效率与非 SMT 相同（平均为 1.0），但有两个执行状况不佳的基准测试拉低了这一数值；如果没有 tradebeans 和 pjbb2005，Java 基准测试的平均能耗效率为 1.06，几乎与 PARSEC 基准测试一样好。在 PARSEC 基准测试中，SMT 的能耗降低了 1 – (1/1.08)=7%。这种在降低能耗方面的性能改进是非常罕见

的。当然，在两种情况下，都会因为 SMT 而需要增大静态功率，所以这些结果在能耗性能收益方面可能稍有夸大。

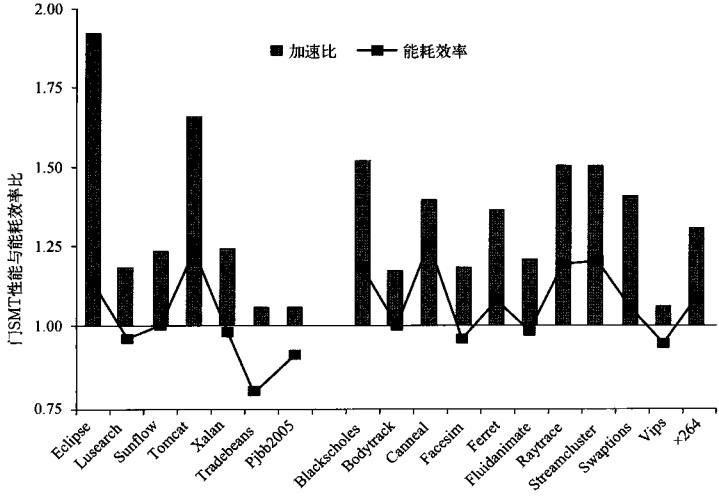


图 3-17 在 i7 处理器的一个核心中使用多线程时，Java 基准测试的平均加速比为 1.28，PARSEC 基准测试为 1.31（使用非加权调和均值，在这种工作负载中，执行单线程基本集中每个基准测试的总时间相同）。能耗效率平均值分别为 0.99 和 1.07（调和均值）。回想一下，能耗效率高于 1.0 就意味着这一特性在缩短执行时间方面的收益高于增大平均功率方面的不足。两个 Java 基准测试的加速比很小，因此，对能耗效率的负面影响非常严重。在所有情景中都关闭了 Turbo Boost。这些数据由 Esmaeilzadeh 等人[2011]使用 Oracle(Sun) HotSpot build 16.3-b01 Java 1.6.0 虚拟机和 gcc v4.4.1 原始编译器进行收集和分析

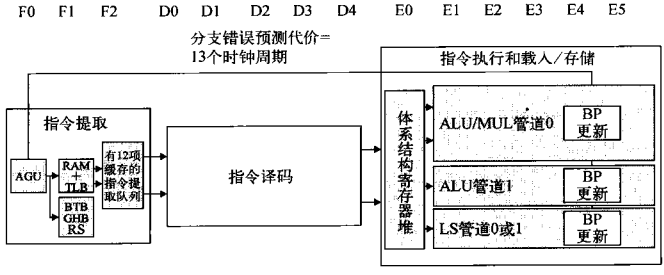


图 3-18 A8 流水线的基础结构为 13 级。3 个时钟周期用于指令提取，4 个用于指令译码，还有一个 5 周期的整数流水线。这样，分支预测错误的代价就是 13 个周期。指令提取单元会尝试保持包含 12 项的指令队列处于填满状态

这些结果清楚地显示了在广泛支持 SMT 的积极推测处理器中，SMT 可以采用一种高能耗效率的方式来提高性能，这是更积极的 ILP 方法无法做到的。是提供多个较简单核心，还是提供较少的更高级核心，这两者之间的天平在 2011 年倾向了提供多个核心，每个核心通常是 3～4 发射超标量的，其 SMT 支持 2～4 个线程。事实上，Esmaeilzadeh 等人[2011]表明：在 Intel i5（一种类似于 i7 的处理器，但其缓存较少，时钟频率较低）和 Intel Atom（一种为上网本市场设计的 80x86 处理器，在 3.14 节介绍）上，通过 SMT 获得的能耗效能改进还要更大一些。

231 ↓ 232

3.13 融会贯通：Intel Core i7 和 ARM Cortex-A8

本节，我们研究两种多发射处理器的设计：一种是 ARM Cortex-A8 核心，它是 iPad 中 Apple A9 处理器、Motorola Droid 和 iPhone 3GS、4 中处理器的基础，另一种是 Intel Core i7，一种高端、动态调度、推测处理器，主要为高端桌面应用程序和服务器应用程序设计。我们首先从较简单的处理器开始。

3.13.1 ARM Cortex-A8

A8 是一种双发射、静态调度超标量处理器，具有动态发射检测功能，允许处理器在每个时钟周期内发射一条或两条指令。图 3-18 显示了 13 级流水线的基本流水线结构。

A8 使用一种动态分支预测器，具有一个 512 项 2 路组相联分支目标缓冲区和一个 4K 项全局历史缓冲区，由分支历史和当前 PC 进行索引。当出现分支目标缓冲区缺失时，则在全局历史缓冲区中进行预测，然后用预测值计算分支地址。此外，还维护一个 8 项返回栈，用于跟踪返回地址。一次错误预测会导致 13 个时钟周期的代价，用于刷新流水线。

图 3-19 显示了指令译码流水线。利用循序发射机制，每个时钟周期最多可以发射两条指令。可以使用一种简单的记分板结构来跟踪何时能够发射一条指令。通过发射逻辑可以处理一对相关指令，当然，除非它们的发射方式能使转发路径消除两者之间的相关，否则会在记分板上对它们进行序列化。

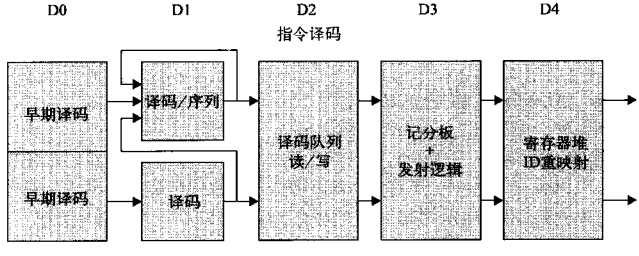


图 3-19 A8 的 5 级指令译码。在第一级中，使用取指单元生成的 PC（或者来自分支目标缓冲区，或者来自 PC 递增器）从缓存中提取大小为 8 字节的块。最多对两条指令进行译码，并将它们放在译码队列中；如果两条指令都不是分支，则将 PC 递增，为下一次取指做准备。一旦进入译码队列，则由记分板逻辑决定何时可以发射这些指令。在发射时，读取寄存器操作数；回想在简单的记分板中，操作数总是来自寄存器。寄存器操作数和操作码被发送到流水线的指令执行部分

图 3-20 显示了 A8 处理器的执行流水线。指令 1 或指令 2 可以进入这个载入/存储流水线。在这些流水线之间支持完全旁路。ARM Cortex-A8 流水线使用简单的双发射静态调度超标量，可以在较低功率下实现相当高的时钟频率。与之相对，i7 使用一种相当积极的 4 发射动态调度推理流水线结构。

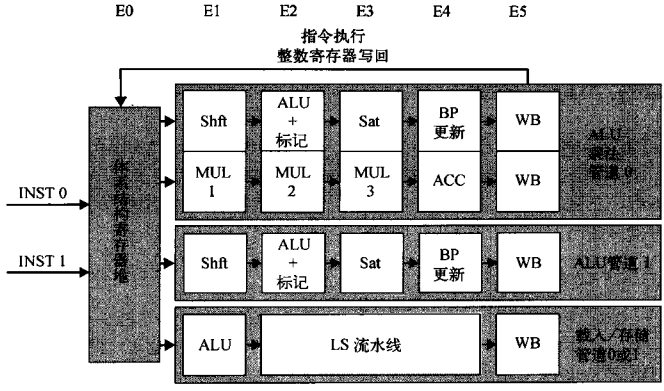


图 3-20 A8 的一级指令译码。乘法运算总是在 ALU 流水线 0 中执行

A8 流水线的性能

由于 A8 采用双发射结构，所以它的理想 CPI 为 0.5。可能会因为以下 3 种来源而产生流水线停顿。

233~234 (1) 功能冒险，如果被选择同时发射的两个相邻指令使用同一功能流水线，就会出现功能冒险。由于 A8 是静态调度的，所以避免此类冲突是编译器的任务。如果不能避免此类冲突，A8 在这个时钟周期内最多只能发射一条指令。

(2) 数据冒险，在流水线的早期进行侦测，可能使两条指令停顿（如果第一条指令不能发射，第二条总是会被停顿），也可能是一对指令中的第二条指令停顿。编译器负责尽可能防止此类停顿。

(3) 控制冒险，仅在分支预测错误时发生。

除了流水线停顿之外，L1 和 L2 缺失都会导致停顿。

图 3-21 是影响 Minnespec 基准测试实际 CPI 的各项因素的估计值，我们在第 2 章曾经见过这些基准测试。可以看到，这一 CPI 的主要影响因素是流水线延迟，而不是存储器停顿。出现这一结果的部分原因是 Minnespec 的缓存印记要小于全套 SPEC 或其他大型程序。 235

流水线停顿会造成性能大幅下降，深刻认识到这一点，可能对于决定将 ARM Cortex-A9 设计为动态调度超标量处理器起到了重要作用。A9 和 A8 相似，每个时钟最多发射两条指令，但它使用了动态调度和推测。在一个时钟周期内可以开发执行多达 4 条未完成指令（2 个 ALU、1 个载入/存储或浮点/多媒体指令、1 个分支指令）。A9 使用一种功能更为强劲的分支预测器、指令缓存预取和非阻塞 L1 数据缓存。图 3-22 表明，在使用相同时钟频率和几乎相同的缓存配置时，A9 的平均性能是 A8 的 1.28 倍。

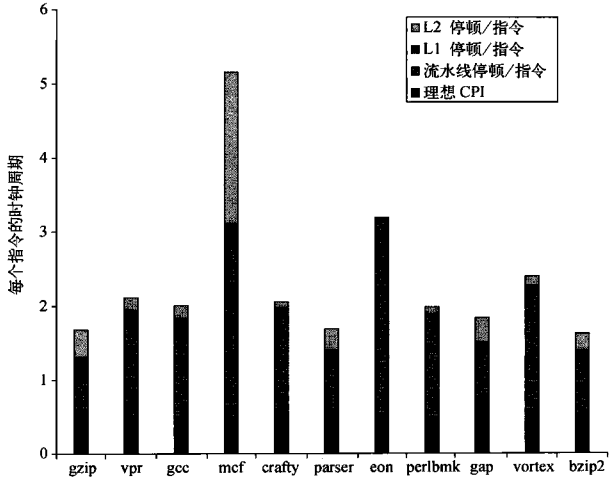


图 3-21 对 ARM A8 上 CPI 各组成分量的估计值表明：流水线停顿是增大基本 CPI 的主要因素。eon 值得专门一提，它完成基于整数的图形计算（光线跟踪），而且缓存缺失很少。由于大量使用乘法，计算非常密集，所以单个乘法流水线可能会成为主要瓶颈。这一估计值是利用 L1 和 L2 缺失率与代价来计算每个指令中因为 L1 和 L2 生成的停顿而获得的。从具体模拟器测得的 CPI 中减去这些估计值即可获得流水线停顿。流水线停顿包含所有这 3 种冒险再加上一些次要影响，比如路预测错误等

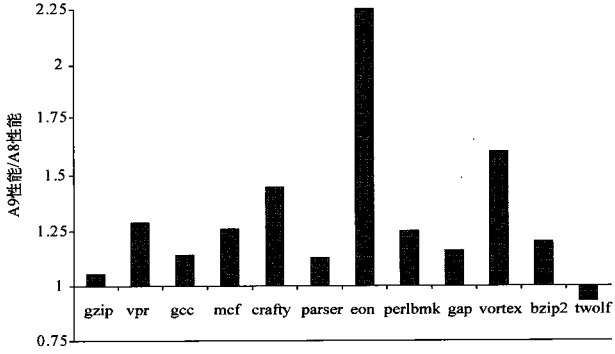


图 3-22 在时钟频率均为 1 GHz、L1 与 L2 缓存大小相同时，A9 与 A8 的性能比表明：A9 大约快 1.28 倍。两者都使用 32 KB 主缓存和 1 MB 次级缓存，A8 采用 8 路组相关，A9 使用 16 路组相联。A8 处理器缓存中的块大小为 64 字节，A9 为 32 字节。在图 3-21 的图题中曾经提及，eon 大量使用了整数乘法，动态调度与快速乘法流水线的组合使用显著提高了 A9 的性能。由于 A9 的 L1 块较小，所以 twoif 的缓存表现不佳，可能是由于这一因素，twoif 的速度略有减缓

3.13.2 Intel Core i7

i7 采用一种非常积极的乱序推测微体系结构，具有较深的流水线，目的是通过综合应用多发射与高时钟速率来提高指令吞吐量。图 3-23 显示了 i7 流水线的整体结构。我们在研究流水线时，按照如下步骤，首先从指令提取开始，接下来是指令提交。 236

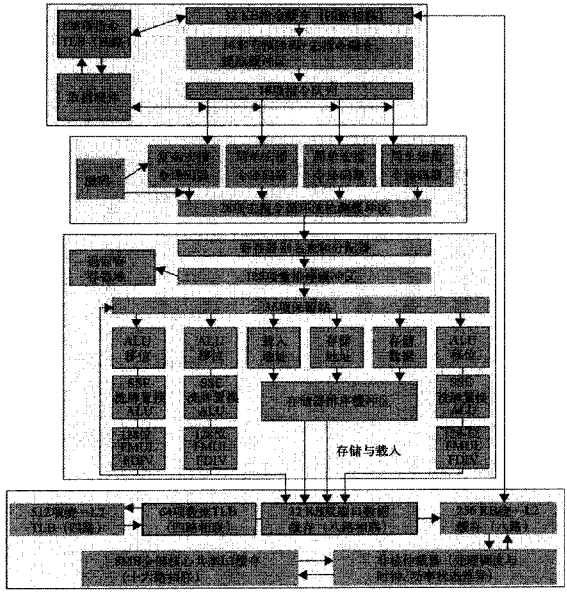


图 3-23 Intel Core i7 流水线结构，一同给出了存储器系统组件。总流水线深度为 14 级，分支错误预测成本为 17 个时钟周期。共有 48 个载入缓冲区和 32 个存储缓冲区。6 个独立功能单元可以在同一时钟周期分别开始执行准备就绪的微操作

(1) 指令提取——处理器使用一个多级分支目标缓冲区，在速度与预测准确度之间达到一种平衡。还有一个返回地址栈，用于加速函数返回。错误预测会损失大约 15 个时钟周期。利用预测地址，指令提取单元从指令缓存中提取 16 个字节。

(2) 16 个字节被放在预译码指令缓冲区中——在这一步，会执行一个名为微指令融合的进程。**微指令融合**接收指令组合（比如先对比后分支），然后将它们融合为一个操作。这个预译码过程还将 16 个字节分解为单独的 x86 指令。由于 x86 指令的长度可能是 1 ~ 17 字节中的任何一种长度，所以这一预译码非常重要，预译码器必须查看许多字节才能知道指令长度。然后将单独的 x86 指令（包括一些整合指令）放到包含 18 项的指令队列中。 237

(3) 微指令译码——各个 x86 指令被转换为微指令。微指令是一些类似于 MIPS 的简单指令，可以直接由流水线执行；这种方法将 x86 指令集转换为更容易实现流水化的简单操作，1997 年在 Pentium Pro 中引入，一直使用至今。3 个译码器处理可以直接转换为一个微指令的 x86 指令。

对于那些语义更为复杂的 x86 指令，有一个微码引擎可供用于生成微指令序列；它可以在每个时钟周期中生成多达 4 条微指令，并一直持续下去，直到生成必要的微指令序列为止。按照 x86 指令的顺序，将这些微指令放在一个包含 28 项的微指令缓冲区中。

(4) 微指令缓冲区执行**循环流检测**和**微融合**——如果存在一个包含循环的小指令序列（长度少于 28 条指令，或不足 256 个字节），循环流检测器会找到这个循环，直接从缓冲区中发射微指令，不再需要启动指令提取与指令译码级。微整合则合并指令对，比如载入/ALU 运算和 ALU 运算/存储，并将它们发射到单个保留站（在保留站仍然可以独立发射这些指令），从而提高了缓冲区的利用率。在对 Intel Core 体系结构的一项研究中（这种结构也合并了微整合和宏融合），Bird 等人[2007]发现微整合几乎对性能没有什么影响，而宏融合则对整数性能有一定的下面影响，对浮点性能也几乎没有什么影响。

(5) 执行基本指令发射——在寄存器表中查看寄存器位置，对寄存器重命名、分配重排序缓冲区项，从寄存器或重排序缓冲区中提取任意结果，然后向保留站发送微指令。

(6) i7 使用一个包括 36 项的集中保留站，供 6 个功能单元共享。每个时钟周期最多可以向这些功能单元分发 6 个微指令。

(7) 微指令由各个功能单元执行，然后将结果发送给任何正在等待的保留站以及寄存器退回单元，一旦知道指令不再具有推测之后，将在这里更新寄存器状态。重排序缓冲区中与该指令相对应的数目被标记为完成。

(8) 当重排序缓冲区头部的一条或多条指令被标记为完成之后，则执行寄存器退回单元中的未完成写入操作，并将这些指令从重排序缓冲区中删除。 238

i7 的性能

在前面几节中，我们研究了 i7 分支预测器的性能和 SMT 的性能。在这本节，我们将研究单线程流水线性能。由于积极推测与非阻塞缓存的存在，所以很难准确描述理想性能与实际性能之间的差距。我们将会看到，因为指令不能发射而导致的停顿很少。例如，只有大约 3%的载入指令是因为没有可用保留站而导致的。大多数损失不是来自分支错误预测就是来自缓存缺失。分支预测错误的成本为 15 个时钟周期，而 L1 缺失的成本大约为 10 个时钟周期；L2 缺失的成本比 L1 缺失的 3 倍略多一些，而 L3 缺失的成本大约是 L1 缺失成本的 13 倍（130~135 个时钟周期）！尽管在发生 L3 缺失和某些 L2 缺失时，处理器会尝试寻找一些替代指令来执行，但有些缓冲区会在缺失完成之前填满，从而导致处理器停止发射指令。

为了研究错误预测和错误推测的成本，图 3-24 给出了未退回工作（即它们的结果未被取消）相对于所有微指令分发指令所占的比例（根据分发到流水线中的微指令数目来测量）。比如，对 sjeng 来说，由于在所分发的微指令中，有 25%从来未被退回，所以浪费了 25%的工作。 239

注意，在某些情况下，被浪费的工作与图 3-3 所示的分支错误预测率吻合，而在几种实例中，比如 mcf 中，被浪费的工作似乎要高于错误预测率。从存储器行为的角度也许能够解释这些情况。当数据缓存缺失率非常高时，只要有足够的保留站可供停顿存储器引用使用，mcf 就将在错误的推测期间分发许多指令。在检测到分支预测错误时，与这些指令相对应的微指令将被刷新，但当推测存储器引用试图完成时，可能会产生缓存争用。对处理器来说，在启动缓存请求之后，没有一种简单方法可以使其停止。

图 3-25 显示了 19 个 SPECCPU2006 基准测试的总 CPI。整数基准测试的 CPI 为 1.06，方差很大（标准偏差为 0.67）。MCF 和 OMNETPP 是两个主要例外，它们的 CPI 都大于 2.0，而其他基准测试都接近或小于 1.0（gcc 是第二高，为 1.23）。这种偏差是由于分支预测准确度和缓存缺 240

失率方面的差别造成的。对于整数基准测试，L2 缺失率是 CPI 的最佳预测值，L3 缺失率（非常小）几乎没有什么影响。

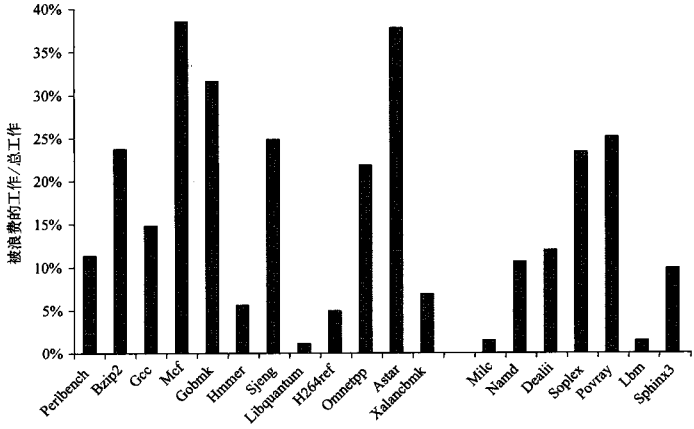


图 3-24 通过计算所有已分发微指令中未退回微指令所占的比值，绘制了“被浪费工作”的数量。例如，sjeng 的比值为 25%，也就是说在已分发、执行的微指令中有 25%被抛弃。本节的数据由路易斯安那州大学的 Lu Peng 教授和 Ying Zhang 博士生收集

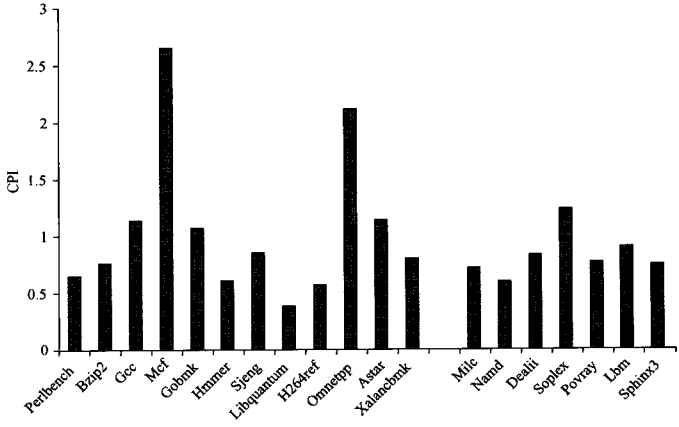


图 3-25 19 个 SPECCPU2006 的 CPI 表明，尽管行为表现有很大不同，但浮点与整数基准测试的平均 CPI 都是 0.83。整数基准测试的 CPI 值变化范围为 0.44 ~ 2.66，标准偏差为 0.77，而浮点基准测试的变化范围为 0.62 ~ 1.38，标准偏差为 0.25。本节的数据由路易斯安那州大学的 Lu Peng 教授和 Ying Zhang 博士生收集

浮点基准测试的性能较高：平均 CPI 较低（0.89）、标准偏差较低（0.25）。对于浮点基准测试来说，L1 和 L2 对于确定 CPI 是同等重要的，而 L3 则扮演着小而重要的角色。尽管 i7 的动态调度和非阻塞功能可以隐藏一些缺失延迟，但缓存存储器表现仍然是一个重要因素。这进一步强化了多线程作为另一种方法来隐藏存储器延迟的作用。

3.14 谬论与易犯错误

这里介绍的几点谬论主要集中在根据单一测量值（比如时钟频率或 CPI）来预测性能、能耗效率以及进行推断的难度。我们还将表明：对于不同基准测试，不同体系结构方法可能会有截然不同的表现。

谬论 如果能够保持技术的稳定，可以很容易地预测同一指令集体系结构两个不同版本的性能与能耗效率。

Intel 为低端上网本和 PMD 制造了一种名为 Atom 230 的处理器，它的微体系结构与 ARM A8 非常相似。有趣的是，Atom 230 和 Core i7 920 都是用相同的 45 nm Intel 技术制造的。表 3-19 对 Intel Core i7、ARM Cortex-A8 和 Intel Atom 230 进行了总结。这些相似性提供了一个非常宝贵的机会，可以在保持基础制造工艺一致的情况下，直接针对同一指令集对比两种截然不同的微体系结构。在我们进行这一对比之前，需要再对 Atom 230 多说两句。

表 3-19 4 核 Intel i7 920、一种典型 Arm A8 处理器芯片（256 MB L2，32 K L1，无浮点运算）和 Intel ARM 230 的概述清楚地显示了处理器 PMD（ARM）或上网本处理器（Atom）与服务器、高端桌面处理器在设计原理方面的区别

领　域	特定属性	Intel i7 920 四个核心，每个核心都有浮点功能	ARM A8 一个核心，没有浮点功能	Intel Atom 230 一个核心，没有浮点功能
物理芯片属性	时钟频率	2.66 GHz	1 GHz	1.66 GHz
	热设计功率	130W	2W	4W
	包装	1366管脚BGA	522管脚BGA	437管脚BGA
存储器系统	TLB	两级全四路组相联 128 I/64 D 512 L2	一组，全相联32 I/32D	两级全四路组相联16 I/16D 64 L2
	缓存	三级 32 KB/32 KB 256 KB 2~8 MB	两级 16/16或32/32 KB 128 KB~1 MB	两级 32/24 KB 512 KB
	峰值存储器带宽	17 GB/s	12 GB/s	8 GB/s
流水线结构	峰值发射速率	4操作/时钟周期 带有融合功能	2操作/时钟周期	2操作/时钟周期
	流水线调度	乱序推测	循序动态发射	循序动态发射
	分支预测	两级	两级 512项BTB 4K全局历史8项返回栈	两级

* 记住，i7 包括 4 个核心，每个核心的性能都要比单核心 A8 或 Atom 快几倍。所有这些处理器都是用相差不大的 45 nm 工艺实现的。

Atom处理器实现x86体系结构，使用标准技术将x86指令转换为类似于RISC的指令（自20世纪90年代以来，所有x86实现都已完成。）Atom使用了一种功能要稍强一点的微运算，可以将算术运算与载入指令或存储指令结为一对。这意味着：对于典型的指令混合体来说，这些指令中只有平均4%需要一个以上的微运算。然后在一个深度为16的流水线中执行这些微运算，能够在每个时钟周期循序发射两条指令，和ARM A8一样。这种处理器包含双整数ALU、用于浮点加法和其他浮点运算的分离流水线，还有两个存储器运算流水线，与ARM A8相比，支持更具一般性的双执行，但仍然受循序发射功能的限制。Atom 230有一个32 KB的指令缓存和一个24 KB的数据缓存，在同一晶片上由一个共享512 KB L2提供后援。（Atom 230还支持双线程的多线程技术，但我们仅考虑单线程对比。）表3-20对i7、A8、Atom处理器及其关键特性进行了总结。

表3-20　3种变化幅度很大的不同Intel处理器

处　理　器	时钟频率	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	3.8 GHz	11.5	12.2
Intel Itanium -2	1.66 GHz	14.5	17.3
Intel i7	3.3 GHz	35.5	38.4

* 尽管Itanium处理器有2个核心，i7有4个，但在这些基准测试中仅使用一个核心。

我们可能会预期，这两种用相同工艺实现、具有相同指令集的处理器在相对性能与能耗方面的表现是可预测的，也就是说功率和性能接近线性。我们使用三组基准测试来检验这一假设。第一组是Java、单线程基准测试，选自DaCapo基准测试和SPEC JVM98基准测试（见Esmaeilzadeh等人[2011]关于基准测试和测量结果的讨论）。第二、第三组基准测试选自SPEC CPU2006，分别由整数和浮点基准测试组成。

241

在图3-26中可以看出，i7的性能明显优于Atom。所有基准测试在i7上都至少快4倍，两个SPECFP基准测试快10倍以上，一个SPECINT基准测试的运行速度要快8倍以上。

242

由于这两个处理器的时钟频率比为1.6，所以大多数优势体现在i7的CPI要低得多：对于Java基本测试为2.8，对于SPECINT基准测试为3.1，SPECFP基准为4.3。

但是，i7的平均功耗略低于43W，而Atom的平均功耗为4.2W，为前者的十分之一！将性能与功率结合起来，可以看出Atom在能耗效率方面通常有1.5倍以上的优势，有时会达到2倍以上！通过对比这两种采用相同底层技术的处理器，可以清楚地看到，一种采用动态调度与推测的积极超标量的性能优化是以能耗效率的显著降低为代价的。

243

谬论　CPI较低的处理器总是要更快一些。

谬论　时钟频率较快的处理器总是要更快一些。

要点在于：性能是由CPI与时钟频率的乘积决定的。在通过实现CPU的深度流水化获得高时钟频率后，还必须保持较低的CPI，才能全面体现快速时钟频率的优势。同理，一个时钟频率很高、CPI很低的简单处理器也可能更慢一些。

在前面讨论的谬论中已经看到，在为不同环境设计的处理器中，即使它们采用相同的ISA，也可能在性能与能耗效率方面有很大不同。事实上，即使是同一公司为高端应用程序设计的同一处理器系列，在性能方面也会有很大差异。表3-20显示的是Intel公司对x86体系结构两种不同实现方式的整数与浮点性能，还有一个是Itanium体系结构，也是Intel出品。

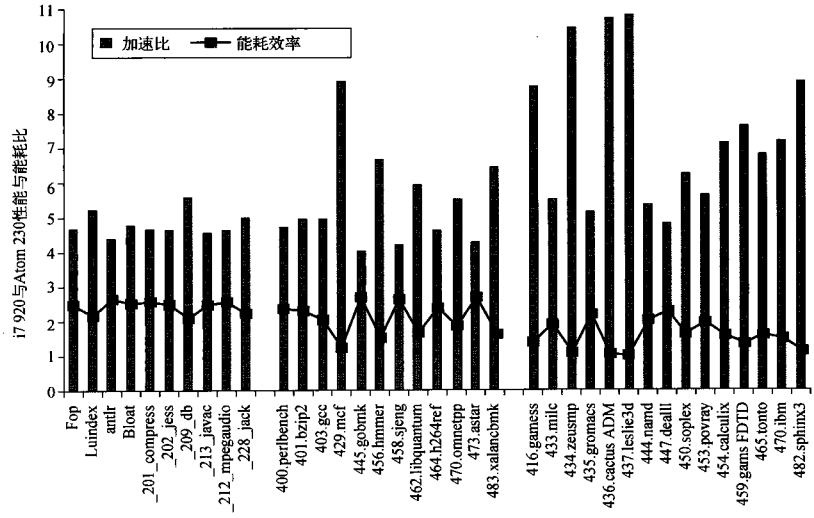


图 3-26 **一组单线程基准测试的相对性能与功耗效率表明，i7 920 比 Atom 230 快 10 倍以上，而功率系数平均只有它的二分之一。**柱形条中显示的性能是 i7 与 Atom 的比值，即执行时间（i7）/执行时间（Atom）。能耗以曲线显示，为能耗（Atom）/能耗（i7）。i7 在能耗效率方法从来都没有打赢 Atom，不过在 4 个基准测试方面的性能基本相当，其中有 3 个是浮点。这里显示的数据由 Esmaeilzadeh 等人[2011]收集。SPEC 基准测试是使用标准 Intel 编译器在打开优化的情况下编译的，Java 基准测试使用 Sun (Oracle) Hostpot Java VM。i7 上只有一个核心是活动的，其余核心处于深度节能模式。i7 上使用了 Turbo Boost，这一功能可以提高其性能优势，但相对的能耗效率会略有降低

Pentium 4 是 Intel 公司生成的最积极的流水线处理器。它使用深度超过 20 级的流水线，有 7 个功能单元，还有缓存微指令，而不是 x86 指令。在这种积极的实施方式中，它的性能相对较差一些，这清楚地表明它在开发更多 ILP 方面的努力失败了（很容易就会同时有 50 条指令正在执行）。Pentium 的功耗与 i7 相似，不过它的晶体管数较少，主存储器大约是 i7 的一半，包括仅有 2 MB 的次级缓存，没有第三级缓存。

Intel Itanium 是一种 VLIW 风格的体系结构，尽管调度超标量相比，它的复杂度可能增加，但它的时钟频率从来都不能与主流 x86 处理器相提并论（尽管它的总 CPI 与 i7 类似）。在研究这些结果时，读者应当明白它们不同的实现技术，对于同等流水线的处理器来说，i7 在晶体管速度上具有优势，从而在时间频率方面也占据上风。不过，性能方面的巨大变化（Pentium 和 i7 之间相差 3 倍以上）还是令人吃惊的。下面的谬论部分将解释这一优势主要来自何处。 244

谬论 有时越大、越被动就越好。

在 2000 年前期，人们大多把注意力都放在构建更积极的处理器，用于开发 ILP，其中就包括 Pentium 4 体系结构（它在一个微处理器中使用了当时最深的流水线）和 Intel Itanium（它每

个时钟周期的峰值发射率是当时最高的）。后来人们很快发现在开发 ILP 时，主要限制因素是存储器系统造成的。尽管推测乱序流水线可以很好地隐藏第一级缺失中 10～15 个时钟周期的大部分缺失代价，但它们在隐藏第二级缺失代价方面几乎是无能为力的，由于涉及主存储器访问，所以第二级缺失代价可能达到 50～100 个时钟周期。

结果就是，尽管使用了数目庞大的晶体管和极为高级、聪明的技术，但这些设计从来未能接近峰值指令吞吐量。下一节将讨论这一两难选择，并从更积极的 ILP 方案转向多核技术，但过去还出现了另外一个变化，放大了这一缺陷。设计人员不再尝试用 ILP 来隐藏更多的存储器延迟，而是直接利用晶体管来创建更大的缓存。Itanium 2 和 i7 使用三级缓存，而 Pentium 4 使用了两级缓存，三级缓存为 9MB 和 8MB，而 Pentium 4 的二级缓存为 2 MB。不用说，构建更多的缓存要比设计 20 多级的 Pentium 4 流水线容易得多，而且从表 3-20 中的数据也可以看出，这种方法也更有效一些。

3.15 结语：前路何方

在 2000 年初，人们对开发指令级并行的关注达到顶峰。Intel 当时要发布 Itanium，它是一种高发射率的静态调度处理器，依靠一种类似于 VLIW 的方法，支持强劲的编译器。采用动态调度推测执行的 MIPS、Alpha 和 IBM 处理器正处于其第二代，已经变得更宽、更快。那一年还发布了 Pentium 4，它采用推测调度，具有 7 个功能单元和 1 个深度超过 20 级的流水线，然而它的发展前景浮现出一些乌云。

诸如 3.10 节介绍的研究表明，要想进一步推动 ILP 是极为困难的，大约三到五年前的第一代推测处理器已经实现了峰值指令吞吐量，而持续指令执行速度的增长要慢得多。

接下来的五年真相大白。人们发现 Itanium 是一个很好的浮点处理器，但在整数处理方面表现泛泛。Intel 仍在生成这一产品，但它的用户不是很多，时钟频率要落后于主流 Intel 处理器，微软不再支持其指令集。Intel Pentium 4 实现了很好的性能，但在性能/瓦特（也就是能量利用）方面的效率很低，这种处理器的复杂程度也使它很难通过提高发射率来进一步提高性能。这条通过开发 ILP 来进一步提高处理器性能的 20 年之路已经走到尽头。人们普遍认为 Pentium 4 已经超越了回报递减点，积极、复杂的 Netburst 微体系结构被放弃。

到 2005 年，Intel 和所有其他主要处理器制造商都调整了自己的方法，将重点放在多核心上。往往通过线程级并行而不是指令级并行来实现更高的性能，高效运用处理器的责任从硬件转移到软件和程序员身上。从流水线和指令级并行的早期发展以来（大约是 25 年之前），这是处理器体系结构的最重大变化。

在同一时间，设计人员开始探索利用更多数据级并行来作为提高性能的另一方法。SIMD 扩展使桌面和服务器微处理器能够适当地提高图形功能及类似功能的性能。更重要的是，GPU 追求更积极地使用 SIMD，用大量数据级并行来实现应用程序的极大性能优势。对于科学应用程序，这些方法可以有效地替代在多核心中开发的更具一般性但效率较低的线程级并行。下一章将研究数据级并行应用方面的这些发展。

许多研究人员预测 ILP 的应用会大幅减少，预计未来会是双发射超标量和更多核心的天下。但是，略高的发射率以及使用推测动态调度来处理意外事件（比如一级缓存缺失）的优势，使适度 ILP 成为多核心设计的主要构造模块。SMT 的添加及其有效性（无论是在性能方面还是在能耗效率方面）都进一步巩固了适度发射、乱序、推测方法的地位。事实上，即使是在嵌入市

场领域，最新的处理器（例如 ARM Cortex-A9）已经引入了动态调度、推测和更宽的发射速率。

未来处理器几乎不会尝试大幅提高发射宽度。因为从硅利用率和功率效率的角度来看，它的效率太低了。考虑一下表 3-21 中的数据，这个表中给出了 IBM Power 系列的 4 种最新处理器。在过去 10 年里，Power 处理器对 ILP 的支持已经有了一定的改进，但所增加的大部分晶体管（从 Power 4 到 Power7 增加了差不多 7 倍）用来提高每个晶片的缓存和核心数目。甚至对 SMT 支持扩展的重视也多于 ILP 吞吐量的增加：从 Power4 到 Power7 的 ILP 结构由 5 发射变为 6 发射，从 8 个功能单元变为 12 个（但最初的 2 个载入/存储单元没有变化），而 SMT 支持从零变为 4 个线程/处理器。显然，即使是 2011 年最高级的 ILP 处理器（Power7），其重点也超越了指令级并行。下面两章将重点介绍开发数据级和线程级并行的方法。 246

表 3-21 4 种 IBM Power 处理器的特性

	Power4	Power5	Power6	Power7
发布时间	2001	2004	2007	2010
最初时钟频率（GHz）	1.3	1.9	4.7	3.6
晶体管数目（百万）	174	276	790	1200
每时钟周期的发射数	5	5	7	6
功能单元	8	8	9	12
每芯片的核心数	2	2	2	8
SMT线程	0	2	2	4
总片上缓存（MB）	1.5	2	4.1	32.3

* 除 Power6 之外的所有处理器都是动态调度的，Power6 是静态、循序的，所有处理器都支持载入/存储流水线。除十进制单元之外，Power6 的功能单元与 Power5 相同。Power7 使用 DRAM 作为 L3 缓存。

3.16 历史回顾与参考文献

附录 L.5 节讨论了流水线和指令级并行的发展。我们为深入阅读和探讨此类主题提供了大量参考文献。附录 L.5 节涵盖了第 3 章和附录 H 的内容。

案例研究与练习（Jason D.Bakos 和 Robert P.Colwell 设计）

案例研究：探讨微体系结构技术的影响

本案例研究说明的概念

- 基本指令调度、重排序、分发
- 多发射和冒险
- 寄存器重命名
- 乱序和推测执行
- 乱序资源花费在哪里

你受命设计一种新的处理器体系结构，正在尝试找出分配硬件资源的最佳方式。你应当运用在第 3 章学到的哪些硬件与软件技术？我们已经拥有功能单元、存储器以及某些代表性代码的延迟列表。对于这种新设计的性能要求，你的老板有些含糊不清，但你从经验中知道，在其他条件相同的情况下，速度越快通常就会越好。让我们从基础做起，表 3-22 提供了一个指令序列和延迟列表。 247

3.1 [10] <1.8、3.1、3.2>如果在先前指令执行完毕之前，不会开始执行新的执行，那表3-22中代码序列的基准性能如何（用每次循环迭代的时钟周期表示）？忽略前端提取与译码过程。假定执行进程没有因为缺少下一条指令而停顿，但每个周期只能发射一条指令。假定该分支被选中，而且存在一个时钟周期的分支延迟槽。

3.2 [10] <1.8、3.1、3.2>思考一下延迟数目到底意味着什么——它们表示一个给定函数生成其输出结果所需要的时钟周期数，没有别的意思。如果整个流水线在每个功能单元的延迟周期中停顿，那么至少要保证任何一对“背靠背”指令（生成结果的指令后面紧跟着使用结果的指令）正确执行。但并非所有指令对具有这种“生成者/使用者”的关系。有时，两条相邻指令之间没有任何关系。如果流水线检测到真正的数据相关，而且只会因为这些真数据相关而停顿，而不会仅仅因为有某个功能单元繁忙就盲目停顿，那表3-22代码序列中的循环体需要多少个时钟周期？在代码中需要容纳所述延迟的时候插入<stall>。（提示：延迟为+2的指令需要在代码序列中插入两个<stall>时钟周期。可以这样来考虑：一条需要一个时钟周期的指令的延迟为1+0，也就是不需要额外的等待状态。那么延迟1+1就意味着1个停顿周期，延迟1+N有N个额外停顿周期。

表3-22 练习3.1至练习3.6的代码与延迟

			超过一个时钟周期的延迟	
Loop:	LD	F2,0(RX)	存储器LD	+4
I0:	DIVD	F8,F2,F0	存储器SD	+1
I1:	MULTD	F2,F6,F2	整数ADD，SUB	+0
I2:	LD	F4,0(Ry)	分支	+1
I3:	ADDD	F4,F0,F4	ADDD	+1
I4:	ADDD	F10,F8,F2	MULTD	+5
I5:	ADDI	Rx,Rx,#8	DIVD	+12
I6:	ADDI	Ry,Ry,#8		
I7:	SD	F4,0(Ry)		
I8:	SUB	R20,R4,Rx		
I9:	BNZ	R20,Loop		

3.3 [15] <3.6、3.7>考虑一种多发射设计。假定有两个执行流水线，每个流水线可以在每个时钟周期开始执行一条指令，前端有足够的取指/译码带宽，所以它不会造成执行停顿。假定可以马上将结果从一个执行单元转发给另一个单元。进一步假设执行流水线停顿的唯一理由就是观察到真正的数据相关。现在这一循环需要多少个周期？

248

3.4 [10] <3.6、3.7>在练习3.3的多发射设计中，你可能已经发现一些微妙的问题。尽管这两个流水线的指令表完全相同，但这两个流水线既不相同，也不能互换，因为它们之间存在一个隐含顺序，必须反映原程序中的指令排序。如果指令N在管道0上开始执行的同时，指令N+1也开始在执行管道1上执行，而N+1碰巧需要的执行延迟要比N短，那么N+1会在N之前完成（而程序排序隐含的情况并非如此）。给出至少两个理由，说明为什么可能存在冒险，为什么在微体系结构中需要特殊考虑。从表3-22的代码中选出这样两条指令，作为说明这一冒险的示例。

3.5 [20] <3.7>调整指令顺序，提高表3-22中代码的性能。假定采用练习3.3中的双管道机器，而且练习3.4中的乱序执行问题已经成功解决。现在只需要考虑观察到真正的数据相关和功能延迟的情景即可。重排序后的代码需要多少个时钟周期？

3.6 [10/10/10] <3.1、3.2>从硬件发挥潜力的角度来看，任何一个未在管道中启动新操作的时钟周期都是一次机会的丧失。

a. [10] <3.1、3.2>在练习3.5重新排序后的代码中，按两个管道进行计数，被浪费的时钟周期（也

就是没有启动新操作的时钟时期）占总时钟周期的比例是多少？

b. [10] <3.1、3.2>循环展开是在代码中查找更多并行的一种标准编译器技术，尽量避免浪费提高性能的机会。手工展开练习 3.35 中重新排序后代码循环的两个迭代。

c. [10] <3.1、3.2>所实现的加速比为多少？（对于这个练习，仅将 N+1 次迭代的指令设为绿色，将它们与第 N 次迭代的指令区分开来；如果你正在实际展开这个循环，必须重新安排寄存器，以防止迭代之间产生争用。）

3.7 [15] <2.1>计算机将大多数时间都花费在循环上，因此，为了使 CPU 资源保持繁忙，在多个循环迭代中进行推理执行是大有所为的。但没有什么事情是轻而易举的；编译器只是省略了循环代码的一个副本，所以即使多个迭代正在处理不同数据，但它们仍然会使用相同寄存器。为避免多个迭代出现寄存器争用情况，我们对它们的寄存器进行重命名。表 3-23 给出了一段希望硬件进行重命名的示例代码。编译器就是展开循环，使用不同的寄存器来避免冲突，但如果我们希望硬件来展开循环，它也必须进行寄存器重命名。怎么做呢？假定硬件有一个临时寄存器池（将这个寄存器称为 T 寄存器，假定共有 64 个此类寄存器，即 T0 ~ T63），它们可以代替编译器指定的寄存器。这一重命名硬件按 src（源）寄存器目标索引，表中的值是上一个指向该寄存器的目的地的 T 寄存器。（可以把这些值看作“生产者”，把 src 寄存器看作“使用者”；生产者将其结果放在哪里并不重要，只要它的使用者能够找到它就行。）考虑表 3-23 中的代码序列。每次看到代码中的目标寄存器时，就用从 T9 开始的下一个可用 T 来代替。然后相应更新所有 src 寄存器，从而保持真正数据相关。给出最终得到的结果。（提示：见表 3-24。）

249

表3-23 寄存器重命名实践的示例代码

```
Loop:  LD      F4,0(Rx)
I0:    MULTD   F2,F0,F2
I1:    DIVD    F8,F4,F2
I2:    LD      F4,0(Ry)
I3:    ADDD    F6,F0,F4
I4:    SUBD    F8,F8,F6
I5:    SD      F8,0(Ry)
```

表3-24 提示：寄存器的期望输出

```
I0:    LD      T9,0(Rx)
I1:    MULTD   T10,F0,T9
...
```

3.8 [20] <3.4>练习 3.7 研究了简单的寄存器重命名：当硬件寄存器重命名程序看到一个源寄存器时，它会代替上一条指向该源寄存器的指令的目标 T 寄存器。当重命名表看到目标寄存器时，它会为其代入下一个可用 T，但超标量设计需要在计算机的每一级的每一个时钟周期处理多条指令，包括寄存器重命名在内。简单的标量处理器会为每一条指令搜索 src 寄存器映射，并在每个时钟周期分配一个新的 dest 映射。超标量处理器也必须能够做到这一点，它们还必须确保正确地处理两个并发指令之间的 dest-src 关系。考虑表 3-25 中的示例代码序列。另外假定，在对两条指令进行重命名的时钟周期开始时，已经知道了下面将要使用的两个可用 T 寄存器。从概念上来说，我们希望为第一条指令进行重命名表查询，然后根据其目标的 T 寄存器来更新重命名表。然后第二条指令会做完全相同的工作，从而可以正确处理指令之间的所有相关。但是，在一个时钟周期内，没有足够的时间将 T 寄存器目标写到重命名表中，然后再为第二条指令进行查

250

询。寄存器替换工作必须实时完成（与寄存器重命名表更新并行完成）。图 3-27 显示了一个路线图，它使用多工器和比较器来完成必要的实时寄存器重命名。你的任务是对于表 3-25 中所示代码中的每一条指令，给出重命名表在每个时钟周期的状态。假定这个表的每个开始项目与其索引相同（T0=0；T1=1，……）。

表3-25 超标量寄存器重命名的示例代码

I0:	SUBD	F1,F2,F3
I1:	ADDD	F4,F1,F2
I2:	MULTD	F6,F4,F1
I3:	DIVD	F0,F2,F6

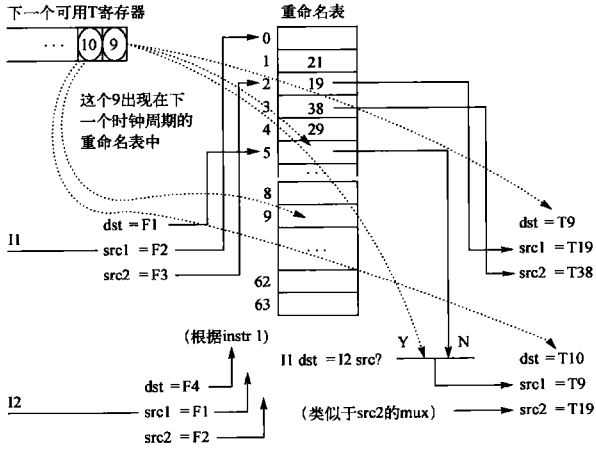


图 3-27 超标量计算机的重命名表和实时寄存器替换。（注意 src 为源，dest 为目的地。）

251

3.9 [5] <3.4>如果你不清楚寄存器重命名程序必须做哪些工作，可以回过头来看看正在执行的汇编代码，问问自己，必须具备哪些条件才能获得正确结果。例如，考虑一个 3 路超标量机器，同时对下面这 3 条指令进行重命名：

```
ADDI R1, R1, R1
ADDI R1, R1, R1
ADDI R1, R1, R1
```

如果 R1 的值在初始为 5，那么执行这一序列之后，它的值应当是多少？

3.10 [20] <3.4、3.9>超长指令字（VLIW）设计人员在为寄存器的使用制定体系结构方面的规则时有几种基本选项。假定一种 VLIW 设计有自排水执行流水线：一旦启动了一个操作，它的结果在目标寄存器中最多存在 L 个时钟周期（其中 L 是该操作的延迟）。从来没有足够的寄存器，所以人们希望能够将现有寄存器发挥到极致。考虑表 3-26。如果负载的延迟为 1+2 个时钟周期，将此循环展开一次，并说明一个能够在每个时钟周期内完成两次载入和两次加法操作的 VLIW 如何使用最少数目的寄存器，假定没有任何流水线中断或停顿。在使用自排水流水线的情况下，给出一个事件示例，它能够破坏流水线、生成错误结果。

表3-26 具有两次加法、两次载入和两次停顿的示例VLIW代码

```
Loop: LW      R4,0(R0) ;   ADDI   R11,R3,#1
      LW      R5,8(R1) ;   ADDI   R20,R0,#1
      <stall>
      ADDI    R10,R4,#1;
      SW      R7,0(R6) ;   SW     R9,8(R8)
      ADDI    R2,R2,#8
      SUB     R4,R3,R2
      BNZ     R4,Loop
```

3.11 [10/10/10] <3.3>假定一个五级单流水线微体系结构（取指、译码、执行、访问存储器、写回）以及表 3-27 中的代码。除了 LW、SW 和分支指令之外，所有操作的执行时间都是一个时钟周期，LW 和 SW 为 1+2 个时钟周期，分支指令为 1+1 个时钟周期。不进行转发操作。针对该循环的一次迭代，说明每个时钟周期内每条指令的各个阶段。

a. [10] <3.3>每个循环迭代有多少时钟损失在分支开销上？

b. [10] <3.3>假定一个静态分支预测器，能够在译码阶段识别出反向分支。现在，有多少个时钟周期浪费在分支开销上？

c. [10] <3.3>假定一个动态分支预测器。有多少时钟周期损失在正确预测上？

252

表3-27 练习3.11的代码循环

```
Loop: LW    R3,0(R0)
      LW    R1,0(R3)
      ADDI  R1,R1,#1
      SUB   R4,R3,R2
      SW    R1,0(R3)
      BNZ   R4, Loop
```

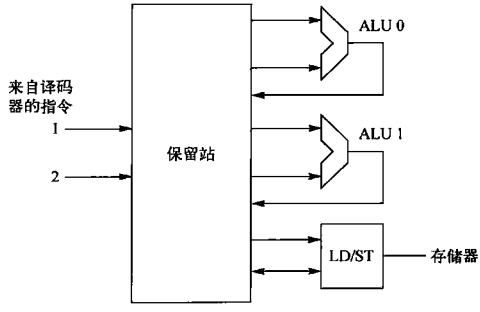


图 3-28 一种乱序微体系结构

3.12 [15/20/20/10/20] <3.4、3.7、3.14>让我们考虑一下动态调度可以实现什么。假定如图 3-28 所示的微体系结构。假定算术逻辑单元（ALU）可以完成所有算术运算（MULTD、DIVD、ADDD、ADDI、SUB）和分支指令，而且保留站（RS）在每个时钟周期最多可以向每个功能单元分配一个运算（向每个 ALU 分配一个运算，再加上向 LD/ST 分配一个存储器访问指令）。

a. [15] <3.4>假定表 3-22 序列中的所有指令都存在于保留站中，还没有进行重命名。突出标示

出此代码中可以通过寄存器重命名提高性能的所有指令。（提示：查找“写后读”和“写后写”冒险。假定功能单元的延迟与表 3-22 中相同。）

b. [20] <3.4>假定(a)部分中代码的寄存器重命名版本在时钟周期 N 中存储于保留站中，延迟如表 3-22 所示。说明保留站应当如何以每个时钟周期为单位乱序分配这些指令，以优化此代码的性能。（假定保留站约束条件与(a)部分相同。还假定这些结果必须写回保留站后才能供其他指令使用——没有旁路。）这个代码序列会占用多少个时钟周期？

253

c. [20] <3.4>(b)部分让保留站尝试优化调度这些指令。但在实际中，所关注的整个指令序列通常不会存在于保留站中。各种事件会清除保留站，当新的代码序列流入、流出译码器时，保留站必须选择如何分配它拥有的内容。假定保留站是空的。在周期 0 中，这个序列前两个经过寄存器重命名的指令出现在保留站中。假定分配任何指令都需要一个时钟周期，并假定功能单元的延迟与练习 3.2 中相同。进一步假定前端（译码器/寄存器重命名器）将持续在每个时钟周期中提供两条新指令。给出 RS 逐个时钟周期的分配顺序。这一代码序列现在需要多少个时钟周期？

d. [10] <3.14>如果我们希望改进(c)部分的结果，以下哪种方法最有帮助：(1)采用另一种 ALU？(2)采用另一个 LD/ST 单元？(3)将 ALU 结果完全旁路给后续操作？或者(4)将最长的延迟截为两段？加速比为多少？

e. [20] <3.7>现在让我们考虑推理，在超越一或多个条件分支时的取指、解码和执行过程。我们这样做是有双重动机的：我们在(c)部分中提供的分配表中有许多“无指令”；我们知道计算机把大多数时间花费在执行循环上（这也就意味着返回循环顶部的分支具有很强的可预测性）。循环告诉我们到哪里去找更多的工作来做；稀疏分配表提醒我们有机会提前做点工作。在部分(d)中，我们发现了贯穿这个循环的关键路径。设想一下将这个路径的第二副本迭入部分(b)中获得的分配表中。要完成两个循环的工作，需要增加多少个时钟周期（假定所有指令都在保留站中）？（假定所有功能单元都完全实现了流水化。）

练习

3.13 [25] <3.13>在这个练习中，我们将研究三种处理器之间的性能权衡，这三种处理器采用了不同类型的多线程技术。每个处理器都是超标量、使用循序流水线，在所有载入与分支指令之后需要固定停顿三个时钟周期，它们的 L2 缓存相同。在同一周期中由相同线程发射的指令按程序顺序读取，不能包含任何数据相关或控制相关。

- 处理器 A 是一种超标量 SMT 体系结构，能够在每个周期内从两个线程发射最多 2 条指令。
- 处理器 B 是一种精细 MT 体系结构，能够在每个周期内从一个线程发射最多 4 条指令，并在任意流水线停顿时切换线程。
- 处理器 C 是一种粗放 MT 体系结构，能够在每个周期内从一个线程发射最多 8 条指令，并在 L1 缓存缺失时切换线程。

254

我们的应用程序是一个列表搜索程序，它对一个存储器区域进行扫描，在由 R16 和 R17 指定的地址范围之间搜索 R9 中存储的特定值。这一搜索过程是平行完成的：将搜索空间平均分为 4 个同等大小的连续区块，并为每个块指定一个搜索线程（产生 4 个线程）。每个线程的大多数运行时间花费在下面已经展开的循环体上：

```
loop:  LD R1,0(R16)
       LD R2,8(R16)
       LD R3,16(R16)
       LD R4,24(R16)
       LD R5,32(R16)
       LD R6,40(R16)
```

```
LD R7,48(R16)
LD R8,56(R16)
BEQAL R9,R1,match0
BEQAL R9,R2,match1
BEQAL R9,R3,match2
BEQAL R9,R4,match3
BEQAL R9,R5,match4
BEQAL R9,R6,match5
BEQAL R9,R7,match6
BEQAL R9,R8,match7
DADDIU R16,R16,#64
BLT R16,R17,loop
```

作出以下假定：

- 使用一个屏障来确保所有线程同时开始；
- 在该循环迭代两次之后出现第一次 L1 缓存缺失；
- BEQAL 分支都未被选中；
- BLT 总是被选中；
- 所有 3 个处理器都以轮询方式调度线程。

判断每个处理器完成该循环的前两次迭代需要多少个周期。

3.14 [25/25/25] <3.2、3.7>在这个练习中，我们研究如何利用软件技术从一个常见的向量循环中提取指令级并行（ILP）。下面的循环是所谓的 DAXPY 循环（双精度 a*X*加 *Y*），它是高斯消元法的核心运算。下面的代码实现 DAXPY 运算 *Y*=a*X*+*Y*，向量长度为 100。最初，R1 被设置为数组 *X* 的基地址，R2 被设置为 *Y* 的基地址： 255

```
      DADDIU   R4,R1,#800  ; R1 = upper bound for X
foo:  L.D      F2,0(R1)   ; (F2) = X(i)
      MUL.D    F4,F2,F0   ; (F4) = a*X(i)
      L.D      F6,0(R2)   ; (F6) = Y(i)
      ADD.D    F6,F4,F6   ; (F6) = a*X(i) + Y(i)
      S.D      F6,0(R2)   ; Y(i) = a*X(i) + Y(i)
      DADDIU   R1,R1,#8   ; increment X index
      DADDIU   R2,R2,#8   ; increment Y index
      DSLTU    R3,R1,R4   ; test: continue loop?
      BNEZ     R3,foo     ; loop if needed
```

假定功能单元的延迟如下表所示。假定在 ID 阶段解决一个延迟为 1 周期的分支。假定结果被完全旁路。

产生结果的指令	使用结果的指令	延迟（单位：时钟周期）
浮点乘	浮点ALU运算	6
浮点加	浮点ALU运算	4
浮点乘	浮点存储	5
浮点加	浮点存储	4
整数运算和所有载入	任何指令	2

a. [25] <3.2>假定一个单发射流水线。说明在编译器未进行调度以及对浮点运算和分支延迟进

行调度之后，该循环是什么样的，包括所有停顿或空闲时间周期。在未调度和已调度情况下，结果向量 Y 中每个元素的执行时间为多少个时钟？为使处理器硬件独自匹配调度编译器所实现的性能改进，时钟频率应当为多少？（忽略加快时钟速度会对存储器系统性能产生的影响。）

b. [25] <3.2>假定一个单发射流水线。根据需要对循环进行任意次展开，使调度中不存在任何停顿，消除循环开销指令。必须将此循环展开多少次？给出指令调度。结果中每个元素的执行时间为多少？

256

c. [25] <3.7>假定一个 VLIW 处理器，其指令中包含 5 个操作，如表 3-11 所示。我们对比两种程度的循环展开。首先，将该循环展开 6 次以提取 ILP，并对其进行调度，使其没有任何停顿（所谓停顿就是完全空闲的发射周期）、消除循环开销指令，然后重复该进程，将循环展开 10 次。忽略分支延迟槽。给出两个调度表。对于每个调度而言，结果向量中每个元素的执行时间为多少？每个调度中，所使用的操作槽占多大比例？两种调度中的代码相差多少？两种调度的总共需要多少寄存器？

3.15 [20/20] <3.4、3.5、3.7、3.8>在这个练习中，我们将研究 Tomasulo 算法的各种变体在执行练习 3.14 中的循环时的表现。功能单元（FU）如下表所述。

功能单元类型	EX中的循环数	功能单元数	保留站数
整数	1	1	5
浮点加法器	10	1	3
浮点乘法器	15	1	2

作出如下假设：

- 功能单元未实现流水化；
- 功能单元之间不存在转发，结果由公共数据总线（CDB）传送；
- 执行级（EX）既进行有效地址计算，又进行存储器访问，以完成载入和存储指令。因此，这个流水线为 IF/ID/IS/EX/WB；
- 载入指令需要一个时钟周期；
- 发射（IS）和写回（WB）结果级各需要一个时钟周期；
- 共有 5 个载入缓冲区槽和 5 个存储缓冲区槽；
- 假定“等于/不等于 0 时转移”（BNEZ）指令需要一个时钟周期；

a. [20] <3.4 ~ 3.5>对这个问题来说，使用图 3-4 的单发射 Tomasulo MIPS 流水线，流水线延迟如上表所示。对于该循环的 3 个迭代，给出每个指令的停顿周期数以及每个指令在哪个时钟周期中开始执行（即，进入它的第一个 EX 周期）。每个循环迭代需要多少个时钟周期？以表格方式给出你的答案，表中应当具有以下列标头：

- 迭代（循环迭代数）；
- 指令；
- 发射（发射指令的周期）；
- 执行（执行指令的周期）；
- 存储器访问（访问存储器的周期）；
- 写 CDB（将结果写到 CDB 的周期）；
- 注释（对指令正在等待的事件的说明）。

257

在表中给出这个循环的 3 次迭代，可以忽略第一条指令。

b. [20] <3.7、3.8>重复(a)部分，但这一次假定双发射 Tomasulo 算法和完全流水化的浮点单元（FPU）。

3.16 [10] <3.4>Tomasulo 的算法有以下缺点：每个 CDB 每个时钟周期仅能计算一个结果。使用上一

个问题的硬件配置与延迟，并找出一段不超过 10 个指令的代码序列，其中 Tomasulo 算法必须因为 CDB 争用而停顿。在序列中指出发生这一停顿的位置。

3.17 [20] <3.3>(m, n)相关分支预测器利用最近执行的 m 个分支的行为从 2^m 个预测器中作出选择，这些预测器都是 n 位预测器。两级局部预测器以类似方式工作，但仅跟踪每个独立分支过去的行为来预测未来的行为。

这些预测器中涉及一种设计折中：相关预测器几乎不需要存储器来进行历史记录，使它们能够针对大量独立分支来维持 2 位预测器（降低了分支指令重复利用同一预测器的概率），而本地预测器则需要相当多的存储器来记录历史，因此只能跟踪较少数量的分支指令。对于这一练习来说，考虑一个（1，2）相关预测器，它可以根据 4 个分支（需要 16 位），而一个（1，2）局部预测器使用相同数量的存储器只能跟踪两个分支。对于下面的分支结果，提供每次预测、用于作出预测的表项以及根据预测结果对表的更新，还有每个预测器的最终错误预测率。假定到这一点的所有分支都已经选定。如下表所示对预测器进行初始化。 258

相关预测器

项　目	分　支	上一个结果	预　测
0	0	T	T（一次错误预测）
1	0	NT	NT
2	1	T	NT
3	1	NT	T
4	2	T	T
5	2	NT	T
6	3	T	NT（一次错误预测）
7	3	NT	NT

局部预测器

项　目	分　支	前两个结果（右边的是最近的）	预　测
0	0	T,T	T（一次错误预测）
1	0	T,NT	NT
2	0	NT,T	NT
3	0	NT	T
4	1	T,T	T
5	1	T,NT	T（一次错误预测）
6	1	NT,T	NT
7	1	NT,NT	NT

分支PC（字地址）	输　出
454	T
543	NT
777	NT
543	NT
777	NT
454	T
777	NT
454	T
543	T

3.18 [10] <3.9>假定有一个深度流水线处理器，为其实现分支目标缓冲区，仅用于条件分支。假定错误预测的代价都是4个周期，缓冲缺失代价都是3个周期，假定命中率为90%、准确率为90%、分支频率为15%。与分支代价固定为两个周期的处理器相比，采用这一分支目标缓冲区的处理器要快多少？假定每条指令的时钟周期数（CPI）为基本CPI，没有分支停顿。

3.19 [10/5] <3.9>考虑分支目标缓冲区，正确条件分支预测、错误预测和缓存缺失的代价分别为0、2和2个时钟周期。考虑一种区分条件与无条件分支的分支目标缓冲区设计，而条件分支存储目标地址，对于无条件分支则存储目标指令。

a. [10] <3.9>当缓冲区中发现无条件分支时，代价为多少个时钟周期？

b. [10] <3.9>判断对于无条件分支进行分支折合所获得的改进。假定命中率为90%，无条件分支频率为5%，缓冲区缺失的代价为两个时钟周期。这样可以获得多少改进？对于这一改进来说，必须达到多高的命中率才能提供性能增益？

259

4

向量、SIMD 和 GPU 体系结构中的数据级并行

我们将这些算法称为数据并行算法，是因为它们的并行源于对大型数据集的同时操作，而不是来自多个控制线程。

——W. Daniel Hillis 和 Guy L. Steele，
"Data Parallel Algorithms"，*Comm. ACM* (1986)

如果要耕种一块地，那你会选择两头强壮的公牛呢，还是 1024 只小鸡？

——Seymour Cray，超级计算机之父

（争论的是选择两个功能强大的向量处理器，还是选择许多简单的处理器。）

260 ~ 261

4.1 引言

关于单指令多数据（SMID）体系结构（在第1章介绍），人们总是会问一个问题：有多少应用程序拥有大量的数据级并行（DLP）。50年后，这个问题的答案不只包含科学运算中的矩阵计算，还包括媒体图像和声音处理。此外，由于单条指令可以启动许多数据运算，所以SIMD在能耗效率方面可能要比多指令多数据（MIMD）更高效一些，MIMD每进行一次数据运算都需要提取和执行一条指令。这两个答案使SIMD对于个人移动设备极具吸引力。最后，SMID与MIMD相比的最大优势可能就是：由于数据操作是并行的，所以程序员可以采用顺序思维方式但却能获得并行加速比。

本章介绍SIMD的3种变体：向量体系结构、多媒体SIMD指令集扩展和图形处理单元（GPU）。[①]

第一种变体的出现要比其他两个早30年以上，它实际上就是以流水线形式来执行许多数据操作。与其他SIMD变体相比，这些**向量体系结构**更容易理解和编译，但过去一直认为它们对于微处理器来说太过昂贵了，这一看法直到最近才有所改变。这种体系结构的成本，一部分用在晶体管上，另一部分用于提供足够的DRAM带宽，因为它广泛依赖于缓存来满足传统微处理器的存储器性能要求。

第二种SIMD变体借用SIMD名称来表示基本同时进行的并行数据操作，在今天支持多媒体应用程序的大多数指令集体系结构中都可以找到这种变体。x86体系结构的SIMD指令扩展是在1996年以MMX（多媒体扩展）开始的，在接下来的10年间出现了几个SSE（流式SIMD扩展）版本，一直发展到今天的AVX（高级向量扩展）。为了使x86计算机达到最高计算速度，通常需要使用这些SIMD指令，特别是对于浮点程序。

SIMD的第三种变体来自GPU社区，它的潜在性能要高于当今传统多核计算机的性能。尽管GPU的一些特征与向量体系结构相同，但它们有自己的一些独特特征，部分原因在于它们的发展生态系统。在GPU的发展环境中，除了GPU及其图形存储器之外，还有系统处理器和系统存储器。事实上，为了辨识这些差别，GPU社区将这种体系结构称为**异类**。

262

对于拥有大量数据并行的问题，所有这三种SIMD变体都有一个共同的好处：与经典的并行MIMD编程相比，程序员的工作更轻松一些。为了对比SIMD与MIMD的重要性，图4-1绘制了x86计算机中MIMD的核心数与SIMD模式中每个时钟周期的32位及64位运算数随时间的变化曲线。

对于x86计算机，我们预期每个芯片上每两年增加两个核心，SIMD的宽度每四年翻一番。给定这些假设，在接下来的10年里，由SIMD并行获得的潜在加速比为MIMD并行的两倍。因此，尽管MIMD并行最近受到的关注要多得多，但理解SIMD并行至少与理解MIMD并行一样重要。对于同时具有数据级并行和线程级并行的应用程序，2020年的潜在加速比将比今天的加速比高一个数量级。

---

① 本章的基础材料包括：本书第4版的附录F“向量处理器”（Krste Asanovic编写）和附录G“VLIW和EPIC的硬件与软件”；*Computer Organization and Design*第4版的附录A“图形与计算GPU”（John Nickolls和David Kirk编写）；还参考了Joe Gebis和David Patterson于2007年4月在*IEEE Computer*杂志上发表的文章“Embracing and Extending 20th-Century Instruction Set Architectures”。

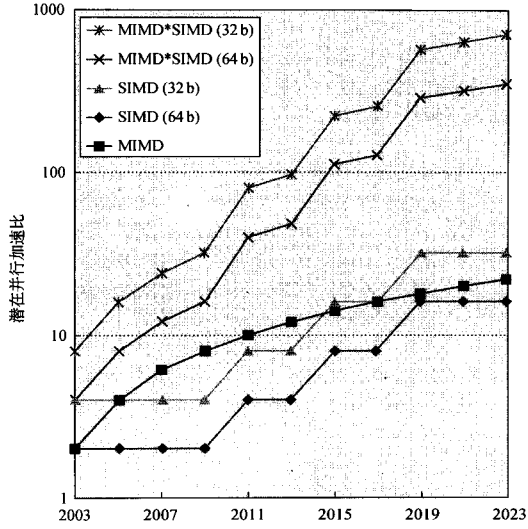


图 4-1 x86 计算机通过 MIMD、SIMD、MIMD 与 SIMD 组合中的并行而在不同时间获得的潜在加速比。这一曲线假定 MIMD 的每个芯片上每两年增加两个核心，SIMD 的运算数每四年翻番

本章的目的是让架构师理解向量为什么比多媒体 SIMD 更具一般性，以及向量与 GPU 体系结构之间的相似与区别。由于向量体系结构是多媒体 SIMD 指令的超集（包括一个更好的编译模型），而且由于 GPU 与向量体系结构有一些相似性，所以我们首先从向量体系结构入手，为以下两节奠定基础。下一节介绍向量体系结构，而附录 G 对这一主题的讨论要深入得多。 263

4.2 向量体系结构

执行可向量化应用程序的最高效方法就是向量处理器。

——Jim Smith

计算机体系结构国际研讨会（1994 年）

向量体系结构获得在存储器中散布的数据元素集，将它们放在一些大型的顺序寄存器堆中，对这些寄存器堆中的数据进行操作，然后将结果放回存储器中。一条指令对数据向量执行操作，从而会对独立数据元素进行数十个“寄存器–寄存器”操作。

这些大型寄存器堆相当于由编译器控制的缓冲区，一方面用于隐藏存储器延迟，另一方面用于充分利用存储器带宽。由于向量载入和存储是尝试流水化的，所以这个程序仅在每个向量载入或存储操作中付出较长的存储器延迟时间，而不需要在载入或存储每个元素时耗费这一时间，从而将这一延迟时间分散在比如 64 个元素上。事实上，向量程序会尽力使存储器保持繁忙状态。

4.2.1 VMIPS

我们首先看一个向量处理器，它由图 4-2 所示的主要组件组成。这个处理器大体以 Cray-1 为基础，它是本节讨论的基础。我们将这种指令集体系结构称为 VMIPS；它的标量部分为 MIPS，它的向量部分是 MIPS 的逻辑向量扩展。这一小节的其他部分研究 VMIPS 的基本体系结构与其他处理器有什么关系。

VMIPS 指令集体系结构的主要组件如下所示。

- **向量寄存器**——每个向量寄存器都是一个固定长度的寄存器组，保存一个向量。VMIPS 有 8 个向量寄存器，每个向量寄存器保留 64 个元素，每个元素的宽度为 64 位。向量寄存器堆需要提供足够的端口，向所有向量功能单元馈送数据。这些端口允许将向量操作高度重叠，发送到不同向量寄存器。利用一对交叉交换器将读写端口（至少共有 16 个读取端口和 8 个写入端口）连接到功能单元的输入或输出。
- **向量功能单元**——每个单元都完全实现流水化，它可以在每个时钟周期开始一个新的操作。需要有个控制单元来检测冒险，既包括功能单元的结构性冒险，又包括关于寄存器访问的数据冒险。图 4-2 显示 VMIPS 有 5 个功能单元。为简单起见，我们仅关注浮点功能单元。

264

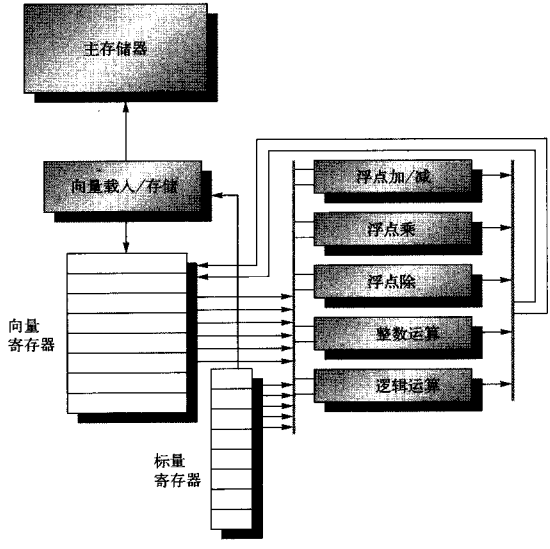


图 4-2 向量体系结构 VMIPS 的基本结构。这一处理器拥有类似于 MIPS 的标量体系结构。它还有 8 个 64 元素向量寄存器，所有功能单元都是向量功能单元。这一章为算术和存储器访问定义了特殊的向量指令。图中显示了用于逻辑运算与整数运算的向量单元，所以 VMIPS 看起来像是一种通常包含此类单元的标准向量处理器；但是，我们不会讨论这些单元。这些向量与标量寄存器有大量读写端口，允许同时进行多个向量运算。一组交叉交换器（粗灰线）将这些端口连接到向量功能单元的输入和输出

- **向量载入/存储单元**——这个向量存储器单元从存储器中载入向量或者将向量存储到存储器中。VMIPS 向量载入与存储操作是完全流水化的，所以在初始延迟之后，可以在向量寄存器与存储器之间以每个时钟周期一个字的带宽移动字。这个单元通常还会处理标量载入和存储。
- **标量寄存器集合**——标量寄存器还可以提供数据，作为向量功能单元的输入，还可以计算传送给向量载入/存储单元的地址。它们通常是 MIPS 的 32 个通用寄存器和 32 个浮点寄存器。在从标量寄存器堆读取标量值时，向量功能单元的一个输入会闩锁住这些值。

265

表 4-1 列出了 VMIPS 向量指令。在 VMIPS 中，向量运算使用的名字与标量 MIPS 指令的名字相同，但后面追加了字母“VV”。因此，ADDVV.D 就是两个双精度向量的加法。向量指令的输入或者为一对向量寄存器（ADDVV.D），或者为一个向量寄存器和一个标量寄存器，通过附加“VS”来标识（ADDVS.D）。在后一种情况下，所有操作使用标量寄存器的相同值来作为一个输入：运算 ADDVS.D 将向量寄存器中的每个元素都加上标量寄存器的内容。向量功能单元在发射时获得标量值的一个副本。大多数向量运算有一个向量目标寄存器，尽管其中一些（比如人口计数）会产生标量值，这个值将存储在标量寄存器中。

表 4-1 VMIPS 向量指令，仅给出双精度浮点操作

指令	操作数	功能
ADDVV.D	V1,V2,V3	将V2与V3中的元素相加，然后将结果放在V1中
ADDVS.D	V1,V2,F0	将F0加到V2中的每个元素，然后将结果放在V1中
SUBVV.D	V1,V2,V3	从V2中减去V3的元素，然后将结果放在V1中
SUBVS.D	V1,V2,F0	从V2的元素中减去F0，然后将结果放在V1中
SUBSV.D	V1,F0,V2	从F0中减去V2的元素，然后将结果放在V1中
MULVV.D	V1,V2,V3	将V2和V3中的元素相乘，然后将结果放在V1中
MULVS.D	V1,V2,F0	将V2中的每个元素乘以F0，然后将结果放在V1中
DIVVV.D	V1,V2,V3	将V2中的元素除以V3，然后将结果放在V1中
DIVVS.D	V1,V2,F0	将V2中的元素除以F0，然后将结果放在V1中
DIVSV.D	V1,F0,V2	将F0除以V2的元素，然后将结果放在V1中
LV	V1,R1	将从地址R1开始的存储器内容载入向量寄存器V1中
SV	R1,V1	将向量寄存器V1的内容存储到从地址R1开始的存储器中
LVWS	V1,(R1,R2)	从地址R1开始载入V1，步幅为R2（即，R1 + i × R2）
SVWS	(R1,R2),V1	将V1存储到地址R1处，步幅为R2（即，R1 + i × R2）
LVI	V1,(R1+V2)	将向量内容载入V1，向量的元素位于R1 + V2(i)（即，V2为索引）
SVI	(R1+V2),V1	将V1存储到向量中，它的元素位于R1 + V2(i)（即，V2为索引）
CVI	V1,R1	创建索引向量，将值0,1 × R1,2 × R1,...,63 × R1存储到V1中
S--VV.D S--VS.D	V1,V2 V1,F0	对比V1和V2中的元素（EQ、NE、GT、LT、GE、LE）。如果条件为真，则将在相应的位向量中放入1；否则放入0。将所得到的位向量放在向量遮罩寄存器中（VM）。指令S--VS.D执行同样的对比操作，但它的一个操作数为标量值
POP	R1,VM	计算向量遮罩寄存器VM中1的个数，并将计数值存储到R1中
CVM		将向量遮罩寄存器设为全1
MTC1	VLR,R1	将R1的内容移到向量长度寄存器VL中
MFC1	R1,VLR	将向量长度VL的内容移到R1中
MVTM	VM,F0	将F0的内容移动到向量遮罩寄存器VM中
MVFM	F0,VM	将向量遮罩寄存器VM的内容移动到F0中

* 除了向量寄存器外，还有两个特殊寄存器 VLR 和 VM，下面将进行讨论。这些特殊寄存器假定存在于 MIPS 协处理器 1 空间中，与 FPU 寄存器位于一起。后面将解释带有步幅的运算以及索引创建及索引载入/存储操作的应用。

266

名字 LV 和 SV 表示向量载入和向量存储，它们载入或存储整个双精度数据向量。一个操作数是要载入或存储的向量寄存器，另一个操作数是 MIPS 通用寄存器，它是该向量在存储器中的起始地址。后面将会看到，除了这些向量寄存器之外，我们还需要两个通用寄存器：向量长度寄存器和向量遮罩寄存器。当向量长度不是 64 时使用前者，当循环中涉及 IF 语句时使用后者。

功率瓶颈使架构师非常看重具有以下特点的体系结构：一方面能够提供高性能，另一方面又不需要高度乱序超标量处理器的能耗与设计复杂度。向量指令天生就与这一趋势吻合，架构师可以用它们来提高简单循序标量处理器的性能，而又不会显著增大能耗要求和设计复杂度。在实践中，开发人员可以采用向量指令的方式来表达许多程序，采用数据级并行可以很高效地在复杂乱序设计中运行，Kozyrakis 和 Pattrson[2002]证明了这一点。

采用向量指令，系统可以采用许多方式对向量数据元素进行运算，其中包括对许多元素同时进行操作。因为有了这种灵活性，向量设计可以采用慢而宽的执行单元，以较低功率获得高性能。此外，向量指令集中各个元素是相互独立的，这样不需要进行成本高昂的相关性检查就能调整功能单元，而超标量处理器是需要进行这一检查的。

向量本身就可以容纳不同大小的数据。因此，如果一个向量寄存器可以容纳 64 个 64 位元素，那同样可以容纳 128 个 32 位元素、256 个 16 位元素，甚至 512 个 8 位元素。向量体系结构之所以既能用于多媒体应用，又能用于科学应用，就是因为具备这种硬件多样性。

4.2.2 向量处理器如何工作：一个示例

通过查看 VMIPS 的向量循环，可以更好地理解向量处理器。让我们来看一个典型的向量问题，在本节将一直使用这个例子：

```
Y = a × X + Y
```

X 和 Y 是向量，最初保存在存储器中，a 是一个标量。这个问题就是所谓的 SAXPY 或 DAXPY 循环，它们构成了 Linpack 基准测试的内层循环。SAXPY 表示“单精度 a × X 加 Y”（single-precision a × X plus Y）；DAXPY 表示“双精度 a × X 加 Y”（double precision a × X plus Y）。Linpack 是一组线性代数例程，Linpack 基准测试包括执行高斯消去法的例程。

现在假定向量寄存器的元素数或者说其长度为 64，与我们关心的向量运算长度匹配。（稍后将取消这一限制。）

267

例题 给出 DAXPY 循环的 MIPS 和 VMIPS 代码。假定 X 和 Y 的起始地址分别为 Rx 和 Ry。

解答 MIPS 代码如下。

```
        L.D     F0,a          ;载入标量 a
        DADDIU  R4,Rx,#512    ;要载入的最后地址
Loop:   L.D     F2,0(Rx)      ;载入 X[i]
        MUL.D   F2,F2,F0      ;a × X[i]
        L.D     F4,0(Ry)      ;载入 Y[i]
        ADD.D   F4,F4,F2      ;a × X[i] + Y[i]
        S.D     F4,9(Ry)      ;存储到 Y[i]
        DADDIU  Rx,Rx,#8      ;递增 X 的索引
        DADDIU  Ry,Ry,#8      ;递增 Y 的索引
        DSUBU   R20,R4,Rx     ;计算范围
        BNEZ    R20,Loop      ;检查是否完成
```

下面是 DAXPY 的 VMIPS 代码：

```
L.D        F0,a        ;载入标量 a
LV         V1,Rx       ;载入向量 X
MULVS.D    V2,V1,F0    ;向量-标量乘
LV         V3,Ry       ;载入向量 Y
ADDVV.D    V4,V2,V3    ;相加
SV         V4,Ry       ;存储结果
```

最引人注目的差别在于向量处理器大幅缩减了动态指令带宽，仅执行 6 条指令，而 MIPS 几乎要执行 600 条。这一缩减是因为向量运算是对 64 个元素执行的，在 MIPS 中差不多占据一半循环的开销指令在 VMIPS 代码中是不存在的。当编译器为这样一个序列生成向量指令时，所得到的代码会将大多数时间花费在向量运行模式中，我们将这种代码称为**已向量化或可向量化**。如果循环的迭代之间没有相关性（这种相关被称为**循环间相关**，见 4.5 节），那么这些循环就可以向量化。

MIPS 与 VMIPS 之间的另一个重要区别是流水线互锁的频率。在简单的 MIPS 代码中，每个 ADD.D 都必须等待 MUL.D，每个 S.D 都必须等待 ADD.D。在向量处理器中，每个向量指令只会因为等待每个向量的第一个元素而停顿，然后后续元素会沿着流水线顺畅流动。因此，每条向量指令仅需要一次流水线停顿，而不是每个向量元素需要一次。向量架构师将元素相关操作的转发称为**链接**（chaining），因为这些相关操作是被“链接”在一起的。在这个例子中，MIPS 中的流水线停顿频率大约比 VMIPS 高 64 倍。软件流水线或循环展开（参见附录 H）可以减少 MIPS 中的流水线停顿，但很难大幅缩减指令带宽方面的巨大差别。

4.2.3 向量执行时间

向量运算序列的执行时间主要取决于 3 个因素：(1)操作数向量的长度；(2)操作之间的结构冒险；(3)数据相关。给定向量长度和**初始速率**（初始速率就是向量单元接受新操作数并生成新结果的速率），我们可以计算一条向量指令的执行时间。所有现代向量计算机都有具备多条并行流水线（或**车道**）的向量功能单元，它们在每个时钟周期可以生成两个或更多个结果，但这些计算机还可能拥有一些未完全流水化的功能单元。为简便起见，我们的 VMIPS 实现方式有一条车道，各个操作的初始速率为每个时钟周期一个元素。因此，一条向量指令的执行时间（以时钟周期为单位）大约就是向量长度。 268

为了简化对向量执行和向量性能的讨论，我们使用了一种**护航指令组**（convoy）的概念，它是一组可以一直执行的向量指令。稍后可以看到，我们可以通过计算护航指令组的数目来估计一段代码的性能。护航指令组中的指令**不能包含任何结构性冒险**，如果存在这种冒险，则需要在不同护航指令组中序列化和启动这些指令。为了保持分析过程的简单性，假定在开始执行任意其他指令（标量或向量）之前，护航指令都必须已经完成。

除了具有结构性冒险的向量指令序列之外，具有写后读相关冒险的序列也应该位于不同护航指令组中，但通过链接操作可以允许它们位于同一护航指令组中。

链接操作允许向量操作在其向量源操作数的各个元素变为可用状态之后立即启动：链中第一个功能单元的结果被“转发”给第二个功能单元。在实践中经常采用以下方式来实现链接：允许处理器同时读、写一个特定的向量寄存器，不过读写的是不同元素。早期的链接实现类似于标量流水线中的转发，但这限制了链中源指令与目标指令的定时。最近的链接实现采用灵活

链接，这种方式允许向量指令链接到几乎任意其他活动向量指令，只要不生成结构性冒险就行。所有现代向量体系结构都支持灵活链接，这也是本章的假设之一。

为了将护航指令组转换为执行时间，我们需要有一种定时度量，用来估计护航指令组的时间。这种度量被称为**钟鸣**（chime），就是执行护航指令组所花费的时间单位。执行由 m 个护航指令组构成的向量序列需要 m 次钟鸣。当向量长度为 n 时，对于 VMIPS 来说，大约为 $m \times n$ 个时钟周期。钟鸣近似值忽略处理器特有的一些开销，许多此类开销都依赖于向量长度。因此，以钟鸣为单位测量时间时，对于长向量的近似要优于对短向量的近似。我们将使用钟鸣测量结果（而不是每个结果的时钟周期），用来明确表示忽略了特定的开销。

如果知道向量序列中的护航指令组数，那就知道了用钟鸣表示的执行时间。在以钟鸣为单位测试执行时间时，所忽略的一个开销源是对单个时钟周期内启动多条向量指令的限制。如果在一个时钟周期内只能启动一条向量指令（大多数向量处理器都是如此），那钟鸣数会低估护航指令组的实际执行时间。由于向量的长度通常远大于护航指令组中的指令数，所以简单地假定这个护航指令组是在一次钟鸣中执行的。

269

例题 给出以下代码序列在护航指令组中是如何排列的，假定每个向量功能单元只有一个副本：

```
LV          V1,Rx        ;载入向量 X
MULVS.D     V2,V1,F0     ;向量-标量乘
LV          V3,Ry        ;载入向量 Y
ADDVV.D     V4,V2,V3     ;两个向量相加
SV          V4,Ry        ;存储所得之和
```

这个向量序列将花费多少次钟鸣？每个 FLOP（浮点运算）需要多少个时钟周期（忽略向量指令发射开销）？

解答 第一个护航指令组从第一个 LV 指令处开始。MULVS.D 依赖于第一个 LV，但链接操作允许它位于同一护航指令组中。

第二个 LV 指令必须放在另一个护航指令组中，因为它与上一个 LV 指令的载入/存储单元存在结构性冒险。ADDVV.D 与第二个 LV 相关，但它也可以通过链接操作位于同一护航指令组中。最后，SV 与第二个护航指令组中的 LV 存在结构冒险，所以必须把它放在第三护航指令组中。通过这一分析，将得出向量指令在护航指令组的如下排列：

```
1. LV        MULVS.D
2. LV        ADDVV.D
3. SV
```

这个序列需要 3 个护航指令组。由于这一序列需要 3 次钟鸣，而且每个结果有 2 个浮点运算，所以每个 FLOP 的时钟周期数目为 1.5（忽略任何向量指令发射开销）。注意，尽管我们允许 LV 和 MULVS.D 都在第一护航指令组中执行，但大多数向量计算机将需要两个时钟周期来启动这些指令。

这个例子表明，钟鸣近似值对于长向量是相当准确的。例如，对于包括 64 个元素的向量来说，用钟鸣表示的时间为 3，所以这个序列将需要大约 $64 \times 3=192$ 个时钟周期。在两个分离时钟周期中发射护航指令组的开销很小。

另一个开销源要比发射限制明显得多。钟鸣模型中忽略的最重要开销源就是向量**启动时间**。启动时间主要由向量功能单元的流水线延迟决定。对于 VMIPS，我们使用与 Cray-1 相同的流水线深度，不过在更多的现代处理器中，这些延迟有增加的趋势，特别是向量载入操作的延迟。所有功能单元都被完全流水化。浮点加的流水线深度为 6 个时钟周期、浮点乘为 7 个、浮点除为 20 个、向量载入为 12 个。 270

有了这些向量基础知识之后，接下来的几小节将介绍一些优化方式，或者用来提高性能，或者增加可以在向量体系结构中完美运行的程序类型。具体来说，它们将回答如下问题。

- 向量处理器怎样执行单个向量才能快于每个时钟周期一个元素？每个时钟周期处理多个元素可以提高性能。
- 向量处理器如何处理那些向量长度与向量寄存器长度（对于 VMIPS，此长度为 64）不相同的程序？由于大多数应用程序向量与体系结构向量长度不匹配，所以需要一种高效的解决方案来处理这一常见情景。
- 如果要向量化的代码中含有 IF 语句，会发生什么？如果可以高效地处理条件语句，就能向量化更多的代码。
- 向量处理器需要从存储器系统中获得什么？没有充分的存储器带宽，向量执行可能会徒劳无益。
- 向量处理器如何处理多维矩阵？为使向量体系结构能够很好地工作，必须对这个常见数据结构进行向量化。
- 向量处理器如何处理稀疏矩阵？这一常见数据结构也必须进行向量化。
- 如何为向量计算机进行编程？如果体系结构方面的创新不能与编译器技术相匹配，那可能不会被广泛应用。

这一节的后续部分将分别介绍向量体系结构的这些优化技术，附录 G 将更深入地讨论。

4.2.4 多条车道：每个时钟周期超过一个元素

向量指令集的一个重要好处是它允许软件仅使用一条很短的指令就能向硬件传送大量并行任务。一条向量指令可以包含数十个独立运算，而其编码使用的位数与一条传统的标量指令相同。向量指令的并行语义允许实现方式在执行这些元素运算时使用：深度流水化的功能单元（就像我们目前研究过的 VMIPS 实现方式一样）；一组并行功能单元；或者并行与流水线功能单元的组合方式。图 4-3 说明如何使用并行流水线来执行一个向量加法指令，从而提高向量性能。

VMIPS 指令集有一个特性：所有向量算术指令只允许一个向量寄存器的元素 N 与其他向量寄存器的元素 N 进行运算。这一特性极大地简化了一个高度并行向量单元的构造，将其结构设定为多个并行**车道**。和高速公路一样，我们可以通过添加更多车道来提高向量单元的峰值吞吐量。图 4-4 给出了一种四车道向量单元的结构。这样，从单车道变为四车道之后，将一次钟鸣的时钟周期数由 64 个变为 16 个。由于多车道非常有利，所以应用程序和体系结构都必须支持长向量；否则，它们的快速执行速度会耗尽指令带宽，这需要 ILP 技术（见第 3 章）提供足够的向量指令。 271

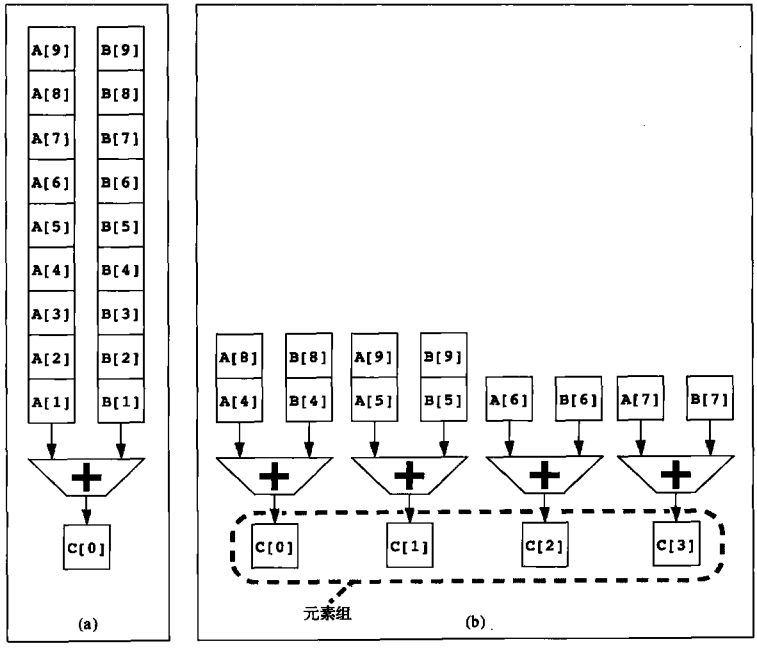


图 4-3 使用多个功能单元提高单个向量加法指令 C=A+B 的性能。左边的向量处理器(a)有一条加法流水线，每个时钟周期可以完成一次加法。右边的向量处理器(b)有 4 条加法流水线，每个时钟周期可以完成 4 次加法。一条向量加法指令中的元素交错存在于 4 条流水线中。通过这些流水线结合在一起的元素集被称为元素组（element group）（此图经 Asanovic[1998]许可后复制。）

每个车道都包含向量寄存器堆的一部分和来自每个向量功能单元的一个执行流水线。每个向量功能单元使用多条流水线，以每个时钟周期一个元素组的速度执行向量指令，每个车道一条流水线。第一个车道保存所有向量寄存器的第一个元素（元素 0），所以任何向量指令的第一个元素都会将其源操作数与目标操作数放在第一车道中。这种分配方式使该车道本地的算术流水线无须与其他车道通信就能完成运算。主存储器的访问也只需要车道内的连接。通过避免车道间的通信减少了构建高并行执行单元所需要的连接成本与寄存器堆端口，有助于解释向量计算机为什么能够在每个时钟周期内完成高达 64 个运算（跨越 16 个车道的 2 个算术单元和 2 个载入/存储单元）。

272

增加多个车道是一种提高向量性能的常见技术，它不需要增加太多控制复杂性，也不需要对现有机器代码进行修改。它还允许设计人员在晶片面积、时钟频率、电压和能耗之间进行权衡，而且不需要牺牲峰值性能。如果向量处理器的时钟频率减半，只需要使车道数目加倍就能保持原性能。

273

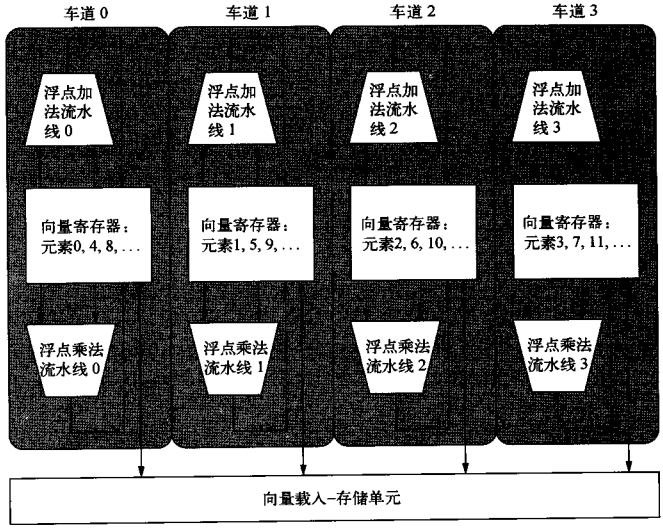


图 4-4 包含 4 个车道的向量单元的结构。向量寄存器存储分散在各个车道中，每个车道保存每个向量寄存器每 4 个元素中的 1 个。此图显示了三个向量功能单元：一个浮点加法、一个浮点乘法和一个载入-存储单元。向量算术单元各包含 4 条执行流水线，每个车道 1 条，它们共同完成一条向量指令。注意，向量寄存器堆的每一部分只需要为其车道本地的流水线提供足够的端口即可。本图没有给出为向量-标量指令提供标量操作数的路径，而是由标量处理器（或控制处理器）向所有车道广播标量值

4.2.5 向量长度寄存器：处理不等于 64 的循环

向量寄存器处理器有一个自然向量长度，这一长度由每个向量寄存器中的元素数目决定。对于 VMIPS 来说，这一长度为 64，它不大可能与程序中的实际向量长度相匹配。此外，在实际程序中，特定向量运算的长度在编译时通常是未知的。事实上，一段代码可能需要不同的向量长度。例如，考虑以下代码：

```
for (i=0; i <n;  i=i+1)
    Y[i] = a * X[i] + Y[i];
```

所有这些向量运算的大小都取决于 n，而它的取值不可能在运行之前获知。n 的值还可能是某个过程（该过程中包含上述循环）的参数，从而会在执行时发生变化。

对这些问题的解决方案就是创建一个**向量长度寄存器**（VLR）。VLR 控制所有向量运算的长度，包括向量载入与存储运算。但 VLR 中的值不能大于向量寄存器的长度。只要实际长度小于或等于**最大向量长度**（MVL），就能解决上述问题。MVL 确定了体系结构的一个向量中的数据元素数目。这个参数意味着向量寄存器的长度可以随着计算机的发展而增大，不需要改变指令集；在下一节将会看到，多媒体 SIMD 扩展没有与 MVL 对等的参数，所以在每次增大向量长度时都会改变指令集。

如果 n 的值在编译时未知，从而可能大于 MVL，那该怎么办呢？为了解决向量长于最大长度的第二问题，可以使用一种名为**条带挖掘**（strip mining）的技术。条带挖掘是指生成一些代码，使每个向量运算都是针对小于或等于 MVL 的大小来完成的。我们创建两个循环，一个循环处理迭代数为 MVL 倍数的情况，另一个循环处理所有其他迭代及小于 MVL 的情况。在实践中，编译器通常会生成一个条带挖掘循环，为其设定一个参数，通过改变长度来处理这两种情况。我们以 C 语言给出 DAXPY 循环的条带挖掘版本：

```
low = 0;
VL = (n % MVL); /*使用求模运算%找出不规则大小部分*/
for (j = 0; j <= (n/MVL);  j=j+1) { /*外层循环*/
    for (i = low; i < (low+VL);  i=i+1) /*执行长度 VL */
        Y[i] = a * X[i] + Y[i] ; /*主运算*/
     low = low + VL; /*开始下一个向量*/
     VL = MVL; /*将长度复位为最大向量长度*/
}
```

274

n/MVL 项表示截短整数除法。这一循环的效果是将向量分段，然后由内层循环进行处理。第一段的长度为(n %MVL)，所有后续段的长度为 MVL。图 4-5 说明如何将这个长向量分到各个段中。

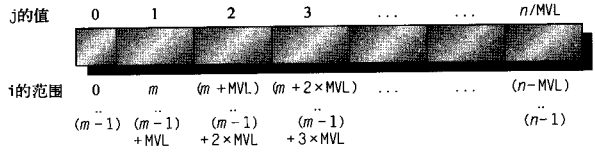


图 4-5 用条带处理的任意长度的向量。除第一块外，所有其他块的长度都是 MVL，充分利用了向量处理器的功能。本图中使用变量 *m* 来表示表达式(n % MVL)（C 运算符%表示求模。）

以上代码的内层循环可以进行向量化，长度为 VL，或者等于(n % MVL)，或者等于 MVL。在此代码中，必须对 VLR 寄存器设置两次，也就是在代码中为变量 VL 进行赋值时各设置一次。

4.2.6 向量遮罩寄存器：处理向量循环中的IF语句

根据 Amdahl 定律，我们知道对于中低向量化级别的程序，加速比是非常有限的。循环内部存在条件（IF 语句）、稀疏矩阵是向量化程度较低的两个主要原因。如果程序的循环中包含 IF 语句，由于 IF 语句会在循环中引入控制相关，所以不能使用前面讨论的技术以向量模式运行这种程序。同样，利用前面看到的各项功能也不能高效地实现稀疏矩阵。我们现在将讨论处理条件执行的策略，稀疏矩阵留待后文讨论。

考虑以 C 语言编写的以下循环：

```
for (i = 0; i < 64;  i=i+1)
  if (X[i] != 0)
    X[i] = X[i] - Y[i];
```

由于这一循环体需要条件执行，所以它通常是不能进行向量化的；但是，如果对于 X[i] ≠ 0 的迭代可以运行内层循环，那就可以实现减法的向量化。

这一功能的常见扩展称为**向量遮罩控制**。遮罩寄存器可以用来实现一条向量指令中每个元素运算的条件执行。向量遮罩控制使用布尔向量来控制向量指令的执行，就像条件执行指令使用布尔条件来决定是否执行标量指令一样。在启用**向量遮罩寄存器**时，任何向量指令都只会针

275

对符合特定条件的向量元素来执行，即这些元素在向量遮罩寄存器中的相应项目为 1。目标向量寄存器中的其他项目（在遮罩寄存器中的相应项目为 1）不受这些向量操作的影响。清除向量遮罩寄存器会将其置为全 1，后续向量指令将针对所有向量元素执行。我们现在可以为以上循环使用下列代码，假定 X、Y 的起始地址分别为 Rx 和 Ry：

```
LV          V1,Rx        ;将向量 X 载入 V1
LV          V2,Ry        ;载入向量 Y
L.D         F0,#0        ;将浮点零载入 F0
SNEVS.D     V1,F0        ;若 V1(i)!=F0，则将 VM(i)设置为 1
SUBVV.D     V1,V1,V2     ;在向量遮罩下执行减法
SV          V1,Rx        ;将结果存到 X 中
```

编译器写入程序调用转换过程，使用条件执行 IF 转换将 IF 语句修改为直行代码序列。

但是，使用向量遮罩寄存器是有开销的。对于标量体系结构，条件执行的指令在不满足条件时也需要执行时间。不过，通过消除分支和有关的控制相关性，即使有时会做一些无用功，也可以加快条件指令的执行速度。与此类似，对于采用向量遮罩执行的向量指令，即使遮罩为 0 的元素，仍然会占用相同的执行时间。与此类似，即使遮罩中有大量 0，使用向量遮罩控制的速度也仍然远快于使用标量模式的速度。

在 4.4 节将会看到，向量处理器与 GPU 之间的一个区别就是它们处理条件语句的方式不同。向量处理器将遮罩寄存器作为体系结构状态的一部分，依靠编译器来显式操控遮罩寄存器。而 GPU 则是使用硬件来操控 GPU 软件无法看到的内部遮罩寄存器，以实现相同效果。在这两种情况下，无论遮罩是 1 还是 0，硬件都需要花费时间来执行向量元素，所以 GFLPS 速率在使用遮罩时会下降。

4.2.7 内存组：为向量载入/存储单元提供带宽

载入/存储向量单元的行为要比算术功能单元的行为复杂得多。载入操作的开始时间就是它从存储器向寄存器中载入第一个字的时间。如果在无停顿情况下提供向量的其他元素，那么向量初始化速率就等于提取或存储新字的速度。这一初始化速率不一定是一个时钟周期，因为存储器组的停顿可能会降低有效吞吐量，这一点不同于较简单的功能单元。

一般情况下，载入/存储单元的起始代价要高于算术单元的这一代价——在许多处理器中要多于 100 个时钟周期。对于 VMIPS，我们假定起始时间为 12 个时钟周期，与 Cray-1 相同。（最近的向量计算机使用缓存来降低向量载入与存储的延迟。）

276

为了保持每个时钟周期提取或存储一个字的初始化速率，存储器系统必须能够生成或接受这么多的数据。将访问对象分散在多个独立的存储器组中，通常可以保证所需速率。稍后将会看到，拥有大量存储器组可以很高效地处理那些访问多行或多列数据的向量载入或存储指令。

大多数向量处理器都使用存储器组，允许进行多个独立访问，而不是进行简单的存储器交错，其原因有以下 3 个。

(1) 许多向量计算机每个时钟周期可以进行多个载入或存储操作，存储器组的周期时间通常比处理器周期时间高几倍。为了支持多个载入或存储操作的同时访问请求，存储器系统需要有多个组，并能够独立控制对这些组的寻址。

(2) 大多数向量处理器支持载入或存储非连续数据字的功能。在此类情况下，需要进行独立的组寻址，而不是交叉寻址。

(3) 大多数向量计算机支持多个共享同一存储器系统的处理器，所以每个处理器会生成其自己的独立寻址流。

这些特征综合起来，就有了大量的独立存储器组，如下例所示。

例题 Cray T90（Cray T932）的最高配置有32个处理器，每个处理器每个时钟周期可以生成4个载入操作和2个存储操作。处理器时钟周期为2.167 ns，而存储器系统所用SRAM的周期时间为15 ns。计算：为使所有处理器都以完全存储器带宽运行，最少需要多少个存储器组。

解答 每个时钟周期产生的最大存储器引用数目为192：每个处理器每个时钟周期产生6次引用共32个处理器。每个SRAM组的繁忙时钟周期数为15/2.167=6.92，四舍五入为7个处理器时钟周期。因此，至少需要192×7＝1344个存储器组！

Cray T932实际上有1024个存储器组，所以早期型号不能让所有处理器都同时维持完全带宽。后来对存储器进行升级时，用流水化同步SRAM代替了15 ns的异步SRAM，存储器周期时间缩短一半，从而可以提供足够的带宽。

从更高一级的角度来看，向量载入/存储单元与向量处理器中的预取单元扮演着类似的角色，它们都是通过向处理器提供数据流来尝试提供数据带宽。

277

4.2.8 步幅：处理向量体系结构中的多维数组

向量中的相邻元素在内存中的位置可能并不一定是连续的。考虑下面一段非常简单的矩阵乘法C语言代码：

```
for (i = 0; i < 100;  i=i+1)
   for (j = 0; j < 100;  j=j+1) {
      A[i][j] = 0.0;
      for (k = 0; k < 100; k=k+1)
         A[i][j] = A[i][j] + B[i][k] * D[k][j];
   }
```

我们可以将B的每一行与D的每一列的乘法进行向量化，以k为索引变量对内层循环进行条带挖掘。

为此，我们必须考虑如何对B中的相邻元素及D中的相邻元素进行寻址。在为数组分配内存时，该数组是线性化的，其排序方式要么以行为主（如C语言），要么以列为主（如Fortran语言）。这种线性化意味着：要么行中的元素在内存中不相邻，要么列中的元素在内存中不相邻。例如，上面的C代码是按照以行为主的排序来分配内存的，所以内层循环中各次迭代在访问D元素时，这些元素之间的间隔等于行大小乘以8（每一项的字节数），共为800个字节。在第2章中，我们已经知道在基于缓存的系统中通过分层有可能提高局域性。对于没有缓存的向量处理器，需要使用另一种方法来提取向量在内存中不相邻的元素。

对于那些要收集到一个寄存器中的元素，它们之间的距离称为步幅。在这个例子中，矩阵D的步幅为100个双字（800个字节），矩阵B的步幅可能为1个双字（8个字节）。对于以列为主的排序（Fortran语言采用这一顺序），这两个步幅的大小会颠倒过来。矩阵D的步幅为1，也就是说连续元素之间相隔1个双字（8个字节），而矩阵B的步幅为100，也就是100个双字（800个字节）。因此，如果不对循环进行重新排序，编译器就不能隐藏矩阵B和D中连续元素之间的较长距离。

将向量载入向量寄存器后，它的表现行为就好像它的元素在逻辑上是相邻的。仅利用具有步幅功能的向量载入及向量存储操作，向量处理器可以处理大于 1 的步幅，这种步幅称为**非单位步幅**。向量处理器的主要优势之一就是能够访问非连续存储器位置，并对其进行调整，放到一个密集结构中。缓存在本质上是处理单位步幅数据的。增大块大小有助于降低大型科学数据集（其步幅为单位步幅）的缺失率，但增大块大小也可能会对那些以非单位步幅访问的数据产生负面影响。尽管分块技术可以解决其中一些问题（见第 2 章），但在 4.7 节将会看到，高效访问非连续数据的功能仍然是向量处理器的一个优势。

在 VMIPS 结构中，可寻址单位为 1 个字节，所以我们示例的步幅将为 800。由于矩阵的大小在编译时可能是未知的，或者就像向量长度一样，在每次执行相同语句时可能会发生变化，所以必须对步幅值进行动态计算。向量步幅可以和向量起始地址一样，放在通用寄存器中。然后，VMIPS 指令 LVWS（load vector with stride）将向量提取到向量寄存器中。同样，在存储非单位步幅向量时，使用指令 SVWS（store vector with stride）。 278

为了支持大于 1 的步幅，会使存储器系统变得复杂。在引入非单位步幅之后，就有可能频繁访问同一个组。当多个访问对一个存储器组产生竞争时，就会发生存储器组冲突，从而使某个访问陷入停顿。如果满足以下条件则会产生组冲突，从而产生停顿。

$$\frac{\text{组数}}{\text{步幅与组数的最小公倍数}} < \text{组繁忙时间}$$

例题 假定有 8 个存储器组，组繁忙时间为 6 个时钟周期，总存储器延迟为 12 个时钟周期。要以步幅 1 完成一个 64 元素的向量载入操作，需要多长时间？步幅为 32 呢？

解答 由于组数大于组繁忙时间，所以当步幅为 1 时，该载入操作将耗费 12+64=76 个时钟周期，也就是每个元素需要 1.2 个时钟周期。最糟糕的步幅是存储器组数目的倍数，在本例中就是步幅为 32、存储器组为 8 的情况。（在第一次访问之后，）对存储器的每次访问都会与上一次访问发生冲突，必须等候长度为 6 个时钟周期的组繁忙时间。总时间为 12+1+6×63=391 个时钟周期，即每个元素 6.1 个时钟周期。

4.2.9 集中-分散：在向量体系结构中处理稀疏矩阵

前面曾经提到，稀疏矩阵是很常见的，所以非常需要一些技术，能够以向量模式运行那些处理稀疏矩阵的程序。在稀疏矩阵中，向量的元素通常以某种紧凑形式存储，然后对其进行间接访问。假定有一种简化的稀疏结构，我们可能会看到类似下面的代码：

```
for (i = 0; i < n;  i=i+1)
   A[K[i]] = A[K[i]] + C[M[i]];
```

这一代码实现数组 A 与数组 C 的稀疏向量求和，用索引向量 K 和 M 来指出 A 与 C 中的非零元素。（A 和 C 的非零元素数必须相等，共有 n 个，所以 K 和 M 的大小相同。）

用于支持稀疏矩阵的主要机制是采用索引向量的**集中-分散操作**。这种运算的目的是支持在稀疏矩阵的压缩表示（即不包含零）和正常表示（即包含零）之间进行转换。**集中**操作是取得**索引向量**，并在此向量中提取元素，元素位置的确定是将基础地址加上索引向量中给定的偏移量。其结果是向量寄存器中的一个密集向量。在以密集形式对这些元素进行操作之后， 279

再使用同一索引向量，通过**分散存储操作**，以扩展方式存储这一稀疏向量。对此类操作的硬件支持被称为**集中-分散**，几乎所有现代向量处理器都具备这一功能。VMIPS 指令为 LVI（载入索引向量，也就是集中）和 SVI（存储索引向量，也就是分散）。例如，如果 Ra、Rc、Rk 和 Rm 中包含以上序列中向量的起始地址，就可以用向量指令来对内层循环进行编码，如下所示：

```
LV        Vk, Rk        ;载入 K
LVI       Va, (Ra+Vk)   ;载入 A[K[]]
LV        Vm, Rm        ;载入 M
LVI       Vc, (Rc+Vm)   ;载入 C[M[]]
ADDVV.D   Va, Va, Vc    ;求和
SVI       (Ra+Vk), Va   ;存储 A[K[]]
```

这一技术允许以向量模式运行带有稀疏矩阵的代码。简单的向量化编译器可能无法自动实现以上源代码的向量化，因为编译器可能不知道 K 的元素是离散值，因此也就不存在相关性。相反，应当由程序员发出的指令告诉编译器，可以放心地以向量模式来运行这一循环。

尽管索引载入与存储（集中与分散）操作都可以流水化，但由于存储器组在开始执行指令时是未知的，所以它们的运行速度通常远低于非索引载入或存储操作。每个元素都有各自的地址，所以不能对它们进行分组处理，在存储器系统的许多位置都可能存在冲突。因此，每次访问都会招致严重的延迟。但是，如 4.7 节所示，如果架构师不是对此类访问采取放任态度，而是针对这一情景进行设计，使用更多的硬件资源，那存储器系统就能提供更好的性能。

在 4.4 节将会看到，在 GPU 中，所有载入操作都是集中，所有存储都是分散。为了避免在常见的单位步幅情景中缓慢运行，应当由 GPU 程序员来确保一次集中或分散操作中的所有地址都处于相邻位置。此外，GPU 硬件在执行时间必须能够识别这些地址序列，将集中与分散操作转换为更高效的存储器单位步幅访问。

4.2.10 向量体系结构编程

向量体系结构的优势在于编译器可以在编译时告诉程序员：某段代码是否可以向量化，通常还会给出一些暗示，说明这段代码为什么不能向量化。这种简单的执行模型可以让其他领域的专家了解如何通过修改自己的代码来提高性能，如果可以假定操作之间是相互独立的（比如集中-分散数据传送操作），那也可以给编译器提供一点提示。这就是编译器与程序员之间的对话，每一方都为对方提供一些关于如何提高性能的线索，从而简化向量计算的编程。

今天，以向量模式运行的程序能否成功，其主要影响因素是程序本身的结构：循环是否有真正的数据相关（见 4.5 节），能否调整它们的结构，使其没有此类相关？这一因素受选定算法的影响，在一定程度上还受编码方式的影响。

让我们看一下在 Perfect Club 基准测试中观测到的向量化水平，用以指示科学程序中所能实现的向量化水平。表 4-2 显示了两种代码版本在 Cray Y-MP 上运行时，以向量模式运行的运算比例。第一个版本仅对原代码进行了编译器优化，而第二个版本则利用了 Cray Research 程序员团队给出的一些提示。对向量处理器上的应用程序性能进行多次研究后发现，编译器向量化水平的变化范围很大。

表 4-2 在 Cray Y-MP 上执行 Perfect Club 基准测试所获得的向量化水平[Vajapeyam 1991]

基准测试名称	以向量模式执行的运算，编译器优化	以向量模式执行的运算，有程序员提供帮助	根据提示进行优化后获得的加速比
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

* 第一列显示在没有提示下用编译器获得的向量化水平，而第二列是在根据 Cray Research 程序员团队的提示对代码进行改进后的结果。

281

对于编译器自身不能很好地完成向量化的代码来说，根据大量提示进行修改后的版本会大幅提高向量化水平，现在有超过 50%的代码可以进行向量化了。平均向量化水平从大约 70%提高至大约 90%。

4.3 SIMD 指令集多媒体扩展

SIMD 多媒体扩展源于一个很容易观察到的事实：许多媒体应用程序操作的数据类型要比对 32 位处理器进行针对性优化的数据类型更窄一些。许多图形系统使用 8 位来表示三基色中的每一种颜色，再用 8 位来表示透明度。根据不同的应用程序，音频采样通常用 8 位或 16 位来表示。假定有一个 256 位加法器，通过划分这个加法器中的进位链，处理器可以同时对一些短向量进行操作，这些向量可以是 32 个 8 位操作数、16 个 16 位操作数、8 个 32 位操作数或者 4 个 64 位操作数。这些经过划分的加法器的额外成本很小。表 4-3 总结了典型的多媒体 SIMD 指令。和向量指令一样，SIMD 指令规定了对数据向量的相同操作。一些向量机器拥有大型寄存器堆，比如 VMIPS 向量寄存器，8 个向量寄存器中的每一个都可以保存 64 个 64 位元素，SIMD 指令与之不同，它指定的操作数较少，因此使用的寄存器堆也较小。

向量体系结构专门针对向量化编译器提供了一流的指令集，与之相对，SIMD 扩展主要进行了以下 3 项简化。

- 多媒体 SIMD 扩展固定了操作代码中数据操作数的数目，从而在 x86 体系结构的 MMX、SSE 和 AVX 扩展中添加了数百条指令。向量体系结构有一个向量长度寄存器，用于指定当前操作的操作数个数。一些程序的向量长度小于体系结构的最大支持长度，由于这些向量寄存器的长度可以变化，所以也能够很轻松地适应此类程序。此外，向量体系结构有一个隐含的最大向量长度，它与向量长度寄存器相结合，可以避免使用大量操作码。 282
- 多媒体 SIMD 没有提供向量体系结构的更复杂寻址模式，也就是步幅访问和集中-分散访问。这些功能增加了向量编译器成功向量化的程序数目（见 4.7 节）。

- 多媒体SIMD通常不会像向量体系结构那样，为了支持元素的条件执行而提供遮罩寄存器。这些省略增大了编译器生成SIMD代码的难度，也加大了SIMD汇编语言编程的难度。

表4-3 典型SIMD多媒体运算汇总（运算宽度为256位）

指令类别	操作数
无符号加/减	32个8位、16个16位、8个32位或4个64位
最大/最小	32个8位、16个16位、8个32位或4个64位
平均	32个8位、16个16位、8个32位或4个64位
右/左移位	32个8位、16个16位、8个32位或4个64位
浮点	16个16位、8个32位、4个64位或2个128位

* 注意，IEEE 754-2008浮点标准增加了半精度（16位）和四精度（128位）浮点运算。

对于x86体系结构，1996年增加的MMX指令重新确定了64位浮点寄存器的用途，所以基本指令可以同时执行8个8位运算或4个16位运算。这些指令与其他各种指令结合在一起，包括并行MAX和MIN运算、各种遮罩和条件指令、通常在数字信号处理器中进行的运算以及人们相信在重要媒体库中有用的专用指令。注意，MMX重复使用浮点数据传送指令来访问存储器。

1999年推出的后续流式SIMD扩展（SSE）添加了原来宽128位的独立寄存器，所以现在的指令可以同时执行16个8位运算、8个16位运算或4个32位运算。它还执行并行单精度浮点运算。由于SSE拥有独立寄存器，所以它需要独立的数据传送指令。Intel很快在2001年的SSE2、2004年的SSE3和2007年的SSE4中添加了双精度SIMD浮点数据类型。拥有四个单精度浮点运算或两个并行双精度运算的指令提高了x86计算机的峰值浮点性能，只要程序员将操作数并排放在一起即可。在每一代计算机中都添加了一些专用指令，用于加快一些重要的特定多媒体功能的速度。

2010年增加的高级向量扩展（AVX）再次将寄存器的宽度加倍，变为256位，并提供了一些指令，将针对所有较窄数据类型的运算数目翻了一番。表4-4给出了可用于进行双精度浮点计算的AVX指令。AVX进行了一些准备工作，以便在将来的体系结构中将宽度扩展到512位和1024位。

表4-4 在双精度浮点程序中有用的x86体系结构AVX指令

AVX指令	说明
VADDPD	加上4个紧缩双精度操作数
VSUBPD	减去4个紧缩双精度操作数
VMULPD	乘以4个紧缩双精度操作数
VDIVPD	除以4个紧缩双精度操作数
VFMADDPD	乘、加4个紧缩双精度操作数
VFMSUBPD	乘、减4个紧缩双精度操作数
VCMPxx	对比4个紧缩双精度操作数，结果为EQ、NEQ、LT、LE、GT、GE，等等
VMOVAPD	移动对齐的4个紧缩双精度操作数
VBROADCASTSD	将一个多精度操作数广播至256位寄存器中的4个位置

* 256位AVX的紧缩双精度是指以SIMD模式执行的4个64位操作数。当AVX指令的宽度增大时，数据置换指令的添加也变得更为重要，以允许将来自宽寄存器中不同部分的窄操作数结合起来。AVX中的一些指令可以在256位寄存器中分散32位、64位或128位操作数。比如，BROADCAST在AVX寄存器中将一个64位操作数复制4次。AVC还包含大量结合在一起的乘加/乘减指令，这里仅给出了其中的两个。

一般来说，这些扩展的目的是加快那些精心编制的库函数运行速度，而不是由编译器来生成这些库（参见附录 H），但近来的 x86 编译器正在尝试生成此类代码，尤其是针对浮点计算密集的应用程序。

既然有这些弱点，那多媒体 SIMD 扩展为什么还如此流行呢？第一，它们不需要花费什么成本就能添加标准算术单元，而且易于实施。第二，与向量体系结构相比，它们不需要什么额外状态，上下文切换次数总是要考虑这一因素。第三，需要大量存储器带宽来支持向量体系结构，而这是许多计算机所不具备的。第四，当一条能够生成 64 个存储器访问的指令在向量中间发生页面错误时，SIMD 不必处理虚拟内存中的问题。SIMD 扩展对于操作数的每个 SIMD 组使用独立的数据传送（这些操作数在存储器中是对齐的），所以它们不能跨越页面边界。固定长度的简短 SIMD "向量" 还有另一个好处：能够很轻松地引入一些符合新媒体标准的指令，比如执行置换操作的指令或者所用操作数少于或多于所生成向量的指令。最后，人们还关注向量体系结构在使用缓存方面的表现。最近的向量体系结构已经解决了所有这些问题，但由于过去一些缺陷的影响，架构师还是对向量抱有怀疑态度。 283

例题 为了了解多媒体指令是什么样子的，假定我们向 MIPS 中添加了 256 位 SIMD 多媒体指令。在这个例子中主要讨论浮点指令。对于一次能够对 4 个双精度运算数执行操作的指令添加后缀 "4D"。和向量体系结构一样，可以把 SIMD 处理器看作是拥有车道的处理器，在本例中为 4 个车道。MIPS SIMD 会重复利用浮点寄存器，作为 4D 指令的操作数，就像原始 MIP 中的双精度运算重复利用单精度寄存器一样。这一示例显示了 DAXPY 循环的 MIPS SIMD 代码。假定 X 和 Y 的起始地址分别为 Rx 和 Ry。用下划线划出为添加 SIMD 而对 MIPS 代码进行的修改。 284

解答 下面是 MIPS 代码：

```
        L.D     F0,a            ;载入标量 a
        MOV     F1, F0          ;将 a 复制到 F1,以完成 SIMD MUL
        MOV     F2, F0          ;将 a 复制到 F2,以完成 SIMD MUL
        MOV     F3, F0          ;将 a 复制到 F3,以完成 SIMD MUL
        DADDIU  R4,Rx,#512      ;要载入的最后一个地址
Loop:   L.4D    F4,0(Rx)        ;载入 X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D  F4,F4,F0        ;a·X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D    F8,0(Ry)        ;载入 Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D  F8,F8,F4        ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D    F8,0(Rx)        ;存储到 Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU  Rx,Rx,#32       ;将索引递增至 X
        DADDIU  Ry,Ry,#32       ;将索引递增至 Y
        DSUBU   R20,R4,Rx       ;计算范围
        BNEZ    R20,Loop        ;检查是否完成
```

这些修改包括将所有 MIPS 双精度指令用对应的 4D 等价指令代替，将递增步长由 8 变为 32，将寄存器由 F2 和 F4 改为 F4 和 F8，以在寄存器堆中为 4 个连续双精度操作数获取足够的空间。所以，对于标量 a，每个 SIMD 车道都将拥有自己的一个副本，我们将 F0 的值复制到寄存器 F1、F2 和 F3。（真正的 SIMD 指令扩展有一条指令，可以向组中的所有其他寄存器广播一个值。）因此，这一乘法将完成 F4*F0、F5*F1、F6*F2 和 F7*F3。尽管 SIMD MIPS 没有像 VMIPS 那样，将动态指令宽带降低 100 倍，但也降低了 4 倍，共有 149 条，而 MIPS 则为 578 条指令。

4.3.1 多媒体SIMD体系结构编程

由于SIMD多媒体扩展的特有本质，使用这些指令的最简便方法就是通过库或用汇编语言编写。

最近的扩展变得更加规整，为编译器提供了更为合理的目标。通过借用向量化编译器的技术，这些编译器也开始自动生成SIMD指令。例如，目前的高级编译器可以生成SIMD浮点指令，大幅提高科学代码的性能。但是，程序员必须确保存储器中的所有数据都与运行代码的SIMD单元的宽度对齐，以防止编译器为本来可以向量化的代码生成标量指令。

4.3.2 Roofline可视性能模型

有一种直观的可视方法可以对比各种SIMD体系结构变体的潜在浮点性能，那就是Roofline模型[Williams等人，2009]。它将浮点性能、存储器性能和运算密度汇总在一个两维图形中。**运算密度**等于浮点运算数与所访问存储器字节的比值。其计算方法为：获取一个程序的总浮点运算数，然后再除以在程序执行期间向主存储器传送的总数据字节。图4-6给出了几种示例内核的相对运算密度。

285

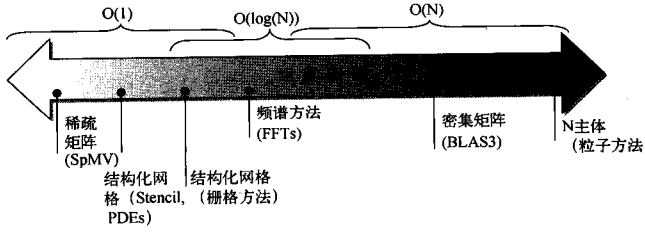


图 4-6 运算密度，定义为：运行程序时所执行的浮点运算数除以在主存储器中访问的字节数[Williams等人，2009]。一些内核的运算密度会随问题的规模（比如密集矩阵）而缩放，但有许多核心的运算密度与问题规模无关

峰值浮点性能可以使用硬件规范求得。这一实例研究中的许多核心都不能放到片上缓存中，所以峰值性能是由缓存背后的存储器系统确定的。注意，我们需要的是可供处理器使用的峰值存储器带宽，而不只是4.7节表4-13中DRAM管脚处的可用宽带。要求出（所提供的）峰值存储器性能，其中一种方法是运行Stream基准测试。

图4-7在左侧给出NEC SX-9向量处理器的Roofline模型，在右侧给出Intel Core i7 920多核计算机的相应模型。垂直的Y轴是可以实现的浮点性能，为2～256 GFLOP/s。水平的X轴是运算密度，在两个图中都是从1/8 FLOP/DARM访问字节到16 FLOP/DARM访问字节。注意，该图为对数-对数图尺，Roofline对于一种计算机仅完成一次。

对于一个给定内核，我们可以根据它的运算密度在X轴上找到一个点。如果过该点画一条垂线，此内核在该计算机上的性能必须位于该垂线上的某一位置。我们可以绘制一个水平线，显示该计算机的浮点性能。显然，由于硬件限制，实际浮点性能不可能高于该水平线。

如何绘制峰值存储器性能呢？由于X轴为FLOP/字节，Y轴为FLOP/s，所以字节/s就是

图中 45 度角的对角线。因此，我们可以画出第三条线，显示该计算机的存储器系统对于给定运算密度所能支持的最大浮点性能。我们可以用公式来表示这些限制，以绘制图 4-7 中的相应曲线： 286

可获得的 GFLOP/s=Min（峰值存储器带宽×运算密度，峰值浮点性能）

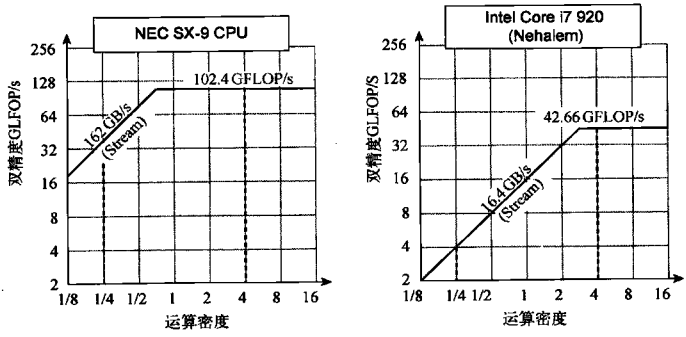


图 4-7 左图为 NEC SX-9 向量处理器上的 Roofline 模型，右图为采用 SIMD 扩展的 Intel Core i7 920 多核计算机的相应模型[Williams 等人，2009]。这个 Roofline 模型针对的单位步幅的存储器访问和双精度浮点性能。NEC SC-9 是在 2008 年发布的超级向量计算机，耗费了数百万美元。根据 Stream 基准测试，它的峰值 DP FP 性能为 102.4 GFLOP/s，峰值存储器宽度为 162GB/s。Core i7 920 的峰值 DP FP 性能为 42.66 GFLOP/s 和峰值存储器带宽为 16.4 GB/s。在运算密度为 4 FLOP/字节处的垂直虚线显示两个处理器都以峰值性能运行。在这个示例中，102.4 GFLOP/s 处的 Sx-9 要比 42.66 GFLOP/s 处的 Core i7 快 2.4 倍。在运算密度为 0.25 FLOP/字节处，SX-9 为 40.5 GFLOP/s，比 Core i7 的 4.1 GFLOP/s 快 10 倍

水平线和对角线给出了这个简单模型的名字，并指出了它的取值。Roofline 根据内核的运算密度设定了其内核的性能上限。如果我们把运算密度看作是触及房顶的柱子，它既可能触及房顶的平坦部分（表示这一性能是受计算功能限制的），也可能触及房顶的倾斜部分（表示这一性能最终受存储器带宽的限制）。在图 4-7 中，右侧的垂直虚线（运算密度为 4）是前者的示例，左侧的垂直虚线（运算密度为 1/4）是后者的示例。给定一台计算机的 Roofline 模型，就可以重复应用它，因为它是不会随内核变化的。

注意对角线与水平线交汇的“屋脊点”，通过它可以深入了解这台计算机的性能。如果它非常靠右，那么只有运算密度非常高的内核才能实现这台计算机的最大性能。如果它非常靠左，那么几乎所有内核都可能达到最高性能。后面将会看到，与其他 SIMD 处理器相比，这个向量处理器的存储器带宽要高得多，屋脊点非常靠左。

图 4-7 显示 SX-9 的峰值计算性能比 Core i7 快 2.4 倍，但存储器性能要快 10 倍。对于运算密度为 0.25 的程序，SX-9 快 10 倍（40.5 GFLOP/s 比 4.1 GFLOP/s）。更宽的存储器带宽将屋脊点从 Core i7 的 2.6 移动到 SX-9 的 0.6，这就意味着有更多的程序可以在这个向量处理器上达到峰值计算性能。 287

4.4 图形处理器

只需要几百美元，任何人都能买一个具有数百个并行浮点单元的GPU，从而更容易实现高性能计算。当GPU的计算潜力与一种简化GPU编程的编程语言相结合时，人们对GPU计算的兴趣大增。因此，当今许多科学与多媒体应用程序的程序员都在考虑是使用GPU还是使用CPU。

GPU和CPU在计算机体系结构谱系中不会上溯到同一个祖先；并没有哪个“过渡环节”可以解释这两者之间的关系。如4.10节中的介绍，GPU的祖先是图形加速器，而极强的图形处理能力正是GPU得以存在的原因。尽管GPU正在转向主流计算领域，但它们不能放弃继续在图形处理领域保持优异表现的责任。因此，对于能够出色处理图形的硬件，当架构师询问应当如何进行补充才能提高更广泛应用程序的性能时，GPU的设计就可能体现出更重要的价值。

注意，这一节主要讨论使用GPU进行计算。若要了解GPU计算如何与传统的图形加速角色相结合，请参阅John Nickolls和David Kirk等人的文章“图形与计算GPU”（本书作者编著的《计算机组成与设计》一书第4版的附录A）。

由于这一体系结构的术语和一些硬件功能都与向量和SIMD体系结构有很大不同，所以我们认为，在介绍这一体系结构之前，首先从GPU的一种简化编程模型入手会更容易一些。

4.4.1 GPU编程

> 关于如何表示算法中的并行，CUDA绝对是一种非常出色的解决方案，尽管不能表示所有算法中的并行，却也足够了。它在某种方式上与我们的思考与编码方式相吻合，可以更轻松、更自然地表达超越任务级别的并行。
>
> ——Vincent Natol
> “Kudos for CUDA”，*HPC Wire*（2010）

CPU程序员的挑战不只是在GPU上获得出色的性能，还要协调系统处理器与GPU上的计算调度、系统存储器与GPU存储器之间的数据传输。此外，在本节后面将会看到，GPU几乎拥有所有可以由编程环境捕获的并行类型：多线程、MIMD、SIMD，甚至还有指令级并行。

288

NVIDIA决定开发一种与C类似的语言和编程环境，通过克服异质计算及多种并行带来的双重挑战来提高GPU程序员的生产效率。这一系统的名称为CUDA，表示“计算统一设备体系结构”（Compute Unified Device Architecture）。CUDA为系统处理器（**主机**）生成C/C++，为GPU（**设备**，也就是CUDA中的D）生成C和C++方言。一种类似的编程语言是OpenGL，几家公司共同开发这一语言，为多种平台提供一种与供应商无关的语言。

NVIDIA认为，所有这些并行形式的统一主题就是**CUDA线程**。以这种最低级别的并行作为编程原型，编译器和硬件可以将数以千记的CUDA线程聚合在一起，利用CPU中的各种并行类型：多线程、MIMD、SIMD和指令级并行。因此，NVIDIA将CUDA编程模型定义为“单指令多线程”（SIMT）。这些线程进行了分块，在执行时以32个线程为一组，称为**线程块**，我们马上会明白其原因。我们将执行整个线程块的硬件称为**多线程SIMD处理器**。

我们只需要几个细节就能给出 CUDA 程序的示例。

- 为了区分 GPU（设备）的功能与系统处理器（主机）的功能，CUDA 使用__device__或__global__表示前者，使用__host__表示后者。
- 被声明为__device__或__global__functions 的 CUDA 变量被分配给 GPU 存储器（见下文），可以供所有多线程 SIMD 处理器访问。
- 对于在 GPU 上运行的函数 *name* 进行扩展函数调用的语法为：

  name<<<dimGrid, dimBlock>>>(... *parameter list* ...)

  其中 dimGrid 和 dimBlock 规定了代码的大小（用块表示）和块的大小（用线程表示）。
- 除了块识别符（blockIdx）和每个块的线程识别符（threadIdx）之外，CUDA 还为每个块的线程数提供了一个关键字（blockDim），它来自上一个细节中提到的 dimBlock 参数。

在查看 CUDA 代码之前，首先来看看 4.2 节 DAXPY 循环的传统 C 代码：

```
// 调用 DAXPY
daxpy(n, 2.0, x, y);
// C 语言编写的 DAXPY
void daxpy(int n, double a, double *x, double *y)
{
        for (int i = 0; i < n; ++i)
            y[i] = a*x[i] + y[i];
}
```

289

下面是 CUDA 版本。我们在一个多线程 SIMD 处理器中启动 n 个线程，每个向量元素一个线程，每个线程块 256 个 CUDA 线程。GPU 功能首先根据块 ID、每个块的线程数以及线程 ID 来计算相应的元素索引 i。只要这个索引没有超出数组的范围（i<n），它就会执行乘法和加法。

```
// 调用 DAXPY，每个线程块中有 256 个线程
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// CUDA 中的 DAXPY
__device__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

对比 C 代码和 CUDA 代码，我们可以看出一种用于实现数据并行 CUDA 代码并行化的共同模式。C 版本中有一个循环的所有迭代都与其他迭代相独立，可以很轻松地将这个循环转换为并行代码，其中每个循环迭代都变为一个独立线程。（前面曾经提到，向量化编译器也要求循环的迭代之间没有相关性，这种相关被称为循环间相关，4.5 节将详细介绍。）程序员通过明确指定网格大小及每个 SIMD 处理器中的线程数，明确指出 CUDA 中的并行。由于为每个元素都分配了一个线程，所以在向存储器中写入结果时不需要在线程之间实行同步。

行执行和线程管理由 GPU 硬件负责，而不是由应用程序或操作系统完成。为了简化硬件处理的排程，CUDA 要求线程块能够按任意顺序独立执行。尽管不同的线程块可以使用全局存储器中的原子存储器操作进行协调，但它们之间不能直接通信。

马上可以看到，许多 GPU 硬件概念在 CUDA 中不是非常明显。从程序员生产效率的角度来看，这是一件好事，但大多数程序员使用 GPU 而不是 CPU 来提高性能。重视性能的程序员在用 CUDA 编写程序时必须时刻惦记着 GPU 硬件。他们知道需要将控制流中的 32 个线程分为

一组，以从多线程SIMD处理器中获得最佳性能，并在每个多线程SIMD处理器中另外创建许多线程，以隐藏访问DRAM的延迟，稍后将解释其原因。它们还需要将数据地址保持在一个或一些存储器块的局部范围内，以获得所期望的存储器性能。

和许多并行系统一样，CUDA在生产效率和性能之间进行了一点折中：提供一些本身固有的功能，让程序员能够显式控制硬件。一方面是生产效率，另一方面是使程序员能够表达硬件所能完成的所有操作，在并行计算中，这两个方面之间经常会发生竞争。了解编程语言在这一著名的生产效率与性能大战中如何发展，了解CUDA是否能够在其他GPU或者其他类型的体系结构中变得普及，都将是非常有意义的一件事。

290

4.4.2　NVIDIA GPU计算结构

上文提到的这些罕见传统可以帮助解释为什么GPU拥有自己的体系结构类型，为什么拥有与CPU独立的专门术语。理解GPU的一个障碍就是术语，有些词汇的名称甚至可能导致误解。克服这一障碍的难度非常之大，这一章经过多次重写就是一个例证。为了让读者既能理解GPU的体系结构，又能学习许多采用非传统定义的GPU术语，我们最终的解决方案是使用CUDA术语来描述软件，而在开始时使用更具描述性的术语来介绍硬件，有时还会借用OpenCL使用的一些术语。在用我们的术语解释GPU体系结构之后，再将它们对应到NVIDIA GPU的官方术语。

表4-5从左至右列出了本节使用的一些更具描述性的术语、主流计算中的最接近术语、我们关心的官方NVIDIA GPU术语，以及这些术语的简短描述。本节的后续部分将使用该表左侧的描述性术语来解释GPU的微体系结构特征。

表4-5　本章所用GPU术语快速指南。第一列为硬件术语

类　型	更具描述性的名称	除GPU术语之外的最接近旧术语	官方CUDA/NVIDIA GPU术语	书中描述
程序抽象	可向量化循环	可向量化循环	网格	在GPU上执行的可向量化循环，由一个或多个可以并行执行的线程块（向量化循环体）构成
	向量化循环体	（条带挖掘后的）向量化循环体	线程块	可以在多线程SIMD处理器上执行的向量化循环，由一个或多个SIMD指令线程构成。它们可以通过局部存储器通信
	SIMD车道操作序列	标量循环的一次迭代	CUDA线程	SIMD指令线程的垂直抽取，对应于一个SIMD车道所执行的一个元素。根据遮罩和预测寄存器对结果进行存储
机器对象	SIMD指令线程	向量指令线程	Warp	一种传统线程，但它仅包含在多线程SIMD处理器上执行的SIMD指令。根据每个元素的遮罩来存储结果
	SIMD指令	向量指令	PTX指令	在多个SIMD车道上执行的单一SIMD指令
处理硬件	多线程SIMD处理器	（多线程）向量处理器	流式多处理器	多线程SIMD处理器执行SIMD指令的线程，与其他SIMD处理器无关
	线程块调度程序	标量处理器	Giga线程引擎	将多个线程块（向量化循环体）指定给多线程SIMD处理器

（续）

类 型	更具描述性的名称	除GPU术语之外的最接近旧术语	官方CUDA/NVIDIA GPU术语	书中描述
处理硬件	SIMD线程调试程序	多线程CPU中的线程调度器	Warp调度程序	当SIMD指令线程做好执行准备之后，用于调度和发射这些线程的硬件；包括一个记分板，用于跟踪SIMD线程执行
	SIMD车道	向量车道	线程处理器	SIMD车道执行一个SIMD指令线程中针对单个元素的操作。根据遮罩存储结果
存储器硬件	GPU存储器	主存储器	全局存储器	可供GPU中所有多线程SIMD处理器访问的DRAM存储器
	专用存储器	栈或线程局部存储（操作系统）	局部存储器	每个SIMD车道专用的DRAM存储器部分
	局部存储器	局部存储器	共享存储器	一个多线程SIMD处理器的快速本地SRAM，不可供其他SIMD处理器使用
	SIMD车道寄存器	向量车道寄存器	线程处理器寄存器	跨越完整线程块（向量化循环体）分配的单一SIMD车道中的寄存器

* 这 11 种术语分为 4 个组。从上至下为：程序抽象、机器对象、处理硬件和存储器硬件。表 4-8 将向量术语与这里的最接近术语关联在一起，表 4-10 和表 4-11 揭示了官方 CUDA/NVIDIA 和 AMD 术语与定义，以及 OpenCL 使用的术语。

我们将以 NVIDIA 系统为例，它们是 GPU 体系结构的代表。具体来说，我们将使用上面 CUDA 并行编程语言的术语，以 Fermi 体系结构为例（见 4.7 节）。

和向量体系结构一样，GPU 只能很好地解决数据级并行问题。这两种类型都拥有集中-分散数据传送和遮罩寄存器，GPU 处理器的寄存器要比向量处理器更多。由于它们没有一种接近的标量处理器，所以 GPU 有时会在运行时以硬件实现一些功能，而向量计算机通常是在编译时用软件来实现这些功能的。与大多数向量体系结构不同的是，GPU 还依靠单个多线程 SIMD 处理器中的多线程来隐藏存储器延迟（见第 2 章和第 3 章）。但是，要想为向量体系结构和 GPU 编写出高效代码，程序员还需要考虑 SIMD 操作分组。

网格是在 GPU 上运行、由一组**线程块**构成的代码。表 4-5 给出了网格与向量化循环、线程块与循环体（已经进行了条带挖掘，所以它是完整的计算循环）之间的相似之处。作为一个具体例子，假定我们希望把两个向量乘在一起，每个向量的长度为 8192 个元素。本节中，我们将反复使用这一示例。图 4-8 给出了这个示例与前两个 GPU 术语之间的关系。执行所有 8192 个元素乘法的 GPU 代码被称为**网格**（或向量化循环）。为了将它分解为更便于管理的大小，网格可以由**线程块**（或向量化循环体）组成，每个线程块最多 512 个元素。注意，一条 SIMD 指令一次执行 32 个元素。由于向量中有 8192 个元素，所以这个示例中有 16 个线程块（16=8192÷512）。网络和线程块是在 GPU 硬件中实现的编程抽象，可以帮助程序员组织自己的 CUDA 代码。（线程块类似于一个向量长度为 32 的条带挖掘向量循环。）

291 ~ 293

线程块调度程序将线程块指定给执行该代码的处理器，我们将这种处理器称为**多线程 SIMD 处理器**。线程块调度程序与向量体系结构中的控制处理器有某些相似。它决定了该循环所需要的线程块数，在完成循环之前，一直将它们分配给不同的多线程 SIMD 处理器。在这个示例中，会将 16 个线程块发送给多线程 SIMD 处理器，计算这个循环的所有 8192 个元素。

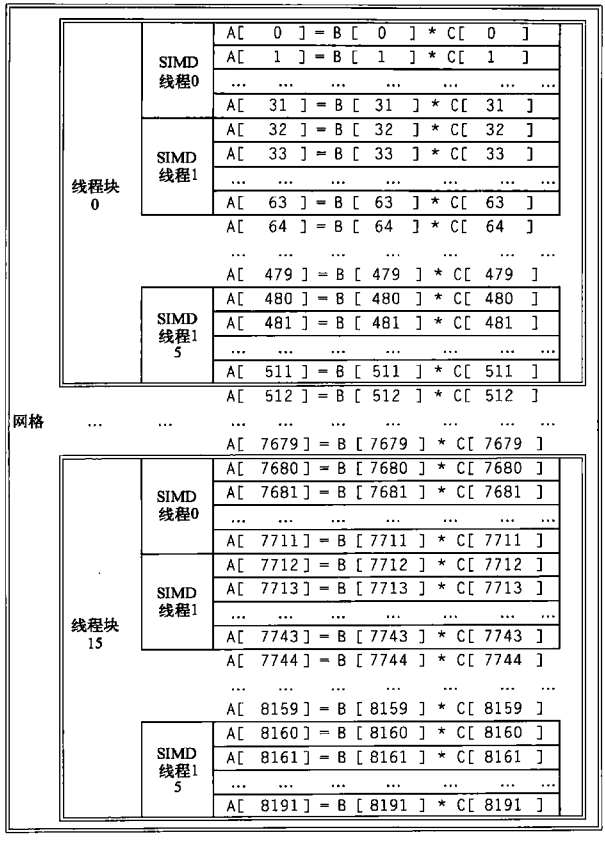


图 4-8　网格（可向量化循环）、线程块（SIMD 基本块）和 SIMD 指令线程与向量-向量乘法的对应，每个向量的长度为 8192 个元素。每个 SIMD 指令线程的每条指令计算 32 个元素，在这个示例中，每个线程块包含 16 个 SIMD 指令线程，网格包含 16 个线程块。硬件线程块调度程序将线程块指定给多线程 SIMD 处理器，硬件线程调度程序选择某个 SIMD 指令线程来运行一个 SIMD 处理器中的每个时钟周期。只有同一线程块中的 SIMD 线程可以通过本地存储器进行通信。（对于 Tesla 代的 GPU，每个线程块可以同时执行的最大 SIMD 线程数为 16，后来 Fermi 一代的 GPU 为 32。）

图 4-9 显示了多线程 SIMD 处理器的简化框图。它与向量处理器类似，但它有许多并行功能单元都是深度流水化的，而不是像向量处理器一样只有一小部分如此。在图 4-8 中的编程示例中，向每个多线程 SIMD 处理器分配这些向量的 512 个元素以进行处理。SIMD 处理器都是具有独立 PC 的完整处理器，使用线程进行编程（见第 3 章）。

294

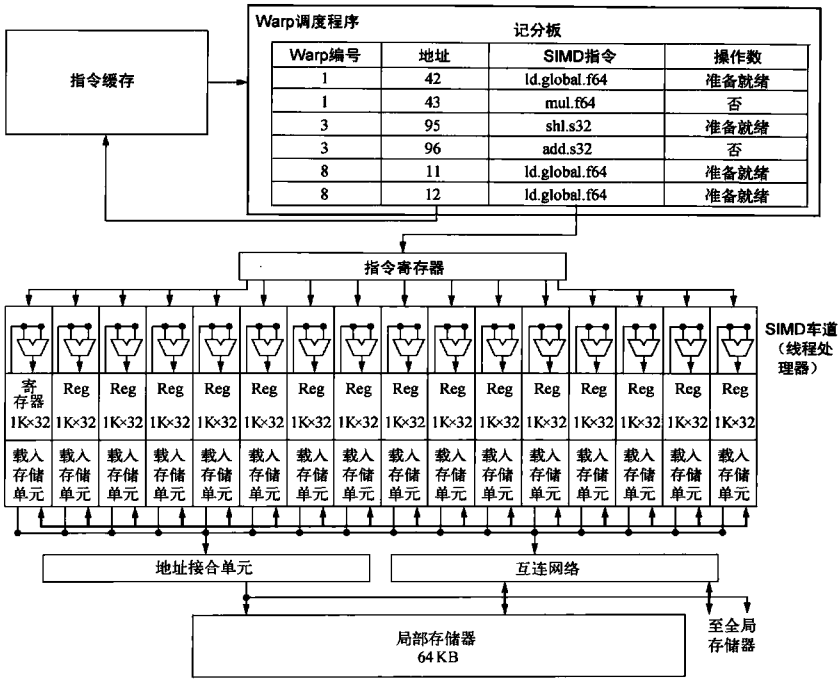


图 4-9 多线程 SIMD 处理器的简化框图。它有 16 个 SIMD 车道。SIMD 线程调度程序拥有大约 48 个独立的 SIMD 指令线程，它用一个包括 48 个 PC 的表进行调度

GPU 硬件包含一组用来执行线程块网络（向量化循环体）的多线程 SIMD 处理器，也就是说，GPU 是一个由多线程 SIMD 处理器组成的多处理器。

Fermi 体系结构的前四种实现拥有 7、11、14 或 15 个多线程 SIMD 处理器；未来的版本可能仅有 2 个或 4 个。为了在拥有不同个多线程 SIMD 处理器的 GPU 型号之间实现透明的可伸缩功能，线程块调度程序将线程块（向量化循环主体）指定给多线程 SIMD 处理器。图 4-10 给出了 Fermi 体系结构的 GTX 480 实现的平面图。

具体地说，硬件创建、管理、调度和执行的机器对象是 **SIMD 指令线程**。它是一个包含专用 SIMD 指令的传统线程。这些 SIMD 指令线程有其自己的 PC，它们运行在多线程 SIMD 处理器上。**SIMD 线程调度程序**包括一个记分板，让你知道哪些 SIMD 指令线程已经做好运行准备，然后将它们发送给分发单元，以在多线程 SIMD 处理器上运行。它与传统多线程处理器中的硬件线程调度程序相同（见第 3 章），就是对 SIMD 指令线程进行调度。因此 GPU 硬件有两级硬件调度程序：(1)**线程块调度程序**，将线程块（向量化循环体）分配给多线程 SIMD 处理器，确保线程块被分配给其局部存储器拥有相应数据的处理器，(2)SIMD 处理器内部的 SIMD 线程调度程序，由它来调度应当何时运行 SIMD 指令线程。

295

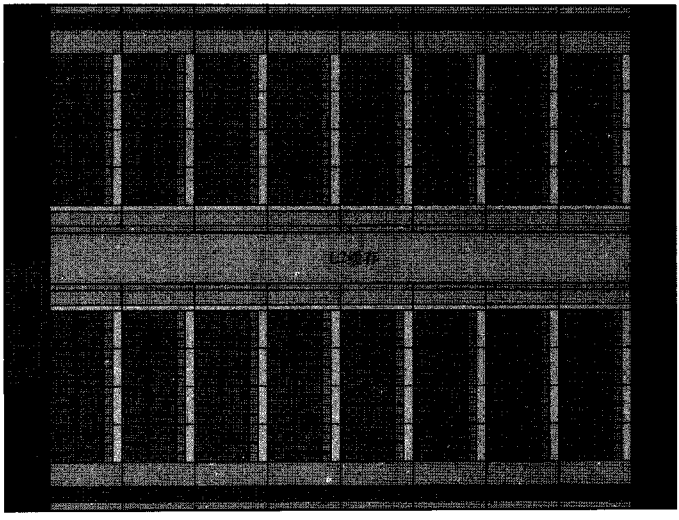


图4-10　Fermi GTX 480 GPU的平面图。本图显示了16个多线程SIMD处理器。在左侧突出显示了线程块调度程序。GTX 480有6个GDDR5端口，每个端口的宽度为64位，支持最多6 GB的容量。主机接口为PCI Express 2.0 × 16。Giga线程是将线程块分发给多处理器的调度程序名称，其中每个处理器都有其自己的SIMD线程调度程序

这些线程的SIMD指令的宽度为32，所以这个示例中每个SIMD指令线程将执行32个元素运算。在本示例中，线程块将包含512 ÷ 32=16 SIMD线程（见图4-8）。

由于线程由SIMD指令组成，所以SIMD处理器必须拥有并行功能单元来执行运算。我们称之为**SIMD车道**，它们与4.2节的向量车道非常类似。

每个SIMD处理器中的车道数在各代GPU中是不同的。对于Fermi，每个宽度为32的SIMD指令线程被映射到16个物理SIMD车道，所以一个SIMD指令线程中的每条SIMD指令需要两个时钟周期才能完成。每个SIMD指令线程在锁定步骤执行，仅在开始时进行调度。将SIMD处理器类比为向量处理器，可以说它有16个车道，向量长度为32，钟鸣为2个时钟周期。（我们之所以使用术语“SIMD处理器”，而不是“向量处理器”，就是因为这种既宽且浅的本质，前者的描述性更强一些。）

根据定义，由于SIMD指令的线程是独立的，SIMD线程调度程序可以选择任何已经准备就绪的SIMD指令线程，而不需要一直盯着线程序列中的下一条SIMD指令。SIMD线程调度程序包括一个记分板（见第3章），用于跟踪多达48个SIMD线程，以了解哪个SIMD指令已经做好运行准备。之所以需要这个记分板，是因为存储器访问指令占用的时钟周期数可能无法预测，比如存储器组的冲突就可能导致这一现象。图4-11给出的SIMD线程调度程序在不同时间以不同顺序选取SIMD指令线程。GPU架构师假定GPU应用程序拥有如此之多的SIMD指令线程，因此，实施多线程既可以隐藏到DRAM的延迟，又可以提高多线程SIMD处理器的使用率。

但是，为了防止损失，最近的 NVIDIA Fermi GPU 包含了一个 L2 缓存（见 4.7 节）。

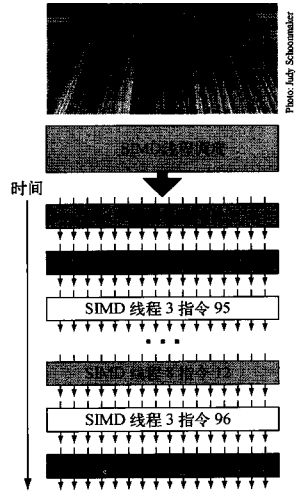


图 4-11 SIMD 指令线程的调度。调度程序选择一个准备就绪的 SIMD 指令线程，并同时向所有执行该 SIMD 线程的 SIMD 车道发出一条指令。由于 SIMD 指令线程是独立的，所以调度程序可以每次选择不同的 SIMD 线程

继续探讨向量乘法示例，每个多线程 SIMD 处理器必须将两个向量的 32 个元素从存储器载入寄存器中，通过读、写寄存器来执行乘法，然后将乘积从寄存器存回存储器中。为了保存这些存储器元素，SIMD 处理器拥有 32 768 个 32 位寄存器，给人以深刻印象。就像向量处理器一样，从逻辑上在向量车道之间划分这些寄存器，这里自然是在 SIMD 车道之间划分。每个 SIMD 线程被限制为不超过 64 个寄存器，所以我们可以认为一个 SIMD 线程最多拥有 64 个向量寄存器，每个向量寄存器有 32 个元素，每个元素的宽度为 32 位。（由于双精度浮点操作数使用两个相邻的 32 位寄存器，所以另一种意见是每个 SIMD 线程拥有 32 个各包括 32 个元素的向量寄存器，每个宽度为 64 位。） 296

由于 Fermi 拥有 16 个物理 SIMD 车道，各包含 2048 个寄存器。（GPU 没有尝试根据位来设计硬件寄存器，使其拥有许多读取端口和写入端口，而是像向量处理器一样，使用较简单的存储器结构，但将它们划分为组，以获得足够的带宽。）每个 CUDA 线程获取每个向量寄存器中的一个元素。为了用 16 个 SIMD 车道处理每个 SIMD 指令线程的 32 个元素，线程块的 CUDA 线程可以共同使用 2048 个寄存器的一半。

为了能够执行许多个 SIMD 指令线程，需要在创建 SIMD 指令线程时在每个 SIMD 处理器上动态分配一组物理寄存器，并在退出 SIMD 线程时加以释放。

注意，CUDA 线程就是 SIMD 指令线程的垂直抽取，与一个 SIMD 车道上执行的元素相对应。要当心，CUDA 线程与 POSIX 线程完全不同；不能从 CUDA 线程进行任意系统调用。

297

现在可以去看看 GPU 指令是什么样的了。

4.4.3 NVIDA GPU指令集体系结构

与大多数系统处理器不同，NVIDIA编译器的指令集目标是硬件指令集的一种抽象。PTX（并行线程执行）为编译器提供了一种稳定的指令集，可以实现各代GPU之间的兼容性。它向程序员隐藏了硬件指令集。PTX指令描述了对单个CUDA线程的操作，通常与硬件指令一对一映射，但一个PTX可以扩展到许多机器指令，反之亦然。PTX使用虚拟寄存器，所以编译器指出一个SIMD线程需要多少物理向量寄存器，然后，由优化程序在SIMD线程之间划分可用的寄存器存储。这一优化程序还会清除死亡代码，将指令打包在一起，并计算分支发生发散的位置和发散路径可能会聚的位置。

尽管x86微体系结构与PTX之间有某种类似，这两者都会转换为一种内部形式（x86的微指令），区别在于：对于x86，这一转换是在执行过程中在运行时以硬件实现的，而对于GPU，则是在载入时以软件实现的。

PTX指令的格式为：

```
opcode.type d, a, b, c;
```

其中d是目标操作数，a、b和c是源操作数；操作类型如表4-6所示。

表4-6　PTX操作类型

类　型	类型区分符
无类型位8、16、32和64位	.b8,.b16,.b32,.b64
无符号整数8、16、32和64位	.u8,.u16,.u32,.u64
有符号整数8、16、32和64位	.s8,.s16,.s32,.s64
浮点16、32和64位	.f16,.f32,.f64

源操作数为32位或64位整数或常值。目标操作数为寄存器，存储指令除外。

表4-7显示了基本PTX指令集。所有指令都可以由1位谓词寄存器进行判定，这些寄存器可以由设置谓词指令（setp）来设定。控制流指令为函数call和return，线程exit、branch以及线程块内线程的屏障同步（bar.sync）。在分支指令之前放置谓词就可以提供条件分支。编译器或PTX程序员将虚拟寄存器声明为32位或64位有类型或无类型值。例如，R0，R1...用于32位值，RD0，RD1...用于64位寄存器。回想一下，将虚拟寄存器指定给物理寄存器的过程是在载入时由PTX进行的。

298

表4-7　基本PTX GPU线程指令

分　组	指　令	示　例	含　义	注　释
算术	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a - b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	乘加
	div.type	div.f32 d, a, b	d = a / b;	多条微指令
	rem.type	rem.u32 d, a, b	d = a % b;	整余数
	abs.type	abs.f32 d, a	d = |a|;	

（续）

分组	指令	示例	含义	注释
算术	neg.type	neg.f32 d, a	d = 0 - a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	浮点选择非NaN
	max.type	max.f32 d, a, b	d = (a >b)? a:b;	浮点选择非NaN
	setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b) ;	比较和设定谓词
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	移动
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	用谓词选择
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a) ;	将atype转换为dtype
特殊函数	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	倒数
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	平方根
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a) ;	平方根的倒数
	sin.type	sin.f32 d, a	d = sin(a) ;	正弦
	cos.type	cos.f32 d, a	d = cos(a) ;	余弦
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	二进制对数
	ex2.type	ex2.f32 d, a	d = 2 ** a;	二进制指数
逻辑	logic.type = .pred,.b32, .b64			
	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
	xor.type	xor.b32 d, a, b	d = a ^ b;	
	not.type	not.b32 d, a, b	d = ~a;	1的补数
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	C逻辑非
	shl.type	shl.b32 d, a, b	d = a << b;	左移位
	shr.type	shr.s32 d, a, b	d = a >>b;	右移位
存储器访问	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	从存储器空间载入
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	存储到存储器空间
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b) ;	纹理查询
	atom.spc.op.type	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b,	atomic { d = *a; *a = op(*a, b); }	原子读改写操作
	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
控制流	branch	@p bra target	if (p) goto target;	条件分支
	call	call (ret), func, (params)	ret = func(params) ;	调用函数
	ret	ret	return;	从函数调用返回
	bar.sync	bar.sync d	wait for threads	屏障同步
	exit	exit	exit;	终止线程执行

下面的PTX指令序列是4.4.1节DAXPY循环一次迭代的指令：

```
shl.u32 R8, blockIdx, 9   ; 线程块ID * 块大小 (512 或 2⁹)
add.u32 R8, R8, threadIdx ; R8 = i = 我的 CUDA 线程 ID
shl.u32 R8, R8, 3         ; 字节偏移
ld.global.f64 RD0, [X+R8] ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8] ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4     ; 在 RD0 中求乘积 RD0 = RD0 * RD4 (标量 a)
add.f64 RD0, RD0, RD2     ; 在 RD0 中求和 RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0 ; Y[i] = sum (X[i]*a + Y[i])
```

如上所述，CUDA编程模型为每个循环迭代指定一个CUDA，为每个线程块指定一个唯一的识别编号（blockIdx），也为块中的每个CUDA线程指定一个唯一识别编号（threadIdx）。因此，它创建8192个CUDA线程，并使用唯一编号完成数组中每个元素的寻址，因此，不存在递增和分支编码。前3条PTX指令在R8中计算出唯一的元素字节偏移，会将这一偏移量加到数组的基地址中。以下PTX指令载入两个双精度浮点操作数，对其进行相乘和相加，并存储求和结果。（下面将描述与CUDA代码"if (i < n)"相对应的PTX代码。）

注意，GPU与向量体系结构不同，它们没有分别用于顺序数据传送、步幅数据传送和集中-分散数据传送的指令。所有数据传送都是集中-分散的！为了重新获得顺序（单位步幅）数据传送的效率，GPU包含了特殊的“地址接合”硬件，用于判断SIMD指令线程中的SIMD车道什么时候一同发出顺序地址。运行时硬件随后通知存储器接口单元来请求发送32个顺序字的分块传送。为了实现这一重要的性能改进，GPU程序员必须确保相邻的CUDA线程同时访问可以接合为一个或一些存储器或缓存块的相邻地址，我们的示例就是这样做的。

4.4.4　GPU中的条件分支

和单位步幅数据传送的情况一样，向量体系结构和GPU在处理IF语句方面非常相似，前者主要以软件实现这一机制，硬件支持非常有限，而后者则利用了更多的硬件。后面将会看到，除了显式谓词寄存器之外，GPU分支硬件使用了内部遮罩、分支同步栈和指令标记来控制分支何时分为多个执行路径，这些路径何时会汇合。

在PTX汇编程序级别，一个CUDA线程的控制流是由PTX指令分支、调用、返回和退出描述的，另外还要加上每条指令的各个按线程车道给出的谓词来描述，这些谓词由程序员用每个线程车道的1位谓词寄存器指定。PTX汇编程序分析了PTX分支图，对其进行优化，实现最快速的GPU硬件指令序列。

300

在GPU硬件指令级别，控制流包括分支、跳转、索引跳转、调用、索引调用、返回、退出和管理分支同步栈的特殊指令。GPU硬件提供了每个拥有自己栈的SIMD线程；一个堆栈项包含一个标识符标记、一个目标指令地址和一个目标线程活动遮罩。有一些GPU特殊指令为SIMD项目压入栈项，还有一些特殊指令和指令标记用于弹出栈项或者将栈展开为特殊项，并跳转到具有目标线程活动遮罩的目标指令地址。GPU硬件指令还拥有一些为不同车道设置的不同谓词（启用/禁用），这些谓词是利用每个车道的1位谓词寄存器指定的。

PTX汇编程序通常会将用PTX分支指令编码的简单外层IF/THEN/ELSE语句优化为设有谓词的GPU指令，不采用任何GPU分支指令。更复杂控制流的优化通常会混合采用谓词与GPU分支指令，这些分支指令带有一些特殊指令和标记，当某些车道跳转到目标地址时，这些GPU分支指令会使用分支同步栈压入一个栈项，而其他各项将会失败。在这种情况下，NVIDIA称为发生了分支**分岔**。当SIMD车道执行同步标记或**汇合**时，也会使用这种混合方式，它会弹出

一个栈项，并跳转到具有栈项线程活动遮罩的栈项地址。

PTX 汇编程序识别出循环分支，并生成 GPU 分支指令，跳转到循环的顶部，用特殊栈指令来处理各个跳出循环的车道，并在所有车道完成循环之后，使这些 SIMD 车道汇合。GPU 索引跳转和索引调用指令向栈中压入项目，以便在所有车道完成开关语句或函数调用时，SIMD 线程汇合。

GPU 设定谓词指令（表 4-7 中的 setp）对 IF 语句的条件部分求值。PTX 分支指令随后将根据该谓词来执行。如果 PTX 汇编程序生成了没有 GPU 分支指令的有谓词指令，它会使用各个车道的谓词寄存器来启用或禁用每条指令的每个 SIMD 车道。IF 语句 THEN 部分线程中的 SIMD 指令向所有 SIMD 车道广播操作。谓词被设置为 1 的车道将执行操作并存储结果，其他 SIMD 车道不会执行操作和存储结果。对于 ELSE 语句，指令使用谓词的补数（与 THEN 语句相对），所以原来空闲的 SIMD 车道现在执行操作，并存储结果，而它们前面的对应车道则不会执行相关操作。在 ELSE 语句的结尾，会取消这些指令的谓词，以便原始计算能够继续进行。因此，对于相同长度的路径，IF-THEN-ELSE 的工作效率为 50%。

IF 语句可以嵌套，因而栈的使用也可以嵌套，所以 PTX 汇编程序通常会混合使用设有谓词的指令和 GPU 分支与特殊分支指令，用于复杂控制流。注意，尝试嵌套可能意味着大多数 SIMD 车道在执行嵌套条件语句期间是空闲的。因此，等长路径的双重嵌套 IF 语句的执行效率为 25%，三重嵌套为 12.5%，以此类推。与此类似的情景是仅有少数几个遮罩位为 1 时向量处理器的运行情况。

301

具体来说，PTX 汇编程序在每个 SIMD 线程中的适当条件分支指令上设置“分支同步”标记，这个标记会在栈中压入当前活动遮罩。如果条件分支分岔（有些车道进行跳转，有些失败），它会压入栈项，并根据条件设置当前内容活动遮罩。分支同步标记弹出分岔的分支项，并在 ELSE 部分之前翻转遮罩位。在 IF 语句的末尾，PTX 汇编程序添加了另一个分支同步标记，它会将先前的活动遮罩从栈中弹出，放入当前的活动遮罩中。

如果所有遮罩位都被设置为 1，那么 THEN 结束的分支指令将略过 ELSE 部分的指令。当所有遮罩位都为零时，对于 THEN 部分也有类似优化，条件分支将跳过 THEN 指令。并行的 IF 语句和 PTX 分支经常使用没有异议的分支条件（所有车道都同意遵循同一路径），所以 SIMD 指令不会分岔到各个不同的车道控制流。PTX 汇编程序对此类分支进行了优化，跳过 SIMD 线程中所有车道都不会执行的指令块。这种优化在错误条件检查时是有用的，在这种情况下，必须进行测试，但很少会被选中。

以下是一个类似于 4.2 节的条件语句，其代码为：

```
if (X[i] != 0)
  X[i] = X[i] - Y[i];
else X[i] = Z[i];
```

这个 IF 语句可以编译为以下 PTX 指令（假定 R8 已经拥有经过调整的线程 ID），*Push、*Comp、*Pop 表示由 PTX 汇编程序插入的分支同步标记，用于压入旧遮罩、对当前遮罩求补，弹出恢复旧遮罩：

```
ld.global.f64 RD0, [X+R8]        ; RD0 = X[i]
setp.neq.s32 P1, RD0, #0         ; P1 是谓词寄存器 1
@!P1, bra ELSE1, *Push           ; 压入旧遮罩，设定新遮罩位
                                 ; if P1 为假，则转至 ELSE1
ld.global.f64 RD2, [Y+R8]        ; RD2 = Y[i]
sub.f64 RD0, RD0, RD2            ; RD0 中的差
st.global.f64 [X+R8], RD0        ; X[i] = RD0
@P1, bra ENDIF1, *Comp           ; 对遮罩位求补
```

```
                                  ; if P1 为真，则转至 ENDIF1
ELSE1:  ld.global.f64 RD0, [Z+R8]  ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0  ; X[i] = RD0
ENDIF1: <next instruction>, *Pop   ; 弹出以恢复旧遮罩
```

同样，IF-THEN-ELSE语句中的所有指令通常都是由SIMD处理器执行。一些SIMD车道是为THEN语句启用的，另一些车道是为ELSE指令启用的。前面曾经提到，在非常常见的情况中，各个车道都一致选择设定谓词的分支，比如，根据参数值选择分支，而所有车道的这个参数值都相同，所有活动遮罩位或者都为0，或者都为1，因此，分支会跳过THEN指令或ELSE指令。

302

这一灵活性清楚地表明元素有其自己的程序计数器，但是，在最缓慢的情况下，只有一个SIMD车道可以每两个时钟周期存储其结果，其余车道则会闲置。在向量体系结构中有种与之类似的最缓慢情景，那就是仅有一个遮罩位被设置为1时进行操作的情况。这一灵活性可能会导致GPU编程新手无法获得较佳性能，但在早期编程开发阶段可能是有帮助的。但要记住，在一个时钟周期内，SIMD车道的唯一选择就是执行在PTX指令中指定的操作或者处于空闲状态；两个SIMD车道不能同时执行不同指令。

这一灵活性还有助于解释为SIMD指令线程中每个元素指定的名称——CUDA线程，它会给人以独立运行的错觉。编程新手可能会认为这一线程抽象意味着GPU能够更出色地处理条件分支。一些线程会沿一条路径执行，其他线程则会沿另一路径执行，只要你不着急，那似乎就是如此。每个CUDA线程要么与线程块中的所有其他线程执行相同指令，要么就处于空闲状态。利用这一同步可以较轻松地处理带有条件分支的循环，这是因为遮罩功能可以关闭SIMD车道，自动检测循环的结束点。

最终得到的性能结果可能会与这种简单的抽象不相符。如果编写一些程序，以这种高度独立的MIMD模式来操作SIMD车道，就好像是编写了一些程序，在一个物理存储器很小的计算机上使用大量虚拟地址空间。这两种程序都是正确的，但它们的运行速度可能非常慢，程序员可能会对结果感到不快。

向量编译器可以用遮罩寄存器做到GPU用硬件完成的小技巧，但可能需要使用标量指令来保存、求补和恢复遮罩寄存器。条件执行就是这样一个例子：GPU在运行时用硬件完成向量体系结构在编译时完成的工作。有一种优化方法，可以在运行时针对GPU应用，但不能在编译时对向量体系结构应用，那就是在遮罩位全0或全1时略过THEN或ELSE部分。

因此，GPU执行条件分支的效率决定了分支的分岔频率。例如，某个特征值计算具有深度条件嵌套，但通过代码测试表明，大约82%的时钟周期发射将32个遮罩位中的29至32位设置为1，所以GPU执行这一代码的效率可能要超出人们的预期。

注意，同一机制处理向量循环的条带挖掘——当元素数与硬件不完全匹配时。本节开始的例子表明，用一个IF语句检查SIMD车道元素数（在上例中，该数目存储在R8中）是否小于限值（i<n），并适当设置遮罩。

303

4.4.5　NVIDIA GPU存储器结构

图4-12给出了NVIDIA GPU的存储器结构。多线程SIMD处理器中的每个SIMD车道获得片外DRAM的一个专用部分，称之为**专用存储器**，用于栈帧、溢出寄存器和不能放在寄存器中的私有变量。SIMD车道不共享专用存储器。最近的GPU将这一专用存储器缓存在L1和L2缓存中，用于辅助寄存器溢出并加速函数调用。

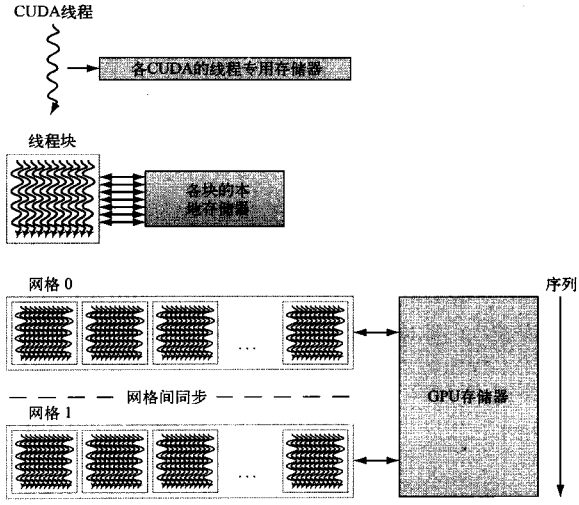


图 4-12 GPU 存储器结构。GPU 存储器由所有网格（向量化循环）共享，本地存储器由线程块（向量化循环体）中的所有 SIMD 指令线程共享，专用存储器由单个 CUDA 线程专用

我们将每个多线程 SIMD 处理器本地的片上存储器称为**本地存储器**。这一存储器由多线程 SIMD 处理器内的 SIMD 车道共享，但这一存储器不会在多线程 SIMD 处理器之间共享。多线程 SIMD 处理器在创建线程块时，将部分本地存储器动态分配给此线程块，当线程块中的所有线程都退出时，释放此存储器。这一本地存储器部分由该线程块专用。

最后，我们将由整个 GPU 和所有线程块共享的片外 DRAM 称为 **GPU 存储器**。这里的向量乘法示例仅使用 GPU 存储器。

304

被称为**主机**的系统处理器可以读取或写入 GPU 存储器。本地存储器不能供主机使用，它是每个多线程 SIMD 专用的。专用存储器也不可供主机使用。

GPU 通常不是依赖大型缓存来包含应用程序的整个工作集，而是使用较少的流式缓存，依靠大量的 SIMD 指令多线程来隐藏 DRAM 的较长延迟，其主要原因是它们的工作集可能达到数百 MB。在利用多线程隐藏 DRAM 延迟的情况下，系统处理器中供缓存使用的芯片面积可以用于计算资源和大量的寄存器，以保存许多 SIMD 指令线程的状态。如前文所述，向量的载入和存储与之相对，是将这些延迟分散在许多元素之间，因为它只需要有一次延迟，随后即可实现其余访问的流水化。

尽管隐藏存储器延迟是一种优选方法，但要注意，最新的 GPU 和向量处理器都已经添加了缓存。例如，最近的 Fermi 体系结构已经添加了缓存，但它们要么被看作带宽滤选器，以减少对 GPU 存储器的要求，要么被看作有限几种变量的加速器，这些变量的修改不能通过多线程来隐藏。因此，用于栈帧、函数调用和寄存器溢出的本地存储器与缓存是绝配，这是因为延迟对于函数调用是有影响的。由于片上缓存访问所需要的能量要远远小于对多个外部 DRAM 芯片的访问，所以使用缓存还可以节省能量。

为了提高存储器带宽、降低开销，如上所述，当地址属于相同块时，PTX数据传送指令会将来自同一SIMD线程的各个并行线程请求接合在一起，变成单个存储器块请求。对GPU程序设置的这些限制，多少类似于系统处理器程序在硬件预取方面的一些准则（见第2章）。GPU存储器控制器还会保留请求，将一些请求一同发送给同一个打开的页面，以提高存储器带宽（见4.6节）。第2章非常详细地介绍了DRAM，可帮你理解对相关地址进行分组所带来的潜在好处。

4.4.6 Fermi GPU体系结构中的创新

Fermi多线程SIMD处理器要比图4-13中的简化版本更复杂一些。为了提高硬件利用率，每个SIMD处理器有两个SIMD线程调度程序和两个指令分派单元。双重SIMD线程调度程序选择两个SIMD指令线程，并将来自每个线程的一条指令发射给由16个SIMD车道、16个载入/存储单元或4个特殊功能单元组成的集合。因此，每两个时钟周期将两个SIMD指令线程调度至这些集合中的任何一个。由于这些线程是独立的，所以不需要检查指令流中的数据相关性。这一创新类似于多线程向量处理器，它可以发射来自两个独立线程的向量指令。

图4-13展示了发射指令的双重调度程序，图4-14展示了Fermi GPU的多线程SIMD处理器的框图。

305

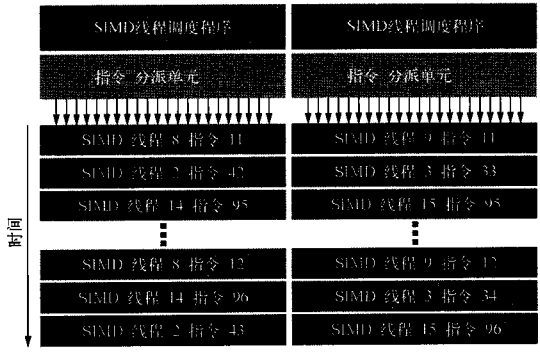


图4-13　Fermi双SIMD线程调度程序的框图。将这一设计与图4-11中的单SIMD线程设计进行对比

Fermi引入了几种创新，使GPU与主流系统处理器的接近程度远远超过Tesla和前几代GPU体系结构。

- **快速双精度浮点运算**——Fermi对比发现，传统处理器的相对双精度速度大约为单精度速度的一半，是先前Tesla代处理器单精度速度的十分之一。也就是说，当准确性需要双精度时，使用单精度在速度方面没有太大的诱惑力。在使用乘加指令时，峰值双精度性能从过去GPU的78 GFLOP/s增长到515 GFLOP/s。

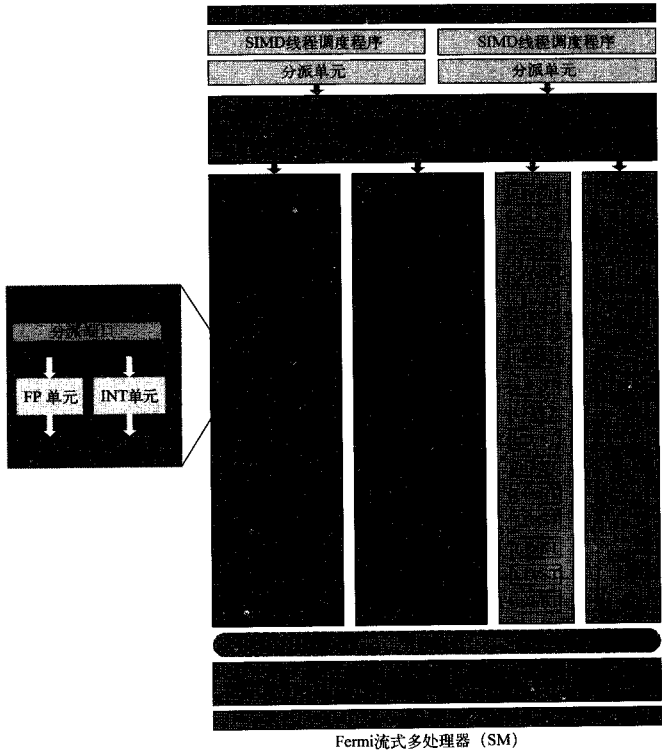


图 4-14 Fermi GPU 多线程 SIMD 处理器的框图。每个 SIMD 车道有一个流水线浮点单元、一个流水线整数单元、还有某些逻辑，用于将指令和操作数分发给这些单元，以及一个队列用于保存结果。4 个特殊函数单元（SFU）计算诸如平方根、求倒数、正弦和余弦等函数

- **GPU 存储器**——尽管 GPU 的基本思想是使用足够多的线程来隐藏 DRAM 延迟，但仍然需要在线程之间使用一些变量，比如前面提到的局部变量。Fermi 在 GPU 中为每个多线程 SIMD 处理器包含了 L1 数据缓存和 L1 指令缓存，还包含了由所有多线程 SIMD 处理器共享的单个 768 KB L2 缓存。如上所述，除了降低对 GPU 存储器的带宽压力之外，缓存因为驻留在芯片上，不用连到片外 DRAM，所以还能节省能量。L1 缓存实际上与本地存储器使用同一 SRAM。Fermi 有一个模式位，为用户提供了两种使用 64KB SRAM 的选择：两种 16 KB L1 缓存和 48 KB 本地存储器，另一种是 48 KB L1 缓存和 16 KB 本地存储器。注意，GTX 480 有一个倒转的存储器层次结构：聚合寄存器堆的大小为 2 MB，所有 L1 数据缓存的大小介于 0.25 与 0.75 MB 之间（取决于它们是 16 KB，还是 48 KB），L2 缓存的大小为 0.75 MB。了解这一反转比值对 GPU 应用程序的影响是有意义的。

306

- **全部GPU存储器的64位寻址和统一地址空间**——利用这一创新可以非常轻松地提供C和C++所需要的指针。
- **纠错码检测和纠正存储器与寄存器中的错误**（见第2章），为了提高数千个服务器上长期运行的应用程序的可靠性，ECC是数据中心的一种标准配置（见第6章）。
- **更快速的上下文切换**——由于多线程SIMD处理器拥有大量状态，所以Fermi以硬件支持大幅加速上下文的切换速度。Fermi可以在不到25微秒内完成切换，比之前的处理器大约快10倍。

307

- **更快速的原子指令**——这一特征最早包含在Telsa体系结构中，Fermi将原子指令的性能提高了5～20倍，达到几微秒的级别。有一个与L2缓存相关的特殊硬件单元（不是在多线程SIMD处理器内部）用来处理原子指令。

4.4.7　向量体系结构与GPU的相似与不同

我们已经看到，向量体系结构与GPU之间确实有许多相似之处。这些相似之处和GPU那些怪异的术语一样，也让体系结构圈的人们难以真正了解新奇的GPU本质。既然我们现在已经了解了向量体系结构和GPU的一些内幕，那就可以体味一下它们的相似与不同了。这两种体系结构都是为了执行数据级并行程序而设计的，但它们选取了不同的路径，对比它们是希望更深入地了解DLP硬件到底需要什么。表4-8首先给出向量术语，然后给出GPU中最接近的对等术语。

表4-8　向量术语的对等GPU术语

类型	向量术语	最接近的CUDA/NVIDIA GPU术语	注　释
编程抽象	向量化循环	网格	概念相似，GPU使用了描述性较差的术语
	钟鸣	—	由于一条向量指令（PTX指令）的完成在Fermi上只需要2个时钟周期，在Tesla上只需要4个时钟周期，所以钟鸣在GPU中很短
机器对象	向量指令	PTX指令	SIMD线程的PTX指令会广播到所有SIMD车道，所以它与向量指令类似
	集中/分散	全局载入/存储 (ld.global/st.global)	所有GPU载入与存储都是集中和分散，因为每个SIMD车道会发送一个唯一地址。如果来自SIMD车道的地址允许，将由GPU接合单元来实现单位步幅性能
	遮罩寄存器	谓词寄存器和内部遮罩寄存器	向量遮罩寄存器是体系结构状态的组成部分，而GPU遮罩寄存器位于硬件内部。GPU条件硬件添加了超越谓词寄存器的新特征，可以动态管理遮罩
处理与存储器硬件	向量处理器	多线程SIMD处理器	这些概念是类似的，但SIMD处理器倾向于拥有许多车道，每个车道只需要几个时钟周期完成一个向量，而向量体系结构的车道较少，需要许多时钟周期才能完成一个向量。SIMD还实现了多线程，而向量通常不会
	控制处理器	线程块调度程序	最接近的是线程块调度程序，它将线程块指定给多线程SIMD处理器。但GPU没有标量-向量运算，没有单位步幅或步幅化数据传送指令，而控制处理器经常会提供上述指令
	标量处理器	系统处理器	由于缺少共享存储器，而且通过PCI总线进行通信的延迟较高（时钟周期时间达1000秒），所以GPU中的系统处理器很少执行标量处理器在向量体系结构中执行的相同任务
	向量车道	SIMD车道	这两者基本上都是带有寄存器的功能单元
	向量寄存器	SIMD车道寄存器	向量寄存器的对等术语就是多线程SIMD处理器中所有32个SIMD车道的相同寄存器（这个多线程SIMD处理器在运行SIMD指令线程）。每个SIMD线程的寄存器数目是灵活的，但最大为64，所以向量寄存器的最大数目为64
	主存储器	GPU存储器	GPU存储器对应向量处理器中的系统存储器

SIMD 处理器与向量处理器类似。GPU 中的多个 SIMD 处理器像独立 MIMD 核心一样操作，就好像是许多向量计算机拥有多个向量处理器。这种观点将 NVIDIA GTX 480 看作一个具有多线程硬件支持的 15 核心机器，其中每个核心有 16 个车道。两者之间最大的区别是多线程，它是 GPU 的基本必备技术，而大多数向量处理器则没有采用。

看一下这两种体系结构中的寄存器，VMIPS 寄存器堆拥有整个向量，也就是说，由 64 个双精度值构成的连续块。相反，GPU 中的单个向量会分散在所有 SIMD 车道的寄存器中。VMIPS 处理器有 8 个向量寄存器，各有 64 个元素，总共 512 个元素。一个 GPU 的 SIMD 指令线程拥有多达 64 个寄存器，各有 32 个元素，总共 2048 个元素。这些额外的 GPU 寄存器支持多线程。

图 4-15 的左边是向量处理器执行单元的框图，右侧是 GPU 的多线程 SIMD 处理器。为便于讲解，假定向量处理器有 4 个车道，多线程 SIMD 处理器也有 4 个 SIMD 车道。此图表明，4 个 SIMD 车道的工作方式非常像 4 车道向量单元，SIMD 处理器的工作方式与向量处理器非常类似。

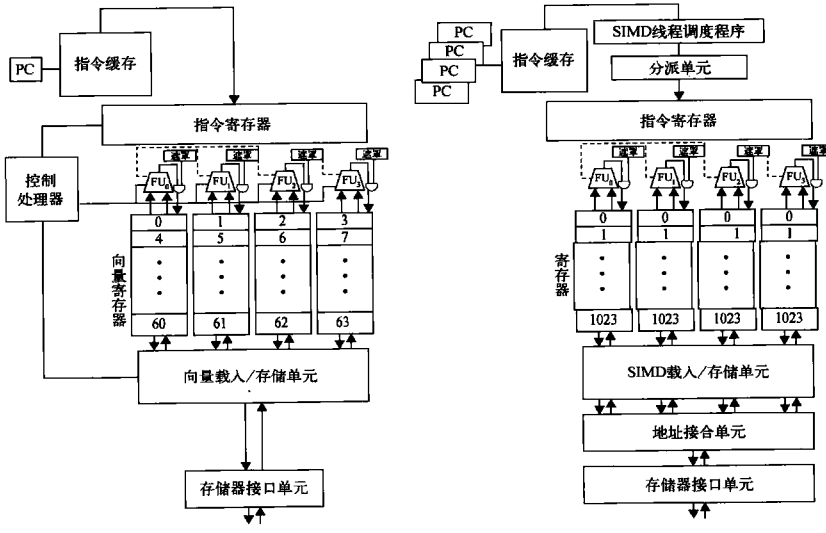


图 4-15 左侧为具有 4 个车道的向量处理器，右侧为 GPU 的多线程 SIMD 处理器。（GPU 通常有 8～16 个 SIMD 车道。）控制处理器为标量-向量运算提供标量操作数，为对存储器进行单位步幅或非单位步幅访问而递增地址，执行其他“记账类型”（accounting-type）的运算。只有当地址接合单元可以发现本地寻址时，才会在 GPU 中实现峰值存储器性能。与此类似，当所有内部遮罩位被设置为相同时，会实现峰值计算性能。注意，SIMD 处理器中每个 SIMD 线程有一个 PC，以帮助实现多线程

实际上，GPU中的车道要多很多，所以GPU“钟鸣”更短一些。尽管向量处理器可能拥有2～8个车道，向量长度例如为32（因此，钟鸣为4～16个时钟周期），多线程SIMD处理器可能拥有8～16个车道。SIMD线程的宽度为32个元素，所以GPU钟鸣仅为2或4个时钟周期。这一差别就是为什么要使用“SIMD处理器”作为更具描述性术语的原因，这一术语更接近于SIMD设计，而不是传统的向量处理器设计。

308～309 与向量化循环最接近的GPU术语是网格，PTX指令与向量指令最接近，这是因为SIMD线程向所有SIMD车道广播PTX指令。

关于两种体系结构中的存储器访问指令，所有GPU载入都是集中指令，所有GPU存储都是分散指令。如果CUDA线程的地址引用同一缓存/存储器块的邻近地址，那GPU的地址接合单元将会确保较高的存储器带宽。向量体系结构采用显式单位步幅载入并存储指令，而GPU编程则采用隐式单位步幅，这两者的对比说明为什么在编写高效GPU代码时，需要程序员从SIMD运算的角度来思考，尽管CUDA编程模型与MIMD看起来非常类似。由于CUDA线程可以生成自己的地址、步幅以及集中-分散，所有在向量体系结构和GPU中都可以找到寻址向量。

我们已经多次提到，这两种体系结构采用了非常不同的方法来隐藏存储器延迟。向量体系结构通过深度流水化访问让向量的所有元素分担这一延迟，所以每次向量载入或存储只需要付出一次延迟代价。因此，向量载入和存储类似于在存储器和向量寄存器之间进行的块传送。与之相对的是，GPU使用多线程隐藏存储器延迟。（一些研究人员正在研究为向量体系结构添加多线程，以实现这两者的最佳性能。）

310

关于条件分支指令，两种体系结构都使用遮罩寄存器来实现。两个条件分支路径即使在未存储结果时也会占用时间以及（或者）空间。区别在于，向量编译器以软件显式管理遮罩寄存器，而GPU硬件和汇编程序则使用分支同步标记来隐式管理它们，使用内部栈来保存、求补和恢复遮罩。

前面曾经提到，GPU的条件分支机制很好地处理了向量体系结构的条带挖掘问题。如果向量长度在编译时未知，那么程序必须计算应用程序向量长度的模和最大向量长度，并将它存储在向量长度寄存器中。条带挖掘循环随后将向量长度寄存器重设剩余循环部分的最大向量长度。这种情况用GPU处理起来要更容易一些，因为它们将会一直迭代循环，直到所有SIMD车道到达循环范围为止。在最后一次迭代中，一些SIMD车道将被遮罩屏蔽，然后在循环完成后恢复。

向量计算机的控制处理器在向量指令的执行过程中扮演着重要角色。它向所有向量车道广播操作，并广播用于向量-标量运算的标量寄存器值。它还执行一些在GPU中显式执行的隐式计算，比如自动为单位步幅和非单位幅载入、存储指令递增存储器地址。GPU中没有控制处理器。最类似的是线程块调度程序，它将线程块（向量循环体）指定给多线程SIMD处理器。GPU中的运行时硬件机制一方面生成地址，另一方面还会查看它们是否相邻，这在许多DLP应用程序中都是很常见的，其功耗效率可能要低于控制处理器。

向量计算机中的标量处理器执行向量程序的标量指令。也就是说，它执行那些在向量单元中可能速度过慢的运算。尽管与GPU相关联的系统处理器与向量体系结构中的标量处理器最为相似，但独立的地址空间再加上通过PCIe总线传送，往往会耗费数千个时钟周期的开销。对于在向量计算机中执行的浮点计算，标量处理器可能要比向量处理器慢一些，但它们的速度比值不会达到系统处理器与多线程SIMD处理器的比值（在给定开销的前提下）。

因此，GPU中的每个“向量单元”必须执行本来指望在向量计算机标量处理器上进行的计

算。也就是说，如果不是在系统处理器上进行计算然后再发送结果，而是使用谓词寄存器和内置遮罩禁用其他 SIMD 车道，仅留下其中一个 SIMD 车道，并用它来完成标量操作，那可以更快一些。向量计算机中比较简单的标量处理器可能要比 GPU 解决方案更快一些、功耗效率更高一些。如果系统处理器和 GPU 将来更紧密地结合在一起，那了解一下系统处理器能否扮演标量处理器在向量及多媒体 SIMD 体系结构中的角色，那将是很有意义的。 311

4.4.8 多媒体SIMD计算机与GPU之间的相似与不同

从较高级别的角度来看，具有多媒体 SIMD 指令扩展的多核计算机的确与 GPU 有一些相似之处。表 4-9 总结了它们之间的相似与不同。

表4-9 具有多媒体SIMD扩展的多核心与最新GPU之间的相似与不同

组　件	具有SIMD的多核	GPU
SIMD处理器	4～8	8～16
SIMD车道/处理器	2～4	8～16
对SIMD线程的多线程硬件支持	2～4	16～32
单、双精度性能的典型比值	2：1	2：1
最大缓存	8MB	0.75MB
存储器地址的大小	64位	64位
主存储器的大小	8～256 GB	4～6GB
页面级别的存储器保护	是	是
需求分页	是	否
集成标量处理器/SIMD处理器	是	否
缓存一致性	是	否

这两种多处理器的处理器都使用多个 SIMD 车道，只不过 GPU 的处理器更多一些，车道数要多很多。它们都使用硬件多线程来提高处理器利用率，不过 GPU 为大幅增加线程数目提供了硬件支持。由于 GPU 中最近的一些创新，现在这两者的单、双精度浮点运算性能比相当。它们都使用缓存，不过 GPU 使用的流式缓存要小一些，多核计算机使用大型多级缓存，以尝试完全包含整个工作集。它们都使用 64 位地址空间，不过 GPU 中的物理主存储器要小得多。尽管 GPU 支持页面级别的存储器保护，但它们都不支持需求分页。

除了在处理器、SIMD 车道、硬件线程支持和缓存大小等大量的数字差异之外，还有许多体系结构方面的区别。在传统计算机中，标量处理器和多媒体 SIMD 指令紧密集成在一起；它们由 GPU 中的 I/O 总线隔离，它们甚至还有独立的主存储器。GPU 中的多个 SIMD 处理器使用单一地址空间，但这些缓存不是像传统的多核计算机那样是一致的。多媒体 SIMD 指令与 GPU 不同，它不支持集中-分散存储器访问，4.7 节表明，这是一个非常重要的简化。 312

4.4.9 小结

现在，GPU 的神秘面纱已经揭开，可以看出 GPU 实际上就是多线程 SIMD 处理器，只不过与传统的多核计算机相比，它们的处理器更多、每个处理器的车道更多，多线程硬件更多。例

如，Fermi GTX 480拥有15个SIMD处理器，每个处理器有16个车道，为32个SIMD线程提供硬件支持。Fermi甚至包括指令级并行，可以从两个SIMD线程向两个SIMD车道集合发射指令。另外，它们的缓存存储器较少——Fermi的L2缓存为0.75 MB，而且与标量处理器不一致。

313

CUDA编程模型将所有这些形式的并行包含在一种抽象中，即CUDA线程中。因此，CUDA程序员可以看作是在对数千个线程进行编程，而实际上他们是在许多SIMD处理器的许多车道上执行各个由32个线程组成的块。希望获得良好性能的CUDA程序员一定要记住，这些线程是分块的，一次执行32个，而且为了从存储器系统获得良好性能，其地址需要是相邻的。

尽管本节使用了CUDA和NVIDIA GPU，但我们确信在OpenCL编程语言和其他公司的GPU中也采用了相同思想。

在读者已经很好地理解了GPU的工作原理之后，现在可以揭示真正的术语了。表4-10和4-11将本节的描述性术语及定义与官方CUDA/NVIDIA和AMD术语及定义对应起来，而且还给出了OpenCL术语。我们相信，GPU学习曲线非常陡峭，一部分原因就是因为使用了如下术语：用“流式多处理器”表示SIMD处理器，“线程处理器”表示SIMD车道，“共享存储器”表示本地存储器，而本地存储器实际上并非在SIMD处理器之间共享！我们希望，这种“两步走”方法可以帮助读者更快速地沿学习曲线上升，尽管这种方法有些不够直接。

表4-10　由本章使用的术语转换为官方NVIDIA/CUDA和AMD术语

类型	本书使用的描述性更强的名称	官方CUDA/NVIDIA术语	书中定义及AMD和OpenCL术语	官方CUDA/NVIDIA定义
编程抽象	可向量化循环	网格	一种可向量化循环，在GPU上执行，由一个或多个可以并行执行的“线程块”（或向量化循环体）组成。OpenCL名称为“索引范围”，AMD名称为“NDRange”	网格是一组可以同时、顺序或混合执行的线程块
	向量化循环体	线程块	在多线程SIMD处理器上执行的向量化循环，由一个或多个SIMD指令线程组成。这些SIMD线程可以通过本地存储器通信。AMD和OpenCL称为“工作组”	线程块是一组CUDA线程，这些线程可以一起同时执行，并通过共享存储器和屏障同步进行协调与通信。线程块在其网格内有一个线程块ID
	SIMD车道运算序列	CUDA线程	SIMD指令线程的垂直抽取，与SIMD车道上执行的一个元素相对应。根据遮罩位来存储结果。AMD和OpenCL将CUDA线程称为“工作项”	CUDA线程是一种轻量级线程，可以执行顺序程序，可以与同一线程块中执行的其他CUDA线程合作CUDA线程在其线程块内有一个线程ID
机器对象	SIMD指令线程	Warp	一种传统线程，但它仅包含在多线程SIMD处理器上执行的SIMD指令。根据每个元素的遮罩位来存储结果。AMD称为“波前”	Warp是一组并行CUDA线程（比如32个），可以在多线程SIMT/SIMD处理器中一起执行相同指令
	SIMD指令	PTX指令	在SIMD车道之间执行的单条SIMD指令。AMD名称为“AMDIL”或“FSAIL”指令	PTX指令指定了由CUDA线程执行的指令

* OpenCL名称在第4列给出。

表 4-11 由本章使用的术语转换为官方 NVIDIA/CUDA 和 AMD 术语

类型	本书使用的描述性更强的名称	官方CUDA/NVIDIA术语	书本定义及AMD和OpenCL术语	官方CUDA/NVIDIA定义
处理硬件	多线程SIMD处理器	流式多处理器	多线程SIMD处理器，独立于其他SIMD处理器执行SIMD指令线程。AMD和OpenCL都将其称为“计算单元”。但是，CUDA程序员是为一个车道编程，而不是为多个SIMD车道的“向量”编写	流式多处理器（SM）是一种多线程SIMT/SIMD处理器，它执行CUDA线程的warp。SIMD程序指定一个CUDA线程的执行，而不是多个SIMD车道的向量
	线程块调度程序	Giga线程引擎	将多个向量化循环体分派给多线程SIMD处理器，AMD称为“超线程分派引擎”	当资源可用时，将网格的线程块分派、调度至流式多处理器
	SIMD线程调度程序	Warp调度程序	硬件单元，当SIMD指令线程做好执行准备后调度和发射这些SIMD指令线程，包括跟踪SIMD线程执行的记分板。AMD称为“工作组调度程序”	流式多处理器中的Warp调度程序调度Warp，以在下一条指令做好执行准备后执行
	SIMD车道	线程处理器	硬件SIMD车道，执行SIMD指令线程中对单一元素进行的操作。根据遮罩存储结果。OpenCL将其称为“处理元素”。AMD也称为“SIMD车道”	线程处理器是流式多处理器中的一条数据路径和寄存器堆部分，执行一个Warp中一条或多条车道的运算
存储器硬件	GPU存储器	全局存储器	可供一个GPU中所有多线程SIMD处理器访问的DRAM存储器。OpenCL称为“全局存储器”	全局存储器可供任意网格任意线程块中的所有CUDA线程访问。实现为DRAM的一个区域，可被缓存
	专用存储器	本地存储器	每个SIMD车道专用的DRAM存储器部分。AMD和OpenCl称为“专用存储器”	CUDA线程的专用“线程本地”存储器。实现为DRAM的缓存区域
	本地存储器	共享存储器	一个多线程SIMD处理器的快速本地SRAM，不可供其他SIMD处理器使用。OpenCL称为“本地存储器”，AMD称为“组存储器”	由线程块中CUDA线程共享的快速SRAM存储器，由该线程块专用。用于在屏障同步点在一个线程块中的CUDA线程之间进行通信
	SIMD车道寄存器	寄存器	在向量化循环体之间分配的单个SIMD车道中的寄存器。AMD也称为“寄存器”	CUDA线程的专用寄存器。实现为每个线程处理器的几个Warp中特定车道的多线程寄存器堆

* 注意这里的描述性术语“本地存储器”和“专用存储器”使用了 OpenCL 术语。NVIDIA 在描述流式多处理器时使用了 SIMT（单指令多线程）而不是 SIMD。SIMT 优于 SIMD 是因为按线程分支和控制流与任何 SIMD 机器都不同。

4.5 检测与增强循环强并行

程序中的循环是我们前面讨论以及将在第 5 章讨论的许多并行类型的根源。本节，我们讨论用于发现并行以在程序中加以开发的编译器技术，以及这些编译器技术的硬件支持。我们准确地定义一个循环何时是并行的（或可向量化的）、相关性是如何妨碍循环成为并行的，以及用于消除几类相关性的技术。发现和利用循环级并行对于开发 DLP 和 TLP 以及将在附录 H 中研究的更主动静态 ILP 方法（例如，VLIW）都至关重要。

循环级并行通常是在源代码级别或接近级别进行分析的，而在编译器生成指令之后，就完成了对ILP的大多数分析。循环级分析需要确定循环的操作数在这个循环的各次迭代之间存在哪种相关性。就目前来说，我们将仅考虑数据相关，在某一时刻写入操作数，并在稍后时刻读取时会出现这种相关性。名称相关也是存在的，利用第3章讨论的重命名技术可以消除这种相关。

循环级并行的分析主要是判断后续迭代中的数据访问是否依赖于在先前迭代中生成的数据值；这种相关被称为**循环间相关**。我们在第2章和第3章考虑的大多数示例都没有循环间相关，而是循环级并行的。为了了解一个循环是并行的，让我们首先看看源代码：

314
~
315

```
for (i=999; i>=0; i=i-1)
      x[i] = x[i] + s;
```

在这个循环中，对x[i]的两次使用是相关的，但这是同一个迭代内的相关，不是循环间相关。在不同迭代中对i的连续使用之间存在循环间相关，但这种相关涉及一个容易识别和消除的归纳变量。我们在第2章的2.2节讨论循环展开时见到了一些示例，说明如何消除涉及归纳变量的相关。

要寻找循环之间的并行，需要识别诸如循环、数组引用和归纳变量计算之类的结构，所以与机器码级别相比，编译器在源代码级别或相近级别进行这一分析要更轻松一些。让我们看一个更复杂的例子。

例题　考虑下面这样一个循环：

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

假定A、B和C是没有重叠的不同数组。（在实践中，这些数组有时可能相同，或可能重叠。因为这些数组可能是作为参数传递给包含这一循环的过程，为了判断数组是否重叠或相同，通常需要对程序进行复杂的过程间分析。）在这个循环中，语句S1和S2之间的数据相关如何？

解答　共有以下两种不同相关。

(1) S1使用一个在先前迭代中由S1计算的值，这是因为迭代i计算A[i+1]，然后在迭代i+1中读取它。对B[i]和B[i+1]来说，S2也是如此。

(2) S2使用由同一迭代中S1计算的值A[i+1]。

这两种相关是不同的，拥有不同的效果。为了了解它们如何不同，我们假定此类相关只能同时存在一个。因为语句S1依赖于S1的先前迭代，所以这种相关是循环间相关。这种相关迫使这个循环的连续迭代必须按顺序执行。

第二种相关（S2对S1的依赖）位于一个迭代内，不是循环间相关。因此，如果它是仅有的相关，那这个循环的多个迭代就能并行执行，只要一个迭代中的每对语句保持相对顺序即可。我们在2.2节的例子中看到过这种类型的相关，通过循环展开可以暴露这种并行。这种循环内的相关是很常见的，例如，使用链接（chaining）的向量指令序列就存在此类相关。

316

还有可能存在一种不会妨碍并行的循环间相关，如下例所示。

例题 考虑下面这样一个循环：

```
for (i=0; i<100; i=i+1) {
      A[i] = A[i] + B[i];    /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

S1 和 S2 之间是什么样的相关？这一循环是否为并行的？如果不是，说明如何使之成为并行循环。

解答 语句 S1 使用了在上一次迭代中由语句 S2 指定的值，所以在 S2 与 S1 之间存在循环间相关。尽管存在这一循环间相关，依然可以使这一循环变为并行。与前面的循环不同，这种相关不存在环式相关：这些语句都没有依赖于自身，而且尽管 S1 依赖于 S2，但 S2 没有依赖于 S1。如果可以将一个循环改写为没有环式相关的形式，那这个循环就是并行的，因为没有这种环式相关形式就意味着这种相关性对语句进行了部分排序。

尽管以上循环中没有环式相关，但必须对其进行转换，以符合部分排序，并暴露出并行。两个观察结果对于这一转换至关重要。

(1) 不存在从 S1 到 S2 的相关。如果存在这种相关，那就可能存在环式相关，那循环就不是并行的。由于没有其他相关，所以两个语句之间的互换不会影响 S2 的执行。

(2) 在循环的第一次迭代中，语句 S2 依赖于 B[0]值，它是在开始循环之前计算的。这两个观察结果可以让我们用以下代码序列来代替以上循环：

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

这两个语句之间的相关不再是循环间相关，所以循环的各次迭代可以重叠，只要每次迭代中的语句保持相对顺序即可。

我们的分析需要首先找出所有循环间相关。这一相关信息是不确切的，也就是说，它告诉我们此相关可能存在。考虑以下示例：

```
for (i=0;i<100;i=i+1) {
      A[i] = B[i] + C[i]
      D[i] = A[i] * E[i]
}
```

317

这个例子中对 A 的第二次引用不需要转换为载入指令，因为我们知道这个值是由上一个语句计算并存储的；因此，对 A 的第二个引用可能就是引用计算 A 的寄存器。执行这一优化需要知道这两个引用总是指向同一存储器地址，而且不存在对相同位置的干扰访问。通常，数据相关分析只会告诉我们一个引用可能依赖于另一个引用；要确定两个引用一定指向同一地址，那就需要进行更复杂的分析。在上面的例子中，进行这一简单分析就足够了，因为这两个引用都处于同一基本块中。

循环间相关经常会是一种**递归**（recurrence）形式。如果要确定变量的取值，需要先知道该变量在前面迭代中的取值时，就会发生**递归**，这个先前迭代往往就是前面的相邻近代，如以下

代码段所示：

```
for (i=1;i<100;i=i+1) {
      Y[i] = Y[i-1] + Y[i];
}
```

检测递归是非常重要的，原因有两个：其一，一些体系结构（特别是向量计算机）对执行递归提供特殊支持；其二，在ILP环境中，仍然可能开发相当数量的并行。

4.5.1　查找相关

显然，查找程序中的相关对于确定哪些循环可能包含并行以及如何消除名称相关都很重要。诸如C或C++语言中存在数组和指针，Fortran中存在按引用传送的参数传递，这些也都增加了相关分析的复杂度。由于标量变量引用明确指向名称，所以用别名对它们进行分析是比较轻松的，因为指针和引用参数会增加分析过程的复杂性和不确定性。

编译器通常是如何检测相关的呢？几乎所有相关分析算法都假定数组索引是仿射的（affine）。用最简单的话说，一维数组索引可以写为 $a \times i+b$ 的形式，其中 a 和 b 是常数，i 是循环索引变量，也就说这个索引是仿射的。如果多维数组每一维的索引都是仿射的，那就称这个多维数组的索引是仿射的。稀疏数组访问（其典型形式为 x[y[i]]）是非仿射访问的主要示例之一。

要判断一个循环中对同一数组的两次访问之间是否存在相关，等价于判断两个仿射函数能否针对不同索引取同一个值（这些索引当然没有超出循环范围）。例如，假定我们已经以索引值 $a \times i+b$ 存储了一个数组元素，并以索引值 $c \times i+d$ 从同一数组中载入，其中 i 是FOR循环索引变量，其变化范围是 $m \sim n$。如果满足以下两个条件，则存在相关性。

318

(1) 有两个迭代索引 j 和 k，它们都在循环范围内。即 $m \leqslant j \leqslant n$、$m \leqslant k \leqslant n$。

(2) 此循环以索引 $a \times j+b$ 存储一个数组元素，然后以 $c \times k+d$ 提取同一数组元素。即 $a \times j+b=c \times k+d$。

一般来说，我们在编译时不能判断是否存在相关。例如，a、b、c 和 d 的值可能是未和的（它们可能是其他数组中的值），从而不可能判断是否存在相关。在其他情况下，在编译时进行相关测试的开销可能非常高，但的确可以确定是否存在；例如，可能要依靠多重嵌套循环的迭代索引来进行访问。但是，许多项目主要包含一些简单的索引，其中 a、b、c 和 d 都是常数。对于这些情况，有可能设计出一些合理的测试程序，在编译时测试。

举个例子，最大公约数（GCD）测试非常简单，但足以判定不存在相关的情况。它基于以下事实：如果存在循环间相关，那么 GCD(c, a)必须能够整除（$d-b$）。（回想一下，有两个整数 x、y，在计算 y/x 除法运算时，如果能够找到一个整数商，使运算结果没有余数，则说 x 能够整除 y。）

例题　使用GCD测试判断以下循环中是否存在相关：

```
for (i=0; i<100; i=i+1) {
      X[2*i+3] = X[2*i] * 5.0;
}
```

解答　给定值 a=2、b=3、c=2、d=0，那么 GCD(a,c)=2、$d-b$=−3。由于2不能整除−3，所以不可能存在相关。

GCD测试足以确保不存在相关，但在某些情况下，GCD测试成功但却不存在相关。例如，

这种情况可能因为 GCD 测试没有考虑循环范围。

一般来说，要确定是否实际存在相关，就是一个 NP 完全（NP-complete）问题。但实际上，有许多常见情况能够以低成本来准确分析。最近出现了一些既准确又高效的方法，它们使用不同层次的精确测试，通用性和成本都有所提高。（如果一个测试能够确切地判断是否存在相关，就说这一测试是确切的。尽管一般情况是“NP 完全”的，但对于一些有一定限制的情况，是存在确切测试的，其成本也要低廉得多。）

除了检测是否存在相关以外，编译器还希望划分相关的种类。编译器可以通过这种分类来识别名称相关，并在编译时通过重命名和复制操作来消除这些相关。 319

例题 下面的循环有多种类型的相关。找出所有真相关、输出相关和反相关，并通过重命名消除输出相关和反相关。

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

解答 4 个语句之间存在以下相关。

(1) 由于 Y[i]的原因，从 S1 至 S3、从 S1 至 S4 存在真相关。这些相关不是循环间相关，所以它们并不妨碍将该循环看作是并行的。这些相关将强制 S3 和 S4 等待 S1 完成。

(2) 从 S1 到 S2 有基于 X[i]的反相关。

(3) 从 S3 到 S4 有关于 Y[i]的反相关。

(4) 从 S1 到 S4 有基于 Y[i]的输出相关。

以下版本的循环消除了这些假（或伪）相关。

```
for (i=0; i<100; i=i+1 {
    T[i] = X[i] / c; /* Y 重命名为 T，以消除输出相关*/
    X1[i] = X[i] + c; /* X 重命名为 X1，以消除反相关*/
    Z[i] = T[i] + c; /* Y 重命名为 T，以消除反相关*/
    Y[i] = c - T[i];
}
```

在这个循环之后，变量 X 被重命名为 X1。在此循环之后的代码中，编译器只需要用 X1 来代替名称 X 即可。在这种情况下，重命名不需要进行实际的复制操作，通过替换名字或寄存器分配就可以完成重命名。但在其他情况下，重命名是需要复制操作的。

相关分析是一种非常关键的技术，不仅对开发并行如此，对于第 2 章介绍的转换分块也是如此。相关分析是检测循环级别并行的一种基本工具。要针对向量计算机、SIMD 计算机或多处理器进行有效的程序编译，都依赖于这一分析。相关分析的主要缺点是它仅适用于非常有限的一些情况，也就是用于分析单个循环嵌套中引用之间的相关以及使用仿射索引功能的情景。因此，在许多情况下，面向数组的相关分析不能告诉我们希望知道的内容；例如，在分析用指针而不是数据索引完成的访问时，可能要困难得多。（对于许多为并行计算机设计的科学应用程序，Fortran 仍然优于 C 和 C++，上述内容就是其中一个理由。）同理，分析过程调用之间的引 320

用也极为困难。因此，尽管依然需要分析那些以顺序语言编写的代码，但我们也需要一些编写显式并行循环的方法，比如OpenMP和CUDA。

4.5.2 消除相关计算

上面曾经提到，相关计算的最重要形式之一就是“递归”。点积是递归的一个完美示例：

```
for (i=9999; i>=0; i=i-1)
      sum = sum + x[i] * y[i];
```

这个循环不是并行的，因为它的变量求和存在循环间相关。但是，我们可以将它转换为一组循环，其中一个是完全并行的，而另一个可以是部分并行的。第一个循环将执行这一循环的完全并行部分。它看起来如下所示：

```
for (i=9999; i>=0; i=i-1)
      sum[i] = x[i] * y[i];
```

注意，这一求和已经从标量扩展到向量值（这种转换被称为**标量扩展**），通过这一转换使新的循环成为完全并行的循环。但是，在完成转换时，需要进行约简步骤，对向量的元素求和。类似如下所示：

```
for (i=9999; i>=0; i=i-1)
      finalsum = finalsum + sum[i];
```

尽管这个循环不是并行的，但它有一种非常特殊的结构，称为**约简**（reduction）。约简在线性代数中很常见，在第6章将会看到，它们还是仓库级计算机中所有主要并行原型MapReduce的关键部分。一般来说，任何函数都可用作约简运算符，常见情况中包含着诸如max和min之类的运算符。

在向量和SIMD体系结构中，约简有时是由特殊硬件处理的，使约简步骤的执行速度远快于在标量模式中的完成速度。具体做法是实施一种技术，类似于可在多处理器环境中实现的技术。尽管一般转换可以使用任意个处理器，但为简便起见，假定有10个处理器。在对求和进行约简的第一个步骤中，每个处理器执行以下运算（p是变化范围为0～9的进程号）：

```
for (i=999; i>=0; i=i-1)
      finalsum[p] = finalsum[p] + sum[i+1000*p];
```

这个循环在10个处理器中的每个处理器上对1000个元素求和，它是完全并行的。最后用简单的标量循环来完成最后10个总和的计算。向量和SIMD处理器中使用了类似的方法。

321

以上转换依赖于加法的结合性质，观察到这一点是非常重要的。尽管拥有无限范围与精度的算术运算具有结合性质，但计算机运算却不具备结合性：对于整数运算来说，是因为其范围有限；对于浮点运算来说，既有范围原因，又有精度原因。因此，使用这些重构技术有时可能会导致一些错误行为，尽管这种现象很少会发生。为此，大多数编译器要求显式启用那些依赖结合性的优化。

4.6 交叉问题

4.6.1 能耗与DLP：慢而宽与快而窄

数据级并行体系结构的主要功耗优势来自第1章的能耗公式。由于我们假定有充足的数据级并行，所以，如果将时钟频率折半、执行资源加倍：将向量计算机的车道数加倍，将多

媒体 SIMD 的寄存器和 ALU 加宽、增加 GPU 的 SIMD 车道数，那性能是一样的。如果我们在降低时钟频率的同时降低电压，那就可以降低计算过程的功耗和功率，同时保持峰值性能不变。因此，DLP 处理器的时钟频率可以低于系统处理器，后者依靠高时钟频率来获取性能（见 4.7 节）。

与乱序处理器相比，DLP 处理器可以采用较简单的控制逻辑，在每个时钟周期中启动大量计算；例如，这一控制对于向量处理器中的所有车道都是相同的，没有用于决定多指令发射的逻辑和推测执行逻辑。利用向量体系结构还可以轻松地关闭芯片中的未使用部分。在发射指令时，每条向量指令都明确指明它在大量周期内所需要的全部资源。

4.6.2 分组存储器和图形存储器

4.2 节提到了实际存储器带宽对于向量体系结构支持单位步幅、非单位步幅和集中-分散访问的重要性。

为了实现高性能，GPU 也需要充足的存储器带宽。专为 GPU 设计的特殊 DRAM 芯片可以帮助提供这一带宽，这种芯片被称为 GDRAM，即图形 DARM。与传统 DARM 芯片相比，GDRAM 芯片的带宽较高，容量较低。为了提供这一带宽，GDRAM 芯片经常被直接焊接在 GPU 所在的同一电路板上，而不是像系统存储器那样设置在 DIMM 模块中，DIMM 是插在主板插槽中的。DIMM 模块便于系统升级和提供更大的容量，这一点与 GDRAM 不同。这一有限容量（2011 年大约为 4 GB）与解决更大问题的目标相冲突，随着 GPU 计算能力的增长，这冲突将成为它的一个必然趋势。

322

为了提供最佳性能，GPU 试图考虑 GDRAM 的所有特性。它们在内部通常被安排为 4 ~ 8 组，行数是 2 的幂（通常为 16 384），每行的位数也是 2 的幂（通常为 8192）。第 2 章介绍了 GPU 尝试匹配的一些 DRAM 行为细节。

在给出计算任务及图形加速任务对 GDRAM 的所有潜在要求之后，存储器系统可能会面对大量的不相关请求。然而，这种多样性会伤害到存储器性能。为了应对这种情况，GPU 的存储器控制器为不同 GDRAM 组设定分离的通信量限度队列，要等到具有足够的通信量后才会打开一行，并同时传送所请求的全部数据。这一延迟提升了带宽，但使延迟时间增长，控制器必须确保所有处理过程不会因为等待数据而“挨饿”，否则，相邻的处理器可能会处于空闲状态。4.7 节显示了集中-分散技术，与基于缓存的传统体系相比，那些考虑了存储器组的访问技术可以提高性能。

4.6.3 步幅访问和TLB缺失

步幅访问的一个问题是它们如何与转换旁视缓冲区（TLB）进行交换，以在向量体系结构或 GPU 中获得虚拟存储器。（GPU 使用 TLB 来实现存储器映射。）根据 TLB 的组织方式以及存储器中受访数组的大小，甚至有可能在每次访问数组的元素时都会遇到一次 TLB 缺失。

4.7 融会贯通：移动与服务器 GPU、Tesla 与 Core i7

由于图形应用程序如此普及，所以现在的移动客户端以及传统的服务器或高效桌面计算机中都可以看到 GPU 的身影。表 4-12 列出了 NVIDIA Tegra 2 和 Fermi GPU 的重要特性，前者用于 LG Optimus 2X 中，运行 Android 操作系统，后者用于服务器中。GPU 服务器工程师希望能

够在一部电影发行后的五年内完成仿真动画。GPU移动工程师则希望再过五年后，移动客户端能够完成今天的服务器或游戏主机所完成的工作。更具体地说，总体目的是，希望到2015年能够在服务器GPU上实时实现诸如《阿凡达》[①]这样一部电影的图形质量，而到2020年可以在移动GPU上实现同一质量。

表4-12　用于移动客户端及服务器的GPU的关键特性

	NVIDIA Tegra 2	NVIDIA Fermi GTX 480
市场	移动客户端	桌面机、服务器
系统处理器	双核ARM Cortex-A9	不适用
系统接口	不适用	PCI Express 2.0 × 16
系统接口带宽	不适用	6 GB/s（每个方向）、12 GB/s（总共）
时钟频率	最高1 GHz	1.4 GHz
SIMD多处理器	不可用	15
SIMD车道/SIMD多处理器	不可用	32
存储器接口	32位LP-DDR2/DDR2	384位GDDR5
存储器带宽	2.7 GB/s	177 GB/s
存储器容量	1 GB	1.5 GB
晶体管	242 M	3030 M
工艺	40 nm TSMC 工艺G	40 nm TSMC 工艺G
晶片面积	57 mm^2	520 mm^2
功率	1.5 瓦	167 瓦

* Tegra 2是Android OS的参考平台，LG Optimus 2X移动电话中采用了这一处理器。

供移动设备使用的NVIDIA Tegra 2在使用单一物理存储器的单块芯片上同时提供系统处理器和GPU。系统处理器是一个双核ARM Cortex-A9，每个核心采用乱序执行和双指令发射。每个核心包括可选的浮点单元。

GPU为可编程像素着色、可编程顶点与光照和3D图形提供了硬件加速，但没有包含运行CUDA或OpenCL程序所需要的GPU计算功能。

323

在40 nm TSMC工艺中，晶片大小为57 mm^2（7.5 × 7.5 mm），包含2.42亿个晶体管。功率为1.5瓦。

表4-12中的NVIDIA GTX 480是第一次实现Fermi体系结构的GPU。时钟频率为1.4 GHz，其中包括15个SIMD处理器。芯片本身拥有16个处理器，但为了提高合格率，对于这一产品而言，只需要其中的15个处理器正常工作。连到GDDR5存储器的路径宽度为384（6 × 64），时钟频率为1.84 GHz，在双数据率存储器的两个时钟沿上传送数据，所以峰值存储器带宽为177 GB/s。它通过PCI Express 2.0 × 16链路连接到主机系统处理器和存储器，其峰值双向速率为12 GB/s。

GTX 480晶片的所有物理特性都非常庞大：包含30亿个晶体管，采用40 nm TSMC工艺，晶片大小为520 mm^2（22.8 mm× 22.8 mm），典型功率为167瓦。整个模块的功率为250瓦，其中包括GPU、GDRAM、风扇、功率调节器，等等。

① 电影《阿凡达》的上映时间是2010年。——编者注

对比GPU与具有多媒体SIMD的MIMD

Intel 的一群研究人员发表了一篇论文[Lee 等人 2010]，将具有多媒体 SIMD 扩展的四核 Intel i7（见第 3 章）与上一代 GPU-Tesla GTX 280 进行对比。表 4-13 列出了两个系统的特性。这两种产品都是在 2009 年秋天生产的。Core i7 采用 Intel 的 45 nm 半导体技术，而 GPU 则采用 TSMC 的 65 nm 技术。尽管由中立方或两个利益方都进行这一对比更公正一些，但这一节的目的不是确定这件产品比那件产品快多少，而是希望理解这两种相对体系结构类型的相对特征值。 324

表 4-13 Intel Core i7-960、NVIDIA GTX 280 和 GTX 480 技术指标

	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
处理元素数目（核心或SM）	4	30	15	7.5	3.8
时钟频率（GHz）	3.2	1.3	1.4	0.41	0.44
晶片大小	263	576	520	2.2	2.0
工艺	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1.6	1.0
功率（芯片，不是模块）	130	130	167	1.0	1.3
晶体管	700 M	1400 M	3030M	2.0	4.4
存储器带宽（GB/s）	32	141	177	4.4	5.5
单精度SIMD宽度	4	8	32	2.0	8.0
双精度SIMD宽度	2	1	16	0.5	8.0
峰值单精度标量FLOPS（GFLOP/s）	26	117	63	4.6	2.5
峰值单精度SIMD FLOPS（GFLOP/s）	102	311～933	515～1344	3.0～9.1	6.6～13.1
（SP 1加或乘）	N.A.	(311)	(515)	(3.0)	(6.6)
（SP 1指令融合乘-加）	N.A.	(622)	(1344)	(6.1)	(13.1)
（罕见的SP双精度融合乘-加与乘）	N.A.	(933)	N.A.	(9.1)	—
峰值双精度SIMD FLOPS（GFLOP/s）	51	78	515	1.5	10.1

* 最右边两列给出 GTX 280 与 GTX 480 与 Core i7 之比。对于 GTX 280 上的单精度 SIMD FLOPS，较高速度（933）源于一种非常罕见的情况：融合乘-加与乘法的双发射。对于单融合乘-加来说，较合理的值是 622。尽管此实例研究是在 280 与 i7 之间的对比，但由于本章描述了 480，所以图中也包含了 480，以显示它与 280 的关系。注意，图 4-16 中的这些存储器带宽更高一些，这是因为本表中为 DRAM 管脚带宽，而图 4-16 中则是由基准测试程序在处理器测出的。（摘自 Lee 等人[2010]论文中的表 2。）

图 4-16 中 Core i7 920 和 GTX 280 Roofline 模型显示了两种计算机之间的差别。920 的时钟频率慢于 960（分别是 2.66 GHz、3.2 GHz），但系统的其他部分都是相同的。GTX 280 不仅拥有高得多的存储器带宽和双精度浮点性能，而且它的双精度脊点也要明显靠左。如前文所述，Roofline 的脊点越靠左，就越容易达到峰值计算性能。GTX 280 的双精度脊点为 0.6，而 Core i7 则为 2.6。对于单精度性能，脊点大幅右移，这是因为它的单精度性能过高，所以要达到其峰值性能要难得多。注意，内核的运算密度取决于进入主存储器的字节数，而不是进入缓存存储器的字节数。因此，如果假定大多数引用实际指向缓存，那缓存可以改变特定计算机上内核的运算密度。Roofline 可以帮助解释这一实例研究中的相对性能。还要注意到，在两种体系结构中，这一带宽都是针对单位步幅访问的。后面将会看到，对于那些没有接合的集中-分散寻址，在 GTX 280 中要比在 Core i7 中慢一些。 325

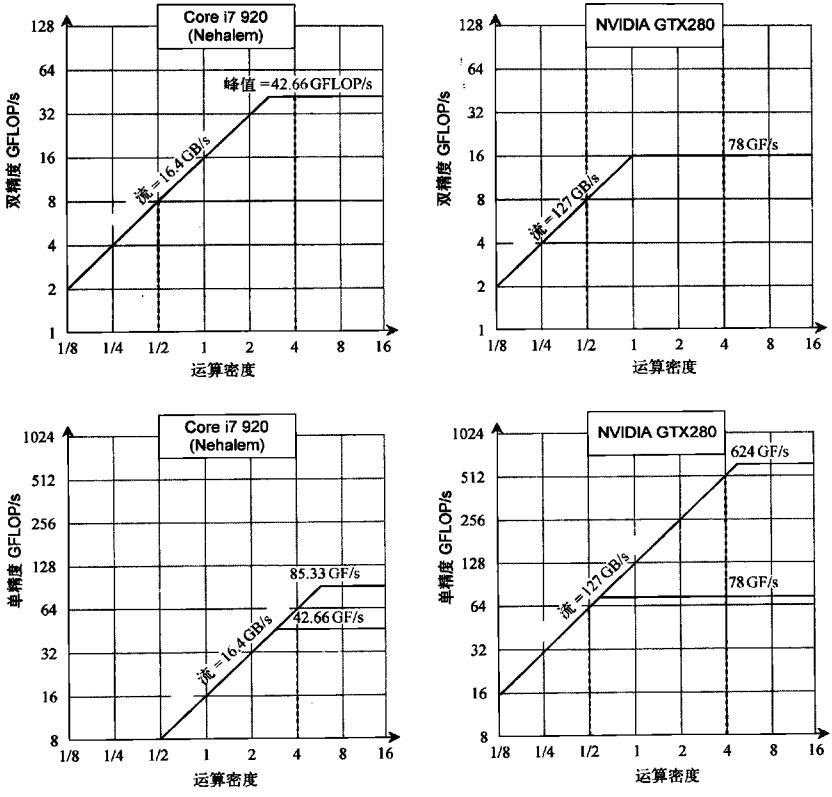


图 4-16　Roofline 模型，Williams 等人[2009]。这些 Roofline 在上半部分显示了双精度浮点性能，在下半部分中显示了单精度性能。（还给出了 DP FP 性能值，希望提供一些观察角度。）左边 Core i7 920 的峰值 DP FP 性能为 42.66 GFLOP/s，SP FP 峰值性能为 85.33 GFLOP/s，峰值存储器带宽为 16.4 GB/s。NVIDIA GTX 280 的 DP FP 峰值性能为 78 GFLOP/s，SP FP 峰值为 624 GFLOP/s，存储器带宽为 127 GB/s。左侧的垂直虚线表示等于 0.5 FLOP/B 的运算密度。对于 Core i7，它受存储器带宽的限制，不超过 8 DP GFLOP/s 或 8 SP GFLOP/s。右侧的虚线表示等于 4 FLOP/B 的运算密度。在 Core i7 上，它仅受计算限制，为 42.66 DP GFLOP/s 和 64 SP GFLOP/s，在 GTX 280 上为 78 DP GFLOP/s 和 512 DP GFLOP/s。为了达到 Core i7 的最高计算速率，需要使用所有 4 个核心和具有相同数量乘-加的 SSE 指令。对于 GTX 280，需要在所有多线程 SIMD 处理器上使用整合乘-加指令。Guz 等人[2009]给出了这两种体系结构的一种重要分析模型

据这些研究人员所述，他们在选择基准测试程序时，分析了最近提出的 4 个基准测试套件的计算特性与存储器特性，然后“设计了一组收集这些特性数值的吞吐量计算内核”。表 4-14 描述了这 14 种内核，表 4-15 显示了性能结果，数值越大，表示速度越快。

326
|
327

表 4-14 吞吐量计算内核特性值（摘自 Lee 等人[2010]的表 1）

内核	应用	SIMD	TLP	特性
SGEMM（SGEMM）	线性代数	常规	跨2D贴图	贴图之后的计算限制
蒙特卡洛（MC）	计算金融	常规	跨路径	计算限制
卷积（Conv）	图像分析	常规	跨像素	计算限制，小型滤波器的BW限制
FFT（FFT）	信号处理	常规	跨更小的FFT	计算限制或取决于大小的BW限制
SAXPY（SAXPY）	点积	常规	跨向量	大型向量的BW限制
LBM（LBM）	时间迁移	常规	跨单元	BW限制
约束解算器（Solv）	刚体物理学	集中/分散	跨约束条件	同步限制
SpMV（SpMV）	稀疏矩阵解算器	集中	跨非零	典型大矩阵的BW限制
GJK（GJK）	冲突检测	集中/分散	跨对象	计算限制
排序（Sort）	数据库	集中/分散	跨元素	计算限制
光线投射（RC）	体渲染	集中	跨射线	4～8 MB一级工作集，超过500 MB的末级工作集
搜索（Search）	数据库	集中/分散	跨查询	小型树的计算限制，大型树底的BW限制
柱状图（Hist）	图像分析	需要冲突检测	跨像素	还原/同步限制

* 括号中的名称表示本节的基准测试名称。作者指出，两种机器的代码都进行了同样的优化努力。

表 4-15 在两个平台上测量的原始性能及相对性能

内核	单位	Core i7-960	GTX 280	GTX 280/i7-960
SGEMM	GFLOP/s	94	364	3.9
MC	十亿路径/秒	0.8	1.4	1.8
Conv	百万像素/秒	1250	3500	2.8
FFT	GFLOP/s	71.4	213	3.0
SAXPY	GB/s	16.8	88.8	5.3
LBM	百万查询/秒	85	426	5.0
Solv	帧/秒	103	52	0.5
SpMV	GFLOP/s	4.9	9.1	1.9
GJK	帧/秒	67	1020	15.2
Sort	百万元素/秒	250	198	0.8
RC	帧/秒	5	8.1	1.6
Search	百万查询/秒	50	90	1.8
Hist	百万像素/秒	1517	2583	1.7
Bilat	百万像素/秒	83	475	5.7

* 在这一研究中，SAXPY 仅用作存储器带宽的一种度量，所以右边的单位为 GB/s，而不是 GFLOP/s（根据[Lee 等人 2010]中的表 3。）

已知 GTX 280 的原始性能技术指标由较低的 2.5 倍变化到较快的 7.5 倍（每芯片的核心数），而性能由较慢的 2.0 倍（Solv）变化到 15.2 倍（GJK），Intel 研究人员研究了这些差异的原因主要有以下几方面。

- **存储器带宽**。GPU的存储器带宽为4.4倍，它可以帮助解释为什么LBM和SAXPY的运行速度要快5.0和5.3倍；它们的工作集为数百MB，因此不能放到Core i7缓存中。（为了密集访问存储器，它们没有在SAXPY上使用缓存分块。）因此，Roofline的斜率解释了它们的性能。SpMV还有一个大型工作集，但由于GTX 280的双精度浮点仅比Core i7快1.5倍，所以SpMV的速度仅为1.9倍。（回想一下，Fermi GTX 480的双精度比Tesla GTX 280快4倍。）
- **计算带宽**。其余内核中有5个是计算限制：SGEMM、Conv、FFT、MC和Bilat。GTX分别快3.9、2.8、3.0、1.8和5.7。前3个使用单精度浮点运算，GTX 280单精度快3～6倍。（只有在GTX 280每个时钟周期可发射一个融合乘加和乘法指令的极特殊情况下，才会出现表4-13所示比Core i7快9倍的情况。）MC使用双精度，它解释了为什么DP性能仅快1.5倍，MC仅快1.8倍。Bilat使用超越函数，GTX 280直接支持这些函数（见表4-7）。Core i7将三分之二的时间花费在计算超越函数上，所以GTX 280快5.7倍。这一观察结果有助于指出硬件支持对工作负载中某些运算的价值：双精度浮点，还有可能包括超越函数。

328

- **缓存优势**。光线投射（RC）在GTX上仅快1.6倍，这是因为利用Core i7进行缓存分块可以防止它像在GPU上一样，变为存储器带宽限制。缓存分块也可以帮助Search。如果索引树很小，可以放在缓存中，那么Core i7将会快2倍。索引树较大时，则会成为存储器带宽限制。整体上，GTX 280执行搜索的速度快1.8倍。缓存分块也对Sort有所帮助。大多数程序员不会在SIMD处理器上运行Sort，可以用一个称为分割的1位Sort原型来编写它。但是，分割算法执行的指令数要比标量排序多很多。结果，GTX 280的运行速度仅是Core i7的0.8倍。注意，缓存还可以帮助Core i7上的其他内核，这是因为缓存分块允许SGEMM、FFT和SpMV成为计算限制。这一观察结果再次强调了第2章缓存分块优化的重要性。（了解Fermi GTX 480中的缓存将会如何影响这一段提到的6种内核是很有意义的。）
- **集中-分散**。如果数据分散在主存储器内，那多媒体SIMD扩展的帮助就不是很大；只有当数据与16字节边界对齐时，才会产生优化性能。因此，在Core i7上，GJK从SIMD中获得的好处很少。前面曾经提到，GPU提供了集中-分散寻址，在向量体系结构中使用了这一技术，但在SIMD扩展中则被忽略。地址接合单元也可以提供帮助：合并对相同DRAM线的访问，从而减少集中与分散的数目。存储器控制器还会对相同DRAM页面进行批访问。这种组合意味着GTX 280运行GJK的速度要比Core i7快出15.2倍之多，这一数字要比表4-13中给出的任一单个物理参数都要大。这一观测结果再次强调了集中-分散对向量与GPU体系结构的重要性，这是SIMD扩展中所没有的。
- **同步**。同步的性能受原子更新的限制，尽管Core i7中有硬件提取与递增指令，但原子更新仍然占Core i7总运行时间的28%。因此，Hist在GTX 280上仅快1.7倍。前面曾经提到，Fermi GTX 480的原子更新比Tesla GTX 280上快5～20倍，所以在这一较新GPU上运行Hist同样是有意义的。Solv以少量带有屏障同步的计算克服了一批独立约束。Core i7获益于原子指令和存储器一致性模型，即使还没有完成对存储器层次结构的所有先前访问，也能确保得到正确结果。GTX 280版本中没有存储器一致性模型，它会从系统处理器启动一些批处理指令，使GTX 280的运算速度为Core i7的二分之一。这一观察点指出，同步性能对于某些数据并行问题是多么的重要。

329

Intel研究人员选择的内核在Tesla GTX 280中暴露了许多弱点，而这些弱点往往在Tesla的后续体系结构中得以克服，例如，Fermi拥有更快速的双精度浮点性能、原子操作和缓存。（在一项相关研究中，IBM研究人员给出了相同的观测结果[Bordawekar 2010]。）还有一点也非常重要：向量体系结构对分散-集中的支持要比SIMD指令早几十年，它对于有效应用这些SIMD扩展非常重要，在进行这一对比之前就已经有人对此进行了预测[Gebis和Patterson 2007]。Intel研究人员指出，这14个内核中的6个能够更好地利用SIMD，在Core i7上拥有更高效的集中-分散支持。这一研究同样确定了缓存分块的重要性。了解未来各代多核及GPU硬件、编译器和库是如何响应这些提高内核性能的特性，也是很有意义的。

我们希望对比更多的多核GPU。注意，这一对比中省略了一个重要的特性，也就是为没有给出为在这两种系统上获得运行结果所付出的努力。未来的对比最好还发布这两种系统上使用的代码，使其他人能够在不同硬件平台上再现相同试验，并可能对结果进行改进。

4.8 谬论与易犯错误

尽管，从程序员的角度来看，数据级并行是ILP之后的最简单并行形式，从架构师的角度来看也可能是最简单的形式，但它仍然有许多谬论和易犯错误。

谬论 *GPU因作为协处理器而窘迫。*

尽管主存储器与GPU之间的分割有一些缺点，但与CPU保持一定距离也有一些优势。

例如，PTX之所以存在，部分原因就在于GPU实质上是一种I/O设备。编译器与硬件之间的间接水平为GPU架构师提供了很大的灵活性，这一点要远远高于系统处理器架构师。人们通常很难事先知道一种体系结构的创新是否受到编译器和库的支持，是否对应用程序非常重要。有时，一种新的机制对于一两代是有效的，之后，其重要性就会随着IT世界的变化而下降。PTX使GPU架构师大胆尝试创新，如果它们令人失望或者重要性下降，就会在后续GPU中删除，从而鼓励架构师进行实践。一项创新要被包含在系统处理器中，那就必须具备非常充分的理由（此后可以进行的试验要少得多），这是因为：一旦发布二进制机器码，通常就意味着在该体系结构的所有后续版本中都必须支持这些新特性。

Fermi体系结构快速地改变了硬件指令集（从类似于x86的面向存储器转为类似于MIPS的面向寄存器，另外，还将地址大小加倍，变为64位），但没有对NVIDIA软件栈产生任何影响，这一事例充分体现了PTX的价值。 330

易犯错误 *关注向量体系结构的峰值性能，忽略启动开销。*

早期存储器-存储器向量处理器（比如TI ASC和CDC STAR-100）的启动时间都很长。对于某些向量问题，向量长度必须大于100时才能使向量代码快于标量代码！在CYBER 205（派生自STAR-100）上，DAXPY的启动开销为158个时钟周期，它充分提高了收支平衡点。如果Cray-1和CYBER 205的时钟频率相同，当向量长度小于64时，Cray-1要更快一些。因为Cray-1的时钟也要更快一些（尽管205更新），所以当向量长度大于100之后，CYBER 205才会更快。

易犯错误　提高向量性能，但却没有相应地提高标量性能。

在许多早期向量处理器中，这些不均衡性都成为一个问题，这也正是Seymour Cray（Cray计算机的架构师）改写规则的地方。许多早期向量处理器的标量单元都比较慢（启动开销也比较大）。即使到了今天，向量性能较低但标量性能较佳的处理器，也要优于峰值向量性能较佳的处理器。良好的标量性能可以降低开销成本（比如条带挖掘），并降低Amdahl定律的影响。

有一个好的例子就是对比快速标量处理器和标量性能较差的向量处理器。Livermore Fortran内核是一组24个向量化程度不同的科学内核。表4-16显示了在这一基准测试上测得两种不同处理器的性能。尽管向量处理器的峰值性能较高，但由于其标量性能较低，从而拖慢了快速标量处理器，从测得的调和平均值可以看出这一点。

表4-16　Livermore Fortran内核在两个不同处理器上的性能测量值

处理器	任意循环的最低速率（MFLOPS）	任意循环的最高速率（MFLOPS）	所有24个循环的调和平均值（MFLOPS）
MIPS M/120-5	0.80	3.89	1.85
Stardent-1500	0.41	10.08	1.72

* MIPS M/120-5和Stardent-1500（正式名称为Ardent Titan-1）都使用16.7 MHz MIPS R2000芯片作为主CPU。Startdent-1500使用其向量单元进行标量浮点运算，其标量性能（用最低速率测试）是MIPS M/120-5的一半，后者使用MIRS R2010 FP芯片。对于可以调度向量化的循环，向量处理器要快2.5倍以上（最高频率）。但是，在对所有24个循环测量总性能的调和平均值时，Stardent-1500较差的标量性能抵消了其较好的向量性能。

331

今天，这一危险颠倒了过来：提高了向量性能（比如通过增加车道数目），而不提高标量性能。这种短视是另一个导致非平衡计算机的因素。

下一个谬论与此密切相关。

谬论　可以在不提高存储器带宽的情况下获得良好的向量性能。

通过DAXPY循环和Roofline模型可以看出，存储器带宽对所有SIMD体系结构都非常重要。DAXPY进行每次浮点运算需要1.5次存储器引用，这一数值在许多科学代码都非常典型。即使浮点运算不占时间，Cray-1也不能提高所用向量序列的性能，因为它受到存储器的限制。当编译器使用分块来改变计算，以在向量寄存器中保存所有值时，Cray-1执行Linpack的性能会有跳跃。这种方法降低了每个FLOP的存储器引用数目，将性能提升将近两倍！因此，对于以前需要更多带宽的循环来说，Cray-1上的存储器带宽就足够了。

谬论　在GPU上，如果存储器性能不够好，只需要添加更多线程就可以了。

GPU使用许多CUDA线程来隐藏到主存储器的延迟。如果存储器访问被分散，或者各个CUDA线程之间没有相关，那么存储器系统在响应每个请求时会明显变慢。最终，即使有许多线程也不能隐藏延迟。为使“增加CUDA线程”的策略生效，不仅需要大量CUDA线程，而且这些CUDA线程自身还必须在存储器地址局域性方面有良好表现。

4.9　结语

随着个人移动设备上各种应用的普及，显示了音频、视频和游戏和这些游戏上的重要性，数据级并行对个人移动设备的重要性也在增加。如果有一种比任务级并行更简单的编程模型，

而且可能具有更佳的性能效率，很容易就能预测到数据级并行在接下来十年中的复兴。事实上，我们已经看到一些产品对数据级并行的重视，GPU 和传统处理器都已经在增加 SIMD 车道的数目，其增加速度至少与添加处理器的速度一样快（见图 4-1）。

因此，我们看到系统处理器正在拥有更多的 GPU 特性，反之亦然。传统处理器和 GPU 的一个最大性能差别是集中-分散寻址。传统的向量体系结构说明如何向 SIMD 指令添加此类寻址，我们希望随着时间的推移，越来越多的在向量体系结构中得到证明的好思想能够添加到 SIMD 扩展中。

332

在 4.4 节开头时曾经说过，GPU 问题并不是简单地问一句哪种体系结构最好，而是当硬件投入能够出色地完成图形处理时，如何对其进行改进以支持更具一般性的计算任务？尽管向量体系结构名义上有许多优势，但向量体系结构能否像 GPU 一样成为出色的图形处理基础，还有待证明。

GPU SIMD 处理器和编译器仍然是相对简单的设计。为了提高 GPU 的利用率，特别是因为 GPU 计算应用程序刚刚开始进入开发阶段，所以随着时间的推移可能会引入更积极的技术。通过研究这些新程序，GPU 设计人员肯定会发现和实现新的机器优化方法。有这样一个问题：在向量处理器中用于节省硬件和能耗的标量处理器（或控制处理器），是否会出现在 GPU 中？

Fermi 体系结构已经引入了传统处理器中的许多功能，使 GPU 更具主流特性，但两者之间还是有一些需要弥补的差距。以下是我们希望在不远的将来能够解决的问题。

- **可虚拟化 GPU**。事实已经证明，虚拟化对于服务器非常重要，它是云计算的基础（见第 6 章）。为了在云中包括 GPU，它们也应当像它们要连接的处理器和存储器一样是可虚拟化的。
- **GPU 存储器的容量较小**。加快运算速度的意义就体现在解决更大规模的问题上，而更大规模的问题通常需要更大的存储器足迹。GPU 在速度与大小方面的这种不一致性可以通过增加存储器容量来解决。其挑战是要在提高容量的同时还能保持高带宽。
- **直接对 GPU 存储器进行 I/O 操作**。真实程序对存储器设备和帧缓冲区进行 I/O 操作，大型程序可能需要大量 I/O 操作和相当多的存储器。今天的 GPU 系统必须在 I/O 设备和系统存储器之间进行传送，然后在系统存储器和 GPU 存储器之间传送。这种额外的跳转会显著降低一些程序的 I/O 性能，降低 GPU 的吸引力。Amdahl 定律警告我们：如果在加快一项任务的处理速度时忽视其中一部分，可能会发生什么样的问题。我们希望未来的 GPU 将所有 I/O 都看作一等公民，就像它今天处理帧缓冲区 I/O 一样。
- **统一的物理存储器**。对于以上两个问题有一种替代解决方案：为系统和 GPU 采用同一个物理存储器，一些廉价的 GPU 在 PMD 和膝上电脑上就是这样做的。AMD Fusion 体系结构是在刚刚完成本书英文版第 5 版时发布的，它是传统 GPU 和传统 CPU 的最初结合体。NVIDIA 也发布了 Project Denver，它在同一个地址空间中合并了 ARM 标量处理器和 NVIDIA GPU。在这些系统交付时，了解它们是如何紧密集成的，以及这种集成对于数据级并行及图形应用产生的性能与能耗影响，都是很有意义的。

在研究了众多 SIMD 版本之后，下一章将开始钻研 MIMD 领域。

333

4.10　历史回顾与参考文献

附录L.6节讨论了Illiac IV(早期SIMD体系结构的代表)和Cray-1(向量体系结构的代表)。该附录还研究了多媒体SIMD扩展和GPU的历史。

案例研究与练习（Jason D. Bakos设计）

案例研究：在向量处理器和GPU上实施向量内核

本案例研究说明的概念

- 向量处理器编程
- GPU编程
- 性能评估

MrBayes是一个著名的常用计算生物学应用程序，可以根据一组输入物种的多重比对DNA序列数据（长度为n）来推测这一物种的进化历史。MrBayes的工作方式是对所有二叉树拓扑空间（这些输入物种就是这个树的叶子）执行试探性搜索。要对具体树进行求值，应用程序必须为每个内部节点计算一个$n\times4$的条件似然表（名称为clP）。这个表是另外4个表的函数，其中两个表分别是该节点两个子节点的条件似然表（clL和clR，单精度浮点），另外两个表是相关的$n\times4\times4$转移概率表（tiPL和tiPR，单精度浮点）。这个应用程序的内核之一就是计算这个条件似然表，如下所示：

```
for (k=0; k<seq_length; k++) {
   clP[h++] = (tiPL[AA]*clL[A] + tiPL[AC]*clL[C] + tiPL[AG]*clL[G] + tiPL[AT]*clL[T])
   *(tiPR[AA]*clR[A] + tiPR[AC]*clR[C] + tiPR[AG]*clR[G] + tiPR[AT]*clR[T]);
   clP[h++] = (tiPL[CA]*clL[A] + tiPL[CC]*clL[C] + tiPL[CG]*clL[G] + tiPL[CT]*clL[T])
   *(tiPR[CA]*clR[A] + tiPR[CC]*clR[C] + tiPR[CG]*clR[G] + tiPR[CT]*clR[T]);
   clP[h++] = (tiPL[GA]*clL[A] + tiPL[GC]*clL[C] + tiPL[GG]*clL[G] + tiPL[GT]*clL[T])
   *(tiPR[GA]*clR[A] + tiPR[GC]*clR[C] + tiPR[GG]*clR[G] + tiPR[GT]*clR[T]);
   clP[h++] = (tiPL[TA]*clL[A] + tiPL[TC]*clL[C] + tiPL[TG]*clL[G] + tiPL[TT]*clL[T])
   *(tiPR[TA]*clR[A] + tiPR[TC]*clR[C] + tiPR[TG]*clR[G] + tiPR[TT]*clR[T]);
   clL += 4;
   clR += 4;
   tiPL += 16;
   tiPR += 16;
}
```

334

4.1　[25] <4.2、4.3>假定有如表4-17所示的常量。给出MIPS和VMIPS的代码。假定我们不能使用分散-集中载入或存储。假定tiPL、tiPR、clL、clR和clP的起始地址分别在RtiPL、RtiPR、RclL、RclR和RclP中。假定VMIPS寄存器长度可由用户编程设定，可以通过设定特殊寄存器VL（即li VL 4）来指定。为便于简化向量加法，假定向VMIPS中添加以下指令：

```
SUMR.S Fd, Vs Vector Summation Reduction Single Precision:
```

表4-17　本案例研究的常量和取值

常　量	值
AA、AC、AG、AT	0、1、2、3
CA、CC、CG、CT	4、5、6、7
GA、GC、GG、GT	8、9、10、11
TA、TC、TG、TT	12、13、14、15
A、C、G、T	0、1、2、3

这个指令对向量寄存器 Vs 执行求和化简，并将总和写到标量寄存器 Fd 中。

4.2 [5] <4.2、4.3>假定 seq_length = 500，两种实施的动态指令数为多少？

4.3 [25] <4.2、4.3>假定在向量功能单元上执行向量化简指令，它类似于向量加法指令。假定每个向量功能单元只有一个实例，给出代码序列是如何安排的。此代码需要多少次钟鸣？每个 FLOP 需要多少个周期（忽略向量指令发射开销）？

4.4 [15] <4.2、4.3>现在假定我们可以使用分散-集中载入和存储指令（LVI 和 SVI）。假定 tiPL、tiPR、clL、clR 和 clP 在存储器中顺序排列。例如，如果 seq_length=500，则 tiPR 数组将在 tiPL 数组之后 500*4 字节开始。这将如何影响为这个内核编写 VMIPS 代码的方式？假定可以使用以下技术以整数来初始化向量寄存器，此技术将以数值（0，0，2000，2000）来初始化向量寄存器：

```
LI R2,0
SW R2,vec
SW R2,vec+4
LI R2,2000
SW R2,vec+8
SW R2,vec+12
LV V1,vec
```

335

假定最大向量长度为 64。使用集中-分散装入指令有没有办法提高性能？如果有，可以提高多少？

4.5 [25] <4.4>现在假定我们希望使用单个线程块在 GPU 上实施 MrBayes 内核。使用 CUDA 改写内核的 C 代码。假定指向条件似然表和转移概率表的指令以内核参数的形式指定。为循环的每个迭代调用一个线程。对于任何需要重复使用的值，应当先将其载入共享存储器中，然后再进行操作。

4.6 [15] <4.4>利用 CUDA，我们可以使用块级别的粗粒度并行来并行计算多个节点的条件似然。假定我们希望由树的底部向上计算条件似然。假定条件似然和转移概率数组在存储器中的组织方式如问题 4 所述，12 个叶节点中每个节点的表组也按照节点顺序存储在连续存储器地址中。假定我们希望计算为节点 12 至节点 17 计算条件似然，如图 4-17 所示。修改在解答练习 4.5 时计算数组索引的方法，将块编号包含在内。

4.7 [15] <4.4>将练习 4.6 的代码转换为 PTX 代码。这个内核需要多少条指令？

4.8 [10] <4.4>你认为这一代码在 GPU 上执行得怎么样？对你的回答给出解释。

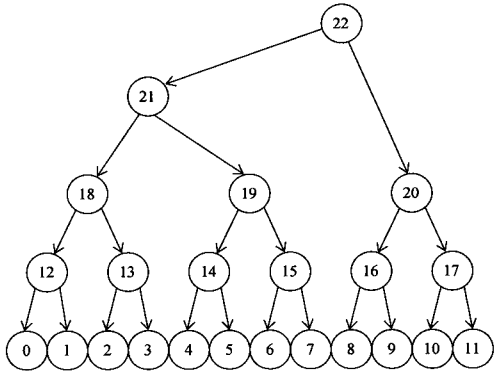


图 4-17 示例树

336

练习

4.9　[10/20/20/15/15] <4.2>考虑以下代码，它将两个包含单精度复数值的向量相乘：

```
for (i=0;i<300;i++) {
        c_re[i] = a_re[i] * b_re[i] - a_im[i] * b_im[i];
        c_im[i] = a_re[i] * b_im[i] + a_im[i] * b_re[i];
}
```

假定处理器的运行频率为700MHz，最大向量长度为64。载入/存储单元的启动开销为15个时钟周期，乘法单元为8个时钟周期，加法/减法单元为5个时钟周期。

a. [10] <4.2>这个内核的运算密度为多少？给出理由。

b. [20] <4.2>将此循环转换为使用条带挖掘的VMIPS汇编代码。

c. [20] <4.2>假定采用链接和单一存储器流水线，需要多少次钟鸣？每个复数结果值需要多少个时钟周期（包括启动开销在内）？

d. [15] <4.2>如果向量序列被链接在一起，每个复数结果值需要多少个时钟周期（包含开销）？

e. [15] <4.2>现在假定处理器有三条存储器流水线和链接。如果该循环的访问过程中没有组冲突，每个结果需要多少个时钟周期？

4.10　[30] <4.4>在这个问题中，我们将对比向量处理器与一种混合系统的性能，这一混合系统包含一个标量处理器和一个基于GPU的协处理器。在混合系统中，主机处理器的标量性能优于GPU，所以在这种情况下，所有标量代码都在主机处理器上执行，而所有向量代码都在GPU上执行。我们将第一种系统称为向量计算机，将第二种系统称为混合计算机。假定你的目标应用程序包含一个向量内核，运算密度为0.5FLOP/被访问DRAM字节；但是，这个应用程序还有一个标量组件，必须在此内核之前和之后执行该组件，以分别准备输入向量和输出向量。对于示例数据集，此代码的标量部分在向量处理器和混合系统的主机上都需要400ms的执行时间。此内核读取包含200MB数据的输入向量，输出数据包含100MB数据。向量处理器的峰值存储器带宽为30GB/s，GPU的峰值存储器带宽为150 GB/s。混合系统有一些额外开销，在调用该内核前后，需要在主存储器和GPU本地存储器之间传送所有输入向量。此混合系统的直接存储器访问（DMA）带宽为10 GB/s，平均延迟为10 ms。假定向量处理器和GPU的性能都受存储器带宽的限制。计算两种计算机执行这一应用程序所需要的执行时间。

337

4.11　[15/25/25] <4.4、4.5>4.5节讨论了化简运算，它通过重复应用一种运算而将向量简化为标量。化简是一种特殊类型的循环递归。下面给出一个例子：

```
dot=0.0;
for (i=0;i<64;i++) dot = dot + a[i] * b[i];
```

向量化编译器可以应用一种被称为标量扩展的转换，它将dot扩展到向量中，并对循环进行分割，从而可以用向量运算来执行乘法，而将化简运算当作独立的标量运算：

```
for (i=0;i<64;i++) dot[i] = a[i] * b[i];
for (i=1;i<64;i++) dot[0] = dot[0] + dot[i];
```

4.5节曾经提到，如果允许浮点加法符合结合律，那就有几种技术可用于实现化简的并行化。

a. [15] <4.4、4.5>一种技术称为递归加倍，它对一个逐渐缩短的向量序列（即两个32元素向量，然后是两个16元素向量，以此类推）进行加法运算。说明C代码如何寻求以这种方式来执行第二个循环。

b. [25] <4.4、4.5>在一些向量处理器中，向量寄存器中的各个元素是可以单独寻址的。在这种情况下，一个向量运算的操作数可能是同一向量寄存器的两个不同部分。这就有了另外一种化简方案——部分求和。其思想是将向量化简为m次求和，其中m是通过该向量功能单元的总延迟，包括操作数读写时间。假定VMIPS向量寄存器是可寻址的（例如，可以用操作

数 V1(16)启动向量运算，表示输入操作数从元素 16 开始。）另外，假定加法运算的总延迟（包括运算数读取和结果写入）为 8 个时钟周期。写一段 VMIPS 代码序列，将 V1 的内容化简为八个部分求和。

c. [25] <4.4、4.5>在 GPU 上执行化简时，输入向量中的每个元素都有一个线程与其相关联。每个线程的第一步是将其相应值写到共享存储器中。接下来，每个线程进入一个循环，对每对输入值求和。每次迭代将元素数减半，在每次迭代后，活动线程数也会减半。为将化简性能提升至最大，应当将通过该循环过程的完整填充 Warp 数提升至最大。换句话说，活动线程应当是连续的。每个线程对共享数组进行索引的方式也应当避免在共享存储器中发生组冲突。以下循环仅违犯了这些指南的第一条，还用到了求模运算符，对 GPU 来说，这种运算符的成本是非常高的：

338

```
unsigned int tid = threadIdx.x;
 for(unsigned int s=1; s < blockDim.x; s *= 2) {
 if ((tid % (2*s)) == 0) {
 sdata[tid] += sdata[tid + s];
 }
 __syncthreads();
 }
```

重写该循环，使其满足这些指南，而且不再使用求模运算符。假定每个 Warp 中有 32 个线程，只要来自相同 Warp 的两个或更多个线程引用索引，而这些索引对 32 求模的结果相同，就会发生组冲突。

4.12 [10/10/10/10] <4.3>以下内核执行有限时域差分法（FDTD）的一部分，用来计算三维空间的 Maxwell 方程，它是 SPEC06fp 基准测试的一部分：

```
for (int x=0; x<NX-1; x++) {
 for (int y=0; y<NY-1; y++) {
 for (int z=0; z<NZ-1; z++) {
 int index = x*NY*NZ + y*NZ + z;
 if (y>0 && x >0) {
 material = IDx[index];
 dH1 = (Hz[index] - Hz[index-incrementY])/dy[y];
 dH2 = (Hy[index] - Hy[index-incrementZ])/dz[z];
 Ex[index] = Ca[material]*Ex[index]+Cb[material]*(dH2-dH1);
}}}}
```

假定 dH1、dH2、Hy、Hz、dy、dz、Ca、Cb 和 Ex 都是单精度浮点数组。假定 IDx 是无符号整数数组。

a. [10] <4.3>这一内核的运算密度为多少？

b. [10] <4.3>这一内核是否可以执行向量或 SIMD？说明理由。

c. [10] <4.3>假定这一内核将在存储器带宽为 30 GB/s 的处理器上执行，这一内核是受存储器的限制还是受计算的限制？

d. [10] <4.3>为这一处理器确定 Roofline 模型，假定其峰值计算吞吐量为 85 GFLOP/s。

4.13 [10/15] <4.4>假定有一种包含 10 个 SIMD 处理器的 GPU 体系结构。每条 SIMD 指令的宽度为 32，每个 SIMD 处理器包含 8 个车道，用于执行单精度运算和载入/存储指令，也就是说，每个非分岔 SIMD 指令每 4 个时钟周期可以生成 32 个结果。假定内核的分岔分支将导致平均 80%的线程为活动的。假定在所执行的全部 SIMD 指令中，70%为单精度运算、20%为载入/存储。由于并不包含所有存储器延迟，所以假定 SIMD 指令平均发射率为 0.85。假定 GPU 的时

339

钟速度为1.5 GHz。

a. [10] <4.4>计算这个内核在这个GPU上的吞吐量，单位为GFLOP/s。

b. [15] <4.4>假定我们有以下选项：

(1) 将单精度车道数增大至16。

(2) 将SIMD处理器数增大至15（假定这一改变不会影响所有其他性能度量，代码会扩展到增加的处理器上）。

(3) 添加缓存可以有效地将存储器延迟缩减40%，这样会将指令发射率增加至0.95，对于这些改进中的每一项。

吞吐量的加速比为多少？

4.14 [10/15/15] <4.5>在这一练习中，我们将研究几个循环，并分析它们在并行化方面的潜力。

a. [10] <4.5>以下循环是否存在循环间相关？

```
for (i=0;i<100;i++) {
 A[i] = B[2*i+4];
 B[4*i+5] = A[i];
 }
```

b. [15] <4.5>在以下循环中，找出所有真相关、输出相关和反相关。通过重命名来消除输出相关和反相关。

```
for (i=0;i<100;i++) {
 A[i] = A[i] * B[i]; /* S1 */
 B[i] = A[i] + c; /* S2 */
 A[i] = C[i] * c; /* S3 */
 C[i] = D[i] * A[i]; /* S4 */
```

c. [15] <4.5>考虑以下循环：

```
for (i=0;i < 100;i++) {
 A[i] = A[i] + B[i]; /* S1 */
 B[i+1] = C[i] + D[i]; /* S2 */
}
```

340

S1和S2之间是否存在相关？这一循环是否为并行的？如果不是，说明如何使其成为并行的。

4.15 [10] <4.4>列出并介绍至少4种可以影响GPU内核性能的因素。换句话说，哪些由内核代码导致的运行时行为会降低内核执行时的资源利用率？

4.16 [10] <4.4>假定一个虚设GPU具有以下特性：

- 时钟频率为1.5 GHz；
- 包含16个SIMD处理器，每个处理器包含16个单精度浮点单元；
- 片外存储器带宽为100 GB/s。

不考虑存储器带宽，假定所有存储器延迟可以隐藏，则这一GPU的峰值单精度浮点吞吐量为多少GFLOP/s？在给定存储器带宽限制下，这一吞吐量是否可持续？

4.17 [60] <4.4>对于这一编程练习，写出并描述CUDA内核的行为特征，这一内核中不仅包含大量数据级并行，还包含条件执行行为。使用NVIDIA CUDA工具套件和英属哥伦比亚大学的GPU-SIM (http://www.ece.ubc.ca/~aamodt/gpgpu-sim/)或者CUDA Profiler来编写并编译CUDA内核，对256 × 256棋盘执行Conway“生命游戏”的100次迭代，并将棋盘的最终状态返回给主机。假定这一棋盘由主机初始化。为每个单元格关联一个线程。确保在每次游戏迭代之后添加一个屏障。使用以下游戏规则。

- 对于任意存活单元格，如果其相邻的存活单元格少于两个，则该存活单元格死亡。

- 对于任意存活单元格，如果其相邻的存活单元格为两个或三个，则该存活单元格将生存到下一代。
- 对于任意存活单元格，如果其相邻的存活单元格超过三个，则该存活单元格死亡。
- 对于任意死亡单元格，如果它恰有三个相邻的存活单元格，则该死亡单元格变为存活单元格。

在完成该内核后，回答以下问题。

a. [60] <4.4>使用-ptx 选项编译代码，并查看该内核的 PTX 表示方式。有多少个 PTX 指令构成了该内核的 PTX 实施方式？该内核的条件部分是否包含分支指令，还是仅有可预测的非分支指令？

b. [60] <4.4>在模拟器上执行代码之后，动态指令数为多少？所实现的的每周期指令数（IPC）为多少？或者说指令发射率为多少？在控制指令、算术逻辑单元（ALU）指令和存储器指令方面，什么是动态指令分解？是否存在任何存储器组冲突？有效片外存储器带宽为多少？

c. [60] <4.4>对该内核进行改进，其中片外存储器引用被接合在一起，观察运行时性能的差别。

341

5

线程级并行

计算机组织结构在20世纪60年代中期开始偏离传统方式，在提升计算机运行速度的努力中，回报递减定律开始发挥作用……电子电路的运行速度最终受光速的限制……许多电路已经工作在纳秒级别。

——W. Jack Bouknight等人，
The Illiac IV System (1972)

我们所有未来产品的开发都专注于多核设计。我们相信这就是这一行业的转折点。

——Intel总裁Paul Otellini，
在2005年度Intel开发人员
论坛上介绍Intel的未来方向

5.1 引言

正如本章开篇引语所说，很多年之前，一些研究人员就持有这样一种观点：单处理器体系结构的发展已经接近尾声。显然，这种观点有点草率了；事实上，自 20 世纪 50 年代末、60 年代初出现第一批晶体管计算机以来，单处理器性能的增长速度在 1986 年至 2003 年期间达到最高峰，其背后的推动力就是微处理器的发展。

不过，在整个 20 世纪 90 年代，多处理器的重要性一直在不断增加，设计人员试图寻求一种构造服务器和超级计算机的方法，希望其性能优于单个微处理器。与此同时，人们还在探索大众商品化微处理器在性价比方面的巨大优势。我们在第 1 章至第 3 章中已经讨论过，由于开发指令级并行（ILP）的回报在降低，再加上对功率因素关注程度的增加，导致了单处理器的发展速率减缓，而这一减缓又将我们引入了计算机体系结构的一个新时代，在这个时代中，多处理器在从低端到高端的各个领域都扮演了重要角色。本章开头的第二条引语就指出了这个明显的转折点。

多重处理的重要性在不断提升，这反映了以下几个重要因素。

- 在 2000 年至 2005 年期间，设计人员尝试寻找和开发更多的 ILP，而事实表明，这种方法的效率很低，因为功率和硅成本的增长速度要快于性能的增长速度，因此，硅与能量的利用效率在这一时期激剧下降。如何使性能的增长速度超过基础技术的发展速度呢？除了 ILP 之外，（换个角度来看）我们所知道的唯一可伸缩、通用方式就是通过多重处理。
- 随着云计算和软件即服务变得越来越重要，人们对高端服务器的兴趣也在增加。
- 因特网上有海量数据可供利用，推动了数据密集型应用程序的发展。
- 人们认识到提高台式计算机性能的重要性在下降（至少图形处理功能之外的性能如此），要么是因为当前的性能已经可以接受，要么是因为计算高度密集、数据高度密集的应用程序都是在云中完成的。
- 人们更深入地了解到如何才能有效地利用多处理器，特别是在服务器环境中如何加以有效利用，这种环境中存在大量自然并行，而并行源于大型数据集、科学代码中的自然并行，或者大量独立请求之间的并行（请求级并行）。
- 通过可复用设计而不是独特设计来充分发挥设计投入的效用，所有多处理器设计都具备这一特点。

本章主要研究线程级并行（TLP）的开发。TLP 意味着存在多个程序计数器，因此主要通过 MIMD 加以开发。尽管 MIMD 的出现已经有几十年了，但在从嵌入式应用到高端服务器的计算领域中，线程级并行移向前台还是最近的事情。同样，线程级并行大量用于通用应用程序而不只是科学应用程序，也是最近的事情。 344

这一章的重点是**多处理器**，我们将**多处理器**定义为由紧耦合处理器组成的计算机，这些处理器的协调与使用由单一处理器系统控制，通过共享地址空间来共享存储器。此类处理器通过两种不同的软件模型来开发线程级并行。第一种模型是运行一组紧密耦合的线程，协同完成同一项任务，这种情况通常被称为**并行处理**。第二种模型是执行可能由一或多位用户发起的多个相对独立的进程，这是一种**请求级并行**形式，其规模要远小于将在下一章研究的内容。请求级并行可以由单个应用程序开发（这个应用程序在多个处理器上运行，比如响应查询请求的数据库程序），也可以由多个独立运行的应用程序开发，通常称为**多重编程**。

本章所研究多处理器的典型范围小到一个双处理器，大至包括数十个处理器，通过存储器的共享进行通信与协调。尽管通过存储器进行共享隐含着对地址空间的共享，但并不一定意味着只有一个物理存储器。这些多处理器既包括拥有多个核心的单片系统（称为**多核**），也包括由多个芯片构成的计算机，每个芯片可能采用多核心设计。

除了真正的多处理器之外，我们还将再次讨论多线程主题，这一技术支持多个线程以交错形式在单个多发射处理器上运行。许多多核处理器也包括对多线程的支持。

在下一章，我们将研究由大量处理器构建而成的超大规模计算机，这些处理器通过联网技术连接在一起，通常称为**集群**；这些大规模系统通常用于云计算，在其所采用的模型中，要么假定有大量独立请求，要么假定有高度并行、密集计算的任务。当这些集群发展到数万个服务器时，我们就称之为**仓库级计算机**。

除了这里研究的多处理器和下一章的仓库级系统之外，还有一些特殊的大规模多处理器，有时称之为**多计算机**，它们的耦合程度要低于本章研究的多处理器，但高于下一章的仓库级系统。这种多计算机主要应用于高端科学计算。有许多其他书籍对这些系统进行了详细介绍，比如 Cullr、Singh 和 Gupta[1999]。由于多重处理领域不但规模庞大，而且还在不断变化之中（刚刚提到的 Culler 等人撰写的参考文献有 1000 多页，但却只讨论了多重处理！），所以我们选择将注意力放在我们认为是计算领域中最重要的通用部分上。附录 I 结合大规模科学应用程序，讨论了在构建此类计算机时出现的一些问题。

345

综上所述，我们的焦点是含有中小量处理器（2～32 个）的多处理器。无论是在数量方面，还是在金额方面，此类设计都占据着主导地位。对于更大规模的多处理器设计领域（33 个或更多个处理器），我们仅给予一点点关注，主要在附录 I 中对其进行讨论，其中包含了此类处理器设计的更多方面，以及并行科学工作负载的行为性能，这类科学运算是大规模多处理器的一类主要应用。在大规模多处理器中，互连网络是设计的关键部分；附录 F 主要讨论了这一主题。

5.1.1　多处理器体系结构：问题与方法

为了充分利用拥有 n 个处理器的 MIMD 多处理器，通常必须拥有至少 n 个要执行的线程或进程。单个进程中的独立线程通常由程序员确认或由操作系统创建（来自多个独立请求）。在另一极端情况下，一个线程可能由一个循环的数十次迭代组成，这些迭代是由开发该循环数据并行的并行编译器生成的。指定给一个线程的计算量称为**粒度大小**，尽管这一数值在考虑如何高效开发线程级并行时非常重要，但线程级并行与指令级并行的重要定性区别在于：线程级并行是由软件系统或程序员在较高层级确认的；这些线程由数百条乃至数百万条可以并行执行的指令组成。

线程还能用来开发数据级并行，当然，其开销可能高于使用 SIMD 处理器或 GPU 的情况（见第 4 章）。这一开销意味着粒度大小必须大到能够足以高效开发并行。例如，尽管向量处理器或 GPU 也许能够高效地实现短向量运算的并行化，但当并行分散在许多线程之间时，粒度大小会非常小，导致在 MIMD 中开发并行的开销过于昂贵，无法接受。

根据所包含的处理器数量，可以将现有共享存储器的多处理器分为两类，而处理器的数量又决定了存储器的组织方式和互连策略。我们是按照存储器组织方式来称呼多处理器的，因为处理器的数目是多还是少，是可能随时间变化的。

第一类称为**对称（共享存储器）多处理器**（SMP），或**集中式共享存储器多处理器**，其特点是核心数目较少，通常不超过 8 个。由于此类多处理器中的处理器数目如此之小，所以处理器

有可能共享一个集中式存储器，所有处理器能够平等地访问它，这就是对称一词的由来。在多核芯片中，采用一种集中方式在核心之间高效地共享存储器，现有多核心都是 SMP。当连接一个以上的多核心时，每个多核心会有独立的存储器，所以存储器为分布式，而非集中式。

SMP 体系结构有时也称为**一致存储器访问**（UMA）多处理器，这一名称来自以下事实：所有处理器访问存储器的延迟都是一致的，即使存储器的组织方式被分为多个组时也是如此。图 5-1 展示了这类多处理器的基本结构。SMP 的体系结构将在 5.2 节讨论，我们将结合一种多核心来解释这种体系结构。 346

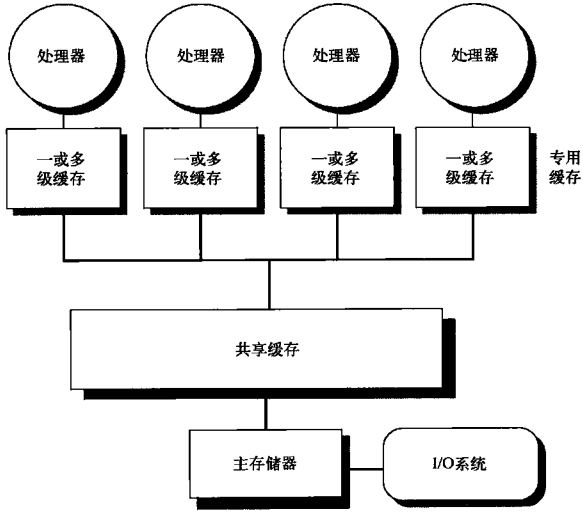


图 5-1 基于多核芯片的集中式共享存储器多处理器的基本结构。多处理器缓存子系统共享同一物理存储器，通常拥有一级共享缓存、一或多级各核心专用缓存。这一结构的关键特性是所有处理器对所有存储器的访问时间一致。在多芯片版本中，将省略共享缓存，将处理器连接至存储器的总线或互连网络分布在芯片之间，而不是一块芯片内部

在另一种设计方法中，多处理器采用物理分布式存储器，称为**分布式共享存储器**（DSM）。图 5-2 展示了此类多处理器的基本结构。为了支持更多的处理器，存储器必须分散在处理器之间，而不应当是集中式的；否则，存储器系统就无法在不大幅延长访问延迟的情况下为大量处理器提供带宽支持。随着处理器性能的快速提高以及处理器存储器带宽需求的相应增加，越来越小的多处理器都优选采用分布式存储器。[①]多核心处理器的推广意味着甚至两芯片多处理器都采用分布式存储器。处理器数目的增大也提升了对高宽带互连的需求，在附录 F 可以看到一些例子。直联网络（即交换机）和间接网络（通常是多维网络）均被用于实现这些互连。 347

① 这里说的“越来越小”是指多处理器中包含的处理器数目或核心数目。——译者注

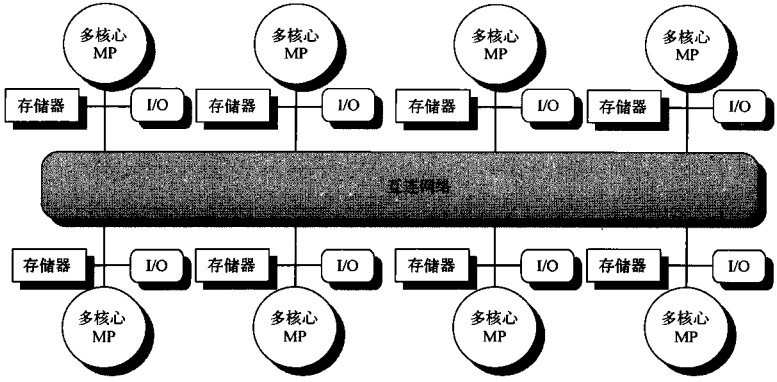


图5-2 2011年的分布式存储器多处理器的基本体系结构通常包括一个带有存储器的多核心多处理器芯片，可能带有I/O和一个接口，连向连接所有节点的互连网络。每个处理器核心共享整个存储器，当然，在访问隶属于该核心芯片的本地存储器时，其速度要远远高于访问远端存储器的速度

将存储器分散在节点之间，既增加了带宽，也缩短了到本地存储器的延迟。DSM多处理器也被称为NUMA（非一致存储器访问），这是因为它的访问时间取决于数据字在存储器中的位置。DSM的关键缺点是在处理器之间传送数据的过程多少变得更复杂了一些，DSM需要在软件中多花费一些力气，以充分利用分布式存储器提升的存储器带宽。因为所有多核心多处理器（处理器芯片或插槽多于一个）都使用了分布式存储器，所以我们将从这个角度来解释分布式存储器多处理器的工作方式。

在SMP和DSM这两种体系结构中，线程之间的通信是通过共享地址空间完成的，也就是说，任何一个拥有正确寻址权限的处理器都可以向任意存储器位置发出存储器引用。与SMP和DSM相关联的**共享存储器**一词是指共享地址空间这一事实。

与之相对，下一章的集群和仓库级计算机看起来更像是由网络连接的独立计算机，如果两个处理器之间没有运行软件协议加以辅助，那一个处理器就无法访问另一个处理器的存储器。在此类设计中，利用消息传送协议在处理器之间传送数据。

348

5.1.2 并行处理的挑战

多处理器的应用范围很广，既可用于运行基本上没有通信的独立任务，也可以运行一些必须在线程之间进行通信才能完成任务的并行程序。有两个重要的障碍使并行处理变得极富挑战性（这两个障碍都可以用Amdahl定律来解释）。这些障碍的难易程度是由应用方式和体系结构来确定的。

第一个障碍与程序中有限的可用并行相关，第二个障碍源于通信的成本较高。由于可用并行的有限性，所以很难在所有并行处理器中都实现良好的加速比，我们的第一个示例就来展示这一点。

例题 假定希望用100个处理器获得80倍的加速比。原计算中的串行部分可以占多大比例?

解答 回想第 1 章的 Amdahl 定律：

$$加速比=\frac{1}{\frac{增强比例}{增强加速比}+(1-增强比例)}$$

为简单起见，假定此程序仅以两种模式运行，一种模式是并行方式，所有处理器都得到充分应用，这是一种增强模式；另一种模式是串行方式，仅使用一个处理器。通过这一简化，增强模式下的加速比就是处理器的数目，增强模式的比例就是并行模式中花费的时间。代入上式得：

$$80=\frac{1}{\frac{并行部分所占比例}{100}+(1-并行部分所占比例)}$$

化简得：

$$0.8\times 并行部分所占比例+80\times(1-并行部分所占比例)=1$$

$$80-79.2\times 并行部分所占比例=1$$

$$并行部分所占比例=\frac{80-1}{79.2}$$

$$并行部分所占比例=0.9975$$

为了以 100 个处理器实现 80 倍的加速比，原计算中只有 0.25%可以是串行的。当然，为实现线性加速比（n 个处理器的加速比为 n），那整个程序就必须都是并行的，没有串行部分。在实践中，程序不会仅在完全并行模式下运行，也不会仅在完全串行模式下运行，而是运行在未利用全部处理器的并行模式下。

349

并行处理的第二个重要挑战涉及并行处理器进行远程访问所带来的长时延迟。在现有的共享存储器多处理器中，分离核心之间的数据通信通常需要耗费 35 ~ 50 个时钟周期，分离芯片上的核心之间进行数据通信可能耗费 100 个时钟周期到 500 甚至更多个时钟周期（对于大规模多处理器而言），具体取决于通信机制、互连网络的类型以及多处理器的规模。长时间的通信延迟显然会造成实际影响。让我们看一个简单的例子。

例题 假定有一个应用程序运行在包含 32 个处理器的多处理器上，它在引用远程存储器时需要的时间为 200 ns。对于这一应用程序，假定除涉及通信的引用之外，其他所有引用都会在本地存储器层次结构中命中，这一假定稍微有些乐观了。处理器会在远程请求时停顿，处理器时钟频率为 3.3 GHz。如果基础 CPI（假定所有引用都在缓存中命中）为 0.5，请对比在没有通信、0.2%的指令涉及远程通信引用这两种情况下，多处理器会快多少?

解答 首先计算每条指令占用的时钟周期数更容易一些。当涉及 0.2%的远程引用时，多处理器的实际 CPI 为：

$$CPI=基础\ CPI+远程请求率\times 远程请求成本$$

$$=0.5+0.2\%\times 远程请求成本$$

远程请求成本为：

$$\frac{\text{远程访问成本}}{\text{周期时间}}=\frac{200\ \text{ns}}{0.3\ \text{ns}}=666\text{个周期}$$

因此，我们可以得出 CPI：

$$\text{CPI}=0.5+1.2=1.7$$

当所有引用均为本地引用时，多处理器要快出 1.7/0.5=3.4 倍。实际的性能分析要复杂得多，因为某些非通信引用会在本地层次结构中缺失，远程访问时间也不是一个常数值。例如，在进行远程引用时，由于许多引用试图利用全局互连从而导致争用，增大延迟，从而使远程引用的成本大幅增加。

这些问题（并行度不足、远程通信延迟太长）是多处理器应用中最大的两个性能难题。应用程序并行度不足的问题必须通过软件来解决，在软件中采用一些能够提供更佳并行性能的新算法，而且软件系统应当尽量利用所有处理器来运行软件。远程延迟过长而导致的影响可以由体系结构和程序员来降低。例如，我们可以利用硬件机制（比如缓存共享数据）或软件机制（比如重新调整数据的结构，增加本地访问的数量）来降低远程访问的频率。我们可以利用多线程（在本章后面讨论）或利用预取（在第2章对这一主题进行了广泛讨论）来尝试容忍这些延迟。

350

本章主要关注的技术是用来降低远程通信延迟过长导致的影响。例如，5.2 节至 5.4 节讨论了如何使用缓存来降低远程访问频率，并同时保持存储器的一致性。5.5 节讨论了同步，由于同步本来就与处理器之间的通信关联在一起，而且还可能会限制并行，所以它是一个重要的潜在瓶颈。5.6 节介绍隐藏延迟的技术和共享存储器的存储器一致性模型。在附录 I 中，我们主要关注更大规模的多处理器，它们绝大多数用于科学工作。这个附录将研究此类应用的本质，以及用数十个乃至数百个处理器来实现加速的挑战。

5.2　集中式共享存储器体系结构

人们发现，使用大型、多级缓存可以充分降低处理器对存储器带宽的需求，这一重要发现刺激了集中式存储器多处理器的发展。最初，这些处理器都是单核心的，经常会占据整个主板，存储器被放置在共享总线上。后来，更高性能的处理器、存储器需求超出了一般总线的能力，最近的微处理器直接将存储器连接到单一芯片中，有时称其为**后端**或**存储器总线**，以便将它与连接至 I/O 的总线区分开来。在访问一个芯片的本地存储器时，无论是为了 I/O 操作，还是为了从另一个芯片进行访问，都需要通过“拥有”该存储器的芯片。因此，对存储器的访问是非对称的：对本地存储器的访问更快一些，而对远程存储器的远程要慢一些。在多核心结构中，存储器由一个芯片上的所有核心共享，但是从一个多核心的存储器访问另一个核心的存储器时，仍然是非对称的。

采用对称共享存储器的计算机通常支持对共享数据与专用数据的缓存。**专用数据**供单个处理器使用，而**共享数据**则由多个处理器使用，基本上是通过读写共享数据来实现处理器之间的通信。在缓存专用项目时，它的位置被移往缓存，缩短平均访问时间并降低所需要的存储器带宽。由于没有其他处理器使用数据，所以程序行为与单处理器中的行为相同。在缓存共享数据时，可能会在多个缓存中复制共享值。除了降低访问延迟和所需要的存储器带宽之外，这一复制过程还可以减少争用，当多个处理器同时读取共享数据项时可能会出现这种争用。不过，共享数据的缓存也引入了一个新问题：缓存一致性。

351

5.2.1 什么是多处理器缓存一致性

遗憾的是，缓存共享数据会引入一个新的问题，这是因为两个不同的处理器是通过各自的缓存来保留存储器视图的，如果不多加防范，它们最终可能会看到两个不同值。表 5-1 展示了这一问题，并说明两个不同处理器如何将同一位置的内容看作两个不同值。这一难题一般被称为**缓存一致性问题**。注意，存在一致性问题是因为既拥有全局状态（主要由主存储器决定），又拥有本地状态（由各个缓存确定，它是每个处理器核心专用的）。因此，在一个多核心中，可能会共享某一级别的缓存（比如 L3），而另外一些级别的缓存则是专用的（比如 L1 和 L2），一致性问题仍然存在，必须加以解决。

表5-1 由两个处理器（A和B）进行读写时，单一存储器位置（X）的缓存一致性问题

时 间	事 件	处理器A的缓存内容	处理器B的缓存内容	位置X的存储器内容
0				1
1	处理器A读取X	1		1
2	处理器B读取X	1	1	1
3	处理器A将0存储到X	0	1	0

* 我们最初假定两个缓存都没有包含该变量，X 的值为 1。我们还假定采用直写缓存，写回缓存会增加一些复杂性，但与此类似。在处理器 A 写入 X 值后，A 的缓存和存储器中都包含了新值，但 B 的缓存中没有，如果 B 读取 X 的值，将会收到数值 1!

通俗地说，如果在每次读取某一数据项时都会返回该数据项的最新写入值，那就说这个存储器系统是一致的。尽管这一定义看起来是正确的，但它有些含混，而且过于简单；实际情况要复杂得多。这一简单定义包含了存储器系统行为的两个方面，这两个方面对于编写正确的共享存储器程序都至关重要。第一个方面称为一致性（coherence），它确定了读取操作可能返回什么值。第二个方面称为连贯性（consistency），它确定了一个写入值什么时候被读取操作返回。首先来看一致性。

如果存储器系统满足以下条件，则说它是一致的。

(1) 处理器 P 读取位置 X，在此之前是由 P 对 X 进行写入，在 P 执行的这一写入与读取操作之间，没有其他处理器对位置 X 执行写入操作，此读取操作总是返回 P 写入的值。 352

(2) 一个处理器向位置 X 执行写入操作之后，另一个处理器读取该位置，如果读写操作的间隔时间足够长，而且在两次访问之间没有其他处理器向 X 写入，那该读取操作将返回写入值。

(3) 对同一位置执行的写入操作**被串行化**，也就是说，在所有处理器看来，任意两个处理器对相同位置执行的两次写入操作看起来都是相同顺序。例如，如果数值 1、数值 2 被依次先后写到一个位置，那处理器永远不可能先从该位置读取到数值 2，之后再读取到数值 1。

第一个特性只是保持了程序顺序——即使在单处理器中，我们也希望具备这一特性。第二个特性定义了一致性存储器视图的含义：如果处理器可能持续读取到一个旧数据值，我们就能明确地说该存储器是不一致的。

对写操作串行化的要求更加微妙，但却同等重要。假定我们没有实现写操作的串行化，而且处理器 P1 先写入地址 X，然后是 P2 写入地址 X。对写操作进行串行化可以确保每个处理器在某一时刻看到的都是由 P2 写入的结果。如果没有对写入操作进行串行化，那某些处理器可能会首先看到 P2 的写入结果，然后看到 P1 的写入结果，并将 P1 写入的值无限期保存下去。避免此类难题的最简单方法是确保对同一位置执行的所有写入操作，在所有处理器看来都是同一顺序；这一特性被称为**写入操作串行化**。

尽管上述三条属性足以确保一致性了，但什么时候才能看到写入值也是一个很重要的问题。比如，我们不能要求在某个处理器向X中写入一个取值之后，另一个读取X的处理器能够马上看到这个写入值。比如，如果一个处理器对X的写入操作仅比另一个处理器对X的读取操作提前很短的一点时间，那就不可能确保该读取操作会返回这个写入值，因为写入值当时甚至可能还没有离开处理器。写入值到底在**多久之后**必须能被读取操作读到？这一问题由**存储器连贯性模型**回答，5.6节将讨论这一主题。

一致性和连贯性是互补的：一致性确定了向同一存储器位置的读写行为，而连贯性则确定了有关访问其他存储器位置的读写行为。现在，作出以下两条假定。第一，只有在所有处理器都能看到写入结果之后，写入操作才算完成（并允许进行下一次写入）。第二，处理器不能改变有关任意其他存储器访问的任意写入顺序。这两个条件是指：如果一个处理器先写入位置A，然后再写入位置B，那么任何能够看到B中新值的处理器也必须能够看到A的新值。这些限制条件允许处理器调整读取操作的顺序，但强制要求处理器必须按照程序顺序来完成写入操作。在5.6节之前将一直采用这一假定，届时将会看到这一定义的内涵以及其替代选择。

353

5.2.2　一致性的基本实现方案

多处理器与I/O的一致性问题尽管在起源上有些类似，但它们却有着不同的特性，会对相应的解决方案产生影响。在I/O情景中很少会出现存在多个数据副本的事件（这是一个应当尽可能避免出现的事件），在多个处理器上运行的程序与此不同，它通常会在几个缓存中拥有同一数据的多个副本。在一致性多处理器中，缓存提供了共享数据项的**迁移**与**复制**。

一致性缓存提供了迁移，可以将数据项移动到本地缓存中，并以透明方式加以使用。这种迁移既缩短了访问远程共享数据项的延迟，也降低了对共享存储器的带宽要求。

一致性缓存还为那些被同时读取的共享数据提供复制功能，在本地缓存中制作数据项的一个副本。复制功能既缩短了访问延迟，又减少了对被读共享数据项的争用。支持迁移与复制功能对于共享数据的访问性能非常重要。因此，多处理器没有试图通过软件来避免这一问题的发生，而是采用了一种硬件解决方案，通过引入协议来保持缓存的一致性。

为多个处理器保持缓存一致性的协议被称为**缓存一致性协议**。实现缓存一致性协议的关键在于跟踪数据块的所有共享状态。目前使用的协议有两类，分别采用不同技术来跟踪共享状态。

- **目录式**——特定物理存储器块的共享状态保存的位置称为**目录**。共有两种不同类型的目录式缓存一致性，它们的差异很大。在SMP中，可以使用一个集中目录，与存储器或其他某个串行化点相关联，比如多核心中的最外层缓存。在DSM中，使用单个目录没有什么意义，因为这种方法会生成单个争用点，当多核心中拥有8个或更多个核心时，由于其存储器要求的原因，很难扩展到许多个多核芯片。分布式目录要比单个目录更复杂，这些设计是5.4节讨论的主题。
- **监听式**——如果一个缓存拥有某一物理存储器块中的数据副本，它就可以跟踪该块的共享状态，而不是将共享状态保存在同一个目录中。在SMP中，所有缓存通常都可以通过某种广播介质访问（比如将各核心的缓存连接至共享缓存或存储器的总线），所有缓存控制器都监听这一介质，以确定自己是否拥有该总线或交换访问上所请求块的副本。监听协议也可用作多芯片多处理器的一致性协议，有些设计在每个多核心内部目录协议的顶层支持监听协议！

采用微处理器（单核）的多处理器和缓存通过总线连接到单个共享存储器，所以监听协议

的应用越来越多。总线提供了一种非常方便的广播介质，用于实现监听协议。在多核体系结构中，所有多核都共享芯片上的某一级缓存，所以这一状况有了大幅改变。因此，一些设计开始转而使用目录协议，因为其开销较低。为便于读者熟悉这两种协议，我们在这里重点介绍监听协议，在谈到 DSM 体系结构时再讨论目录协议。 354

5.2.3 监听一致性协议

有两种方法可以满足上一小节讨论的一致性需求。一种方法是确保处理器在写入某一数据项之前，获取对该数据项的独占访问。这种类型的协议被称为**写入失效协议**（write invalid protocol），因为它在执行写入操作时会使其他副本失效。到目前为止，这是最常用的协议。独占式访问确保在写入某数据项时，不存在该数据项的任何其他可读或可写副本：这一数据项的所有其他缓存副本都作废。

表 5-2 给出了一个失效协议的例子，它采用了写回缓存。为了了解这一协议如何确保一致性，我们考虑在处理器执行写入操作之后由另一个处理器进行读取的情景：由于写操作需要独占访问，所以进行读取的处理器所保留的所有副本都必须失效（这就是这一协议名称的来历）。因此，在进行读取操作时，会在缓存中发生缺失，将被迫提取此数据的新副本。对于写入操作，我们需要执行写入操作的处理器拥有独占访问，禁止任何其他处理器同时写入。如果两个处理器尝试同时写入同一数据，其中一个将会在竞赛中获胜（稍后将会看到如何确定哪个处理器获胜），从而导致另一处理器的副本失效。另一处理器要完成自己的写入操作，必须获得此数据的新副本，其中现在必须包含更新后的取值。因此，这一协议实施了写入串行化。 355

表5-2 失效协议举例，该协议在单一缓存块（X）的监听总线上工作，采用写回缓存

处理器活动	总线活动	处理器A的缓存内容	处理器B的缓存内容	存储器位置X的内容
				0
处理器A读取X	缓存中没有X的内容	0		0
处理器B读取X	缓存中没有X的内容	0	0	0
处理器A将1写至X	对于X失效	1		0
处理器B读取X	缓存中没有X的内容	1	1	1

* 我们假定两个缓存在开始时都没有保存 X 的内容，存储器中的 X 值为 0。处理器和存储器内容给出了完成处理器及总线操作之后的取值。空格表示没有操作或没有缓存副本。当 B 中发生第二次缺失时，处理器 A 反馈该取值，同时取消来自存储器的响应。此外，B 缓存中的内容和 X 的存储器内容都被更新。存储器的这一更新过程是在存储器块变为共享时进行的，这种更新简化了协议，但只能在替换该块时才可能跟踪所有权，并强制进行写回。这就需要引入另外一个名为“拥有者”的状态，表示某个块可以共享，但当拥有该块的处理器在改变或替换它时，需要更新所有其他处理器和存储器。如果多核处理器使用了共享缓存（比如 L3），那么所有存储器都是通过这个共享缓存看到的；在这个例子中，L3 的行为就像存储器一样，一致性必须由每个核心的专用 L1 和 L2 处理。正是由于这一观察结果，一些设计人员选择在多核心处理器中使用目录协议。为使这一方法生效，L3 缓存必须是包含性的（见 5.7.3 节）。

失效协议的一种替代方法是在写入一个数据项时更新该数据项的所有缓存副本。这种类型的协议被称为**写入更新**或**写入广播**协议。由于写入更新协议必须将所有写入操作都广播到共享缓存线上，所以它要占用相当多的带宽。为此，最近的多处理器已经选择实现一种写失效协议，本章的后续部分将仅关注失效协议。

5.2.4 基本实现技术

实现失效协议的关键在于使用总线或其他广播介质来执行失效操作。在较早的多芯片多处理器中，用于实现一致性的总线是共享存储器访问总线。在多核心处理器中，总线可能是专用

缓存（Intel Core i7 中的 L1 和 L2）和共享外部缓存（i7 中的 L3）之间的连接。为了执行一项失效操作，处理器只获得总线访问，并在总线上广播要使其失效的地址。所有处理器持续监听该总线，观测这些地址。处理器检查总线上的地址是否在自己的缓存中。如果在，则使缓存中的相应数据失效。

在写入一个共享块时，执行写入操作的处理器必须获取总线访问权限来广播其失效。如果两个处理器尝试同时写入共享块，当它们争用总线时，会串行安排它们广播其失效操作的尝试。第一个获得总线访问权限的处理器会使它正写入块的所有其他副本失效。如果这些处理器尝试写入同一块，则由总线实现这些写入操作的串行化。这一机制有一层隐含意思：在获得总线访问权限之前，无法实际完成共享数据项的写入操作。所有一致性机制都需要某种方法来串行化对同一缓存块的访问，具体方式可以是串行化对通信介质的访问，也可以是对另一共享结构访问的串行化。

除了使被写入缓存块的副本失效之外，还需要在发生缓存缺失时定位数据项。在直写缓存中，可以很轻松地找到一个数据项的最近值，因为所有写入数据都会发回存储器，所以总是可以从存储器中找到数据项的最新值。（对缓冲区的写入操作可能会增加一些复杂度，必须将其作为额外的缓存项目加以有效处理。）

356

对于写回缓存，查找最新数据值的问题解决起来要困难一些，因为数据项的最新值可能放在专用缓存中，而不是共享缓存或存储器中。令人开心的是，写回缓存可以为缓存缺失和写入操作使用相同的监听机制：每个处理器都监听放在共享总线上的所有地址。如果处理器发现自己拥有被请求缓存块的脏副本，它会提供该缓存块以回应读取请求，并中止存储器（或 L3）访问。由于必须从另一个处理器的专用缓存（L1 或 L2）提取缓存块，所以增加了复杂性，这一提取过程花费的时间通常长于从 L3 进行提取的时间。由于写回缓存对存储器带宽的需求较低，所以它们可以支持更多、更快速的处理器。结果，所有多核处理器都在缓存的最外层级别使用写回缓存，接下来研究以写回缓存来实现缓存的方法。

通常的缓存标记可用于实施监听过程，每个块的有效位使失效操作的实施非常轻松。读取缺失（无论是由失效操作导致，还是某一其他事件导致）的处理也非常简单，因为它们就是依赖于监听功能的。对于写入操作，我们希望知道是否缓存了写入块的其他副本，如果不存在其他缓存副本，那在写回缓存中就不需要将写入操作放在总线上。如果不用发送写入操作，既可以缩短写入时间，还可以降低所需带宽。

若要跟踪缓存块是否被共享，可以为每个缓存块添加一个相关状态位，就像有效位和重写标志位（dirty bit）一样。通过添加 1 个位来指示该数据块是否被共享，可以判断写入操作是否必须生成失效操作。在对处于共享状态的块进行写入时，该缓存在总线上生成失效操作，将这个块标记为**独占**。这个核心不会再发送其他有关该块的失效操作。如果一个缓存块只有唯一副本，则拥有该唯一副本的核心通常被称为该缓存块的**拥有者**。

在发送失效操作时，拥有者缓存块的状态由共享改为非共享（或改为独占）。如果另一个处理器稍后请求这一缓存块，必须再次将状态改为共享。由于监听缓存也能看到所有缺失，所以它知道另一处理器什么时候请求了独占缓存块，应当将状态改为共享。

每个总线事务都必须检查缓存地址标记，这些标记可能会干扰处理器缓存访问。减少这种干扰的一种方法就是复制这些标记，并将监听访问引导至这些重复标记。另一种方法是在共享的 L3 缓存使用一个目录，这个目录指示给定块是否被共享，哪些核心可能拥有它的副本。利用目录信息，可以仅将失效操作发送给拥有该缓存块副本的缓存。这就要求 L3 必须总是拥有 L1 或 L2 中所有数据项的副本，这一属性被称为**包含**，在 5.7 节会再次讨论它。

5.2.5 示例协议

监听一致性协议通常是通过在每个核心中整合有限状态控制器来实施的。这个控制器回应由核心中的处理器和由总线（或其他广播介质）发出的请求，改变所选缓存块的状态，并使用总线访问数据或使其失效。从逻辑上来说，可以看作每个块有一个相关联的独立控制器；也就是说，对不同块的监听操作或缓存请求可以独立进行。在实际实施中，单个控制器允许交错执行以不同块为目标的多个操作。(也就是说，即使仅允许同时执行一个缓存访问或一个总线访问，也可以在一个操作尚未完成之前启动另一个操作。)另外别忘了，尽管我们在以下介绍中以总线为例，但在实现监听协议时可以使用任意互连网络，只要其能够向所有一致性控制器及其相关专用缓存进行广播即可。 357

我们考虑的简单协议有三种状态：无效、共享和已修改。共享状态表明专用缓存中的块可能被共享，已修改状态表明已经在专用缓存中更新了这个块；注意，已修改状态隐含表明这个块是独占的。表 5-3 给出了由一个核心生成的请求（在表的上半部分）和来自总线的请求（在表的下半部分）。这一协议是针对写回缓存的，但可以很轻松地将其改为针对直写缓存：对于直写缓存，只需要将已修改状态重新解读为独占状态，并在执行写入操作时以正常方式更新缓存。这一基本协议的最常见扩展是添加一种独占状态，这一状态表明块未被修改，但仅有一个专用缓存保存了这个块。我们将在 5.2.6 节介绍这一扩展及其他扩展。 358

表5-3 缓存一致性机制接收来自核心处理器和共享总线的请求，并根据请求类型、它在本地缓存中是命中还是缺失、请求中指定的本地缓存块状态来作出回应

请求	源	所寻址缓存块的状态	缓存操作的类型	功能与解释
读取命中	处理器	共享或已修改	正常命中	读取本地缓存中的数据
读取缺失	处理器	无效	正常缺失	将读取缺失放在总线上
读取缺失	处理器	共享	替换	地址冲突缺失：将读取缺失放在总线上
读取缺失	处理器	已修改	替换	地址冲突缺失：写回块，然后将读取缺失放在总线上
写命中	处理器	已修改	正常命中	在本地缓存中写数据
写命中	处理器	共享	一致性	将失效操作放在总线上。这些操作通常称为更新或拥有者缺失，因为它们不能提取数据，只能改变状态
写缺失	处理器	无效	正常缺失	将写缺失放在总线上
写缺失	处理器	共享	替换	地址冲突缺失：将写缺失放在总线上
写缺失	处理器	已修改	替换	地址冲突缺失：写回块，然后将写缺失放在总线上
读取缺失	总线	共享	无操作	允许共享缓存或存储器为读取缺失提供服务
读取缺失	总线	已修改	一致性	尝试共享数据：将缓存块放在总线上，并将状态改为共享
失效	总线	共享	一致性	尝试写共享块，使该块失效
写缺失	总线	共享	一致性	尝试写共享块，使缓存块失效
写缺失	总线	已修改	一致性	尝试将独占块写到其他位置，写回该缓存块，并在本地缓存中使其状态失效

* 第四列将缓存操作类型操作描述为正常命中或缺失（与单处理器缓存看到的情况相同）、替换（单处理器缓存替换缺失）或一致性（保持缓存一致性所需）；正常操作或替换操作可能会根据这个块在其他缓存中的状态而产生一致性操作。对于由总线监听到的读取缺失、写入缺失或无效操作，仅当读取或写入地址与本地缓存中的块匹配，而且这个块有效时，才需要采取动作。

在将一个失效动作或写入缺失放在总线上时，任何一个核心，只要其专用缓存中拥有这个

缓存块的副本，就会使这些副本失效。对于写回缓存中的写入缺失，如果这个块仅在一个专用缓存中是独占的，那么缓存也会写回这个块；否则，将从这个共享缓存或存储器中读取该数据。

图 5-3 显示了单个专用缓存块的有限状态转换图，它采用了写入失效协议和写回缓存。为简单起见，我们将复制这个协议的三种状态，用以表示根据处理器请求进行的状态转换（左图，对应于表 5-3 的上半部分），和根据总线请求进行的状态转换（右图，对应于表 5-3 的下半部分）。图中使用黑体字来区分总线动作，与状态转换所依赖的条件相对。每个节点的状态代表着选定专用缓存块的状态，这一状态由处理器或总线请求指定。

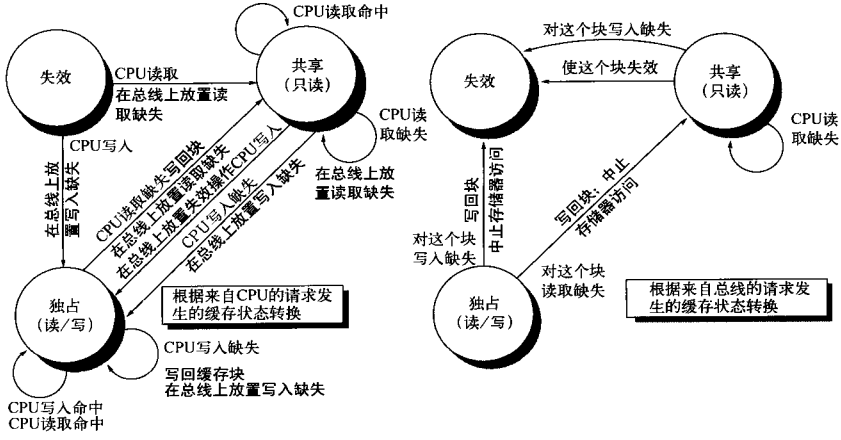


图 5-3　专用写回缓存的写入失效、缓存一致性协议，给出了缓存中每个块的状态及状态转换。缓存状态以圆圈表示，在状态名称下面的括号中给出了本地处理器允许执行但不会产生状态转换的访问。导致状态变换的激励以常规字体标记在转换弧上，因为状态转换而生成的总线动作以黑体标记在转换弧上。激励操作应用于专用缓存的块，而不是缓存中的特定地址。因此，在对一个共享状态的块产生读取缺失时，是对这个缓存块的缺失，而不是对不同地址的缺失。图形左侧显示的状态转换是由于此缓存相关处理器的操作而发生的，右侧显示的状态转换是根据总线上的操作而发生的。当处理器请求的地址与本地缓存块的地址不匹配时，会发生独占状态或共享状态的读取缺失以及独占状态的写入缺失。这种缺失是标准缓存替换缺失。在尝试写入处于共享状态的块时，会生成失效操作。每当发生总线事务时，所有包含此总线事务指定缓存块的专用缓存都会执行右图指定的操作。此协议假定，对于在所有本地缓存中都不需要更新的数据块，存储器（或共享缓存）会在发生对该块的读取缺失时提供数据。在实际实现中，这两部分状态图是结合在一起的。实践中，关于失效协议还有许多非常细微的变化，包括引入独占的未修改状态，说明处理器和存储器是否会在缺失时提供数据。在多核芯片中，共享缓存（通常是 L3，但有时是 L2）充当着存储器的角色，总线就是每个核心的专用缓存与共享缓存之间的总线，再由共享缓存与存储器进行交互

这一缓存协议的所有状态在单处理器缓存中也都是需要的，分别对应于无效状态、有效（与清洁）状态、待清理状态。在写回单处理器缓存中会需要图 5-3 左半部分中弧线所表示的大多数状态转换，但有一个例外，那就是在对共享块进行写入命中时的失效操作。图 5-3 中右半部分弧线所表示的状态转换仅对一致性有用，在单处理器缓存控制器中根本不会出现。

前面曾经提到，每个缓存只有一个有限状态机，其激励或者来自所连接的处理器，或者来自总线。图 5-4 说明图 5-3 中右半部分的状态转换如何与图中左半部分的状态转换相结合，构成每个缓存块的单一状态图。

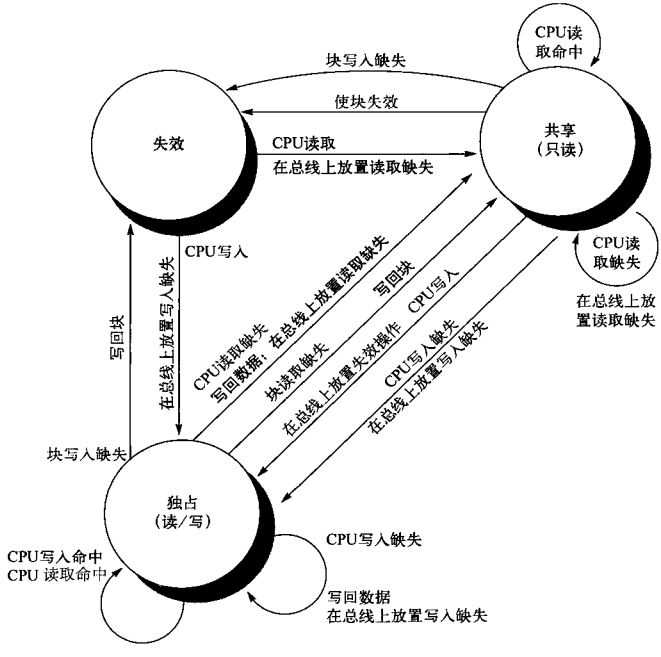


图 5-4 缓存一致性状态图，由本地处理器引起的状态转换用黑色表示，由总线行为引起的以灰色表示。和图 5-3 中一样，有关转换的行为以粗体显示

为了理解这一协议为何能够正常工作，可以观察一个有效缓存块，它要么在一或多个专用缓存中处于共享状态，要么就在一个缓存中处于独占状态。只要转换为独占状态（处理器写入块时需要这一转换），就需要在总线上放置失效操作或写入缺失，从而使所有本地缓存都将这个块记为失效。另外，如果其他某个本地缓存已经将这个块设为独占状态，这个本地缓存会生成写回操作，提供包含期望地址的块。最后，对于处于独立状态的块，如果总线上出现对这个块的读取缺失，拥有其独占副本的本地缓存会将其状态改变共享。

图 5-4 中用灰色表示的操作用来处理总线上的读取缺失与写入缺失，它们实际上就是协议的监听部分。在这个协议及大多数其他协议中，还保留着另外一个特性：任何处于共享状态的存储器块在其外层共享缓存（L2 或 L3，如果没有共享缓存就是指存储器）中总是最新的，这一特性简化了实施过程。事实上，专用缓存之外的层级是共享缓存还是存储器并不重要；关键在于来自核心的所有访问都要通过这一层级。 359

尽管这个简单的缓存协议是正确的，但它省略了许多复杂因素，这些因素大大增加了实施

过程的难度。其中最重要的一点是，这个协议假定这些操作具有原子性——在完成一项操作的过程中，不会发生任何中间操作。例如，这里讨论的协议假定可以采用单个原子动作形式来检测写入缺失、获取总线和接收响应。现实并非如此。事实上，即使读取缺失也可能不具备原子性；在多核处理器的L2中检测到缺失时，这个核心必须进行协调，以访问连到共享L3的总线。非原子性操作可能会导致协议死锁，也就是进入一种无法继续执行的状态。我们将在本节后面研究DSM设计时讨论这些复杂内容。

360

对于多核处理器，处理器核心之间的一致性都在芯片上实施，或者使用监听协议，或者使用简单的集中式目录协议。许多双处理器芯片，包括Intel Xeon和AMD Opteron，都支持多芯片多处理器，这些多处理器可能通过连接高速接口（分别称为Quickpath或Hypertransport）来构建。这些下一级别的互连并不只是共享总线的扩展，而是使用了一种不同方法来实现多核互连。

361

用多个多核芯片构建而成的多处理器将采用分布式存储器体系结构，而且需要一种机制来保持芯片间的一致性，这一机制要高于、超越于芯片内部的此种机制。在大多数情况下，会使用某种形式的目录机制。

5.2.6 基本一致性协议的扩展

我们刚刚介绍的一致性协议是一种简单的三状态协议，经常用这些状态的第一个字母来称呼这一协议——MSI（Modified、Shared、Invalid）协议。这一基本协议有许多扩展，在本节图形标题中提到了这些扩展。这些扩展是通过添加更多的状态和转换来创建的，这些添加内容对特定行为进行优化，可能会使性能得到改善。下面介绍两种最常见的扩展。

(1) MESI 向基本的 MSI 协议中添加了“独占”（Exclusive）状态，用于表示缓存块仅驻存在一个缓存中，而且是清洁的。如果块处于独占状态，就可以对其进行写入而不会产生任何失效操作，当一个块由单个缓存读取，然后再由同一缓存写入时，可以通过这一独占状态得以优化。当然，处于独占状态的块产生读取缺失时，必须将这个块改为共享状态，以保持一致性。因为所有后续访问都会被监听，所以有可能保持这一状态的准确性。具体来说，如果另一个处理器发射一个读取缺失，则状态会由独占改为共享。添加这一状态的好处在于：在由同一核心对处于独占状态的块进行后续写入时，不需要访问总线，也不会生成失效操作，因为处理器知道这个块在这个本地缓存中是独占的；处理器只是将状态改为已修改。添加这一状态非常简单，只需要使用1个位对这个一致状态进行编码，表示为独占状态，并使用重写标志位表示这个块已被修改。流行的MESI协议就采用了这一结构，这一协议是用它所包含的4种状态命名的，即已修改（Modified）、独占（Exclusive）、共享（Shared）和无效（Invalid）。Intel i7使用了MESI协议的一种变体，称为MESIF，它添加了一个状态（Forward），用于表示应当由哪个共享处理器对请求作出回应。这种协议设计用来提高分布式存储器组成结构的性能。

(2) MOESI 向 MESI 协议中添加了“拥有”（Owned）状态，用于表示相关块由该缓存拥有，在存储器中已经过时。在MSI和MESI协议中，如果尝试共享处于“已修改”状态的块，会将其状态改为“共享”（在原共享缓存和新共享缓存中都会做此修改），必须将这个块写回存储器中。而在MOESI协议中，会在原缓存中将这个块的状态由“已修改”改为“拥有”，不再将其写到存储器中。（新共享这个块的）其他缓存使这个块保持共享状态；只有原缓存保持“拥有”状态，表示主存储器副本已经过期，指定缓存成为其拥有者。这个块的拥有者必须在发生缺失时提供该块，因为存储器中没有最新内容，如果替换了这个块，则必须将其写回存储器中。AMD Opteron使用了MOESI协议。

362

下一节研究这些协议在处理多重编程、并行工作负载时的性能；在研究性能时，就能清楚地了解对基本协议进行这些扩展的价值。但是，在研究性能之前，让我们简要地看一下使用对称存储器结构和监听一致性机制的局限性。

5.2.7 对称共享存储器多处理器与监听协议的局限性

随着多处理器中处理器数目的增长，或随着每个处理器对存储器需求的增长，系统中的任何集中式资源都可能变成瓶颈。利用片上提供的更高带宽的连接以及共享的 L3 缓存（它的速度要比存储器快），设计人员可以尝试以对称形式支持 4～8 个高性能核心。这种方法不太可能扩展到远远超过 8 个核心的情况，一旦合并了多个多核心处理器，这种方法就无效了。

每个缓存的监听带宽也可能产生问题，因为每个缓存必须检查总线上的所有缺失。我们曾经提到，复制标记是一种解决方案。另一种方法是在最外层缓存层级放置一个目录，这一方法已经在最近的某些多核处理器中得到应用。这个目录明确指出哪个处理器的缓存拥有最外层缓存中每一项的副本。这就是 Intel 在 i7 和 Xeon 7000 系统中使用的方法。注意，这个目录的使用不会消除因为处理器之间的共享总线及 L3 造成的瓶颈，但它实现起来要远比将在 5.4 节研究的分布式目录机制容易。

设计者如何提高存储器带宽，以支持更多或更快的处理器呢？为了提高处理器与存储器之间的通信带宽，设计者已经采用了多根总线和互连网络，比如交叉开关或小型点对点网络。在此类设计中，可以将存储器系统（主存储器或共享缓存）配置为多个物理组，以提升有效存储器带宽，同时还保持存储器访问时间的一致性。图 5-5 展示了在使用单芯片多核心来实现系统时，它会是什么样子。尽管利用这种方法可以在一块芯片上实现 4 个以上核心的互连，但它不能很好地扩展到那些使用多核心构建模块的多芯片多处理器，因为存储器已经连接到各个多核心芯片上，不再是集中式存储器。

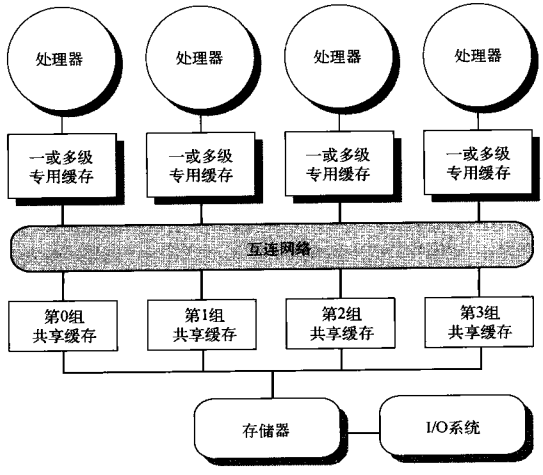


图 5-5 一种多核心单芯片多处理器，通过分组共享缓存实现一致存储器访问，使用互连网络而不是总线

AMD Opteron 表示监听协议与目录协议之间的另一个中间点。存储器被直接连接到每个多核芯片，最多可以连接 4 个多核心芯片。其系统为 NUMA，因为本地存储器多少会更快一些。Opteron 使用点对点连接实现其一致性协议，最多向其他 3 个芯片进行广播。因为处理器之间的链接未被共享，所以一个处理器要想知道失效操作何时完成，唯一的方法就是通过显式确认。因此，一致性协议使用广播来查找可能共享的副本，这一点与监听协议类似，但它却使用确认来确定操作，这一点与目录协议类似。由于在 Opteron 实现中，本地存储器仅比远程存储器快一点点，所以一些软件把 Opteron 多处理器看作拥有一致存储器访问。

363

监听缓存一致性协议可以在没有集中式总线的情况下使用，但仍然需要完成广播，以监听各个缓存，获知潜在共享缓存块的所有缺失。这种缓存一致性通信是处理器规模与速度的另一限制。由于采用较大缓存并不会影响一致性通信，所以当处理器速度很快时，肯定会超出网络的负荷，每个缓存无法响应来自所有其他缓存的监听请求。在 5.4 节，我们研究目录式协议，在发生缺失时不需要向所有缓存进行广播。当处理器速度以及每个处理器的核心数目增大时，更多的设计人员会选择此类协议来避免监听协议的广播限制。

364

5.2.8　实施监听缓存一致性

细节决定成败。

——经典谚语

在 1990 年编写本书第 1 版时，最后的“融会贯通”是一个包含 30 个处理器、单根总线的多处理器，它采用了监听式一致性；总线的容量仅略高于 50 MB/s，到了 2011 年，这种总线带宽恐怕连 Intel i7 中的一个核心也无法提供支持！在 1995 年编写本书第 2 版时，刚刚出现了第一代缓存一致性多处理器，它采用了多根总线，我们当时添加了一个附录，用来描述多总线系统中的监听实现方式。2011 年，大多数仅支持单芯片多处理器的多核处理器已经选择使用共享总线结构，连接到共享存储器或共享缓存。相反，所有支持 16 个或更多个核心的多核多处理器系统都使用互连网络，而不是单根总线，设计人员必须面对一项挑战：在实现监听时，没有为实现事件的串行化而简化总线。

前面曾经说过，在实际实现前面介绍的监听一致性协议时，最复杂的部分在于：在最近的所有多处理器中，写入缺失与更新缺失都不是原子操作。检测写入或更新缺失、其他处理器与存储器通信、为写入缺失获取最新值、确保所有失效操作可以正常进行、更新缓存，这些步骤不能在单个时钟周期内完成。

在单个多核心芯片中，如果（在改变缓存状态之前）首先协调连向共享缓存或存储器的总线，并在完成所有操作之前保持总线不被释放，那就可以有效地使上述步骤变成原子操作。处理器怎么才能知道所有失效操作何时完成呢？在一些多核处理器中，当所有必要失效操作都已收到并在进行处理时，会使用单根信号线发出信号。收到这一信号之后，生成缺失的处理器就可以释放总线，因为它知道在执行与下一次缺失相关的任意行为之前，可以完成所有必要操作。只要在执行这些步骤期间独占总线，处理器就能有效地将各个步骤变为原子操作。

在没有总线的系统中，我们必须寻找其他某种方法，将缺失过程中的步骤变为原子操作。具体来说，必须确保两个尝试同时写入同一数据块的处理器（这种情景称为**竞争**）保持严格排序：首先处理一个写入操作，然后再开始执行下一个。这两次写入操作中的哪一个操作会赢得竞争并不重要，因为只会有一个获胜者，它的一致性操作将被首先完成。在监听系统中，为了

确保一次竞争仅有一个获胜者，会广播所有缺失，并利用互连网络的一些基本性质。这些性质，以及重启竞争失败者缺失处理的能力，是在无总线情况下实现监听缓存一致性的关键。我们将在附录 I 中解释具体细节。

365

还可以将监听式与目录式结合在一起，有一些设计在多核处理器内部使用监听式，在多个芯片之间使用目录式，或者反过来，在多核处理器内部使用目录式，在多个芯片之间使用监听式。

5.3 对称共享存储器多处理器的性能

在使用监听式一致性协议的多核处理器中，其性能通常由几种不同现象共同决定。具体来说，总体缓存性能由两个因素共同决定，一个是由单处理器缓存缺失造成的流量，另一个是通信传输导致的流量，它会导致失效及后续缓存缺失。改变处理器数目、缓存大小和块大小能够以不同方式来影响缺失率的这两个分量，最终得到受这两种因素共同影响的总体系统性能。

附录 B 对单处理器缺失率进行 3C 分类，即容量（capacity）、强制（compulsory）和冲突（conflict），并深入讨论了应用特性和对缓存设计的可能改进。与此类似，因为处理器之间的通信而导致的缺失（经常被称为**一致性缺失**）可以分为两种独立源。

第一种来源是所谓的**真共享缺失**，源自通过缓存一致性机制进行的数据通信。在基于失效的协议中，处理器向共享缓存块的第一次写入操作会导致失效操作，以确保对这个块的拥有关系。此外，当另一处理器尝试修改这个缓存块中的已修改字时，要发生缺失，并传送结果块。由于这两种缺失都是因为处理器之间的数据共享而直接导致的，所以将其划分为真共享缺失。

第二种效果称为**假共享缺失**，它的出现是因为使用了基于失效的一致性算法，这种算法利用了数据块的有效位，每个缓存块只有一个有效位。如果因为写入块中的某个字（不是正被读取的字）而导致一个块失效（而且后续引用会导致失败），就会发生假共享。如果接收到失效操作的处理器真的正在使用要写入的字，那这个引用就是真正的共享引用，无论块大小如何都会导致缺失。但是，如果正被写入的字和读取的字不同，那就不会因为这一失效操作而传送新值，而只是导致一次额外的缓存缺失，所以它是假共享缺失。在假共享缺失中，块被共享，但缓存中的字都没有被实际共享，如果块大小是单个字，那就不会发生缺失。通过下面的例子可以理解这些共享模式。

例题 假定 x1 和 x2 两个字位于同一缓存块中，这个块在 P1 和 P2 的缓存中均为共享状态。假定有以下一系列事件，确认每个缺失是真共享缺失，还是假共享缺失，或是命中。如果块大小为一个字节，那么所发生的所有缺失都被认定为真共享缺失。

366

时　序	P1	P2
1	写x1	
2		读x2
3	写x1	
4		写x2
5	读x2	

解答 下面是按时序进行的分类。

(1) 这一事件是真共享缺失，因为 x1 由 P2 读取，需要由 P2 发出失效操作。

(2) 这一事件是假共享缺失，因为 x2 是由于 P1 中写入 x1 而导致失效的，但 P2 并没有使用 x1 的值。
(3) 这一事件是假共享缺失，因为包含 x1 的块是因为 P2 中的读取操作而被标记为共享状态的，但 P2 并没有读取 x1。在 P2 读取之后，包含 x1 的缓存块将处于共享状态；需要一次写入缺失才能获取对该数据块的独占访问。在一些协议中，会将这种情况作为**更新请求**进行处理，它会生成总线失效，但不会传送缓存块。
(4) 这一事件是假共享缺失，原因与步骤 3 相同。
(5) 这一事件是真共享缺失，因为正在读取的值是由 P2 写入的。

尽管我们将会看到真假共享缺失在商业工作负载中的影响，但对于共享大量用户数据的紧耦合应用程序来说，一致性缺失的角色更重要一些。我们在附录 I 中详细研究它们的效果，届时将考虑并行科学工作负载的性能。

5.3.1　商业工作负载

在这一节，我们将研究一个四处理器共享存储器多处理器在运行通用商业工作负载时的存储器系统特性。我们讨论的这一研究是在 1998 年用一个四处理器 Alpha 系统完成的，但对于一个多处理器在执行此类工作负载时的性能问题，这一研究仍然最全面、最深入。这些结果可以在一个 AlphaServer 4100 上收集，也可以使用一种根据 AlphaServer 4100 建模的可配置模拟器收集。AlphaServer 4100 中的每个处理器都是一个 Alpha 21164，它每个时钟周期最多发射 4 条指令，工作频率为 300 MHz。尽管这个系统中 Alpha 处理器的时钟频率远低于 2011 年所设计系统中的处理器，但这个系统的基本结构由一个四发射处理器和一个三级缓存层级结构构成，非常类似于多核 Intel i7 和其他处理器，如表 5-4 所示。具体来说，Alpha 缓存多少略小一些，但缺失次数也低于 i7。因此，Alpha 系统应当能够让我们深入理解现代多核设计的特性。

367

表5-4　本研究所用Alpha 21164及Intel i7的缓存层级结构特性

缓存级别	特　征	Alpha 21164	Intel i7
L1	大小	8 KB I/8 KB D	32 KB I/32 KB D
	相联度	直接映射	四路 I/八路 D
	块大小	32 B	64 B
	缺失代价	7	10
L2	大小	96 KB	256 KB
	相联度	三路	八路
	块大小	32 B	64 B
	缺失代价	21	35
L3	大小	2 MB	2 MB/核心
	相联度	直接映射	十六路
	块大小	64 B	64 B
	缺失代价	80	约100

* 尽管 i7 的缓存较大、相联度较高，但缺失代价也较高，所以特性可能只有微小不同。例如，根据附录 B，我们可以估计较小 Alpha L1 缓存的缺失率为较大 i7 L1 缓存的 4.9%和 3%，因此，对于 Alpha 来说，每次引用的平均 L1 缺失代价为 0.34，对于 i7 为 0.30。当需要从专用缓存进行传送时，这两种系统的代价都很高（125 个周期，甚至更多）。i7 还在所有核心之间共享它的 L3。

这一研究中使用的工作负载包括以下 3 个应用程序。

(1) 根据 TPC-B（其存储器特性类似于它的较新版本 TPC-C，在第 1 章中介绍）建模的联机事务处理（OLTP）工作负载，并以 Oracle 7.3.2 为底层数据库。这一工作负载由一组发出请求的客户端进程和一组处理这些请求的服务器组成。这些服务器进程占用 85%的用户时间，剩余时间由客户端占用。尽管通过精心调优、利用足够的请求保持处理器繁忙可以隐藏 I/O 延迟，但服务器进程通常会在大约 25 000 条指令之间阻塞 I/O。

(2) 基于 TPC-D 的决策支持系统（DSS）工作负载，（TPC-D 是广泛使用的 TPC-E 的较早版本），这一工作负载也以 Oracle 7.3.2 为底层数据库。这个工作负载仅包含 TPC-D 17 个读取查询中的 6 个，不过这一基准测试中研究的 6 个查询包含了整个基准测试的活动范围。为了隐藏 I/O 延迟，在查询内部和查询之间都开发了并行（在查询内部，会在查询形成期间检测并行）。与 OLTP 基准测试相比，调用阻塞的频率要低得多；这 6 个查询平均在大约 150 万条指令之后产生阻塞。 368

(3) Web 索引搜索（AltaVista）基准测试，其基础是对 AltaVista 数据库存储器映射版本（200GB）的搜索。深入优化了内层循环。因为搜索结构是静态的，所以线程之间几乎不需要同步。在 Google 出现之前，AltaVista 是最流行的 Web 搜索引擎。

表 5-5 显示了在用户模式、内核和空闲循环中所用时间的百分比。I/O 的频率会同时增加内核时间和空闲时间（参见 OLTP 项，它的“I/O-计算”比最大）。AltaVista 将整个搜索数据库映射到存储器中，而且经过了广泛调优，它的内核时间或空闲时间最少。

表5-5 商业工作负载中执行时间的分布

基准测试	用户模式时间百分比	内核时间百分比	处理器空闲时间百分比
OLTP	71	18	11
DSS（对所有查询求平均）	87	4	9
Alta Vista	>98	<1	<1

* OLTP 基准测试中操作系统时间和处理器空闲时间（就是 I/O 等待时间）所占比例最大。DSS 基准测试的操作系统时间少得多，因为它执行的 I/O 操作较少，但仍然有超过 9%的空闲时间。AltaVista 搜索引擎的全面调优通过这些测量值表现得非常清楚。这一工作负载的数据是由 Barroso、Gharachorloo 和 Bugnion [1998]在一个四处理器 AlphaServer 4100 上收集的。

5.3.2 商业工作负载的性能测量

我们先来看看这些基准测试在四处理器系统中的总体处理器执行情况；如 5.3.1 的讨论，这些基准测试包括大量的 I/O 时间，在处理器时间测试数据中忽略了这些时间。我们将 6 个 DSS 查询看作一个基准测试，报告其平均特性。这些基准测试的实际 CPI 变化很大，从 AltaVista Web 搜索的 1.3 到 DSS 工作负载的平均 1.6，再到 OLTP 工作负载的 7.0。图 5-6 显示了如何将执行时间分解为指令执行时间、缓存与存储器系统访问时间及其他停顿（主要是流水线资源停顿，但也包括转换旁视缓冲区（TLB）和分支预测错误停顿）。尽管 DSS 与 AltaVista 工作负载的性能处于合理范围内，但 OLTP 工作负载的性能非常差，这是由于存储器层次结构的性能过差所致。 369

由于 OLTP 工作负载对存储器系统的要求更多，而且存在大量成本高昂的 L3 缺失，所以我们主要研究 L3 缓存大小、处理器数目和块大小对 OLTP 基准测试的影响。图 5-7 显示了增大缓存大小的影响，使用两路组相联缓存，缩减大量冲突缺失。随着 L3 缓存的增大，执行时间会因为 L3 缺失的减少而缩短。令人惊讶的是，几乎所有这些改进都是在 1 ~ 2MB 范围内发生的，超过这一范围之后，尽管当缓存为 2MB 和 4MB 时，缓存缺失仍然是造成大幅性能损失的原因，但几乎没有多少改进了。问题是为什么呢？

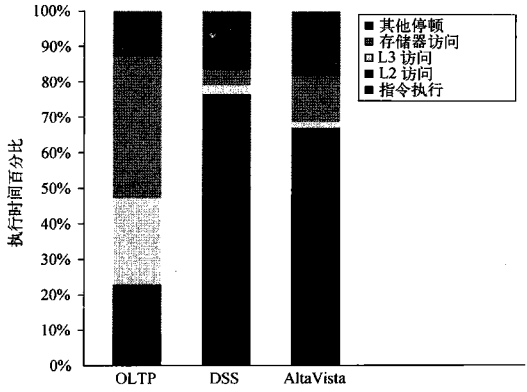


图 5-6 3个程序（OLTP、DSS和AltaVista）在商业工作负载中的执行时间分解。DSS数字是6个不同查询的平均值。CPI的变化很大，从AltaVista较低的1.3，到DSS查询的1.61，再到OLTP的7.0。（分开来说，DSS查询的CPI范围为1.3至1.9。）"其他停顿"包括资源停顿（用21164上的重放陷阱实现）、分支预测错误、存储器屏障和TLB缺失。对于这些基准测试，因为资源而导致的流水线停顿是主要因素。这些数据结合了用户与内核访问的行为。只有OLTP的内核访问占有重要比例，内核访问的表现要优于用户访问！本节给出的所有测量数据都是由Barroso、Gharachorloo和Bugnion[1998]收集

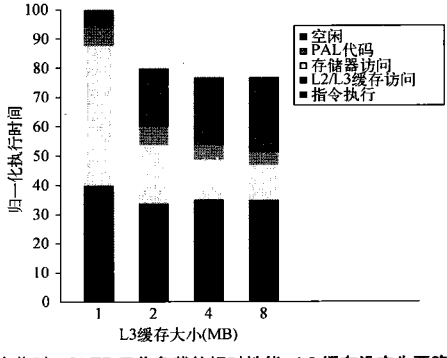


图 5-7 在L3缓存大小变化时，OLTP工作负载的相对性能，L3缓存设定为两路组相联，从1 MB增大到8 MB。空闲时间随缓存大小的增大而延迟，降低了一些性能增益。这一增长是因为当存储器系统停顿较少时，需要更多的服务器进程来隐藏I/O延迟。可以重新调整工作负载，以提高计算/通信平衡性能，将空闲时间保持在可控范围内。PAL代码是一组以优先模式执行的专用操作系统级指令序列；TLB缺失处理程序就是这样一个例子

为了更好地理解这个问题的答案，我们需要确定造成L3缺失的因素，以及当L3缓存增长时，它们是如何变化的。图5-8给出了这些数据，显示了来自5个来源的每条指令所造成的存

储器访问周期数。当 L3 的大小为 1 MB 时，L3 存储器访问周期的两个最大来源是指令和容量/冲突缺失，当 L3 较大时，这两个来源降低为次要因素。遗憾的是，强制、假共享和真共享缺失不受增大 L3 的影响。因此，在 4 MB 和 8 MB 时，真共享缺失占主导地位；当 L3 缓存大小超过 2 MB 时，由于真共享缺失没有变化，从而限制了总体缺失率的减少。 370

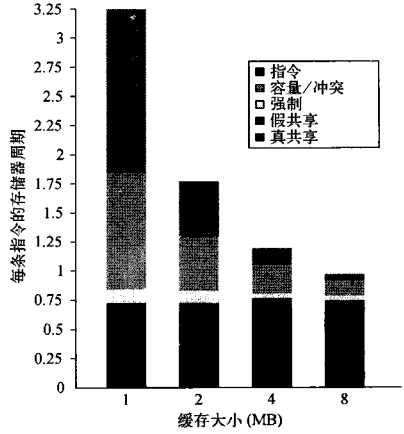


图 5-8 当缓存大小增加时，占用存储器访问周期的各项因素会发生偏移。L3 缓存被模拟为两路组相联

增大缓存大小可以消除大多数单处理器缺失，但多处理器缺失不受影响。增大处理器数目如何影响各种不同类型的缺失呢？图 5-9 给出了这些数据，其中假定所采用的基本配置为 2 MB、两路组相联 L3 缓存。可以预期，真共享缺失率的增加（降低单处理器缺失不会对其有所补偿）导致每条指令的存储器访问周期增大。

我们研究的最后一个问题是：增大块大小对这一工作负载是否有所帮助（增大块大小应当能够降低指令和冷缺失率，在限度范围内，还会降低容量/冲突缺失率，并可能降低真共享缺失率）。图 5-10 显示了当块大小由 32 字节变化到 256 字节时，每千条指令的缺失数目。将块大小由 32 字节变化到 256 字节会影响到 4 个缺失率分量。 371~372

- 真共享缺失率的降低因数大于 2，表示真共享模式中存在某种局域性。
- 强制缺失率显著降低，与我们的预期一致。
- 冲突/容量缺失有小幅降低（降低因数为 1.26，而在增大块大小时的降低因数为 8），表示当 L3 缓存大于 2 MB 时所发生的单处理器缺失没有太高的空间局域性。
- 假共享缺失率接近翻番，尽管其绝对数值较小。

对指令缺失率缺乏显著影响，这一事实是令人惊讶的。如果有一个仅包含指令的缓存具备这一特性，那就可以得出结论：其空间局域性非常差。在采用混合 L2 缓存时，诸如指令数据冲突之类的其他影响也可能会导致较大块中产生较高的指令缓存缺失。其他研究已经表明，在大型数据库和 OLTP 工作负载（它们有许多小的基本块和专用代码序列）的指令流中，空间局域性较低。根据这些数据，可以将块大小为 32 字节的 L3 的缺失代价作为基准，将块大小较大的 L3 的缺失代价表示为前者的倍数。 370

块大小	相对于32字节块缺失代价的缺失代价
64 字节	1.19
128字节	1.36
256字节	1.52

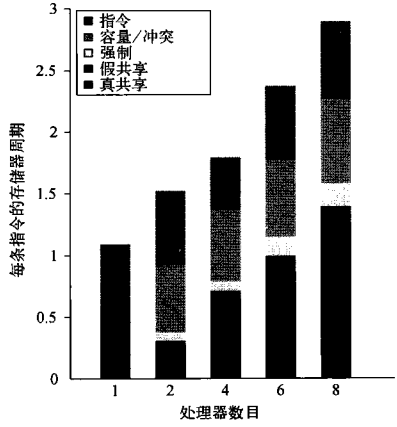


图 5-9　当处理器数目增大时，存储器访问周期的各项导致因素因为真共享缺失的增加而增加。由于每个处理器现在必须处理更多的强制缺失，所以强制缺失会稍有增加

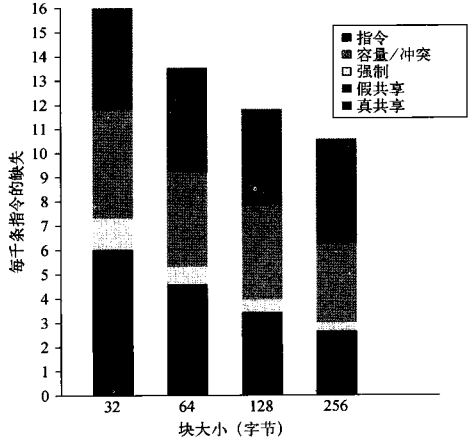


图 5-10　当 L3 缓存的大小增加时，每千条指令的缺失数目稳定下降，所以 L3 块大小至少应当为 128 字节。L3 缓存的大小为 2 MB，两路组相联

由于现代 DDR SDRAM 加快了块访问速度，所以这些数字看起来是可以实现的，特别是块大小为 128 字节的情况。当然，我们还必须考虑增加存储器通信量以及与其他核心争用存储器的影响。后一效果可能很轻松地抵销通过提高单个处理器性能而获得的增益。

5.3.3 多重编程和操作系统工作负载

我们的下一项研究是包括用户行为和操作系统行为的多重编程工作负载。所使用的工作负载是 Andrew 基准测试编译阶段的独立副本，这一基准测试模拟了软件开发环境。其编译阶段使用 8 个处理器执行 Unix "make" 命令的一个并行版本。这一工作负载在 8 个处理器上运行 5.24 秒，生成 203 个进程，对 3 个不同文件系统执行 787 次磁盘请求。运行此工作负载使用了 128 MB 存储器，没有发生分页行为。

此工作负载有 3 个截然不同的阶段：编译基准测试，涉及大量计算行为；将目标文件安装到一个库中；删除目标文件。最后一个阶段完全由 I/O 操作主导；只有两个进程是活动的（每个运行实例有一个进程）。在中间阶段，I/O 也扮演着重要角色，处理器大多处于空闲状态。与经过仔细调优的商业工作负载相比，这个总体工作负载涉及的系统操作和 I/O 操作要多得多。

为进行工作负载的测量，我们假定有以下存储器和 I/O 系统。

- **第一级指令缓存**——32 KB，两路组相联，块大小为 64 字节，命中时间为 1 个时钟周期。
- **第一级数据缓存**——32 KB，两路组相联，块大小为 32 字节，命中时间为 1 个时钟周期。我们改变 L1 数据缓存，以研究它对缓存特性的影响。
- **第二级缓存**——1 MB 一致缓存，两路组相联，块大小为 128 字节，命中时间为 1 个时钟周期。
- **主存储器**——总线上的唯一存储器，访问时间为 100 个时钟周期。
- **磁盘系统**——固定访问延迟为 3 ms（小于正常值，以缩短空闲时间）。

表 5-6 显示如何针对使用上述参数的 8 个处理器来分解其执行时间。执行时间被分解为以下 4 个分量。

(1) **空闲**——在内核模式空闲循环中执行。

(2) **用户**——以用户模式执行。

(3) **同步**——执行或等待同步变量。

(4) **内核**——在既未处于空闲状态也没有进行同步访问的操作系统中执行。

374

表5-6 多重编程并行"make"工作负载中执行时间的分布

	用户执行	内核执行	同步等待	处理器空闲（等待I/O）
所执行的指令	27%	3%	1%	69%
执行时间	27%	7%	2%	64%

* 当 8 个处理器中仅有 1 个处于活动状态时，空闲时间之所以占很大比例是因为磁盘延迟的原因。这些数据及这一工作负载的后续测量由 SimOS 系统收集[Rosenblum 等人，1995]。实际执行及数据收集由斯坦福大学的 M. Rosenblum、S. Herrod 和 E. Bugnion 完成。

这一多重编程工作负载的指令缓存性能损失非常明显，至少对操作系统如此。当块大小为 64 字节、采用两路组相联缓存时，操作系统中的指令缓存缺失率由 32 KB 缓存的 1.7%变为 256 KB 缓存的 0.2%。对于各种缓存大小，用户级指令缓存缺失大体为操作系统缺失率的六分之一。这

一点部分解释了如下事实：尽管用户代码执行的指令数为内核的 9 倍，但这些指令占用的时间仅为内核所执行少量指令的 4 倍。

5.3.4　多重编程和操作系统工作负载的性能

在这一小节，我们研究多重编程工作负载在缓存大小、块大小发生变化时的缓存性能。由于内核特性与用户进程性能之间的差异，我们将这两个分量保持分离。别忘了，用户进程执行的指令是内核的 8 倍，所以整体缺失率部分由用户代码中的缺失率决定，后面将会看到，这一缺失率通常是内核缺失率的五分之一。

尽管用户代码执行更多的指令，但与用户进程相比，操作系统的特性可能导致更多的缓存缺失，除了代码规模较大和缺少局域性之外，还有两个原因。第一，内核在将页面分配给用户之前，会对所有页面进行初始化。第二，内核实际上是共享数据的，因此其一致性缺失率不可忽视。与之相对，用户进程只有在不同处理器上调度进程时才会导致一致性缺失，而这一部分缺失率是很小的。

图 5-11 给出了当数据缓存大小、块大小变化时，数据缺失率的内核及用户部分。增大数据缓存大小对用户缺失率的影响要大于对内容缺失率的影响。增大块大小对于两种缺失率都有正面影响，这是因为很大一部分缺失是因为强制和容量导致，这两者都可能通过增大块大小加以改进。由于一致性缺失相对来说更为罕见，所以增大块大小的负面影响很小。为了了解内核与用户进程的行为为什么会不同，我们可以看看内核缺失是如何表现的。

375

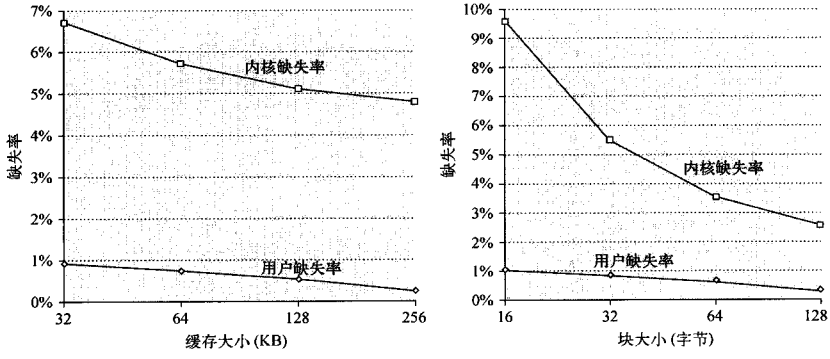


图 5-11　在增大 L1 数据缓存大小（左图）及增大 L2 数据缓存块大小时（右图），数据缺失率的用户分量及内核分量表现不同。将 L1 数据缓存由 32 KB 增大到 256 KB（块大小为 32 字节）导致用户缺失率的降低大于内核缺失率：用户级缺失率的下降因数大约为 3，而内核级缺失率的下降因数仅为 1.3。当 L1 块大小增大时（保持 L1 缓存为 32 KB），缺失率的用户分量及内核分量都会稳定下降。与增大缓存大小的影响相对，增大块大小会显著降低内核缺失率（当块大小由 16 字节变为 128 字节时，内核引用的下降因数仅略低于 4，而用户引用则略低于 3）

图 5-12 显示了缓存大小及块大小增大时，内核缺失的变化。这些缺失被分为三类：强制缺失、一致性缺失（由真、假共享导致）和容量/冲突缺失（包括由于操作系统与用户进程之间和

多个用户进程之间的干扰所导致的缺失）。图 5-12 证明：对于内核引用，增大缓存大小只会降低单处理器容量/冲突缺失率。与之相对，增大块大小会导致强制缺失率的降低。当块大小增大时，一致性缺失率没有大幅增大，这意味着假共享效率可能不是很明显，尽管此类缺失可能会抵消通过降低真共享缺失所带来的增益。

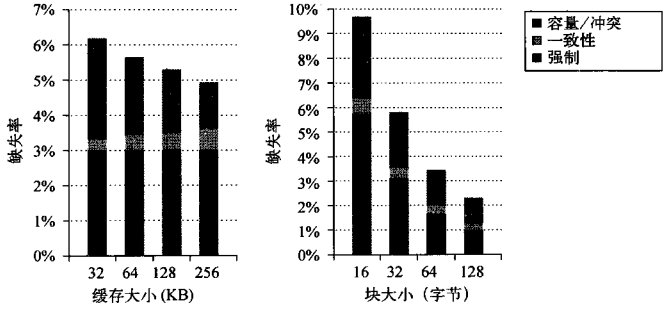


图 5-12 在 8 个处理器上运行多重编程工作负载，当 L1 数据缓存大小由 32 KB 变化为 256 KB 时，内核数据缺失率分量的变化。强制缺失率分量保持不变，因为它不受缓存大小的影响。容量分量的下降因数大于 2，而一致性分量几乎翻番。一致性缺失增大的原因在于：发生冲突的项目会由于容量原因而变少，所以失效操作导致发生缺失的可能性会随着缓存大小的增大而增大。可以预料，L1 数据缓存的块大小增加会大幅降低内核引用中的强制缺失率。它对容量缺失率也有显著影响，在块大小的变化范围中，这一缺失率的降低因数为 2.4。增加块大小只能少量减少一致性通信流量，它在 64 字节时稳定下来，在变为 128 字节时，一致性缺失率没有变化。由于当块大小增加时一致性缺失率没有显著降低，所以因为一致性所导致的缺失率部分由大约 7%增长到大约 15%

如果我们研究每次数据引用所需要的字节数，如图 5-13 所示，可以看到内核的通信流量比较高，会随着块大小的增加而增加。很容易就能看出其原因：当块大小由 16 字节变为 128 字节时，缺失率大约下降 3.7，但每次缺失传送的字节数增大 8 倍，所以总缺失通信量仅提高 2 倍多一点。当块大小由 16 字节变为 128 字节时，用户程序的增大也会超过 2 倍，但它的起始水平要低得多。 376

对于多重编程工作负载，操作系统对存储器系统的要求要严格得多。如果工作负载中包含了更多的操作系统行为或类似于操作系统的行为，而且其特性类似于这一工作负载测量的结果，那就很难构建具有足够功能的存储器系统。一种可能是提高性能的方法是让操作系统更多地了解缓存，可能是通过更好的编程环境，也可能通过程序员的帮助来实现。例如，操作系统会为不同系统调用发出的请求而重复利用存储器。尽管被重复利用的存储器将被完全改写，但硬件并没有意识到这一点，它会尝试保持一致性，即使缓存块不会被读取，也会坚持认为存在这一可能性。这一行为类似于在过程调用时重复利用栈位置。IBM Power 系列就已经允许编译器在过程调用时指示这种行为类型，最新的 AMD 处理器也提供类似支持。系统是很难检测这种行为的，所以可能需要程序员提供帮助，其回报可能要大得多。 377

操作系统与商业工作负载对多处理器存储器系统提出了非常严酷的挑战，而且它们与科学应用程序不同（在附录 I 中进行研究），不太适合进行算法或编译器重构。随着核心数目的增加，

预测此类应用程序的行为也会变得更为困难。一些模拟或仿真技术可以用大型应用程序（包括操作系统）对数百个核心进行仿真，它们对于坚持设计的分析与量化方法至关重要。

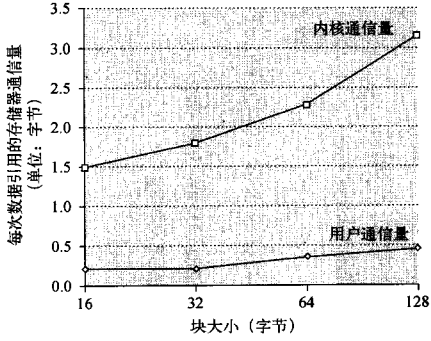


图 5-13　当块大小增加时，对于内核分量与用户分量，每次数据引用所需要的字节数据会增加。将这个图表与附录 I 中的科学程序数据进行对比是很有意义的

5.4　分布式共享存储器和目录式一致性

我们在 5.2 节讨论过，监听协议在每次发生缓存缺失时都需要与所有缓存进行通信，包括对共享数据进行的写入操作。监听式机制没有任何用于跟踪缓存状态的集中式数据结构，这是它的一个基本优点（因为这样可以降低成本），但考虑到可伸缩性时，这也成了它的“阿基里斯脚跟”。

例如，考虑一个由四核多核心组成的多处理器，它能够保持每个时钟周期一次数据引用的速率，时钟速率为 4 GHz。从附录 I 中的 I.5 节的数据可以看出，这些应用程序需要 4 GB/s ~ 170 GB/s 的总线带宽。尽管这些实验中的缓存很小，但大多数通信量都是一致性通信流量，不受缓存大小的影响。尽管现代总线可以达到 4 GB/s 的带宽，但 170 GB/s 还是远远超过了任何总线式系统的能力。在最近几年中，多核处理器的发展迫使所有设计人员转向某种分布式存储器，以支持各个处理器的带宽要求。

378

我们可以通过分布式存储器来提高存储器带宽和互连带宽，如图 5-2 所示；这样会立刻将本地存储器通信与远程存储器通信隔离开来，降低了对存储器系统和互连网络的带宽要求。除非不再需要一致性协议在每次缓存缺失时都进行广播，否则通过分布式存储器不会有太大收益。

如前所述，监听式一致性协议的替代方法是**目录式协议**。目录中保存了每个可缓存块的状态。这个目录中的信息包括哪些缓存（或缓存集合）拥有这个块的副本，它是否需要更新，等等。在一个拥有共享最外层缓存（即 L3）的多核心中，实现目录机制比较容易：只需要为每个 L3 块保存一个位向量，其大小等于核心的数目。这个位向量表示哪些专用缓存的 L3 中可能拥有一个块的副本，失效操作仅会发送给这些缓存。如果 L3 是包含性的，那这一方法对于单个多核心是非常有效的，在 Intel i7 中就是采用了这一机制。

在多核心中使用单个目录时，即使它能避免广播，这种解决方案也是不可能扩展的。这个目录必须是分布式的，但其分布方式必须能够让一致性协议知道去哪里寻找存储器所有缓存块的目录信息。一个容易想到的解决方案是将这个目录与存储器分布在一起，使不同一致性请求

可以进入不同目录，就像不同存储器请求进入不同存储器一样。分布式目录保留了如下特性：块的共享状态总是放在单个已知位置。利用这一性质，再另外维护一些信息，指出其他哪些节点可能缓存这个块，就可以让一致性协议避免进行广播操作。图 5-14 显示了在向每个节点添加目录时，分布式存储器多处理器的样子。

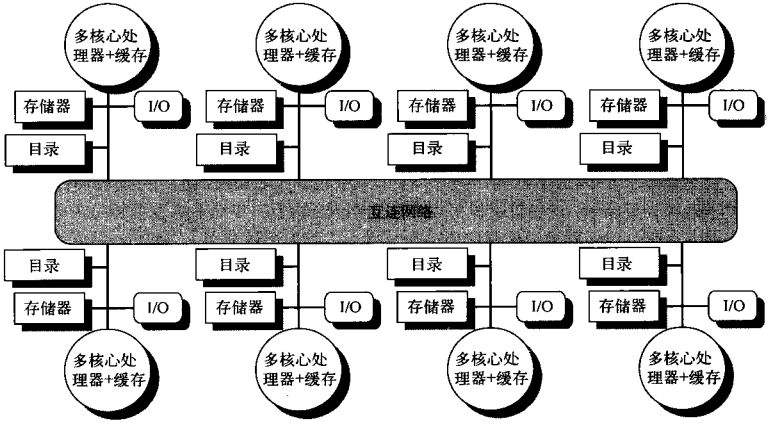


图 5-14 向每个节点添加一个目录，以在分布式存储器多处理器中实施缓存一致性。在本例中，节点被显示为单个多核芯片，相关存储器的目录信息可能驻存在多核心处理器的内部，也可能在其外部。每个目录负责跟踪一些缓存，这些缓存共享该节点内部部分存储器的存储器地址。一致性机制可能会维护多核心节点内部的目录信息，并处理所需要的一致性操作

最简单的目录实现方法是将每个存储器块与目录中的一项相关联。在这种实现方式中，信息量与存储器块数（每个块的大小与 L2 或 L3 缓存块相同）和节点数的乘积成正比，其中一个节点就是在内部实施一致性的单个多核心处理器或一小组处理器。对于处理器少于数百个的多处理器而言（每个处理器可能是多核的），这一开销不会导致问题，因为当块大小比较合理时，目录开销是可以忍受的。对于大型多处理器，需要一些方法来高效地扩展目录结构，不过，只有超级计算机规模的系统才需要操心这一点。 379

5.4.1 目录式缓存一致性协议：基础知识

和监听式协议一样，目录式协议也必须实现两种主要操作：处理读取缺失和处理共享、清洁缓存块的写入操作。（对于当前正被共享的块，其写入缺失的处理就是上述两种操作的组合。）为实现这些操作，目录必须跟踪每个缓存块的状态。在简单协议中，这些状态可能为下列各项之一。

- **共享**——一或多个节点缓存了这个块，存储器的值是最新的（所有缓存中也是如此）。
- **未缓存**——所有节点都没有这个缓存块的副本。
- **已修改**——只有一个节点有这个缓存块的副本，它已经对这个块进行了写操作，所以存储器副本已经过期。这个处理器被称为这个块的**拥有者**。

除了跟踪每个潜在共享存储器块的状态之外，我们还必须跟踪哪些节点拥有这个块的副本，在进行写入操作时需要使这些副本失效。最简单的方法是为每个存储器块保存一个位向量，当 380

这个块被共享时，这个向量的每一位指明相应的原处理器芯片（它可能是一个多核心）是否拥有这个块的副本。当存储器块处于独占状态时，我们还可以使用这个位向量来跟踪块的拥有者。为了提高效率，还会跟踪各个缓存中每个缓存块的状态。

每个缓存中状态机的状态与转换都和监听缓存中使用的状态机相同，只不过在发生转换时的操作稍有不同。用于定位一个数据项独占副本并使其失效的过程有所不同，因为它们需要在发出请求的节点与目录之间进行通信，在目录与一或多个远程节点进行通信。在监听式协议中，这两个步骤通过向所有节点进行广播而结合在一起。

在查看这种协议的状态图之前，先来研究一下为了处理缺失和保持一致性而可能在处理器和目录之间传送的消息类型，这样会有所帮助。表 5-7 给出了节点之间发送的消息类型。**本地节点**是发出请求的节点。**主节点**（home node）就是一个地址的存储器位置及目录项所在的节点。物理地址空间是静态分布的，所以事先知道哪个节点中包含给定物理地址的存储器与目录。例如，高阶位可以提供节点编号，低阶位提供节点上存储器内的偏移。本地节点也可能是主节点。当主节点是本地节点时，由于副本可能存储于第三节点上（称为**远程节点**），所以必须访问该目录。

381

表5-7　在节点之间为保证一致性而发送的可能消息，以及源节点和目标节点、消息内容（p=发出请求的节点编号，A=所请求的地址，D=数据内容），和消息的功能

消息类型	来源	目标	消息内容	消息的功能
读取缺失	本地缓存	主目录	P，A	节点P在地址A发生读取缺失，请示数据并将P设置为读取共享者
写入缺失	本地缓存	主目录	P，A	节点P在地址A发生写入缺失，请求数据并使用P成为独占拥有者
失效	本地缓存	主目录	A	向所有缓存了地址A块的远程缓存发送失败请求
失效	主目录	远程缓存	A	使地址A处数据的共享副本失效
取数据	主目录	远程缓存	A	取回地址A的块，并发送到它的主目录；把远程缓存中A的状态改为共享
取数据/失效	主目录	远程缓存	A	取回地址A的块，并发送到它的主目录；使缓存中的块失效
数据值应答	主目录	本地缓存	D	从主存储器返回数据值
数据写回	远程缓存	主目录	A，D	写回地址A的数据值

* 前 3 条消息是由本地节点发送到主节点的请求。第四个到第六个是当主节点需要数据来满足读取缺失或写入缺失请求时，向远程节点发送的消息。数据应答消息用于由主节点向发出请求的节点传送一个取值。在两种情况下需要对数据值执行写回操作：一种情况是，如果替换了缓存中的一个数据块，且必须写回到它的主存储器中；另一种情况是，对来自主节点的取数据消息或取数据/失效消息做应答时。只要数据块处于共享状态就执行写回操作，这样能简化协议中的状态数目，这是因为任何脏数据块必须处于独占状态并且任何共享块总是可以在主存储器中获取。

远程节点是拥有缓存块副本的节点，这一副本可能独占（在此情况下只有一个副本），也可能共享。远程节点也可能与本地节点或主节点相同。在此类情况下，基本协议不会改变，但处理器之间的消息可能会被处理器内部的消息代替。

在这一节，我们采用存储器一致性的一种简单模型。为了在最大程度上减少这种类型的消息及协议的复杂性，我们假定这些消息的接受及处理顺序与其发送顺序相同。这一假定在实际中并不成立，可能会导致额外的复杂性，在 5.6 节讨论存储器一致性模型时会讨论其中一部分内容。在这一节，我们利用这一假定来确保在传送新消息之前先处理节点发送的失效操作，就像在讨论监听式协议时的假设一样。和在监听情景中一样，我们省略了一些实现一致性协议所必需的细节。具体来说，要想实现写入操作的串行化，并获知某写入的失效操作已经完成，并不像广播式监听机制那样轻松，而是需要采用明确的确认方法来回应写入缺失和失效请求。附录 I 中更详细地讨论了这些问题。

5.4.2 目录式协议举例

目录式协议中缓存块的基本状态与监听式协议中完全相同。目录中的状态也与我们前面展示的状态类似。因此，我们首先看一个简单的状态图，它给出了一个具体缓存块的状态转换，然后再研究与存储器中每一个块相对应的目录项的状态图。和监听情景中一样，这些状态转换图并没有给出一致性协议的所有细节；但是，实际控制器调度依赖于多处理器的大量细节（消息发送特性、缓冲结构，等等）。在这一节，我们给出了基本的协议状态图。附录 I 中研究了在实现这些状态转换图中的一些棘手问题。

图 5-15 显示了一个具体缓存对应的协议操作。所使用的符号与上一节相同，来自节点外部的请求用灰色表示，操作用黑色表示。一个具体缓存的状态转换由读取缺失、写入缺失和状态提取请求导致；图 5-15 显示了这些操作。一个具体缓存也会生成这些读取缺失、写入缺失和失效消息，它们会被发送给主目录。读取缺失与写入缺失要求数值回复，这些事件在改变状态之前会等待回复。如何知道失效操作何时完成，那是另一个问题，另行处理。 382

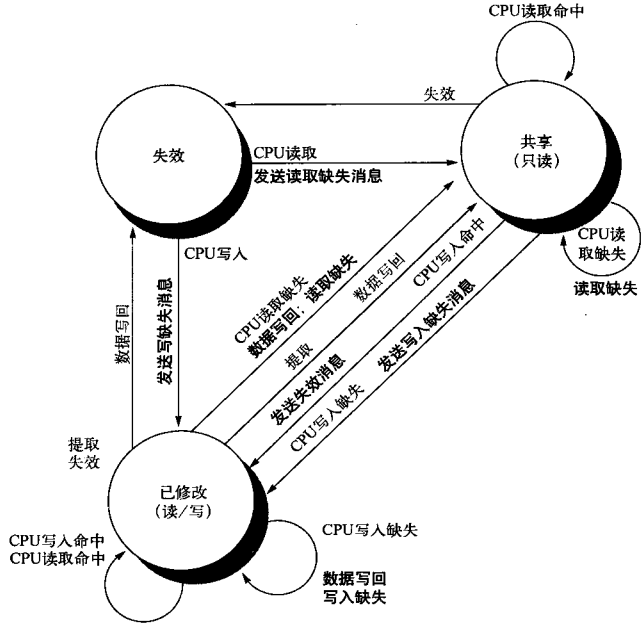


图 5-15 **目录式系统中一个具体缓存块的状态转换图。**本地处理器的请求用黑色表示，来自主目录的请求用灰色表示。这些状态与监听式系统中相同，而且事务非常类似，用显式失效与写回请求来代替向总线正式广播的写入缺失。与监听控制器中一样，我们仍然假定在尝试写入共享缓存块时将被作为缺失而进行处理；在实践中，这样的事务可以看作拥有权请求或升级请求，可以在未请求所提取缓存块的同时提交拥有权 383

图5-15中缓存块状态转换图的操作基本上与监听情景中一样：状态是相同的，激励也几乎相同。写入缺失操作由数据提取和失效操作替代，失效操作由目录控制器选择性地发送，而在监听机制中，写入缺失操作是在总线（或其他网络）上广播的。与监听协议一样，在写入缓存块时，它必须处于独占状态，所有共享块都必须在存储器中进行更新。在许多多核处理器中，在核心之间共享处理器缓存的最外层级（比如Intel i7、AMD Opteron和IBM Power7中的L3），处于这一级别的硬件将在同一芯片上每个核心的专用缓存之间保持一致性，或者使用内部目录实现，或者使用监听实现。因此，只需要与最外层共享缓存进行交互，就能使用芯上多核一致性机制在大量处理器之间扩展一致性。因为这一交互是在L3层级进行的，所以处理器与一致性请求之间的争用就不会导致问题，也可以避免标签的复制。

在目录式协议中，目录实现了一致性协议的另一半。发向目录的一条消息会导致两种不同类型的操作：更新目录状态；发送附加消息以满足请求。目标中的状态表示一个块的三种标准状态；但与监听机制中不同的是，目录状态表示一个存储器块所有缓存副本的状态，而不是表示单个缓存块的相应信息。

存储器块可能未由任何节点缓存，可能缓存于多个节点中并可读（共享），也可能仅在一个节点中独占缓存并可写。除了每个块的状态之外，目录还会跟踪拥有某一缓存块副本的节点集合；我们使用名为**共享器**的集合来执行这一功能。在节点数少于64的多处理器中（每个节点可能表示4～8倍的处理器），这一集合通常表示为位向量。目录请求需要更新这个**共享器集合**，还会读取这个集合，以执行失效操作。

图5-16给出了在目录中为回应所接收消息而采取的操作。目录接收三种不同请求：读取缺失、写入缺失和数据写回。目录发送的回应消息用粗体表示，而集合“共享器”的更新用黑色表示。因为所有激励消息都来自外部，所以所有操作都以灰色表示。我们的简化协议假定一些操作是原子操作，比如请求某个值并将其发送给另一个节点；实际实现时不能采用这一假定。

为了理解这些目录操作，让我们逐个状态查看所接收的请求和所采取的操作。当块处于未缓存状态时，存储器中的副本就是当前值，所以对这个块的请求只能是以下两种。

- **读取缺失**——从存储器向发出请求的节点发送其请求的数据，请求者成为唯一的共享节点。块的状态变为共享。
- **写入缺失**——向发送请求的节点传送取值，该节点变为共享节点。这个块变为独占状态，表明缓存了唯一有效副本。共享器指明拥有者的身份。

384

当块处于共享状态时，存储器值是最新的，所以可能出现相同的两个请求。

- **读取缺失**——从存储器向发出请求的节点发送其请求的数据，请求者被添加到共享集合中。
- **写入缺失**——向请求节点发送取值。向**共享者**集合中的所有节点发送失效消息，**共享者**集合将包含发出请求的节点的身份。这个块的状态变为独占状态。

当块处于独占状态时，这个块的值保存在一个节点的缓存中，这个节点由**共享者**（拥有者）集合识别，所以共有3种可能的目录请求。

385

- **读取缺失**——向拥有者发送数据提取消息，它会将拥有者缓存中这个块的状态转变为共享，拥有者将数据发送给目录，再在这里将其写到存储器中，并发给提出请求的处理器。将发出请求的节点的身份添加到**共享者**集合中，这个集合中仍然包含拥有者处理器的身份（因为这个处理器仍然拥有可读副本）。

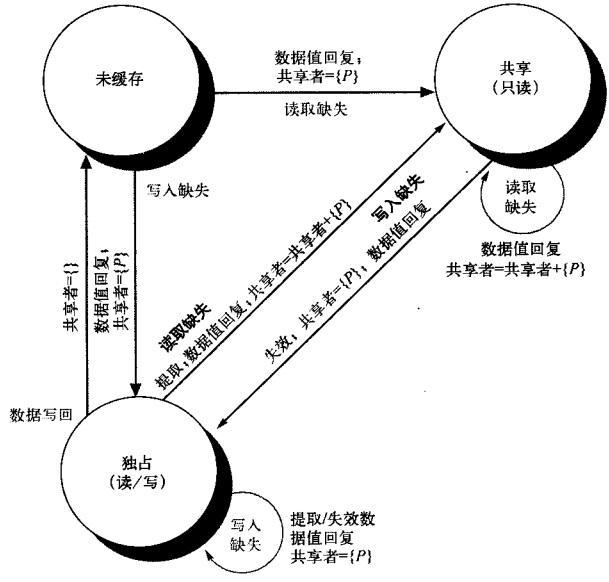


图 5-16 目录的状态转移图与独立缓存的转移图具有相同的状态和结构。由于所有操作都是由外部导致的，所以均以灰色表示。粗体表示该目录回应请求所采取的操作

- **数据写回**——拥有者正在替换这个块，因此必须将其写回。这个写回操作会更新存储器副本（主目录实际上变为拥有者），这个块现在未被缓存，**共享者**集合为空。
- **写入缺失**——这个块有一个新的拥有者。向旧拥有者发送一条消息，将其缓存中的这个块失效，并将值发送给目录，从目录中发送给提出请求的节点，这个节点现在变成新的拥有者。**共享者**被设定为新拥有者的身份，这个块仍然保持独占状态。

图 5-16 中的状态转换图是一种简化图，与监听式缓存的情景相同。在采用目录式协议时，以及用网络而非总线来实现监听机制时，协议需要处理非原子化存储器转换。附录 I 深入探讨了这些问题。

实际多处理器中使用的目录协议还进行了其他一些优化。具体来说，在这种协议中，在针对独占块发生读取缺失或写入缺失时，会首先将这个块发送到主节点上的目录中。再从这里将其存储到主存储器中，并发送给原来发现请求的节点。商用多处理器使用的许多协议都会将数据从拥有者节点直接转发给发出请求的节点（同时对主节点执行写回操作）。由于这些优化方法增大了死锁可能，并增加了必须处理的消息类型，所以通常会提高复杂性。

从 5.2.8 节开始，我们讨论了实现监听式协议的一些挑战，在实施目录式机制时，也要应对大部分此类挑战，而且还增加了一些新的问题，在附录 I 中介绍了这些新问题。在 5.8 节，我们将简要介绍现代多核处理器是如何将一致性扩展到单个芯片之外的。多芯片一致性和多核一致性有 4 种组合方式：监听/监听（AMD Opteron）、监听/目录、目录/监听和目录/目录！

5.5 同步：基础知识

同步机制通常是以用户级软件例程实现的，这些例程依赖于硬件提供的同步指令。对于较小型的多处理器或低争用解决方案，一种很关键的硬件功能是拥有不可中断的指令或指令序列，它们能以原子方式提取和改变一个值。软件同步机制就是利用这一功能实现的。这一节的重点是锁定及非锁定同步操作的实现。可以非常轻松地利用锁定和非锁定来创建互斥，并实现更复杂的同步机制。

386

在高争用情景中，同步可能会成为性能瓶颈，因为争用会引入更多延迟，在此种多处理器中，延迟可能更大一些。附录 I 将会讨论如何将这一节讨论的基本同步机制扩展到处理器数目很大的情况。

5.5.1 基本硬件原语

在多处理器中实施同步时所需要的关键功能就是一组能够以原子方式读取和修改存储器位置的硬件原语。没有这一功能，构建基本同步原语的成本就会过高，并随着处理器数目的增大而增大。基本硬件原语有许多替代方式，所有这些方式都能够以原子形式读取和修改一个位置，还有某种方法可以判断读取和写入是否以原子形式执行。这些硬件原语是一些基本构建模块，用于构建各种用户级别的同步操作，包括诸如锁和屏障之类的内容。一般情况下，架构师不希望用户利用基本硬件原语，而是希望这些原语由系统程序员用来构建同步库，这个过程通常比较复杂，需要一些技巧。我们首先来看这样一个硬件原语，说明如何用它来构建某些基本的同步操作。

一种用于构建同步操作的典型操作就是**原子交换**，它会将寄存器中的一个值与存储器的一个值进行交换。为了明白如何利用这一操作来构建基本的同步操作，假定我们希望构建一个简单锁，数值 0 表示这个锁可以占用，数值 1 表示这个锁不可用。处理器尝试对这个锁进行置位，具体做法是将寄存器中的 1 与这个锁的相应存储器地址进行交换。如果其他某个处理器已经申请了访问权，则这一交换指令将返回 1，否则返回 0。在后一种情况下，这个值也被改变为 1，以防止任意进行竞争的交换指令也返回 0。

例如，考虑两个处理器，每个处理器都尝试同时进行交换：只有一个处理器将会首先执行交换操作，并返回数值 0，第二个处理器进行交换时将会返回 1，所以不会存在竞争问题。使用交换原语来实现同步的关键是这个操作具有原子性：这一交换是不可分的，两个同时交换将由写入串行化机制进行排序。如果两个处理器尝试以这种方式对同步变量进行置位，它们不可能认为自己同时对这个变量进行了置位。

387

还有大量其他原子原语可用于实现同步。它们都拥有一个关键特性：它们读取和更新存储器值的方式可以让我们判断这两种操作是不是以原子形式执行的。在许多较旧的多处理器中存在一种名为**测试并置位**的操作，它会测试一个值，如果这个值通过测试则对其进行置位。比如，我们可以定义一个操作，它会检测 0，并将其值设定为 1，其使用方式与使用原子交换的方式类似。另一个原子同步原语是**提取并递增**：它返回存储器位置的值，并以原子方式使其递增。我们用 0 值来表示同步变量未被声明，可以像使用交换一样使用提取与递增。稍后将会看到，还有其他一些类似于“提取并递增”的操作用法。

实现单个原子存储器操作会引入一些挑战，因为它需要在单个不可中断的指令中进行存储

器读取与写入操作。这一要求增加了一致性实施的复杂性，因为硬件不允许在读取与写入之间插入任何其他操作，而且不能死锁。

替代方法是利用一对指令，其中第二条指令可以返回一个值，根据这个值，可以判断这一对指令是否以原子形式执行。如果任一处理器执行的所有其他指令要么在这对指令之前执行，可么在这对指令之后执行，那就可以认为这对指令具有原子性。因此，如果一个指令对具有原子特性，那所有其他处理器都不能在这个指令对之间改变取值。

这种指令对包含一种名为**链接载入**或**锁定载入**的特殊载入指令和一种名为**条件存储**的特殊存储指令。这些指令是按顺序使用的：对于链接载入指令指定的存储器位置，如果其内容在对同一位置执行条件存储之前发生了改变，那条件存储就会失败。如果在两条指令之间进行了上下文切换，那么存储条件也会失败。条件存储的定义是在成功时返回 1，失败时返回 0。由于链接载入返回了初始值，而条件存储仅在成功时才会返回 1，所以以下序列对 R1 内容指定的存储器位置实现了一次原子交换：

```
try:   MOV R3,R4   ;移动交换值
       LL  R2,0(R1);链接载入
       SC  R3,0(R1);条件存储
       BEQZ R3,try ;分支存储失败
       MOV R4,R2   ;将载入值放入 R4 中
```

在这个序列的末尾，R4 的内容和 R1 指定存储器位置的内容已经实现了原子交换（忽略了延迟分支的影响）。在任意时间，如果处理器介入 LL 和 SC 指令之间，修改了存储器中的取值，那么 SC 在 R3 中返回 0，导致此代码序列再次尝试。

388

链接载入/条件存储机制的益处之一就是它能用于构建其他同步原语。例如，下面是原子的“提取并递增”：

```
try:   LL  R2,0(R1) ;链接载入
       DADDUIR3,R2,#1 ;递增
       SC  R3,0(R1) ;条件存储
       BEQZ R3,try ;条件存储失败
```

这些指令通常是通过在寄存器中跟踪 LL 指令指定的地址来实现的，这个寄存器称为**链接寄存器**。如果发生了中断，或者与链接寄存器中地址匹配的缓存块失效（比如，另一条 SC 使其失效），链接寄存器将被清除。SC 指令只是核查它的地址与链接寄存器中的地址是否匹配。如果匹配，SC 将会成功；否则就会失败。在再次尝试向链接载入地址进行存储之后，或者在任何异常之后，条件存储将会失败，所以在选择向两条指令之间插入的指令时必须非常小心。具体来说，只有寄存器-寄存器指令才是安全的；否则，就有可能造成死锁情景，处理器永远无法完成 SC。此外，链接载入和条件存储之间的指令数应当很小，以尽可能减少无关事件或竞争处理器导致条件存储频繁失败的情景。

5.5.2 使用一致性实现锁

在拥有原子操作之后，就可以使用多处理器的一致性机制来实施**自旋锁**（spin lock）——处理器持续用循环来尝试获取锁，直到成功为止。在两种情况下会用到自旋锁，一种是程序员希望短时间拥有这个锁，另一种情况是程序员希望当这个锁可用时，锁定过程的延迟较低。因为自旋锁会阻塞处理器，在循环中等待锁被释放，所以在某些情况下不适合采用。

最简单的实施方法是在存储器中保存锁变量，在没有缓存一致性时将会使用这种实施方式。处理器可能使用原子操作（比如 5.5.1 节所述的原子交换）持续尝试获得锁，测试这一交换过程

是否返回了可用锁。为释放锁，处理器只需要在锁中存储数值0即可。下面的代码序列使用原子交换来锁定自旋锁，其地址在R1中：

```
          DADDUIR2,R0,#1
lockit:   EXCHR2,0(R1)     ;原子交换
          BNEZR2,lockit    ;已经锁定?
```

如果多处理器支持缓存一致性，就可以使用一致性机制将锁放在缓存中，保持锁值的一致性。将锁放在缓存中有两个好处。第一，它允许采用一种实施方式，允许针对本地缓存副本完成“自旋”过程（在一个紧凑循环中尝试测试和获取锁），不需要在每次尝试获取锁时都请求全局存储器访问。第二个好处来自以下观察结果：锁访问中经常存在局域性；也就是说，上次使用了一个锁的处理器，很可能会在不远的将来再次用到它。在此类情况下，锁值可以驻存在这个处理器的缓存中，大幅缩短获取锁所需要的时间。

要实现第一个好处（能够针对本地缓存副本进行循环，不需要在每次尝试获取锁时都生成存储器请求），需要对这个简单的自旋过程进行一点修改。上述循环中每次尝试进行交换时都需要一次写入操作。如果多个处理器尝试获取这个锁，会分别生成这一写入操作。这些写入操作大多会导致写入缺失，因为每个处理器都是尝试获取处于独占状态的锁变量。

389

因此，应当修改自旋锁过程，使其在自旋过程中读取这个锁的本地副本，直到看到该锁可用为止。然后它尝试通过交换操作来获取这个锁。处理器首先读取锁变量，以检测其状态。处理器不断地读取和检测，直到读取的值表明这个锁未锁定为止。这个处理器随后与所有其他正在进行“自旋等待”的处理器展开竞赛，看谁能首先锁定这个变量。所有进程都使用一条交换指令，这条指令读取旧值，并将数值1存储到锁变量中。唯一的获胜者将会看到0，而失败者将会看到由获胜者放在里面的1。（失败者会继续将这个变量设置到锁定值，但这已经无关紧要了。）获胜的处理器在锁定之后执行代码，完成后将0存储到锁定变量中，以释放这个锁，然后再从头开始竞赛。下面的代码执行这一自旋锁（别忘了，0是未锁定，1是锁定）：

```
lockit:   LDR2,0(R1)       ;载入锁
          BNEZR2,lockit    ;不可用——自旋
          DADDUIR2,R0,#1   ;载入锁定值
          EXCHR2,0(R1)     ;交换
          BNEZR2,lockit    ;如果锁不为0，则跳转
```

让我们看看这一“自旋锁”机制是如何使用缓存一致性机制的。表5-8显示了当多个进程尝试使用原子交换来锁定一个变量时的处理器和总线（或目录）操作。一旦拥有锁的处理器将0存储到锁中，所有其他缓存都将失效，必须提取新值以更新它们保存的锁副本。这种缓存首先获取未定值（0）的一个副本，并执行交换。在满足其他处理器的缓存缺失之后，它们发现这个变量已经被锁定，所以必须回过头来进行检测和自旋。

390

这个例子显示了链接载入/条件存储原语的另一个好处：读取操作与写入操作是明确独立的。链接载入不一定导致任何总线通信。这一事实允许采用以下简单代码序列，它的特性与使用交换的优化版本一样（R1拥有锁的地址，LL代替了LD，SC代替了EXCH）：

```
lockit:   LLR2,0(R1)       ;链接载入
          BNEZR2,lockit    ;不可用——自旋
          DADDUIR2,R0,#1   ;锁定值
          SCR2,0(R1)       ;存储
          BEQZR2,lockit    ;如果失败则跳转
```

391

第一个分支构成了自旋循环，第二个分支化解当两个处理器同时看到锁可用时的竞赛。

表5-8 三个处理器P0、P1、P2的缓存一致性步骤和总线通信

步骤	P0	P1	P2	步骤结束时锁的一致性状态	总线/目录操作
1	拥有锁	开始自旋、判断锁是否为0	开始自旋、判断锁是否为0	共享	以任意顺序满足P1和P2的缓存缺失。锁状态变为共享
2	将锁设为0	（接收到失效操作）	（接收到失效操作）	独占（P0）	来自P0锁变量的写入失效操作
3		缓存缺失	缓存缺失	共享	总线/目录为P2缓存缺失提供服务，从P0写回，状态为共享
4		（当总线/目录忙时等待）	通过锁为0的检测	共享	满足P2的缓存缺失
5		锁为0	执行交换，获得缓存缺失	共享	满足P1的缓存缺失
6		执行交换、获得缓存缺失	完成交换：返回0，并将锁置为1	独占（P2）	总线/目录为P2缓存缺失提供服务，生成失效，锁为独占
7		交换完成、返回1，将锁设置为1	进入关键部分	独占（P1）	总线/目录为P1缓存缺失提供服务，发送失效操作，从P2生成写回操作
8		自旋、检测锁是否为0			无

* 本表假定采用写入失效一致性。在开始时，P0拥有这个锁（步骤1），锁的值为1（即被锁定）；它最初为独占的，在步骤1开始之前由P0拥有。P0退出并解锁（步骤2）。P1和P2竞赛，看看谁能在交换期间看到未锁定值（步骤3至步骤5）。P2赢得竞赛，进入关键部分（步骤6与步骤7），而P1的尝试失败，所以它开始自旋等待（步骤7和步骤8）。在实际系统中，这些事件将耗费更多时间，远多于8次时钟嘀嗒，因为获取总线和回复缺失所需要的时间要长得多。一旦到了步骤8，这一过程就可以从P2开始重复，它最终获得独占访问，并将锁设置为0。

5.6 存储器连贯性模型：简介

缓存一致性保证了多个处理器看到的存储器内容是一致的。但它并没有回答这些存储器内容应当保持何种程度的一致性。我们问“何种程度的一致性”时，实际是在问一个处理器必须在什么时候看到另一个处理器更新过的值？由于处理器通过共享变量进行通信（用于数据值和同步两种目的），于是这个问题便简化为：处理器必须以何种顺序来观测另一个处理器的数据写入？由于“观测另一处理器写入操作”的唯一方法就是通过读取操作，所以问题现在变为：在不同处理器对不同位置执行读取和写入操作时，必须保持哪些特性？

“保持何种程度的一致性”这一问题看起来非常简单，实际上却非常复杂，我们通过一个简单例子来了解一下。下面是来自处理器P1和P2的两段代码，并排列出如下：

```
P1:     A = 0;          P2:     B = 0;
         .....                   .....
        A = 1;                  B = 1;
L1:     if (B == 0)...  L2:     if (A == 0)...
```

假定这些进程运行在不同处理器上，位置A和B最初由两个处理器进行缓存，初始值为0。如果写入操作总是立刻生效，而且马上会被其他处理器看到，那两个IF语句（标有L1和L2）不可能将其条件计算为真，因为能够到达IF语句，说明A或B中必然已经被指定了数值1。我们假定写入失效被延迟，处理器可以在这一延迟期间继续执行。因此，P1和P2在它们尝试读取数值之前，可能还没有（分别）看到B和A的失效。现在的问题是，是否应当允许这一行为？如果应当允许，在何种条件下允许？

存储器连贯性的最简单模型称为**顺序连贯性模型**。顺序连贯性要求任何程序每次执行的结

果都是一样的，就像每个处理器是按顺序执行存储器访问操作的，而且不同处理器之间的访问任意交错在一起。有了顺序连贯性，就不可能再出现上述示例中的某些不明执行情况，因为必须完成赋值操作之后才能启动 IF 语句。

实现顺序连贯性模型的最简单方法是要求处理器推迟完成所有存储器访问，直到该访问操作所导致的全部失效均告完成为止。当然，如果推迟下一个存储器访问操作，直到前一访问操作完成为止，这种做法同样有效。别忘了，存储器连贯性涉及不同变量之间的操作：两个必须保持顺序的访问操作实际上访问的是不同的存储器位置。在我们的例子中，必须延迟对 A 或 B 的读取（A==0 或 B==0），直到上一次写入操作完成为止（B=1 或 A=1）。比如，根据顺序连贯性，我们不能简单地将写入操作放在写入缓冲区中，然后继续执行读取操作。

392

尽管顺序连贯性模型给出了一种简单的编程范例，但它可能会降低性能，特别是当多处理器的处理器数目很大或者互连延迟很长时尤为如此，如下例所示。

例题　假定有一个处理器，一次写入缺失需要 50 个时钟周期来确定拥有权，在确定拥有权之后发射每个失效操作需要 10 个时钟周期，在发射之后，失效操作的完成与确认需要 80 个时钟周期。假定其他 4 个处理器共享一个缓存块，如果处理器保持顺序连贯性，一次写入缺失会使执行写入操作的处理器停顿多长时间？假定必须明确确认失效操作之后，一致性控制器才能知道它们已经完成。假定在为写入缺失获得拥有者之后可以继续执行，不需要等待失效；该写入操作需要多长时间？

解答　在等待失效时，每个写入操作花费的时间等于拥有时间再加上完成失效所需要的时间之和。由于失效操作可以重叠，所以只需要为最后一项操心，它是在确定拥有权之后开始的 10+10+10+10=40 个时钟周期。因此，写入操作的总时间为 50+40+80=170 个时钟周期。与之相比，拥有时间只有 50 个时钟周期。通过实现适当的写入缓冲区，甚至有可能在确定拥有权之前继续进行。

为了提供更好的性能，研究人员和架构师已经研究了两种不同路径。第一，他们开发了强大的实施方式，能够保持顺序连贯性，但使用延迟隐藏技术来降低代价；我们将在 5.7 节讨论这些内容。第二，他们开发了限制条件较低的存储器一致性模型，支持采用更快速的硬件。这些模型可能会影响程序员看到多处理器的方式，所以在讨论这些低限制模型之前，先来看看程序员有什么期望。

5.6.1　程序员的观点

尽管顺序连贯性模型有性能方面的不足，但从程序员的角度来看，它拥有简单性这一优点。挑战在于，要开发一种编程模型，既便于解释，又支持高性能实施方式。

有这样一种支持更高效实施方式的编程模型，它假定程序是**同步的**。如果对共享数据的所有访问都由同步操作进行排序，那就说这个程序同步的。如果满足以下条件，就说数据引用是由同步操作进行排序的：在所有可能的执行情景中，一个处理器对某一变量的写入操作与另一个处理器对这个变量的访问（或者为读取，或者为写入）之间由一对同步操作隔离开来，其中一个同步操作在写入处理器执行写入操作之后执行，另一个同步操作在第二个处理器执行访问操作之前执行。如果变量可以在未由同步操作进行排序的情况下更新，此类情景称为**数据竞赛**，

393

因为操作的执行结果取决于处理器的相对速度，和硬件设计中的竞赛相似，其输出是不可预测的，由此给出另一种同步程序的名字：**无数据竞赛**。

给出一个简单的例子，变量由两个不同处理器读取和更新。每个处理器用锁定和解锁操作将读取和更新操作包围起来，这两种操作是为了确保更新的互斥和读取操作的连贯性。显然，每个写入操作与另一个处理器的读取操作之间现在都由一对同步操作隔离开来：一个是解锁（在写入操作之后），一个是锁定（在读取操作之前）。当然，如果两个处理器正在写入一个变量，中间没有插入读取操作，那这些写入操作之间也必须由同步操作隔离开。

人们普遍认同“大多数程序都是同步的”这一事实。这一观察结果之所以正确，主要是因为：如果这些访问是非同步的，那么到底哪个处理器赢得数据竞赛就由执行速度决定，从而会影响到程序结果，那程序的行为就可能是不可预测的。即使有了顺序连贯性，也很难理清此类程序。

程序员可能尝试通过构造自己的同步机制来确保排序，但这种做法需要很强的技巧性，可能导致充满漏洞的程序，而且在体系结构上可能不受支持，也就是说在以后各代多处理器中可能无法工作。因此，几乎所有的程序员都选择使用同步库，这些库正确无误，而且针对多处理器和同步类型进行了优化。

最后，标准同步原语的使用可以确保：即使体系结构实现了一种比顺序连贯性模型更宽松的连贯性模型，同步程序也会像硬件实施了顺序连贯性一样运行。

5.6.2 宽松连贯性模型：基础知识

宽松连贯性模型的关键思想是允许乱序执行读取和写入操作，但使用同步操作来实施排序，因此，同步程序的表现就像处理器具备顺序连贯性一样。这些宽松模型是多种多样的，可以根据它们放松了哪种读取和写入顺序来进行分类。我们利用一组规则来指定顺序，其形式为 X→Y，也就是说必须在完成操作 X 之后才能执行操作 Y。顺序连贯性模型需要保持所有 4 种可能顺序：R→W、R→R、W→R 和 W→W。宽松模型的确定是看它们放松了这 4 种顺序中的哪一种。 394

(1) 放松 W→R 顺序，将会得到一种称为**完全存储排序或处理器连贯性**的模型。由于这种排序保持了写入操作之间的顺序，所以许多根据顺序连贯性运行的程序也能在这一模型下运行，不用添加同步。

(2) 放松 W→W 顺序，将会得到一种称为**部分存储顺序**的模型。

(3) 放松 R→W 和 R→R 顺序，将会得到许多模型，包括**弱排序**、PowerPC 连贯性模型和**释放连贯性**，具体取决于排序约束条件的细节和同步操作实施排序的方式。

通过放松这些排序，处理器有可能获得显著的性能提升。但是，在描述宽松连贯性模型时存在许多复杂性，包括放松不同顺序的好处与复杂性、准确定义写入完成的含义、决定处理器什么时候看到它自己写入的值。有关宽松模型的复杂性、实施问题以及性能潜力的更多信息，我们强烈推荐 Adve 和 Gharachorloo [1996]的优秀教程。

5.6.3 关于连贯性模型的最后说明

目前，许多正在开发的多处理器都支持某种宽松连贯性模型，既有处理器连贯性，又有释放连贯性。由于同步过程与处理器特性密切相关，而且容易导致错误，所以人们希望大多数程序员使用标准同步库，编写同步程序，使程序员感觉不到弱连贯性模型的选择，并得到更高的性能。

还有一种观点认为，利用推测功能，宽松连贯性模型的大多数性能优势都能通过顺序连贯性或处理器连贯性实现。下一节将更全面地讨论这一内容。

这一观点中关于宽松连贯性的关键部分涉及编译器的角色，以及针对潜在共享变量来优化存储器访问的能力；这一主题也将在5.7节讨论。

5.7　交叉问题

由于多处理器重新定义了许多系统特性（例如，性能评估、存储器延迟和可伸缩性的重要性），所以它们引入了一些贯穿整个领域的重要设计问题，对硬件和软件都产生影响。在这一节，我们将给出一些与存储器连贯性问题有关的示例。随后研究在向多重处理中添加多线程时所能获得的性能。

395

5.7.1　编译器优化与连贯性模型

定义存储器连贯性模型的另一个原因是指定合法的编译器优化范围，可以针对共享数据来执行这些优化。在显式并行程序中，除非明确定义了同步点，而且程序被同步，否则编译器不能交换对两个不同共享数据项的读取操作和写入操作，因为这种转换可能会影响程序的本来语义。因此，一些相对简单的优化方式也无法实施，比如共享数据的寄存器分配，因为这种转换通常会交换读取和写入操作。在隐式并行程序中（比如，用“高性能FORTRAN(HPF)”编写的程序），程序必须被同步，而且同步点已知，所以不会出现这一问题。编译器能否从更宽松的连贯性模型中获得明显好处，无论是从研究的角度来看还是从实践的角度来看，这都依然是一个开放性的问题，由于缺乏统一模型，可能会妨碍编译器的部署进程。

5.7.2　利用推测来隐藏严格连贯性模型中的延迟

在第3章曾经看到，可以利用推测来隐藏存储器延迟。还可用来隐藏因为严格连贯性模型导致的延迟，获得宽松存储器模型的大多数好处。其关键思想是：处理器使用动态调度来重新安排存储器引用的顺序，让它们有可能乱序执行。乱序执行存储器引用可能会违犯顺序连贯性，从而影响程序的执行。利用推测处理器的延迟提交功能，可以避免这种可能性。假定一致性协议是以失效操作为基础的：如果处理器在提交存储器引用之前，收到该存储器引用的失效操作，处理器会使用推测恢复来回退计算，并利用失效地址的存储器引用重新开始。

如果处理器对存储器请求进行重新排序后，新执行顺序的结果不同于在顺序连贯性下看到的结果，处理器将会撤消此次执行。使用这一方法的关键在于：处理器只需要确保其结果与所有访问完全循序完成时是一样的，通过检测两种结果可能在什么时候出现不同，就可以做到这一点。由于很少会触发推测重启，所以这种方法很有吸引力。只有当非同步访问实际导致竞赛时，才会触发推测重启[Gharachorloo、Gupta和Hennessy 1992]。

Hill [1998]提倡将顺序连贯性或处理器连贯性与推测执行结合起来，作为一种连贯性模型。他的观点包括三个部分。第一，积极实现顺序连贯性或处理器连贯性，可以获得更宽松模型的大多数好处。第二，这种实施方式仅对推测处理器增加了非常少的实施成本。第三，这种方法允许程序员考虑使用顺序连贯性或处理器连贯性的更简单编程模型。MIPS R10000设计团队在20世纪90年代中期就深刻认识到这一点，使用R10000的乱序功能来支持顺序连贯性的这种积极实施方式。

396

一个尚未解决的问题是，在优化对共享变量的存储器引用时，编译器技术会取得怎样的成功？共享数据通常是通过指针和数组索引进行访问的，这一事实再加上优化技术的现状，已经

限制了此种优化技术的使用。如果这一技术进入实用状态，而且能够带来显著的性能优势，编译器编写人员可能会希望使用更宽松的编程模型。

5.7.3 包含性及其实现

所有多处理器都使用多级缓存层级结构来减少对全局互连的要求和缓存缺失延迟。如果缓存还提供了**多级包含性**（缓存层次结构的每一级都是距处理器更远一层的子集），所以我们可以使用多级结构来减少一致性通信与处理器通信之间的争用，当监听与处理器缓存访问必须竞争缓存时，就会出现这些争用。许多具有多层缓存的多处理器都具备这种包含性，不过，最近有些多处理器采用较小的 L1 缓存和不同的块大小，有时会选择不实施这种包含特性。这一限制有时也称为**子集特性**，因为每个缓存都是它下一级缓存的子集。

乍看起来，保持多级包含特性是件很简单的事情。考虑一个两级示例：L1 中的所有缺失要么在 L2 命中，要么在 L2 中产生缺失，无论是哪一种情况，缺失块都会进入 L1 和 L2 两级缓存。与此类似，任何在 L2 命中的失效都必然被发送给 L1，如果 L1 中存在这个块，将会使其失效。

难以理解的地方在于当 L1 和 L2 的块大小不同时会发生什么。选择不同块大小是非常合情合理的，因为 L2 通常要大得多，其缺失代价中的延迟分量也要长得多，因此希望使用较大的块大小。当块大小不同时，对于包含性的“自动”实施有什么影响呢？L2 中的一个块对应于 L1 中的多个块，L2 的一次缺失所导致的数据替换对应于多个 L1 块。例如，如果 L2 的块大小是 L1 的 4 倍，那么 L2 中的一次缺失将替换相当于 4 个 L1 块的内容。下面考虑一个详细示例。

例题 假定 L2 的块大小为 L1 块的 4 倍。说明一次导致 L1 和 L2 产生替换的地址缺失将如何违犯包含特性。

解答 假定 L1 和 L2 是直接映射的，L1 的块大小为 b 个字节，L2 的块大小为 $4b$ 个字节。假定 L1 包含两个块，起始地址为 x 和 $x+b$，且 $x \bmod 4b = 0$，也就是说，x 也是 L2 中一个块的起始地址；因此，L2 中的单个块包含着 L1 块 x、$x+b$、$x+2b$ 和 $x+3b$。假定处理器生成一个对块 y 的引用，这个块对应于在两个缓存中都包含 x 的块，从而会产生缺失。由于 L2 产生缺失，所以它会提取 $4b$ 个字节，并替换包含 x、$x+b$、$x+2b$ 和 $x+3b$ 的块，而 L1 取得 b 个字节，并替换包含 x 的块。由于 L1 仍然包含 $x+b$，但 L2 不再包含，因此不再保持包含特性。

397

为了在采用多个块大小时仍然保持包含性，在较低级别完成替换时，必须上溯到层次结构的较高级别，以确保较低级别中替换的所有字在较高级别的缓存中都已失效；相联度的不同级别也会产生同类问题。2011 年，设计人员在实施包含性方面仍然存在分歧。Baer 和 Wang[1988] 详细描述了包含性的优势与挑战。Intel i7 为 L3 应用了包含性，也就是说 L3 总是包含 L2 和 L1 的内容。这样就可以在 L3 实施一种简单的目录机制，在最大程度上降低因为监听 L1 和 L2 而对这些情景造成的干扰，目录中指出 L1 或 L2 中含有一个缓存副本。AMD Opteron 与之相对，使 L2 包含 L1 的内容，但对 L3 没有这一限制。它们使用了监听协议，但除非存在命中情况，否则仅需要在 L2 进行监听，在这种情况下，会向 L1 发送监听。

5.7.4 利用多重处理和多线程的性能增益

这一节将考察两项研究，了解在多核处理器上使用多线程的有效性；在下一节还会讨论这

一主题，届时将研究Intel i7的性能。我们的这两项研究是以Sun T1（在第3章介绍）和IBM Power5处理器为基础的。

我们利用第3章研究过的三个面向服务器的基准测试来研究T1多核处理器的性能，这三个基准测试为TPC-C、SPECJBB（SPEC Java业务基准测试）和SPECWeb99。SPECWeb99基准测试仅在T1的一个四核版本上运行，因为它不能扩展到利用八核处理器的全部32个线程；另两个基准测试以八核心处理器运行，每个核心有4个线程，总共有32个线程。表5-9给出了八核心T1的每线程CPI和每核心CPI，以及实际CPI和每个时钟周期的指令数（IPC）。

表5-9　八核心Sun T1处理器的每线程CPI、每核心CPI、实际八核心CPI和实际IPC（CPI的倒数）

基准测试	每线程的CPI	每核心的CPI	八个核心的实际CPI	八个核心的实际IPC
TPC-C	7.2	1.8	0.225	4.4
SPECJBB	5.6	1.40	0.175	5.7
SPECWeb99	6.6	1.65	0.206	4.8

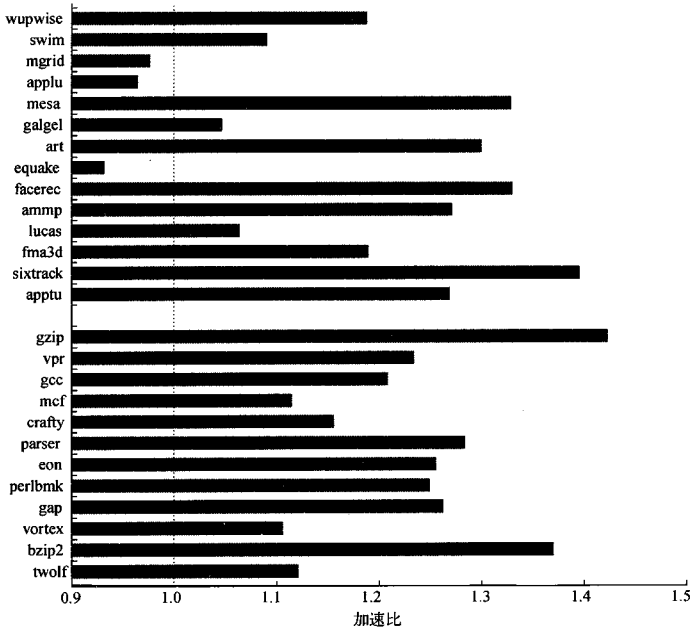


图5-17　在八处理器IBM eServer P5575上对比SMT和单线程（ST）性能。注意，x轴的起始加速比为0.9，表明有性能损失。每个Power5核心中仅有一个处理器是活动的，通过降低存储器的破坏性干扰，应当可以稍稍改善SMT的结果。SMT结果是通过创建16个用户线程获得的，而ST结果仅使用了8个线程；由于每个处理器仅有一个线程，所以操作系统将Power5切换为单线程模式。这些结果是由IBM公司的McCalpin收集的。从数据中可以看出，SPECfpRate结果的标准偏差略高于SPECintRate（0.13比0.07），表明浮点程序的SMT提升可能会有很大的变化范围

IBM Power5是一种支持同时多线程（SMT）的双核处理器。为了研究多线程在多处理器中的性能，我们对一个拥有8个Power 5处理器的IBM系统进行了测试，仅使用了每个处理器上的一个核心。图5-17给出了一个八处理器Power5多处理器在有、无SMT时执行SPECRate2000基准测试的加速比，见图题中所述。平均来说，SPECintRate的速度为1.23倍，而SPECfpRate的速度1.16倍。注意，一些浮点基准测试在SMT模式中的性能会稍有下降，加速比最多会降低0.93。尽管人们希望SMT可能会更好地隐藏SPECFP基准测试的高缺失率，但看起来，在以SMT模式运行这些基准测试时，会遇到存储器系统中的一些限制。

398 ~ 399

5.8 融会贯通：多核处理器及其性能

2011年，多核心成为所有新处理器的主旋律。各种实现方式的变化很大，它们对大型多芯片多处理器的支持也同样有很多不同。这一节研究4种不同多核处理器的设计和一些性能特征。

表5-10给出了4种为服务器应用设计的多核处理器。Intel Xeon的设计基础与i7相同，但它的核心更多、时钟频率稍慢（功率限制了其时钟频率）、L3缓存较大。AMD Opteron和桌面Phenom共享相同的基础核心，而SUN T2与在第3章遇到的SUN T1相关。Power7是Power5的扩展，核心更多，缓存更大。

表5-10 4种最近为服务器设计的高端多核处理器（2010年发布）的特征汇总

特　征	AMD Opteron 8439	IBM Power 7	Intel Xenon 7560	Sun T2
晶体管数	9.04亿	12亿	23亿	5亿
功率（标称）	137 W	140 W	130 W	95 W
每个芯片的最大核心数	6	8	8	8
多线程	无	SMT	SMT	细粒度
每个核心的线程数	1	4	2	8
每个时钟周期发射的指令数	一个线程发射3条	一个线程发射6条	一个线程发射4条	两个线程发射2条
时钟频率	2.8 GHz	4.1 GHz	2.7 GHz	1.6 GHz
最外层缓存	L3、6 MB、共享	L3、32 MB（采用嵌入DRAM）、共享或由各个核心专用	L3、24 MB、共享	L2、4 MB、共享
包含	无、尽管L2是L1的超集	有、L3超集	有、L3超集	有
多核一致性协议	MOESI	扩展MESI，具有行为和局域性暗示（13-状态协议）	MESIF	MOESI
多核一致性实现	监听式	设在L3的目录	设在L3的目录	设在L2的目录
对扩展一致性的支持	利用HyperTransport可以将最多8个处理器连接为一个环，采用目录式或监听式协议。系统为NUMA	使用SMP链接可以将最多32个处理器芯片连接起来。采用动态分布式目录结构。8核心芯片之外的存储器访问是对称的	通过Quickpath互连可以连接最多8个处理器核心。以外部逻辑支持目录	通过每个处理器的四个一致性链接实现，可用于进行监听。最多两个芯片直接相连，最多四个使用外部ASIC的连接

* 表中包含了这些处理器中核心数最多的版本；其中一些处理器还有核心数较低、时钟频率较高的版本。IBM Power7中的L3可以全部共享，也可以划分为各个核心专用的更快速专用区域。我们仅包含了这些多核心处理器的单芯片实现方式。

首先，我们将对比其中三种多核处理器在配置为多芯片多处理器时的性能和性能扩展性

（由于 AMD Opteron 的数据不足，所以省略了这一处理器）。

这三种多处理器对 ILP 和 TLP 的侧重点不尽相同，除此之外，它们的市场定位也有明显不同。因此，我们没有过多地关注绝对性能的对比，而将重点放在增加处理器时的性能扩展能力上。在研究这一数据之后，将会更详细地研究 Intel Core i7 的多核性能。

我们将展示三组基准测试的性能：SPECintRate、SPECfpRate 和 SPECjbb2005。我们将 SPECRate 基准测试聚集在一起，展示这些多处理器在请求级并行方面的性能，这种并行的特征就在于独立程序的并行与重叠执行。具体来说，除系统服务之外的所有内容都未被共享。SPECjbb2005 是一种可扩展 Java 业务基准测试，它对一种三层客户端/服务器系统进行建模，主要关注服务器端，类似于我们在第 1 章研究的 SPECPower 中所使用的基准测试。这些基准测试对 Java 虚拟机、即时编译器、垃圾收集、线程及操作系统其他方面的实现进行了测试；还测试了多处理器系统的可伸缩性。

图 5-18 给出了 SPECRate CPU 基准测试在核心数目增加时的性能变化。随着处理器芯片数及核心数的增加，可以获得近似线性的加速比。

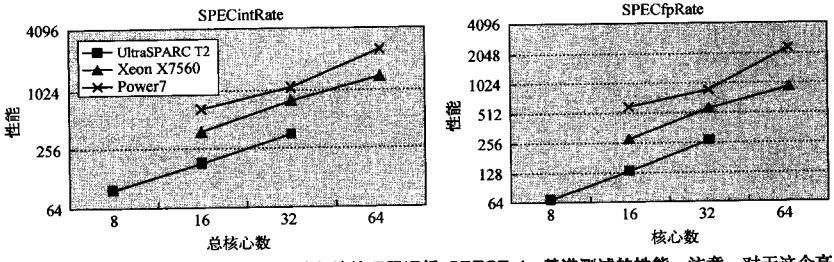


图 5-18　当处理器芯片数增大时，三种多核处理器运行 SPECRate 基准测试的性能。注意，对于这个高度并行的基准测试，得到了近似线性的加速比。这两个曲线都采用对数-对数刻度，所以线性加速比表现为一条直线

图 5-19 给出了 SPECjbb2005 基准测试的类似数据。要在开发更多 ILP 和仅关注 TLP 之间实现平衡是很复杂的，它与具体的工作负载高度相关。SPECjbb2005 工作负载能够在增加更多处理器时进行扩展，使运行时间（而非问题规模）保持恒定。在这种情况下，会有足够的并行，可以通过 64 个核心来实现线性加速比。我们将在结语部分再次讨论这一主题，但现在先让我们仔细地研究一下 Intel Core i7 在单芯片、四核心模式下的性能。

400

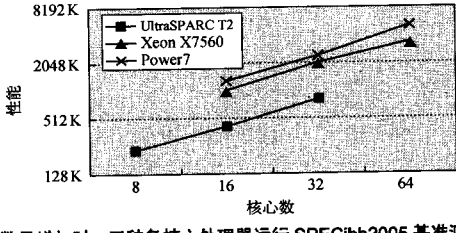


图 5-19　当处理器芯片数目增加时，三种多核心处理器运行 SPECjbb2005 基准测试的性能。注意，对于这一并行基准测试，得到了近似线性的加速比

Intel Core i7 多核的性能与能耗效率

在这一节，我们利用第 3 章考虑过的两组基准测试来研究 i7 的性能，即并行 Java 基准测试和并行 PARSEC 基准测试（在表 3-18 中有详细介绍）。我们首先来看一下在没有使用 SMT 时多核心性能、扩展能力与单核心的对比。然后将多核心和 SMT 功能结合起来。这一节的所有数据与前面 i7 SMT 评估中的数据（3.13 节）一样，都来自 Esmaeilzadeh 等人[2011]。数据集也与前面使用的相同（见表 3-18），只是去除了 Java 基准测试 tradebeans 和 pjbb2005（仅留下了 5 个可伸缩 Java 基准测试）；即使采用 4 个核心、总共 8 个线程，tradebeans 和 pjbb2006 的加速比也不会超过 1.5，因此不适于评估更多核心。 401

图 5-20 绘制了在没有使用 SMT 时 Java 和 PARSEC 基准测试的加速比和能量效率曲线。给出能耗效率曲线意味着我们绘制的是两核心或四核心运行消耗能量与单核心运行消耗能量的比值；因此，能耗效率越高越好，取值为 1.0 时为其平衡点。在所有情景中，没有使用的核心都处于尝试睡眠模式，基本上相当于将这些核心关闭，使其功耗降至最低。在对比单核心和多核心基准测试的数据时，一定要记住，在单核心（及多核心）情景中，L3 缓存和存储器接口的全部能耗成本都是物有所值的。因为这一事实，对于那些能够很好扩展的应用程序，有可能进一步改善其能耗指标。在汇总这些结果时使用了调和均值，其隐含意义见图题。 402

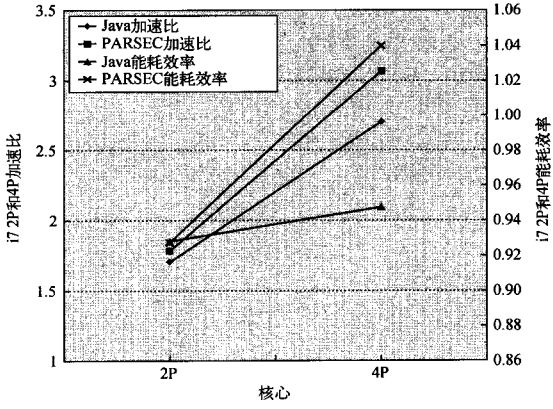


图 5-20 本图给出了未采用 SMT 时，两核和四核处理器执行并行 Java 与 PARSEC 工作负载时的加速比。这些数据由 Esmaeilzadeh 等人[2011]收集，使用的设置与第 3 章所述设置相同。Turbo Boost 功能被关闭。加速比与能耗效率数据使用调和均值汇总，其隐含含义就是在这种工作负载中，运行每个 2p 基准测试所花费的时间是等价的

如图 5-20 所示，PARSEC 基准测试的加速比要优于 Java 基准测试，在四核心处理器上的加速比效率为 76%（即实际加速比除以处理器数目），而 Java 基准测试在四核心处理器上的加速比效率为 67%。尽管从数据中可以很清楚地看出这一结果，但要分析存在这种差异的原因要麻烦一些。例如，很有可能是 Amdahl 定律降低了 Java 工作负载的加速比。此外，处理器体系结构与应用程序之间的交互也可能在其中产生影响（它会影响到同步成本或通信成本等问题）。具

体来说，并行化程度很高的应用程序（比如 PARSEC 中的程序）有时可能因为计算与通信之间的有利比值而获益，这种比值可以降低对通信成本的依赖性（见附录 I）。

这种加速比的差异性可以转换为能耗效率的差异性。例如，相对于单核心版本，PARSEC 基准测试实际上只是稍微提高了能耗效率；这一结果可能受到以下事实的显著影响：L3 缓存在多核运行版本中的使用效率要高于单核情景，而两种情景中的能耗成本是相同的。因此，对于 PARSEC 基准测试，多核方法达到了设计人员从关注 ILP 的设计转向多核设计的目的，即：其性能的增长速度不低于功率的增长速率，从而使能耗效率保持不变，甚至还有所提高。在 Java 情景中我们看到，由于 Java 工作负载的加速比级别较低，（尽管在 2p 运行中，Java 能耗效率与 PARSEC 相同！）所以两核和四核运行版本都没有达到能耗效率的平衡点。四核 Java 情景中的能耗效率相当高（0.94）。对于 PARSEC 或 Java 工作负载，以 ILP 为中心的处理器很可能需要更多的功率才能实现相似的加速比。因此，在提高这些应用程序的性能方面，以 TLP 为中心的方法当然也会优于以 ILP 为中心的方法。

403

将多核与SMT结合起来

最后，我们通过测量两组基准测试在 2～4 个处理器、1～2 个线程（总共 4 个数据点、最多 8 个线程）情况下的结果，来研究多核与多线程的组合方式。图 5-21 给出了在处理器数目为 2 或 4、使用和未使用 SMT 时，在 Intel i7 上获得的加速比和能耗效率，采用调和均值来汇总两组基准测试的结果。显然，如果在多核情景下也有足够的线程级并行，SMT 是可以提高性能的。例如，在四核无 SMT 情景中，Java 和 PARSEC 的加速比效率分别为 67%和 76%。在采用 SMT、四个核心时，这些比值达到了令人惊讶的 83%和 97%！

404

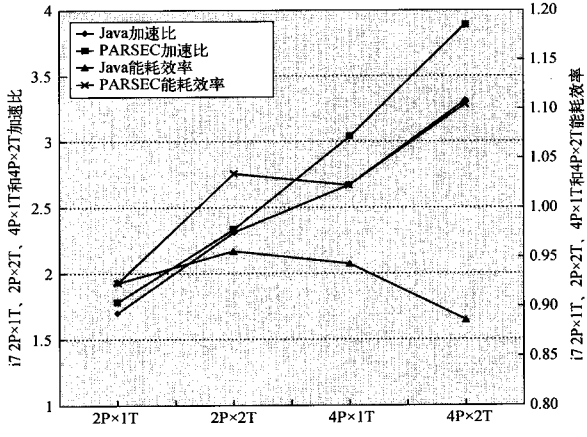


图 5-21 本图给出了在有、无 SMT 时，以两核和四核处理器执行并行 Java 和 PARSEC 工作负载的加速比。注意，以上结果是在线程数由 2 变为 8 时获得的，反映了体系结构的影响和应用程序的特征。汇总结果时采用了调和均值，如图 5-20 的图题所述

能耗效率给出了一幅稍有不同的画面。对于 PARSEC，加速比在四核 SMT 情景中（8 个线

程）基本上为线性，功率的增长要更慢一些，从而使这种情景中的能耗效率达到 1.1。Java 情景要更复杂一些；两核心 SMT（四线程）运行时的能耗效率峰值达到 0.97，在四核心 SMT（8 线程）运行时下降到 0.89。在部署 4 个以上的线程时，Java 基准测试非常有可能遭遇 Amdahl 定律效应。一些架构师已经观察到，多核处理器将提高性能（从而提高能耗效率）的更多责任转嫁给程序员，Java 工作负载的结果显然证实了这一点。

5.9 谬论与易犯错误

由于对并行计算的理解不够成熟，所以存在大量的隐藏易犯错误，这些错误要么会被一些细心的设计人员发现，要么会被一些倒霉的设计人员碰上。由于近年来围绕多处理器进行了大量夸大宣传，所以也存在着很多常见谬论。我们选择其中一些列出如下。

易犯错误 通过随执行时间线性变化的加速比来测量多处理器的性能。

“迫击炮射击”曲线长期用于判断并行处理器的成功与否。（这种曲线显示了线性加速、稳定和最后的下降过程。）尽管加速比是并行程序的一个方面，但它并不是性能的直接度量。第一个问题是所扩展处理器的功率：一个程序的性能线性提高到相当于 100 个 Intel Atom 处理器（在上网本上使用的低端处理器），它的速度可能慢于在一个八核 Xeon 上运行的版本。对于浮点计算密集的程序一定要尤其小心，没有硬件辅助的处理元件也许能够很好地扩展，但整体性能却可能很差。

只有在对比每种计算机上的最佳算法时，对比执行时间才是公平的。在两个计算机上对比相同代码，看起来可能是公平的，但实际并非如此；并行程序在单处理器上的运行速度可能比顺序执行版本要慢一些。开发并行程序有时可能会导致算法方面的改进，所以将过去人们熟知的顺序程序与并行代码进行对比（这看起来是公平的）时，比较的并不是等价算法。为了反映这一问题，有时会使用**相对加速比**（同一程序）和**真实加速比**（最佳程序）等术语。 405

如果结果中呈现**超线性**性能，也就是说一个程序在 n 个处理器上运行时，其速度要比在等同单处理器上的运行速度快 n 倍以上，那就表明这种对比可能是不公平的，尽管在某些情况下已经遇到了“真实的”超线性加速比。例如，一些科学应用程序通常会在小幅增加处理器数目时（2 或 4 增加到 8 或 16），实现超线性加速比。这些结果的出现通常是因为一些关键性的数据结构无法放入拥有 2 个或 4 个处理器的多处理器中的聚合缓存，但却可以放入拥有 8 个或 16 个处理器的多处理器的聚合缓存中。

总而言之，要通过对比加速比来对比性能，运气好时也需要很强的技巧性，运气不好时可能会造成误导。对比两种不同多处理器的加速比并不一定能够告诉我们有关这些多处理器相对性能的相关信息。甚至在相同处理器上对比两种不同算法也需要一些技巧，这是因为我们必须使用真正的加速比而不是相对加速比来获得有效的对比结果。

谬论 Amdahl定律不适用于并行计算机。

1987 年，某个研究组织的领导人宣布，Amdahl 定律（见 1.9 节）已经被 MIMD 多处理器打破。但是，这一声明并不意味着这一定律已经被并行计算机推翻；程序中一些被忽略的部分仍然会对性能产生限制。为了理解这些媒体报道的基础，让我们看看 Amdahl[1967]最初是怎么说的。

> 在这里，我们可以得出一个相当明确的结论：如果不能以近乎相同的幅度来提高串行处理速率，那在提高并行处理速率方面所做的努力都是徒劳无功的。[P483]

对这一定律的一种解释就是：由于每个程序都有串行执行的部分，所以对于经济合理的处理器数目会有一个上限，比如说100个。如果在使用1000个处理器时仍然呈现线性加速比，那就证明对Amdahl定律的这一解释是错误的。

"Amdahl定律已被'推翻'"这一表述的基础就是使用**扩展加速比**（scaled speedup），也称为**弱扩展**（weak scaling）。研究人员对基准测试进行了扩展，使数据大小增大到1000倍，并对比了扩展后基准测试的单处理器与并行执行时间。对于这一特定算法，程序的顺序执行部分恒定，与输入的大小无关，而其余部分是完全并行的，因此在采用1000个处理器时的加速比是线性的。因为运行时间的增长速度要长于线性增长速率，所以这个程序在扩展之后的运行时间要长一些，即使采用1000个处理器也是如此。

在对输入进行扩展的情况下测得的加速比不同于真正的加速比，将其当作真正加速比会造成误导。由于并行基准测试经常在不同规模的多处理器上运行，所以明确指出允许何种应用程序扩展以及如何完成扩展是很重要的。让数据规模随处理器数目的增大而扩展，在大多数情况下并不恰当，但是当处理器数目大量增加时（称为**强扩展**），如果继续采用固定规模的问题，通常也是不恰当的，这是因为当我们向用户提供一个大得多的多处理器时，他们通常会选择运行应用程序的更大、更详细版本。关于这一重要主题的更多讨论，请参阅附录I。

406

谬论　需要线性加速比才能提高多处理器的成本效率。

人们普遍认同，并行计算的主要优点之一是：即使与最快速的单处理器相比，也能在更短的时间内给出计算结果。但是，许多人也持有这样的观点：并行处理器不可能实现与单处理器一样的成本效率，除非它们能够实现完美的线性加速比。这种观点认为，由于多处理器的成本是处理器数目的线性函数，所以只要低于线性加速比，就意味着性能/价格比下降，使并行处理器的成本效率低于使用单处理器的情况。

这种观点的问题在于：成本不仅是处理器数目的函数，也依赖于系统的存储器、I/O和开销（机箱、电源、互连，等等）。在多核时代里，每个芯片上有多个处理器，这一观点就更没有什么意义了。

在系统成本中包含存储器的影响是由Wood和Hill[1995]指出的。我们使用的例子将以最近使用TPC-C和SPECRate基准测试获得的数据为基础，但利用并行科学应用程序工作负载也可以得出同一结论，甚至可以更强烈地表达这一观点。

图5-22给出了TPC-C、SPECintRate和SPECfpRate在IBM eServer p5多处理器上的加速比，此多处理器配有4～64个处理器。图中显示，只有TPC-C获得了优于线性加速比的结果。对于SPECintRate和SPECfpRate，加速比低于线性加速比，成本也是如此，这是因为它们与TPC-C不同，主存储器和所需磁盘数目的扩展也都低于线性扩展。

如图5-23所示，与四处理器配置相比，更多的处理器数目实际上可能更具成本效率。在对比两个计算机的性价比时，必须确保准确评估了总系统成本和可以达到的性能。对于许多具有更高存储器需求的应用程序，这种对比可能极大地增加了使用多处理器的吸引力。

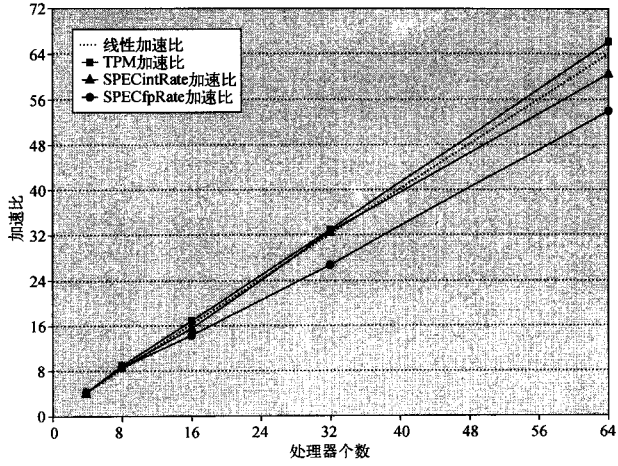


图 5-22 三种基准测试在 IBM eServer p5 多处理器上的加速比，这个多处理器分别配有 4、8、16、32 和 64 个处理器。虚线表示线性加速比

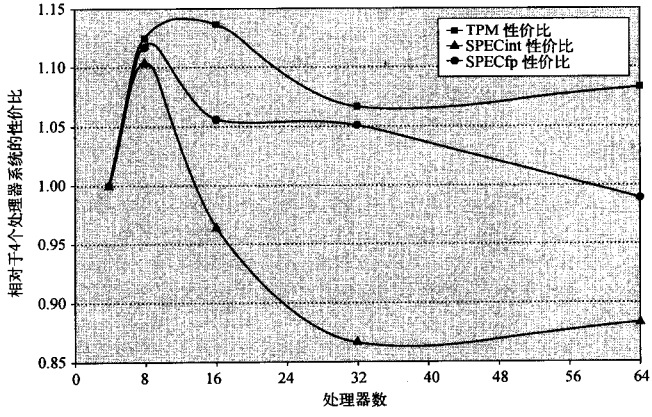


图 5-23 三个基准测试在包含 4～64 个处理器的 IBM eServer p5 多处理器上执行时，其相对于四处理器系统的性能价格比表明：处理器数据较大时，也可能实现与四处理器配置一样的成本效率。对于 TPC-C，这一配置与官方运行时使用的配置相同，这就意味着磁盘与存储器随处理器数目线性扩展，包含 64 个处理器的机器，其成本大约是 32 处理器版本的两倍。与之相对，磁盘和存储器的扩展速度更慢一些（不过，仍然快于在 64 个处理器上实现最佳 SPECRate 所需要的速度）。具体来说，磁盘配置从 4 处理器版本的一个驱动器扩展为 64 处理器版本的四个驱动器（140 GB）。存储器从 4 处理器版本的 8 GB 发展到 64 处理器系统的 20 GB

易犯错误　不要开发利用多处理器体系结构或针对此种结构进行优化的软件。

软件开发长期滞后于多处理的发展，可能是因为软件问题要困难得多。我们给出一个例子来说明这些问题的微妙之处，但实际有许多例子可供我们选择。

在将一个为单处理器设计的软件应用于多处理器环境时，经常会遇到一个问题。例如，2000年的SGI操作系统最初是用单个锁来保护页表数据结构的，当时认为页面分配的出现频率较低。在单处理器中，这种做法并不会带来性能问题。但在多处理器中，可能会成为某些程序的主要性能瓶颈。考虑一个程序，它使用了大量在启动时初始化的页面，UNIX对于静态分配页面就是这样做的。假定实现了这个程序的并行化，由多个进程来分配这些页面。由于页面分配需要使用页表数据结构，而这种结构只要处于使用状态就会被锁定，如果这些进程都试图同时分配它们的页面（这正是我们希望在初始化时做的工作！），即使是允许在操作系统中存在多个线程的操作系统内核也会被串行化。 407

这种页表串行化清除了初始化过程中的并行，对整体并行性能有非常严重的影响。这一性能瓶颈即使在多重编程时也仍然存在。例如，假定我们将并行程序分散到独立进程中，然后在每个处理器上运行一个进程，从而在进程之间不存在共享。（这就是一位用户所做的工作，因为他合情合理地相信这一性能问题是由于其应用程序中的意外共享或干扰造成的。）遗憾的是，那个唯一锁仍然会使所有进程串行化，所以即便是多重编程性能也非常糟糕。这个易犯错误指明，在多处理器上运行软件时，可能会出现一些虽很微小但却影响巨大的性能缺陷。和所有其他关键的软件组件一样，操作系统算法和数据结构在多处理器上下文中也都必须进行重新考虑。在页表的较小部分上设置锁，可以有效地消除这一问题。在存储器结构中也存在类似问题，在没有实际发生共享时，会增大一致性通信流量。 408

由于多核心处理器已经成为从桌面计算机到服务器等各个领域的主旋律，所以并行软件中的投入不足已经变得非常明显。由于重视不够，所以可能还要等待多年之后，我们使用的软件系统才能充分利用这些不断增加的核心。

5.10　结语

30多年来，研究人员和设计人员一直在预测单处理器会终结，会被多处理器超越。但直到本世纪的前几年，这一预测还总被证实是错误的。在第3章已经看到，尝试寻找和利用更多ILP的成本在效率上是难以承受的（在硅面积和功率方面都是如此）。当然，多核并没有解决功率问题，因为它显然增加了晶体管数和晶体管开关的活动数目，而这正是功率消耗的两个主要因素。 409

但是，多核的确改变了这场游戏。因为允许将空闲核心置于功率节省模式，所以也可以在功率效率方面进行一些改进，本章的结论已经证明了这一点。更重要的是，多核技术将保持处理器繁忙的重担更多地交给了TLP，而不再是依靠ILP，TLP由应用程序和程序员负责确认，而ILP则由硬件负责。我们已经看到，这些差别显然在Java与PARSEC基准测试的多核性能与能耗效率方面进一步扩大。

尽管多核技术为克服能耗效率方面的问题提供了一些直接帮助，并将大部分重担移交给软件系统，但仍然存在一些难度很大的挑战和尚未解决的问题。例如，人们也在尝试利用积极推测的线程级版本，但到目前为止，这些努力的结果与尝试利用ILP的努力面临着相同的命运。也就是说，性能有所提高，但不是特别明显，可能低于能耗的增大幅度，所以诸如推测线程或

硬件先行（run-ahead）之类的思想都没有成功地整合到处理器中。和 ILP 的推测一样，除非推测结果总是正确的，否则其成本就会超过其收益。

除了编程语言和编译器技术的重要问题之外，多核技术已经重新开放了计算机体系结构中另一个长期存在的问题：是否值得考虑异构处理器？尽管现在还没有提交这种多核处理器，而且异构多处理器仅在专用计算机或嵌入式系统中取得了有限的成功，但它在多核心环境中的可能性要高得多。和多重处理中的许多问题一样，其答案可能依赖于软件模型和编程系统。如果编译器和操作系统可以有效地使用同构处理器，它们将会变得更加主流。目前，对于许多应用程序来说，现有编译器的能力还不足以有效地应对中等数量的同构核心所带来的压力，但有一类拥有异构核心的多处理器变得越来越常见，它们在功能方面有明显不同，而且有一些用于分解应用程序的明确方法。这些多处理器包括诸如 GPU 和媒体处理器之类的专用处理单元。对能耗效率的重视还可能促使多处理器中包含一些具有不同性能功率比的核心。

在本书的 1995 年版本中，我们在这一章的末尾讨论了当时极具争议的两个问题。

(1) 基于微处理器的超大规模多处理器将会使用哪种体系结构？

(2) 多重处理在未来的微处理器体系结构中扮演何种角色？

这些年来，这两个问题已经大体得到解决。

410

因为超大规模多处理器没有变成一个不断发展的主流市场，所以目前构建这种大规模多处理器的唯一具有成本效率的方法就是使用集群，在集群中，各个节点或者是单个多核微处理器，或者是小规模、共享存储器的多处理器（通常有 2～4 个多核处理器），互连网络采用标准网络技术。这些集群已经扩展到数万个处理器，并且安装在专门设计的“仓库”计算机中，下一章将讨论这一主题。

第二个问题的答案在最近六七年前已经变得非常清晰：多处理器未来性能的增长将源于多核处理器对线程级并行的开发，而不是通过开发更多的 ILP。

因此，核心已经成为芯片的新构建模块，供应商提供了各种不同的芯片，它们以单核设计为基础，采用了不同数目的核心和 L3 缓存。例如，表 5-11 给出了仅使用 Nehalem 核心（用于 Xeon 7560 和 i7 中）构建的 Intel 处理器系列。

表5-11 各种基于Nehalem微体系结构的Intel处理器的特性参数

处理器	系列	核心	L3缓存	功率（典型）	时钟频率（GHz）	价格（美元）
Xeon	7500	8	18～24 MB	130 W	2～2.3	2837～3692
Xeon	5600	4～6（有、无SMT）	12 MB	40～130 W	1.86～3.33	440～1663
Xeon	3400～3500	4（有、无SMT）	8 MB	45～130 W	1.86～3.3	189～999
Xeon	5500	2～4	4～8 MB	80～130 W	1.86～3.3	80～1600
i7	860～975	4	8 MB	82～130 W	2.53～3.33	284～999
i7 mobile	720～970	4	6～8 MB	45～55 W	1.6～2.1	364～378
i5	750～760	4（无SMT）	8 MB	80 W	2.4～2.8	196～209
i3	330～350	2（有、无SMT）	3 MB	35 W	2.1～2.3	

* 本表仍然在每一行中包含了许多项（2～8 项）。其中所列价格是每单达到 1000 个时的单价。

在 20 世纪 80 年代和 90 年代，随着 ILP 的诞生与发展，一些软件通过优化编译器而开发 ILP，成为其成功的关键。与此类似，要想成功开发线程级并行，也要依赖于适当软件系统的发展，就像依赖于计算机体系结构一样。在过去的 30 多年里，并行软件发展迟缓，所以在未来几年里，

411 广泛开发线程级并行仍然富有挑战性。此外，作者相信，出现最佳多核体系结构的机会非常大。为了设计此类体系结构，架构师需要定量设计训练，并能够准确地为数千个运行数万亿条指令的核心建立模型，包括大规模应用程序和操作系统。如果没有这种能力和方法，架构师就像是在黑暗中射击。有时可能会碰上好运，但更常见的还是错过目标。

5.11 历史回顾与参考文献

附录 L.7 节考察了多处理器和并行处理的历史。根据时间段和体系结构的划分，附录 L.7 节讨论了早期的试验性多处理器和在并行处理中的一些著名争论。其中也介绍了最近的进展，并给出了供扩展阅读的参考文献。

案例研究与练习（Amr Zaky 和 David A.Wood 设计）

案例研究 1：单芯片多核多处理器

本案例研究说明的概念

- 监听式一致性协议转换
- 一致性协议的性能
- 一致性协议的优化
- 同步
- 存储器连贯性模型的性能

图 5-24 所示的简单多核多处理器给出了一种经常实现的对称共享存储器体系结构。每个处理器拥有单个专用缓存，使用图 5-4 的监听式一致性协议来保持一致性。每个缓存都是直接映射的，共有四个块，每个块保存两个字。为了简化说明，缓存地址标签中包含了完整的地址，每个字仅显示两个十六进制字符，最低有效字位于右侧。一致性状态表示为 M、S 和 I（已修改、共享和无效）。

5.1 [10/10/10/10/10/10/10] <5.2>对于这一练习的每一部分，假定初始缓存与存储器状态如图 5-24 所示。这个练习的每一部分以如下形式指定由一或多个 CPU 操作组成的序列：

```
P#: <op> <address> [<value>]
```

其中，P#指明 CPU（例如，P0）、<op>是 CPU 操作（例如，读取或写入）、<address>表示存储器地址、<value>指示在写入操作时指定的新字。将以下每个操作看作是独立应用于图 5-24 给定的初始状态。在给定操作之后，缓存和存储器的结果状态是怎样的（即，一致性状态、标签和数据）？仅给出发生变化的块，例如，P0.B0: (I,120,00 01)表示 CPU P0 的 B0 块最终状态为 I，标签为 120，数据字为 00 和 01。另外，每个读取操作返回什么样的值？

412

a. [10] <5.2> P0: read 120
b. [10] <5.2> P0: write 120 <-- 80
c. [10] <5.2> P3: write 120 <-- 80
d. [10] <5.2> P1: read 110
e. [10] <5.2> P0: write 108 <-- 48
f. [10] <5.2> P0: write 130 <-- 78
g. [10] <5.2> P3: write 130 <-- 78

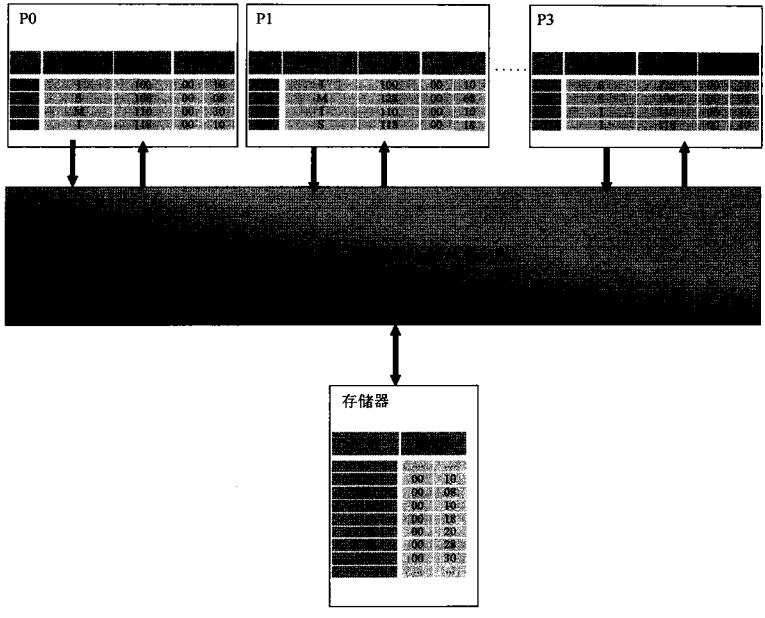


图 5-24 多核（点对点）多处理器

5.2 [20/20/20/20] <5.3>监听式缓存一致性多处理器的性能取决于许多具体的实施问题，它们决定了缓存能够以何种速度作出回应，提供处于独占或 M 状态的块中数据。在一些实施方式中，当一个缓存块在另一个处理器的缓存中处于独占状态时，对这个块的 CPU 读取缺失要快于存储器中一个块的缺失。这是因为缓存要小于主存储器，所以速度也就更快一些。与之相反，在某些实施中，由存储器提供数据的缺失要快于由缓存提供数据的缺失，这是因为缓存通常是针对“前端”或 CPU 引用进行优化的，而不是针对“后端”或监听式访问进行优化的。对于图 5-24 所示的多处理器，考虑在单个 CPU 上执行一系列操作，其中：

413

- CPU 读取和写入命中不会产生停顿时钟周期；
- CPU 读取和写入缺失在分别由存储器和缓存提供数据时，生成 $N_{存储器}$和 $N_{缓存}$个停顿周期；
- 生成失效操作的 CPU 写入命中导致 $N_{失效}$个停顿周期；
- 由于冲突或另一个处理器请求独占块而造成写回块时，会另外增加 $N_{写回}$个停顿周期。

考虑两种实现方式，它们的性能参数不同，汇总于表 5-12 中。考虑以下操作序列，假定其初始缓存状态如图 5-24 所示。为简便起见，假定第二个操作在第一个操作完成之后开始（即使它们由不同处理器执行时也是如此）：

```
P1: read 110
P3: read 110
```

对于实现方式 1，由于第一次读取是由 P0 的缓存提供数据，所以它产生 50 个停顿周期。P1 在等待这个块时停顿 40 个周期，P0 在回应 P1 的请求将其写回存储器时，停顿 10 个周期。之后，P3 的第二次读取生成 100 个停顿周期，因为它的缺失是由存储器提供数据，这个序列总

共生成150个停顿周期。对于以下操作序列，每个实现方式生成多少个停顿周期？

表5-12　监听一致性延迟

参　数	实现方式1	实现方式2
$N_{存储器}$	100	100
$N_{缓存}$	40	130
$N_{失效}$	15	15
$N_{写回}$	10	10

414

a. [20] <5.3>
```
P0: read 120
P0: read 128
P0: read 130
```

b. [20] <5.3>
```
P0: read 100
P0: write 108 <-- 48
P0: write 130 <-- 78
```

c. [20] <5.3>
```
P1: read 120
P1: read 128
P1: read 130
```

d. [20] <5.3>
```
P1: read 100
P1: write 108 <-- 48
P1: write 130 <-- 78
```

5.3　[20] <5.2>许多监听一致性协议拥有更多的状态、状态转换或总线事务，以减少保持缓存一致性的开销。在练习5.2的实现方式1中，当缺失数据由缓存提供时，缺失导致的停顿周期要少于由存储器提供数据时的停顿周期。一些一致性协议尝试通过提高这一情况的出现频率来提高性能。一种常见的协议优化方法是引入**被拥有状态**（通常表示为O）。在那些可能仅读取“被拥有”块的节点中，“被拥有”状态的表现类似于“共享”状态，但在某些节点中，必须在其他节点发生对“被拥有”块的读取和写入缺失时提供数据，“被拥有”状态的表现类似于“已修改”状态。对处于“已修改”状态或“被拥有”状态的块发生读取缺失时，将向发出请求的块提供数据，并转换为“被拥有”状态。对处于“已修改”或“被拥有”状态的块发生写入缺失时，将向发出请求的节点提供数据，并转换为“失效”状态。仅当某个节点替换了处于“已修改”或“被拥有”状态的块时，这种经过优化的MOSI协议才会更新存储器。画出带有附加状态和转换的新协议图。

5.4　[20/20/20/20] <5.2>对于以下代码序列及表5-12中两种实现方式的定时参数，计算基本MSI协议和练习5.3中优化MOSI协议的总停顿周期。假定不需要总线事务的状态转换不会导致额外的停顿周期。

a. [20] <5.2>
```
P0: read 110
P3: read 110
P0: read 110
```

b. [20] <5.2>
```
P1: read 120
P3: read 120
P0: read 120
```

c. [20] <5.2>
```
P0: write 120 <-- 80
P3: read 120
P0: read 120
```

d. [20] <5.2>
```
P0: write 108 <-- 88
P3: read 108
P0: write 108 <-- 98
```

415

5.5　[20] <5.2>一些应用程序首先读取大型数据集，然后修改其中的大多数或全部数据。基本MSI一致性协议将提供所有处于共享状态的缓存块，然后被迫执行失效操作，将它们升级为“已修

改”状态。额外的延迟会对一些工作负载产生严重影响。通过添加一项协议优化，不再需要对那些由同一处理器先读后写的数据块进行更新。这一优化向协议中添加了“独占”（E）状态，表示所有其他节点都没有这个块的副本，但它还没有被修改。当存储器为读取缺失提供数据，而且其他节点都没有有效副本时，缓存块即进入“独占”状态。CPU 对这个块的读取和写入会继续进行，但没有进一步的总线通信流量，但 CPU 写入操作会将一致性状态转换为“已修改”。“独占”状态不同于“已修改”状态，节点可能会悄无声息地替换“独占”块（而“已修改”块则必须写回存储器）。另外，对“独占”块的读取缺失会导致该块转换为“共享”状态，但不会要求节点以数据作出回应（因为存储器中拥有最新副本）。绘制 MESI 协议的新协议图，在其中添加“独占”状态，以及向基本 MIS 协议的“已修改”、“共享”和“无效”状态的转换。

5.6 [20/20/20/20/20] <5.2>假定图 5-24 的缓存内容和表 5-12 中实现方式 1 的定时参数。以下代码序列在基本协议和练习 5.5 的新 MESI 协议中的总停顿周期为多少？假定不需要互连事务的状态转换不会导致额外停顿周期。

a. [20] <5.2>
```
P0: read 100
P0: write 100 <-- 40
```

b. [20] <5.2>
```
P0: read 120
P0: write 120 <-- 60
```

c. [20] <5.2>
```
P0: read 100
P0: read 120
```

d. [20] <5.2>
```
P0: read 100
P1: write 100 <-- 60
```

e. [20] <5.2>
```
P0: read 100
P0: write 100 <-- 60
P1: write 100 <-- 40
```

5.7 [20/20/20/20] <5.5>在大多数商用共享存储器机器中，自旋锁可能是最简单的同步机制。这个自旋锁依靠交换原语来自动载入旧值和存储新值。锁定例程重复执行此交换操作，直到它发现未锁定的锁为止（即返回值为 0）：

416

```
DADDUI R2,R0,#1
lockit:    EXCH R2,0(R1)
           BNEZ R2, lockit
```

要解决一个自旋锁，只需要存储数值 0 即可：

```
unlock: SW R0,0(R1)
```

如 5.5 节中的讨论，经过更多优化的自旋锁利用缓存一致性，并使用载入操作来检查这个锁，允许它以缓存中的共享变量进行自旋：

```
lockit:    LD     R2, 0(R1)
           BNEZ   R2, lockit
           DADDUI R2,R0,#1
           EXCH   R2,0(R1)
           BNEZ   R2, lockit
```

假定处理器 P0、P1 和 P3 都尝试获取位于地址 0x100 的一个锁（即：寄存器 R1 保存着数值 0x100）。假定缓存内容如图 5-24 所示，定时参数如表 5-12 中的实现方式 1 所示。为简便起见，假定关键部分的长度为 1000 个时钟周期。

a. [20] <5.5>使用简单自旋锁，判断每个处理器在获取该锁之前大约导致多少个存储器停顿周期。

b. [20] <5.5>使用优化自旋锁，判断每个处理器在获取该锁之前大约导致多少个存储器停顿周期。

c. [20] <5.5>使用简单自旋锁，大约导致多少个互连事务？

d. [20] <5.5>使用“测试、测试并置位”自旋锁，大约导致多少个互连事务？

5.8 [20/20/20/20] <5.6>顺序一致性（SC）要求所有读取和写入都是按某一总体顺序执行的。这就需要处理器在某些特定情况下，在提交读取或写入指令时停顿下来。考虑以下代码序列：

```
write A
read B
```

其中 write A 导致一次缓存缺失，read B 导致一次缓存命中。根据 SC，处理器必须暂停 read B，直到它可以排定 write A（从而可以执行该操作为止）。SC 的简单实现将使处理器停顿，直到缓存接收到数据，并可以执行写入操作为止。较弱的一致性模型放松了对读取和写入的排序约束条件，减少了处理器必须停顿的情况。总体存储顺序（TSO）一致性模型要求所有写入操作都按某一总体顺序执行，但允许处理器的读取操作越过自己的写入操作。这就允许处理器实现写缓冲区，其中包含已经提交的写入操作，但这些写入操作还没有针对其他处理器的写入操作进行排序。在 TSO 中允许读取操作跳过写缓冲区（这在 SC 中是不允许的。）假定每个时钟周期可以执行一次存储器操作，而且那些在缓存中命名或者可以由写入缓冲区提供数据的操作不会导致停顿周期。未能命中的操作将导致表 5-12 所列的延迟。假定有图 5-24 所列的缓存内容。

417 对于 SC 和 TSO 一致性模型，在每个操作之前有多少个停顿周期？

```
a. [20] <5.6>   P0: write 110 <-- 80
                P0: read 108
b. [20] <5.6>   P0: write 100 <-- 80
                P0: read 108
c. [20] <5.6>   P0: write 110 <-- 80
                P0: write 100 <-- 90
d. [20] <5.6>   P0: write 100 <-- 80
                P0: write 110 <-- 90
```

案例研究 2：简单的目录式一致性

本案例研究说明的概念

- 目录式一致性协议转换
- 一致性协议的性能
- 一致性协议的优化

考虑图 5-25 所示的分布式共享存储器系统。它由两个四核芯片组成。每个芯片中的处理器共享 L2 缓存（L2$），两个芯片通过一个点对点互连连接在一起。系统存储器分散在两个芯片上。图 5-26 放大了这个系统的一部分。Pi,j 表示芯片 j 上的处理器 i。每个处理器有单个直接映射的 L1 缓存，其中包括两个块，每个块包括两个字。每个芯片有单个直接映射的 L2 缓存，其中包括两个块，每个块包括两个字。为了简化图示，缓存地址标签包含完整地址，每个字仅给出两个十六进制字符，最低有效字在右侧。L1 缓存状态用 M、S 和 I 标识，分别表示“已修改”、“共享”和“失效”。L2 缓存和存储器都有目录。目录状态用 DM、DS 和 DI 标识，分别表示“已修改目录”、“共享目录”和“失效目录”。这一简单目录式协议在图 5-15 和图 5-16 中描述。L2 目录列出了本地共享者/拥有者，如果某一行在另一芯片中进行外部共享，则还会增加一些记录；例如，P1,0:E 表示某一行由本地处理器 P1,0 共享，在某一其他芯片上进行外部共享。存储器目录中有一个清单，列出了某一行的芯片共享者/拥有者；例如 C0,C1 表示某一行在芯片 0 和芯片 1 中共享。

5.9 [10/10/10/10/15/15/15/15] <5.4>对于本练习的每一部分，假定初始缓存状态与存储器状态如图 5-26 所示。本练习的每一部分指定了一或多个 CPU 操作组成的序列，其形式如下：

418

```
P#: <op> <address> [ <-- <value> ]
```

其中，P#指明 CPU（例如，P0,0）、<op>是 CPU 操作（例如，读取或写入）、<address>表示存储器地址、<value>指示在写入操作时指定的新字。在给定的 CPU 操作序列完成之后，缓存和存储器的最终状态如何（即，一致性状态、共享者/拥有者、标签和数据）？还有，每个读取操

419 作返回什么值？

```
a. [10] <5.4>   P0,0: read 100
```

b. [10] <5.4> P0,0: read 128

c. [10] <5.4> P0,0: write 128 <-- 78

d. [10] <5.4> P0,0: read 120

e. [15] <5.4> P0,0: read 120
P1,0: read 120

f. [15] <5.4> P0,0: read 120
P1,0: write 120 <-- 80

g. [15] <5.4> P0,0: write 120 <-- 80
P1,0: read 120

h. [15] <5.4> P0,0: write 120 <-- 80
P1,0: write 120 <-- 90

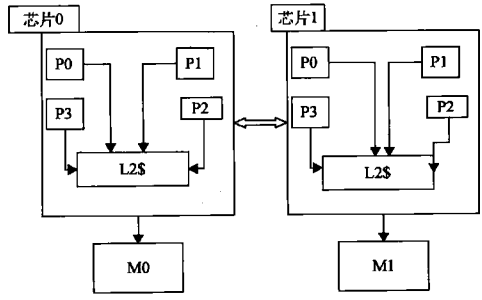


图 5-25 带有 DSM 的多芯片、多核多处理器

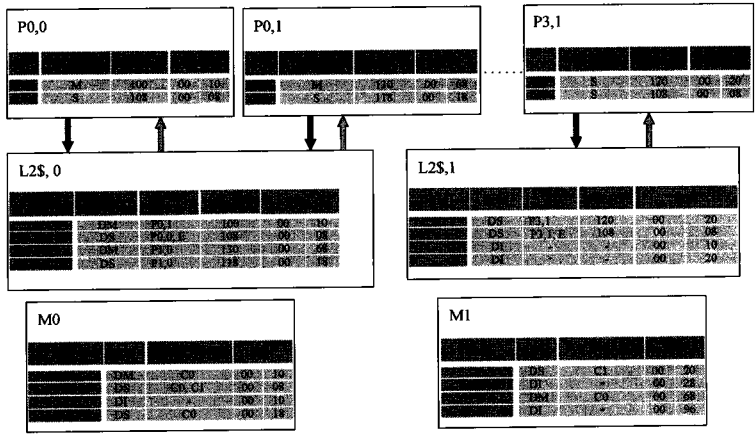


图 5-26 多芯片多核处理器中的缓存与存储器状态

5.10 [10/10/10/10] <5.4>目录式协议比监听式协议的可扩展性更强，因为它们会向那些拥有块副本的节点发送显式请求和失效消息，而监听式协议则向所有节点广播所有请求和失效消息。考虑图 5-25 所示的八处理器系统，假定所有未显示缓存拥有失效块。对于下面的每个序列，确认哪些

节点（芯片/处理器）接收每个请求和失效消息。

a. [10] <5.4>　P0,0: write 100 <-- 80

b. [10] <5.4>　P0,0: write 108 <-- 88

c. [10] <5.4>　P0,0: write 118 <-- 90

d. [10] <5.4>　P1,0: write 128 <-- 98

5.11 [25] <5.4>练习5.3要求向简单的MSI监听协议中添加了“被拥有”状态。重复这一问题，但采用以上简单的目录协议。

5.12 [25] <5.4>试讨论为什么利用简单目录协议添加“独占”状态要比采用监听协议时困难得多。举出一个例子。

案例研究3：高级目录式协议

本案例研究说明的概念

- 目录式一致性协议实施方式
- 一致性协议的性能
- 一致性协议的优化

案例研究2中的目录一致性协议抽象地描述了目录式一致性，但假定采用与简单监听系统非常一致的原子转换。高性能目录系统采用流水化、交换互连，极大地提高了带宽，但也引入了过渡状态和非原子化转换。目录式缓存一致性协议的可扩展性高于监听式缓存一致性协议，原因有两个。第一，监听式缓存一致性协议向所有节点广播请求，限制了其可伸缩性。目录式协议使用一个间接层级（向目录中发送的消息）来确保仅向拥有块副本的节点发送这些请求。第二，监听式系统的地址网络必须按总体顺序提供请求，而目录式协议可以放松这一约束条件。一些目录式协议假定没有网络排序，这样就允许采用自适应路由技术来提高网络带宽，所以是有益的。其他协议依靠点对点排序（即，从节点P0发到节点P1的消息将按顺序到达）。即使有这样一个排序约束条件，目录式协议的过渡状态通常也要多于监听式协议。表5-13给出了一种简化目录式协议的缓存控制器状态转换，这种协议依赖于点对点网络排序。表5-14给出了目录控制器的状态转换。

420

表5-13　广播监听式缓存控制器转换

状态	读取	写入	替换	失效	Forwarded _GetS	Forwarded _GetM	PutM_ Ack	Data	Last Ack
I	发送 GetS/IS^{D}	发送 GetM/IM^{AD}	错误	发送 Ack/I	错误	错误	错误	错误	错误
S	进行读取	发送 GetM/IM^{AD}	I	发送 Ack/I	错误	错误	错误	错误	错误
M	进行读取	进行写入	发送 PutM/MI^{A}	错误	发送数据，发送 PutMS/MS^{A}	发送数据/I	错误	错误	错误
IS^{D}	Z	Z	Z	发送 Ack/ISI^{D}	错误	错误	错误	保存数据，进行读取/S	错误
ISI^{D}	Z	Z	Z	发送 Ack	错误	错误	错误	保存数据，进行读取/I	错误
IM^{AD}	Z	Z	Z	发送 Ack	错误	错误	错误	保存数据，IM^{A}	错误
IM^{A}	Z	Z	Z	错误	IMS^{A}	IMI^{A}	错误	错误	进行读取/M

（续）

状态	读取	写入	替换	失效	Forwarded _GetS	Forwarded _GetM	PutM_ Ack	Data	Last Ack
IMI^A	Z	Z	Z	错误	错误	错误	错误	错误	进行读取，发送数据/I
IMS^A	Z	Z	Z	发送 Ack/IMI^A	Z	Z	错误	错误	进行写入/发送数据/S
MS^A	进行读取	Z	Z	错误	发送数据	发送数据 MI^A	/S	错误	错误
MI^A	Z	Z	Z	错误	发送数据	发送数据/I	/I	错误	错误

421

表5-14 目录控制器转换

状态	GetS	GetM	Put M（拥有者）	PutMS（非拥有者）	PutM（拥有者）	PutMS（非拥有者）
DI	发送数据、加至共享者/DS	发送数据、清除共享者，发送拥有者/DM	错误	发送PutM_Ack	错误	发送PutM_Ack
DS	发送数据、加至共享者/DS	向共享者发送INV，设置拥有者，发送数据/DM	错误	发送PutM_Ack	错误	发送PutM_Ack
DM	转发GetS、加至共享者/DMS^D	转发GetM，向共享者发送INV，清除共享者，设置拥有者	保存数据、发送PutM_Ack/DI	发送PutM_Ack	保存数据，添加到共享者，发送PutM_Ack/DS	发送PutM_Ack
DMS^D	转发GetS、添加到共享者	转发GetM，向共享者发送INV，清除共享者，设置拥有者/DM	保存数据、发送PutM_Ack/DS	发送PutM_Ack	保存数据，添加到共享者，发送PutM_Ack/DS	发送PutM_Ack

对于每个块，目录中都保存着一个状态和当前拥有者字段，或者当前共享者列表（如果有的话）。为了以下讨论及后继问题，假定 L2 缓存被禁用。假定存储器目录以处理器为单位列出了共享者/拥有者。例如，在图 5-26 中，第 108 行的存储器目录为“P0，0：P3，0”，而不是“C0,C1”。另外，假定这些消息可以通过透明方式跨越芯片边界（如果需要的话）。

各行按当前状态索引，各列按事件索引，决定了<操作/后续状态>元组。如果仅列出一个后续状态，则不需要操作。不可能出现的情景标以“错误“，表示错误条件；“z”意味着当前不能处理所请求的事件。

以下示例说明了这一协议的基本操作。假定处理器尝试对处于 I 状态（失效状态）的块进行写入操作。相应的元组为“发送 GetM/IM^{AD}”，表示缓存控制器应当向目录发送 GetM（GetModified）请求，并转换为状态 IM^{AD}。在最简单的情景中，请求消息找到处于 DI（目录失效）状态的目录，表示所有其他缓存都没有副本。目录以 Data 消息作出回应，其中也包含了所希望 Acks 的数目（在此情景中为 0）。在这一简化协议中，缓存控制器将这一条消息看作两条消息：一个 Data 消息，后面跟着一个 Last Ack 事件。首先处理 Data 消息，保存数据并转换到 IM^A。然后处理 Last Ack 事件，转换为状态 M。最后，可以以状态 M 执行写入操作。

如果 GetM 发现处于 DS（目录共享）状态的目录，则该目录向共享者列表中的所有节点发送 Invalidate(INV)消息，向拥有共享者编号的请求者发送数据，并转换为 M 状态。当 INV 消息到达共享者时，它们会找到处于 S 状态或 I 状态（如果它们已经悄悄地使这个块失效）的块。在任何一种情况下，共享者都会直接向发出请求的节点发送一条 Ack。请求者将会计算它接收到的 Ack 数，并与随 Data 消息发回的数字进行对比。当所有 Ack 都已收到后，发生 Last Ack 事件，触发缓存向 M 状态的转换，并允许写入操

422

作继续进行。注意，所有Ack有可能在Data消息之前已经到达，但Last Ack事件并不会发生。这是因为Data消息中包含了Ack数目。因此，此协议假定Data消息是在Last Ack事件之前进行处理的。

5.13 [10/10/10/10/10/10] <5.4>考虑上面介绍的高级目录式协议和图5-26中的缓存内容。在以下每种情况下，哪一个过渡状态序列会影响到缓存块的移动？

a. [10] <5.4>　P0,0: read 100

b. [10] <5.4>　P0,0: read 120

c. [10] <5.4>　P0,0: write 120 <-- 80

d. [10] <5.4>　P3,1: write 120 <-- 80

e. [10] <5.4>　P1,0: read 110

f. [10] <5.4>　P0,0: write 108 <-- 48

5.14 [15/15/15/15/15/15] <5.4>考虑上面介绍的高级目录式协议和图5-26中的缓存内容。在以下每种情况下，哪一个过渡状态序列会影响到缓存块的移动？在所有情况下，假定处理器在同一周期中发出自己的请求，但目录按照自上而下的顺序对这些请求进行排序。假定控制器的操作表现为原子操作（例如，目录控制器将执行DS-->DM转换所需要的全部操作，然后才会处理对同一块的另一请求）。

a. [15] <5.4>　P0,0: read 120
　　P1,0: read 120

b. [15] <5.4>　P0,0: read 120
　　P1,0: write 120 <-- 80

c. [15] <5.4>　P0,0: write 120
　　P1,0: read 120

d. [15] <5.4>　P0,0: write 120 <-- 80
　　P1,0: write 120 <-- 90

e. [15] <5.4>　P0,0: replace 110
　　P1,0: read 110

f. [15] <5.4>　P1,0: write 110 <-- 80
　　P0,0: replace 110

g. [15] <5.4>　P1,0: read 110
　　P0,0: replace 110

423

5.15 [20/20/20/20/20] <5.4>图5-25中所示多处理器（禁用了L2）实现了表5-13及表5-14所述的协议，对此多处理器假定有以下延迟。

- CPU读取与写入命中不产生停顿时钟周期。
- 要完成一次缺失（例如，执行读取或执行写入），仅当它是为回应Last Ack事件而执行时，才会占用L_{ack}个时钟周期（否则，它会在向缓存中复制数据时完成）。
- 生成替换事件的CPU读取或写入操作会在发射PutModified消息之前（比如使用写回缓冲区）发射相应的GetShared或GetModified消息。
- 发送请求或确认消息（例如GetShared）的缓存控制器事件会延迟L_{send_msg}个时钟周期。
- 读取缓存并发送数据消息的缓存控制器事件会延迟L_{send_data}个时钟周期。
- 接收数据消息并更新缓存的缓存控制器事件会延迟L_{rcv_data}个时钟周期。
- 存储器控制器在转发请求消息时会导致L_{send_msg}个时钟周期的延迟。
- 存储器控制器对于它必须发送的每个失效消息会另外导致L_{inv}个时钟周期的延迟。
- 缓存控制器为它接收到的每个失效消息产生L_{send_msg}个时钟周期的延迟（一直延迟到它发送Ack消息为止）。
- 存储器控制器读取存储器和发送数据消息的延迟为L_{read_memory}个时钟周期。
- 存储器控制器向存储器写入数据消息的延迟为L_{write_memory}个时钟周期（一直延迟到它发送Ack消息为止）。

- 非数据消息（例如，请求、失效、Ack）的网络延迟为 L_{req_msg} 个时钟周期。
- 数据消息（例如，请求、失效、Ack）的网络延迟为 L_{data_msg} 个时钟周期。
- 任何一条从芯片 0 传送到芯片 1 的消息都会增加 20 个时钟周期的延迟（反之亦然）。

考虑一种实现方式，其具有表 5-15 汇总的性能指标。

针对以下操作序列、图 5-26 中的缓存内容以及上述目录协议，每个处理器节点观测到的延迟为多少？

a. [20] <5.4> P0,0: read 100
b. [20] <5.4> P0,0: read 128
c. [20] <5.4> P0,0: write 128 <-- 68
d. [20] <5.4> P0,0: write 120 <-- 50
e. [20] <5.4> P0,0: write 108 <-- 80

424

表5-15 目录式一致性协议的延迟

操 作	延 迟
Send_msg	6
Send_data	20
Rcv_data	15
Read-memory	100
Write-memory	20
inv	1
ack	4
Req_msg	15
Data_msg	30

5.16 [20] <5.4>在出现缓存缺失时，前面介绍的交换监听协议和本案例研究中的目录式协议都会尽快执行读取或写入操作。具体来说，它们将这一操作看作是向稳定状态转换过程的一部分，而不是先转换到稳定状态，再简单地重试这一操作。这并非一种优化。而是为了确保转发过程的成功，各种协议实施方式必须确保它们在放弃一个块之前至少执行了一个 CPU 操作。假定一致性协议实施方式没有实现这一点。试解释这种做法可能如何导致活锁（livelock）。给出可以模拟这一行为特性的简单代码示例。

5.17 [20/30] <5.4>一些目录式协议向协议中添加了“被拥有”（O）状态，类似于前面讨论的对监听式协议的优化。在那些可能仅读取“被拥有”块的节点中，“被拥有”状态的表现类似于“共享”状态，但在某些节点中，必须根据其他节点对“被拥有”块的 Get 请求提供数据，“被拥有”状态的表现类似于“已修改”状态。在对一个“已修改”块发出 GetShared 请求时，需要节点向发出请求的处理器和存储器发送数据，而“被拥有”状态消除了这一情景。在 MOSI 目录式协议中，在对处于“已修改”或“被拥有”状态的块发出 GetShared 请求时，会向发出请求的节点提供数据，并转换为“被拥有”状态。在处理“被拥有”状态的 GetModified 请求时，类似于对“已修改”请求的处理。这种经过优化的 MOSI 协议仅在节点替换处于“已修改”或“被拥有”状态的块时才会更新存储器。

a. [20] <5.4>试解释为什么说此协议中的 MSA 状态本质上就是一个“过渡”的“被拥有”状态。

b. [30] <5.4>请修改缓存与目录协议表，以支持一种稳定的“被拥有”状态。

5.18 [25/25] <5.4>前面所述的高级目录式协议依靠一种点对点有序互连来确保操作正确。假定有图 5-38 的初始缓存内容和以下操作序列，试解释，当此互连不能维持点对点排序时，可能会发生什么问题。假定处理器同时执行这些请求，而目录则是按所示顺序进行处理。

425

a. [25] <5.4> P1,0: read 110
P3,1: write 110 <-- 90

b. [25] <5.4> P1,0: read 110
P0,0: replace 110

练习

5.19 [15] <5.1>假定有一个关于应用程序的函数，其形式为 $F(i,p)$，表示在总共提供 p 个处理器的情况下，恰好有 i 个处理器可供使用的时间比例。即：

$$\sum_{i=1}^{p} F(i,p)=1$$

假定在使用 i 个处理器时，应用程序的运行速度加快 i 倍。请改写 Amdahl 定律，将某一应用程序的加速比表示为 p 的函数。

5.20 [15/20/10] <5.1>在这个练习中，我们研究互连网络拓扑对程序的每条指令时钟周期数（CPI）的影响（这些程序运行在包含 64 个处理器的分布式存储器多处理器上）。处理器的时钟频率为 3.3 GHz，应用程序的所有引用都在缓存中命中，其基础 CPI 为 0.5。假定有 0.2%的指令涉及远程通信引用。远程通信引用的成本为（100+10h）ns，其中 h 是指一次远程引用为到达远程处理器存储器并返回时必须在通信网络进行跳转的次数。假定所有通信网络都是双向的。

a. [15] <5.1>当 64 个处理器分别排列为一个环、一个 8×8 处理器网格或者超立方体时，计算在最糟情况下的远程通信成本。（提示：在一个 2^n 超立方体中，最长的通信路径有 n 个链接。）

b. [20] <5.1>将此应用程序在没有远程通信时的基本 CPI 与分别采用(a)部分三种拓扑时获得的 CPI 进行对比。

c. [10] <5.1>与分别采用(a)部分三种拓扑进行远程通信的性能相比，没有远程通信的应用程序可以加快多少倍？

5.21 [15] <5.2>说明可以如何针对直写缓存来修改图 5-4 所示的基本监听协议。与写回缓存相比，在采用直写缓存时不需要哪一项主要硬件功能？

5.22 [20] <5.2>向基本监听缓存一致性协议（见图 5-4）中添加一个洁净的独占状态。以图 5-4 的形式给出这一协议。

5.23 [15] <5.2>关于伪共享问题，有人提出一种解决方案：为每个字添加一个有效位。添加之后，不需要删除整个块就能使一个字失效，这样处理器可以在其缓存中保存块的某一部分，而另一个处理器可以写入这个块的不同部分。如果包含这一功能，需要向基本监听式缓存一致性协议（见图 5-4）中增加什么样的复杂性？注意考虑所有可能出现的协议操作。

426

5.24 [15/20] <5.3>本练习研究的是在共享存储器多处理器系统设计中应用一些非常积极的技术时，对处理器中开发指令级并行的影响。考虑两种除处理器之外完全相同的系统。系统 A 使用的处理器采用简单的单发射循序流水线，而系统 B 使用的处理器具有 4 路发射、乱序执行以及拥有 64 项的重新排序缓冲区。

a. [15] <5.3>按照图 5-6 中的约定，我们将执行时间划分为指令执行、缓存访问、存储器访问和其他停顿。预测系统 A 与系统 B 中这些组件有什么何种不同？

b. [10] <5.3>基于 5.3 节对联机事务处理（OLTP）工作负载行为特性的讨论，OLTP 工作负载与其他基准之间的哪种重要区别限制了它从更积极处理器设计中获得的好处？

5.25 [15] <5.3>如何改变应用程序的代码，以避免伪共享？编译器做些什么？哪里需要有程序员的指示？

5.26 [15] <5.4>假定有一个目录式缓存一致性协议。目录中当前拥有的信息表明处理器 P1 拥有“独占”模式的数据。如果这个目录现在收到处理器 P1 对同一缓存块的请求，这可能意味着什么？目录控制器应当怎么做？（此类情况被称为**竞赛情景**，正是难以设计和验证一致性协议的原因所在。）

5.27 [20] <5.4>目录控制器可以为那些已经被本地缓存控制器替换的行发送失效消息。为了避免此类消息，并保持目录的一致性，人们采用了替换提示。此类消息告诉控制器：某个块已经被替换。修改 5.4 节的目录式一致性协议，以利用此类替换提示。

5.28 [20/30] <5.4>利用全部填充的位向量来直接实现目录时，一个不利因素是目录信息的总大小会以乘积规模扩大（即：处理器数 × 存储器块数）。如果存储器随处理器数目线性增大，那目录的总大小就会以处理器数目的 4 次方增长。在实践中，由于目录只需要为每个存储器块保存一个位（一个存储器块通常为 32 ~ 128 个字节），所以当处理器数目处于中小规模时，这一问题并不严重。例如，假定有一个大小为 128 字节的块，与主存储器相比，目录存储的大小为处理器数目/1024，或者说当有 100 个处理器时，大约增加 10%的存储量。如果发现需要保存的信息量与每个处理器的缓存大小成比例，那就可以避免上述问题。我们将在这些练习中探讨一些解决方案。 427

a. [20] <5.4>一种可以获得可扩展目录协议的方法是将多处理器组织为逻辑层次结构，以处理器作为这个层次结构的叶子，目录位于每个子树的根部。每个子树的目录记录哪些后代缓存了哪些存储器块，以及哪些以该子树中的节点为主节点的存储器块被缓存在该子树的外部。假定每个目录都是完全相联的，计算记录这些目录的处理器信息所需要的存储数量。答案中还应当包括该层次结构每一级中的节点数目以及节点的总数。

b. [30] <5.4>实现目录机制的一种替代方法是实现一些非密集位向量。共有两种策略；一种策略是减少所需要的位向量数目，另一种策略是减少每个向量的位数。可以通过跟踪来对比这些机制。首先，将这个目录实现为一个 4 路组相联缓存，其中存储全部填满的位向量，但仅针对那些被缓存在主节点外部的块。如果发生了目录缓存缺失，选择一个目录项，并使该项失效。第二，将目录实现为每个项目包含 8 个位。如果某个块仅被缓存在其主节点之外的唯一节点中，那这个字段将包含这个节点的编号。如果这个块被缓存在主节点之外的多个节点中，那这个字段就是一个位向量，每一位指示一组 8 个处理器，其中至少有一个处理器缓存了这个块。利用 64 处理器执行过程的跟踪功能，模拟这些机制的行为。假定非共享引用的缓存非常完美，从而只需要关注一致性行为特性。确定当目录缓存大小增大时的外来失效数目。

5.29 [10] <5.5>使用**链接载入/条件存储**指令对实现经典的“测试并置位”指令。

5.30 [15] <5.5>一种常用的性能优化方法是在填充同步变量时，将同步变量所在缓存行中的所有其他有用数据全部排除在外。构造一个可能造成性能损失的反例。假定采用监听写入失效协议。

5.31 [30] <5.5>为多核处理器实现**链接载入/条件存储**对的一种可能方式是将对这些指令设置约束条件，使其使用未缓存的存储器操作。监听单元负责解读所有核心对该存储器的所有读取与写入操作。它跟踪**链接载入**指令的来源，以及在**链接载入**及其相应的**条件存储**指令之间是否发生了任何中间存储操作。监听单元可以防止任何发生失败的条件存储操作写入任何数据，并可以使用互连信号通知处理器：此次存储失败。请为支持四核心对称多处理器（SMP）的存储器系统设计这样一个监听器。考虑以下因素：读取请求和写入请求的数据大小通常是不同的（4、8、16、32 字节）。任何存储器位置都可能是**链接载入/条件存储**对的目标，存储器监听器应当假定：对任意位置进行的**链接载入/条件存储**引用都可能与同一位置的常规访问交错在一起。监听器的复杂性应当与存储器大小无关。 428

5.32 [10/12/10/12] <5.6>如 5.6 节中的讨论内容，存储器一致性模型提供了一个技术规格，说明存储器系统应当如何展现给程序员。考虑以下代码段，其中，初始值为：

```
A=flag=C=0.
P1                    P2
A= 2000               while (flag ==1){;}
flag=1                C=A
```

a. [10] <5.6>在代码段结束时，预测 C 中的取值。

b. [12] <5.6>系统采用通用互连网络、目录式缓存一致性协议，支持非阻塞式载入，结果C等于0。描述一种可能出现这一结果的情景。

c. [10] <5.6>如果希望使该系统保持顺序一致性，需要设置哪些关键的约束条件？假定处理器支持宽松存储器一致性模型。宽松一致性模型需要明确确认同步。假定此处理器支持“屏障”指令，能够确保此屏障指令之前的所有存储器操作都完成之后，才允许开始屏障之后的存储器操作。为了确保能够得到顺序一致性的“直观结果”，应当在以上代码段的什么位置包含屏障指令？

5.33 [25] <5.7>一个两级缓存层次结构，L1更接近处理器。请证明，如果L2的相联度至少与L1相同，两种缓存都使用行可替换单元（LRU）进行替换，两个缓存的块大小相同。

5.34 [讨论] <5.7>在尝试对多处理器系统进行详细的性能评估时，系统设计人员使用以下三种工具之一：分析模型、由跟踪驱动的模拟和由执行驱动的模拟。分析模型使用数学表达式对程序的行为进行建模。由跟踪驱动的模拟在实际计算机上运行应用程序，并生成跟踪轨迹，通常是存储器操作的跟踪轨迹。这些跟踪轨迹可以通过缓存模拟器进行回放，也可以用具有简单处理器模型的模拟器进行回放，在参数发生变化时预测系统的性能。由执行驱动的模拟器模拟整个执行过程，为处理器状态等保持等价结构。这些方法之间的准确性与速度均衡如何？

5.35 [40] <5.7、5.9>在增加处理器的数目时，多处理器和集群的性能通常也会提高，理想情况下应当是 n 个处理器提高 n 倍。这一有偏基准测试的目标是让程序在增加处理器时的性能恶化。例如，这就意味着当多处理器或集群仅有一个处理器时，程序的运行速度最快，有2个处理器时，速度较慢，4个处理器时要比两个处理器还慢，以此类推。在每种组织结构中，是哪些关键的性能特性导致逆线性加速比的？

429

6

以仓库级计算机开发请求级、数据级并行

数据中心就是计算机。

——Luiz André Barroso，
Google（2007）

100 多年前，各家公司不再自行用蒸汽机和发电机发电，而是接入新建成的电网中。电力公用设施提供的廉价电能所改变的内容并非仅限于这些公司的运行方式。这一现象导致了一连串的经济与社会转型，拉开了现代世界的序幕。今天，一场类似的革命正在进行之中。海量信息处理工厂挂在相互联网的全球计算网格上，开始将数据和软件代码注入我们的家中和公司中。这一次变为公用设施的东西是计算能力。

——Nicholas Carr，
《大变革：重新审视世界，从爱迪生到 Google》
（*The Big Switch: Rewiring the World, from Edison to Google*）（2008）

6.1 引言

谁都可以搭建一块快速CPU。难点在于如何搭建一个快速系统。

——Seymour Cray，
公认的超级计算机之父

仓库级计算机（WSC）[①]是许多人每日所用因特网服务的基础，这些服务包括：搜索、社交网络、在线地图、视频共享、网上购物、电子邮件服务，等等。此类因特网服务深受大众喜爱，从而有了创建WSC的必要，以满足公众迅速增长的需求。尽管WSC看起来可能就像是一些大型数据中心，但它们的体系结构和操作有很大的不同，稍后我们将会看到。今天的WSC像是一个巨型机，其成本大约在1亿5千万美元量级，包括机房、配电与制冷基础设施、服务器和联网设备，它们连接和容纳了50 000至100 000个服务器。此外，云计算的快速发展（见6.5节）让每一个拥有信用卡的人都能使用WSC。

计算机体系结构很自然地扩展到WSC的设计中。例如，Google公司Luiz Barroso（前文引用了他的话）的学位论文就是从事计算机体系结构方面的研究。他认为，架构师在设计过程中实现可扩展性、提高可靠性的技巧以及调试硬件的技巧，对于创建和操作WSC都有很大的帮助。

WSC的规模已经达到一种极致，需要在配电、制冷、监控和运行等各个方面作出创新，它是超级计算机的现代衍生物——使Seymour Cray成为当今WSC架构师的教父。他的极限计算机可以处理一些在其他所有地方都无法完成的计算，但又是如此昂贵，只有很少几家公司可以承受得起。而现在的目标是为整个世界提供信息技术，而不再是为科学家和工程师们提供高性能计算（HPC）；因此，相对于Cray的超级计算机在过去发挥的作用，WSC在当今社会中扮演了更为重要的角色。

毫无疑问，WSC的用户要比高性能计算的用户多出好几个量级，它们在IT市场占有的份额也要大得多。无论是按用户数量计算还是按收入计算，Google公司都至少是Cray Research公司的250倍。

432

WSC架构师有许多目标和需求与服务器架构师是一致的。

- **成本-性能**——每元钱能够完成的工作量是至关重要的，部分原因就是它的规模太大了。将WSC的资本成本降低10%就可能节省1500万美元。
- **能耗效率**——配电成本与功率消耗具有函数关系，需要有充足的配电供给才能消耗功率。机械系统成本与功率具有函数关系，需要将导入的热能排出去。因此，峰值功率和消耗功率推高了配电与制冷系统两项成本。另外，能耗效率也是环境管理的一个重要组成部分。因此，单位焦耳完成的工作对于WSC和服务器来说都至关重要，对于仓库级计算机来说，主要是因为建造电力与机械基础设施的成本很高，对于服务器而言，则是因为每月的公共供电账单费用很高。

① 本章基础资料的来源包括：Google公司Luiz André Barroso、Urs Hölzle所著的*The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*[2009]；Amazon Web服务部门James Hamilton在mvdirona.com的博客Perspectives和访谈"Cloud-Computing Economies of Scale"以及"Data Center Networks Are in My Way"[2009, 2010]；Michael Armbrust等人的技术报告"Above the Clouds: A Berkeley View of Cloud Computing"[2009]。

- **通过冗余提高可靠性**——因特网服务的性质要求其必须长时间运行，这就意味着 WSC 中的硬件和软件在整体上至少提供 99.99%的可用性；也就是说，它每年的宕机时间必须低于 1 小时。对于 WSC 和服务器来说，冗余都是提高可靠性的关键。服务器架构师经常利用成本更高的更多硬件来实现高可用性，而 WSC 架构师则依靠多个更具成本效率的服务器，这些服务器用低成本网络连接在一起，由软件实现冗余管理。另外，如果可用性目标远远超过“四个九”，那就需要利用多个 WSC 来屏蔽可能摧毁整个 WSC 的事件。对于广泛部署的服务，还可以利用多个 WSC 降低延迟。
- **网络 I/O**——服务器架构师必须提供一个连向外部世界的出色网络接口，WSC 架构也必须如此。为保持多个 WSC 之间的数据一致性，并能够与公众进行交互，需要进行联网。
- **交互式与批处理工作负载**——对于诸如搜索和社交网络等拥有数百万用户的服务，人们期望它们的工作负载具有很强的交互性，与此同时，WSC 与服务器类似，还运行着大量并行批处理程序，用以计算对此类服务有用的元数据。例如，它们可以执行 MapReduce 作业，将通过爬网返回的页面转换为搜索索引（见 6.2 节）。

当然，WSC 也有一些不同于服务器体系结构的特性。

- **充足的并行**——服务器架构师的一个顾虑是：目标市场中的应用程序是否有足够的并行度，以充分发挥大量并行硬件的功用；为开发这一并行，需要有足够的通信硬件，其成本是否过高。WSC 架构师没有此类顾虑。首先，批处理应用程序获益于大量需要独立处理的独立数据集，比如爬网时的数十亿个 Web 页面。这一处理过程就是**数据级并行**，其适用于存储（storage）中的数据，而不是存储器（memory）中的数据，我们在第 4 章介绍过。第二，交互式因特网服务应用程序（也称为**软件即服务**，SaaS）可获益于交互式因特网服务数以百万计的独立用户。在 SaaS 中，读取与写入很少是相关的，所以 SaaS 很少需要同步。例如，搜索服务使用的是只读索引，而电子邮件通常是读取与写入相独立的信息。我们将这种简单并行称为**请求级并行**，有大量独立工作可以很自然地并行开发，几乎不需要通信或同步；例如，基于日志的更新过程可以降低吞吐量需求。由于 SaaS 和 WSC 所取得的成功，很多传统应用程序（比如关系数据库）已经被弱化，转而依靠请求级并行。为使所提供的存储能够扩展到现代 WSC 的规模，甚至删除了一些读/写相关的特征。

433

- **运行成本计算**——服务器架构师通常是在成本预算内使设计的系统实现峰值性能，对于功率的唯一顾虑就是确保它们不会超出机柜的冷却能力。他们通常会忽略服务器的运行成本，假定其相对于购买成本是微不足道的。WSC 的寿命要长于服务器——机房、配电和制冷基础设施经常要使用 10 年以上，所以运营成本也不可小视：在 10 年中，能源、配电和制冷方面的费用占 WSC 成本的 30%以上。
- **规模、与规模相关的机会/问题**——通常，极端计算机都是极端昂贵的，因为它们需要定制硬件，但又因为极端计算机的制造数目很低，所以就无法有效地分摊定制成本。不过，如果我们购买 50 000 台服务器和相关基础设施，用来构造一个 WSC，那的确可以获得总额折扣。由于 WSC 本身就非常庞大，所以即使没有太多 WSC，也可以实现规模经济效应。在 6.5 节和 6.10 节将会看到，这些规模经济导致了云计算的出现，这是因为 WSC 的单位成本较低，也就是说，一些公司可以向外租借这些设施，其收费低于租借者自行完成这些工作所需要的成本。50 000 台服务器的不利之处就是容易发生故障。图 6-1 给

出了2400台服务器的停用与异常情况。即使一台服务器的平均无故障时间（MTTF）达到了令人惊叹的25年（200 000小时），WSC架构师在进行设计时也要应对每天有5台服务器发生故障的情况。表6-1列出的年度磁盘故障率为2%～10%。如果每台服务器有4块硬盘，它们的年度故障率为4%，那拥有50 000台服务器时，WSC架构师预计每一小时就会看到一块磁盘发生故障。

表6-1　一个由2400台服务器组成的新集群在第一年发生的停用与异常，并给出了近似的发生频率

第1年的近似事件数	原　因	结　果
1或2	电力设施故障	整个WSC失去供电，如果UPS和发电机正常工作（发电器的正常工作时间大约占总时间的99%），不会导致WSC宕机
4	集群升级	计划内停用，用于升级基础设计，许多时间是出于升级网络的需求，比如重新连接线缆、交换机固件升级，等等。每一次计划外停用大约会有9次计划内集群停用
1000s	硬盘故障	2%～10%的年度磁盘故障率[Pinheiro 2007]
	磁盘缓慢	仍然能够运行，但运行速度减缓10～20倍
	存储器损坏	每年一次不可纠正的DRAM错误[Schroeder等人，2009]
	机器配置错误	配置会导致大约30%的服务中断[Barroso和HÖlzle 2009]
	脆弱的机器	大约1%的服务器每星期重启一次以上[Barroso和HÖlzle 2009]
5000	个别服务器崩溃	机器重启，通常需要大约5分钟

* 我们将Google所说的集群标记为阵列；见图6-2。（据Barroso [2010]。）

例题　一个服务运行在表6-1中的2400台服务器上，试计算该服务的可用性。本例题中的服务与实际WSC中的服务不同，它不能容忍硬件或软件故障。假定重启软件的时间为5分钟，修改硬件的时间为1小时。

解答　通过计算因为每个组件发生故障所导致的停用时间，可以用来估计服务可用性。我们保守地取表6-1中每个类别的最低数字，将1000次停用平均分配在4个组件之间。我们忽略了运行缓慢的磁盘（1000次停用的第二个分量）和电力设施故障，这是因为磁盘缓存会影响性能但不会影响可用性，而99%的电力设施故障可以通过不间断电源（UPS）系统加以隐藏。

434

$$\text{停用时间}_{\text{服务}}=(4+250+250+250)\times 1\text{小时}+(250+5000)\times 5\text{分钟}$$
$$=754+438=1192\text{小时}$$

一年有365 × 24=8760小时，所以可用性为：

$$\text{可用性}_{\text{系统}}=\frac{(8760-1192)}{8760}=\frac{7568}{8760}=86\%$$

即，如果没有软件冗余来屏蔽如此之多的停用次数，在这2400台服务器上运行的服务将会达到平均每周一天的宕机时间，其可用性只能达到“零个九”的级别了！

在6.10节将会解释，WSC的先驱是**计算机集群**。集群是一组使用标准局域网（LAN）和商用交换机连接在一起的独立计算机。对于不需要大量通信的工作负载，集群计算的成本效率要远高于共享存储器多处理器。（共享存储器多处理器是第5章所讨论的多核计算机的先驱。）集群在20世纪90年代后期开始流行，先用于科学计算，后来用于互联网服务。关于WSC有

这样一个观点：它们就是过去数百个服务器组成的集群向今天数万个服务器所组成集群的逻辑发展。 435

人们自然会问：WSC 是否与高性能计算（HPC）使用的现代集群类似。尽管有一些 WSC 和 HPC 的规模与成本相近（有些 HPC 设计拥有数百万个处理器，花费数亿美元），但 HPC 的处理器和节点之间的网络通常要比 WSC 中快得多，这是因为 HPC 应用程序的独立性更强，通信更频繁（见 6.3 节）。HPC 设计还倾向于使用定制硬件（特别是在网络中），所以它们通常不能通过使用大众化商用芯片来降低成本。例如，单是 IBM Power 7 微处理器的成本和耗费的功率就高于 Google WSC 中的一个完整服务器节点。其编程环境还强调线程级并行或数据级并行（见第 4 章、第 5 章），通常强调完成单项任务的延迟，而不是通过请求级并行完成许多独立任务的带宽。HPC 集群往往还拥有长时间运行的作业，它们会使服务器满负荷运行，甚至能持续数周以上，而 WSC 中服务器的利用率通常在 10%～50%之间（见图 6-1），而且每天都会发生变化。

WSC 与传统数据中心相比又怎么样呢？传统数据中心的运营商通常会从组织的许多部门收集机器和第三方软件，并集中为他人运行这些机器和软件。它们的关注点通常是将许多服务整合到较少的机器中，这些机器相互隔离，以保护敏感信息。因此，虚拟机在数据中心的重要性日益增加。与 WSC 不同的是，传统数据中心往往拥有各种不同的硬件和软件，为一家组织中的各种不用客户提供服务。WSC 程序员则定制第三方软件或者自行开发软件，WSC 的硬件一致性要强得多；WSC 的目标是让仓库中的硬件/软件看起来就像是只有一台计算机，只是上面运行着各种不同的应用程序。传统数据中心的最大成本通常是维护人员的费用，而在 6.4 节将会看到，在设计完善的 WSC 中，服务器硬件是最大的成本，人力成本从最大成本变得几乎可以忽略。传统数据中心也不具备 WSC 的规模，所以它们无法获得上述的规模经济效益。因此，尽管你可能会将 WSC 看作一种超级数据中心（因为这些计算机都分别放在具有特殊配电和制冷基础设施的空间内），但典型的数据中心通常没有 WSC 面对的挑战和机遇，无论是体系结构方面还是运转方面都是如此。

由于很少有架构师了解WSC中运行的软件，所以我们首先介绍WSC的工作负载和编程模型。

6.2 仓库级计算机的编程模型与工作负载

如果一个问题没有解决的办法，那它可能就不是一个问题，而是一个事实——不是找到其解决办法，而是随着时间的推移找到应对方法。

——Shimon Peres 436

一些面向公众的因特网服务，比如搜索、视频共享和社交网络等，使 WSC 有了名气，除了这些服务之外，WSC 还运行一些批处理应用程序，比如将视频转换为新的格式，或者通过爬网生成搜索索引。

今天，WSC 中最流行的批处理框架是 MapReduce[Dean 和 Ghemawat 2008]和它的开源孪生框架 Hadoop。表 6-2 显示了 Google 公司 MapReduce 的流行程度日益提高。（据估计，2011 年拥有 60 000 台服务器，Facebook 在其中的 2000 台批处理服务器上运行 Hadoop。）受同名 Lisp 函数的启发，Map 首先将程序员提供的函数应用于每条逻辑输入记录。Map 在数千台计算机上运

行，生成由键/值对组成的中间结果。Reduce 收集这些分布式任务的输出，并使用另一个由程序员定义的函数来分解它们。通过适当的软件支持，这两者都是高度并行的，而且其理解和使用都很容易。在 30 分钟之内，程序员新手就可以在数千台计算机上运行 MapReduce 任务。

表6-2　Google公司MapReduce的年度使用数据

	2004年8月	2006年3月	2007年9月	2009年9月
MapReduce作业数	29 000	171 000	2 217 000	3 467 000
平均完成时间（秒）	634	874	395	475
所用的服务器年数	217	2002	11 081	25 562
读取的输入数据（TB）	3288	52 254	403 152	544 130
中间数据（TB）	758	6743	34 774	90 120
写出的输出数据（TB）	193	2970	14 018	57 520
每项作业的平均服务器数	157	268	394	488

* 在 5 年的时间里，MapReduce 作业数增加了 100 倍，每项作业的平均服务器数增长了 3 倍。在后两年中，增长因素分别为 1.6 和 1.2[Dean 2009]。表 6-9 估算：在 Amazon 的云计算服务 EC2 上执行 2009 工作负载时，将会耗费 1.33 亿美元。

例如，一个 MapReduce 程序计算一大组文档中每个英文单词的出现次数。下面是这个程序的简化版本，仅给出了内层循环，并假定所有英文单词仅在文档中出现一次[Dean 和 Ghemawat 2008]：

```
map(String key, String value):
        // key: 文档名
        // value: 文档内容
        for each word w in value:
        EmitIntermediate(w, "1"); // 为所有单词生成清单
reduce(String key, Iterator values):
        // key: 一个单词
        // values: 一个计数清单
        int result = 0;
        for each v in values:
        result += ParseInt(v); // 从键值对中取得整数
        Emit(AsString(result));
```

437

Map 函数中使用的 EmitIntermediate 函数给出文档中的每个单词，并取值 1。然后，Reduce 函数使用 ParseInt()对每个单词在每篇文档中所有取值求和，得出每个单词在所有文档中的出现次数。MapReduce 运行时环境将 map 任务和 reduce 任务调度到 WSC 的节点中。（这个程序的完整版本可以在 Dean 和 Ghemawat[2004]的参考文献中找到。）

MapReduce 可以看作单指令多数据（SIMD）操作（见第 4 章）的推广（只有一点不同：我们向它传递了一个将应用于数据的函数），这一操作后面跟有一个函数，用于对 Map 任务的输出进行约简操作（reduction）。因为约简在 SIMD 程序中很常见，所以 SIMD 硬件经常会为它们提供特殊操作。例如，Intel 最近的 AVX SIMD 指令包含了“横向”（horizontal）指令，对寄存器中相邻的操作数对进行相加。

为了适应数千台计算机的性能变化，MapReduce 调度程序根据各个节点完成先前任务的速度来分配新的任务。显然，哪怕只有一个速度缓慢的任务，也可能会阻挡大型 MapReduce 作业的完成。在 WSC 中，对缓慢任务的解决办法是提供一种软件机制，以应对这一规模所固有的

这种性能变化。这种方法与传统数据库中心中服务器采取的解决方案截然不同，在后一解决方案中，任务缓慢通常意味着硬件损坏，需要替换，或者服务器软件需要调优或重写。对于 WSC 中的 50 000 台服务器来说，性能出现差异是正常现象。例如，当 MapReduce 程序快结束时，系统将开始在其他节点上备份那些尚未完成的任务，并从那些首先完成的任务中获取结果。在将资源利用率提高几个百分点之后，Dean 和 Ghemawat[2008]发现一些大型任务的完成速度可以加快 30%。

另一个说明 WSC 差异的例子是使用数据复制来应对故障。由于 WSC 中的设备如此之多，经常发生故障并不是什么让人惊讶的事情，上个例子就证明了这一点。为了实现 99.99%的可用性，系统软件必须能够应对 WSC 中的这一现实问题。为了降低运行成本，所有 WSC 都使用自动监视软件，使一位操作员可以负责 1000 多台服务器。

编程框架（比如用于批处理的 MapReduce）和面向外部的 SaaS（比如搜索）依靠内部软件服务才能成功运行。例如，MapReduce 依靠 Google 文件系统（GFS）（Ghemawat、Gobioff 和 Leung [2003]）向任意计算机提供文件，因此，可以将 MapReduce 任务调度到任意地方。

除了 GFS 之外，此类可伸缩存储系统的示例还包括 Amazon 的键值存储系统 Dynamo [DeCandia 等人 2007]和 Google 记录存储系统 Bigtable [Chang 2006]。注意，此类系统经常是相互依赖的。例如，Bigtable 将其日志和数据存储在 GFS 中，就像是关系数据库可以利用内核操作系统提供的文件系统。

这些内部服务作出的决定经常不同于类似软件在单个服务器上运行时作出的决定。例如，这些系统并没有假定存储是可靠的（比如使用 RAID 存储器系统来保证其可靠性），而是经常生成数据的完整副本。制作副本有助于提高读取性能和可用性，通过正确放置这些副本，可以解决许多其他系统故障，比如表 6-1 中列出的一些。某些系统采用删除编码，而不是完整复制，但有一点是不变的，那就是实现跨服务器冗余，而不是实现服务器内部或存储阵列内部的冗余。因此，整个服务器或存储设备发生故障时，不会对数据可用性产生负面影响。 438

还有另外一个例子说明 WSC 中采用的不同方法：WSC 存储软件经常使用宽松一致性，而没有遵循传统数据库系统的所有 ACID（原子性、一致性、隔离性和持久性）需求。这里的重点是：数据的多个副本最终一致才是最重要的，而对于大多数应用程序来说，它们并不需要在所有时间都保持一致。例如，视频共享只需要最终保持一致就行。最终一致性大大简化了系统的扩展，而扩展绝对是 WSC 必不可少的要求。

这些公共交互式服务的工作负载需求都会有大幅波动，即使是 Google 搜索这样流行的全球性服务，在一天中的不同时间也可能会有两倍的变化幅度。如果针对某些应用程序考虑周末、假日和一年中高峰时间等因素（比如万圣节之后的图片共享服务，或者圣诞节之前的网上购物活动），将会看到提供因特网服务的服务器在利用率方面都波动较大。图 6-1 给出了 5000 台 Google 服务器在 6 个月时间内的平均利用率。注意，不到 0.5%的服务器平均利用率达到 100%，大多数服务器的利用率介于 10%至 50%之间。换句话说，利用率超过 50%的服务器仅占全部服务器的 10%。因此，对于 WSC 的服务器来说，在工作负载很低时的良好表现要远比峰值负载时的高效运行重要得多，因为它们很少会在峰值状态下运行。

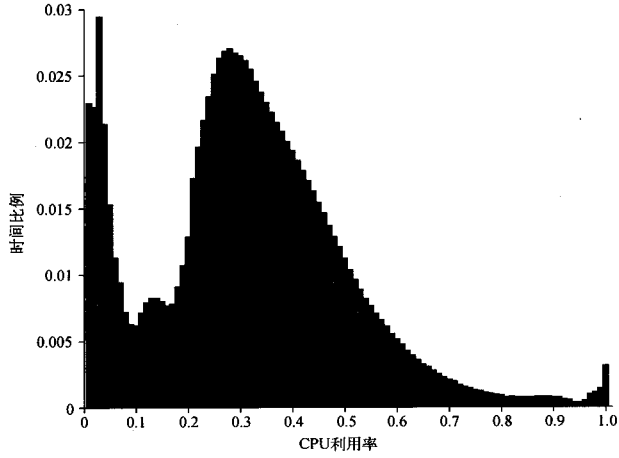


图 6-1　Google 公司 5000 多台服务器在 6 个月时间内的平均 CPU 利用率。服务器很少是完全空闲或全负荷工作的，在大多数时间里，介于其最大利用率的 10%到 50%之间。（据 Barroso 和 Hölzle [2007]）。表 6-3 右侧第三列计算了百分比并加减 5%，以提供权值；因此，90%一行的 1.2%表示有 1.2%的服务器达到了 85%至 95%的利用率

总之，WSC 硬件和软件必须能够应对因为用户需求所造成的负载变化，以及因为硬件在这一规模的各种变化而造成性能和可靠性的变化。

例题　针对不同的负载情况，利用 SPECPower 基准测试对功率和性能进行测试，负载变化范围为 0%～100%，每次递增 10%（见第 1 章），测量结果如图 6-1 所示。可以用一个整体度量来总结这一基准测试的结果，即：将所有性能测试值（单位：服务器端每秒执行的 Java 操作数）之和除以所有功率测量值（单位：瓦）之和。因此，每个级别的可能性相等。如果利用图 6-1 中的利用频率对这些级别进行加权，这一数字汇总数据将如何变化？

解答　表 6-3 给出了与图 6-1 匹配的原权重和新权重。这些权重将性能汇总值降低了 30%，由 3210 ssj_ops/W 降低到 2454 ssj_ops/W。

表6-3　来自表6-10的SPECPower结果，使用图6-1中的权重，而非平均权重

负载	性能	瓦	SPEC权重	加权后的性能	加权后的瓦数	图6-1的权重	加权后的性能	加权后的瓦数
100%	2 889 020	662	9.09%	262 638	60	0.80%	22 206	5
90%	2 611 130	617	9.09%	237 375	56	1.20%	31 756	8
80%	2 319 900	576	9.09%	210 900	52	1.50%	35 889	9
70%	2 031 260	533	9.09%	184 660	48	2.10%	42 491	11
60%	1 740 980	490	9.09%	158 271	45	5.10%	88 082	25
50%	1 448 810	451	9.09%	131 710	41	11.50%	166 335	52

（续）

负载	性能	瓦	SPEC权重	加权后的性能	加权后的瓦数	图6-1的权重	加权后的性能	加权后的瓦数
40%	1 159 760	416	9.09%	105 433	38	19.10%	221 165	79
30%	869 077	382	9.09%	79 007	35	24.60%	213 929	94
20%	581 126	351	9.09%	52 830	32	15.30%	88 769	54
10%	290 762	308	9.09%	26 433	28	8.00%	23 198	25
0%	0	181	9.09%	0	16	10.90%	0	20
总数	15 941 825	4967		1 449 257	452		933 820	380
				ssj_ops/W	3210		ssj_ops/W	2454

由于规模的原因，软件必须能够应对故障，这就意味着没有什么理由再去购买那些降低故障频率的“镀金”硬件。它只能是提高成本。Barroso 和 Hölzle [2009]发现，在运行 TPC-C 数据库基准测试时，高端共享存储器多处理器与大众化 HP 服务器的价格性能比相差 20 倍。可以想到，Google 公司购买的都是低端大众化服务器。 439

此类 WSC 服务还倾向于开发自己的软件，而不是购买第三方商用软件，部分原因是为了应对这种庞大的规模，部分原因是为了节省资金。例如，即使是在 2011 年 TPC-C 的最佳性价比平台上，将 Oracle 数据库和 Windows 操作系统的成本包含在内时，也会使 Dell Poweredge 710 服务器的成本加倍。相反，Google 在其自己的服务器上运行 Bigtable 和 Linux 操作系统，无须为其支付版权费用。

在对 WSC 中的应用程序和系统软件进行以上简单综述后，现在可以开始研究 WSC 的计算机体系结构了。

6.3 仓库级计算机的计算机体系结构

网络是将 50 000 台服务器连接在一起的结缔组织。类似于第 2 章的存储器层次结构，WSC 使用一种网络层次结构。图 6-2 给出了一个示例。理想情况下，这种组合网络提供的性能应当接近于为 50 000 台服务器定制的高端交换机，而每端口的成本则接近于为 50 台服务器设计的大众化交换机。在 6.6 节可以看到，目前的解决方案与理想情况相去甚远，WSC 的网络是一个活跃的探索领域。

19 英寸（48.26 厘米）机架仍然是容纳服务器的标准框架，尽管这一标准要追溯到 20 世纪 30 年代的铁路硬件。服务器的大小按照它们在机架内占用的机架单元（U）数计算。1 U 高 1.75 英寸（4.45 厘米），这是一个服务器可以占用的最小空间。

7 英尺（213.36 厘米）的机架提供 48 U，因此，针对一个机架生成的最流行交换机都是 48 端口以太网交换机，这就不是巧合了。这一产品已经成为一种大众化商品，2011 年，1 Gbit/s 以太网连接的每端口成本只有 30 美元[Barroso 和 Hölzle 2009]。注意，机架内的速度对于每个服务器都是一样的，所以软件将发送器和接收器放在哪个位置都没关系，只要它们位于同一机架就行。从软件的角度来看，这种灵活性是很理想的。

这些交换机通常提供 2 ~ 8 个上行链接，它们离开机架，进入网络层次结构的下一层更高阶交换机。因此，离开机架的速度是机架内速度的 1/6 到 1/24（8/48 到 2/48）。这一比值称为**超额认购率**（oversubscription）。当超额认购率很高时，程序员必须知道将发送机和接收机放在不同机架 440 ~ 441

时导致的性能后果。这样会增大软件调度负担，这是支持数据中心专用网络交换机的另一个理由。

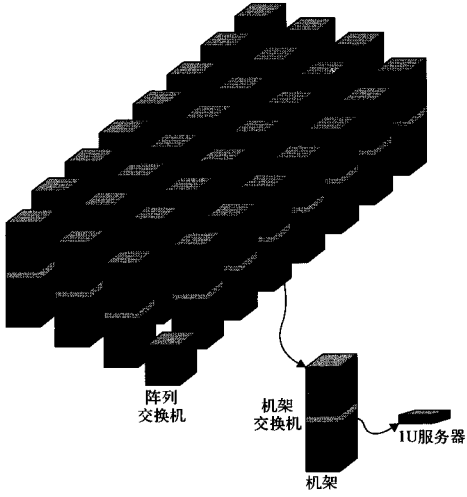


图 6-2　WSC 中的交换机层次结构。（基于 Barroso 和 Hölzle [2009]论文中的图 1-2）

6.3.1　存储

一种很自然的设计是用服务器填充一个机架，当然要扣除大众化以太网机架交换机所需要的空间。这种设计带来一个问题：把存储放在哪儿。从硬件构建的角度来看，最简单的解决方案是将磁盘包含在服务器中，通过以太网连接访问远程服务器磁盘上的信息。另一种替代方案是使用网络连接存储（NAS），可能是通过类似于 Infiniband 的存储网络。NAS 解决方案的每 TB 存储容量成本通常要更高一些，但它提供了许多功能，包括用于提高存储器可靠性的 RAID 技术。

根据上一节表达的思想，可以预计：WSC 通常会依靠本地磁盘，并提供用于处理连接性和可靠性的存储软件。例如，GFS 使用本地磁盘，至少维护三个副本，以克服可靠性问题。这一冗余不仅可以应对本地磁盘故障，还能应对机架和整个集群的电源故障。GFS 的最终一致灵活性降低了使这些副本保持一致所需的成本，还降低了存储系统的网络带宽需求。稍后将会看到，本地访问模式还意味着连向本地存储的高带宽。

442

要小心：在讨论 WSC 的体系结构时，对于集群一词有一点混淆。根据 6.1 节的定义，WSC 就是一个超大型集群。而 Barroso 和 Hölzle [2009]则用集群一词来表示更大一级的计算机组，在本例中大约为 30 个机架。在本章中，为了避免混淆，我们使用阵列一词来表示一组机架，使集群一词保持其最初含义，既可以表示一个机架内的联网计算机组，也可以表示整整一仓库的联网计算机。

6.3.2　阵列交换机

将一个机架阵列连在一起的交换机要比 48 口大众化以太网交换机贵得多。之所以出现这种高成本，部分原因是由于较高的连通性，部分原因是为了减轻超额认购问题，必须大幅增大通

过交换机的带宽。据 Barroso 和 Hölzle [2009]报告，如果一个交换机的二分带宽（基本上就是最糟情况下的内部带宽）是机架交换机二分带宽的 10 倍，那其成本将大约为 100 倍。其中一个原因是：n 端口交换机带宽的成本成长速度与 n^2 成比例。

这种高成本的另一原因是这些产品为其生产公司提供了很高的利润率。为了证明这种高价格的正当性，这些公司的一种做法就是提供一些诸如数据包检测之类的昂贵功能，这些功能之所以昂贵，是因为它们必须以极高速率运转。例如，网络交换机是内容可寻址存储器芯片和现场可编程门阵列（FPGA）的主要用户，这样当然有助于提高这些功能，但这些芯片本身是很昂贵的。这些功能在因特网设置上可能很有价值，但一般不会在数据中心中使用。

6.3.3 WSC存储器层次结构

表 6-4 给出了 WSC 内部存储器层次结构的延迟、带宽和容量，图 6-3 以可视方式显示了同一数据。这些数字基于以下假设[Barroso 和 Hölzle 2009]。

表6-4 WSC存储器层次结构的延迟、带宽和容量[Barroso和Hölzle 2009]

	本 地	机 架	阵 列
DRAM延迟（ms）	0.1	100	300
磁盘延迟（ms）	10 000	11 000	12 000
DRAM带宽（MB/s）	20 000	100	10
磁盘带宽（MB/s）	200	100	10
DRAM容量（GB）	16	1040	31 200
磁盘容量（GB）	2000	160 000	4 800 000

* 图 6-3 绘制了这一相同信息。

443

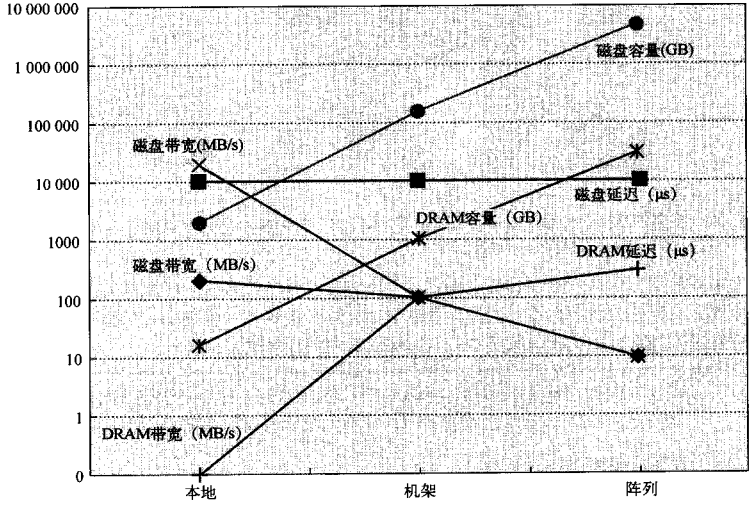


图 6-3 WSC 存储器层次结构的延迟、带宽和容量数据曲线，数据与表 6-4 中相同[Barroso 和 Hölzle 2009]

- 每个服务器包含 16 GB 存储器，访问时间为 100 ns，传输速度为 20 GB/s，还有一个 2 TB 磁盘，访问时间为 10 ms，传输速度为 200 MB/s。每块主板上有两个插槽，它们共享一个 1 Gbit/s 以太网端口。
- 每对机架包括一个机架交换机，容纳 80 个 2U 服务器（见 6.7 节）。联网软件再加上交换机开销将到 DRAM 的延迟增加到 100 ms，磁盘访问延迟增加到 11 ms。因此，一个机架的总存储容量大约为 1 TB 的 DRAM 和 160 TB 的磁盘存储。1 Gbit/s 以太网将连向该机架内的 DRAM 或磁盘的远程带宽限制为 100 MB/s。
- 阵列交换机可以处理 30 个机架，所以一个阵列的存储容量增加 30 倍：DRAM 为 30 TB，磁盘为 4.8 PB。阵列交换机硬件和软件将连向阵列内 DRAM 的延迟增加到 500 ms，磁盘延迟增加到 12 ms。数据交换机的带宽将连向阵列 DRAM 或阵列磁盘的远程带宽限制为 10 MB/s。

444

表 6-4 和图 6-3 显示：网络开销大幅延长了从本地 DRAM 到机架 DRAM 和阵列 DRAM 的延迟，但这两者的延迟性能仍然优于本地磁盘，不到其十分之一。网络消除了机架 DRAM 与机架磁盘之间、阵列 DRAM 和阵列磁盘之间的带宽差别。

这个 WSC 需要 20 个阵列连接到 50 000 个服务器，所以联网层次结构又多了一级。图 6-4 给出了用于将阵列连接在一起并连至因特网的传统 L3 路由器。

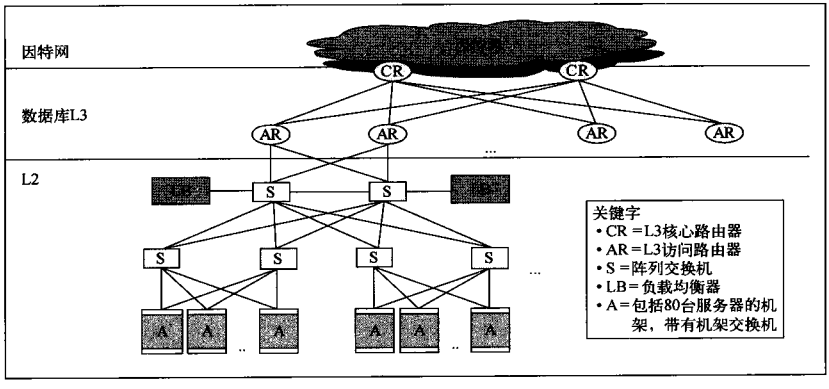


图 6-4　用于将阵列连接在一起并连接到因特网的 L3 网络[Greenberg 等人，2009]。一些 WSC 使用独立的边界路由器将因特网连接到数据中心 L3 交换机

大多数应用程序可以放在 WSC 中的单个阵列上。那些需要多个阵列的应用程序会使用分片或分区，也就是说将数据集分为独立片断，然后再分散到不同阵列中。对整个数据集执行的操作被发送到托管这些片断的服务器，其结果由客户端计算机接合起来。

例题　假定 90%的访问为服务器的本地访问，9%的访问超出服务器但在机架范围内，1%的访问超出机架但在阵列范围内，则平均存储器延迟为多少？

解答　平均存储器访问时间为：

$$(90\% \times 0.1) + (9\% \times 100) + (1\% \times 300) = 0.09 + 9 + 3 = 12.09 \text{ 微秒}$$

或者说，比 100%的本地访问减缓 120 倍以上。显然，实现一个服务器内的访问局域性对于 WSC 性能来说是至关重要的。

445

例题 在服务器内部的磁盘之间、在机架内的服务器之间、在阵列中不同机架内的服务器之间，传递 1000 MB 需要多少时间？在这三种情况下，在 DRAM 之间传递 1000 MB 可以加快多少时间？

解答 在磁盘之间传递 1000 MB 需要的时间为：

在服务器内部=1000/200=5 秒
在机架内部=1000/100=10 秒
在阵列内部=1000/10=100 秒

在存储器之间传送块时需要的时间为：

在服务器内部=1000/20000=0.05 秒
在机架内部=1000/100=10 秒
在阵列内部=1000/10=100 秒

因此，对于单个服务器外部的块传输而言，由于机架交换机和阵列交换机是瓶颈所在，所以数据是在存储器中还是磁盘中并不重要。这些性能限制影响了 WSC 软件的设计，并激发了对更高性能交换机的需求（见 6.6 节）。

知道了 IT 设备的体系结构，我们现在可以看看如何对其进行摆放、供电和冷却，并讨论构建和运行整个 WSC 的成本，与其中 IT 设备本身的成本进行对比。

6.4 仓库级计算机的物理基础设施与成本

要构建 WSC，首先需要建造一个仓库。第一个问题就是：在哪里建造？房地产代理强调位置，但对 WSC 来说，位置意味着接近因特网骨干光纤、电力成本低、环境灾难风险低（比如地震、洪水和飓风）。对于一个拥有许多 WSC 的公司来说，另一个关注点是找到一个在地理上接近当前或未来因特网用户群的地方，以降低通过因特网的延迟。还有其他许多现实考虑因素，比如不动产税率。

配电与制冷的基础设施成本会让 WSC 的构造成本相形见绌，所以我们把主要精力放在前者。图 6-5 和图 6-6 给出了 WSC 内的配电与制冷基础设施。

尽管有许多不同的部署方式，但在北美洲，从 115 千伏的电力塔高压线开始，电力通常经过大约 5 个步骤、四级电压变换到达服务器。

(1) 变电站从 115 000 伏转换为 13 200 伏的中压线，效率为 99.7%。

446

(2) 为防止整个 WSC 在停电时离线，WSC 和一些服务器一样，配备有不间断电源（UPS）。在这种情况下，其中涉及大型柴油机，可以在发生紧急状况时将供电任务从电力公司接管过来，还会涉及电池或惯性轮，用于在电力服务中止但柴油机尚未准备就绪时维持供电。这些发电机和电池可能占用很大的空间，通常将它们和 IT 设备放在不同的房间中。UPS 扮演着三个角色：电力调整（保持正常的电压和其他性能指标）、在发电机启动并正常供电时保持电气负载、在从发电机切换回公共用电时保持电气负载。这种超大型 UPS 的效率为 94%，所以使用 UPS 会损失 6%的电能。WSC UPS 可能占全部 IT 设备成本的 7%～12%。

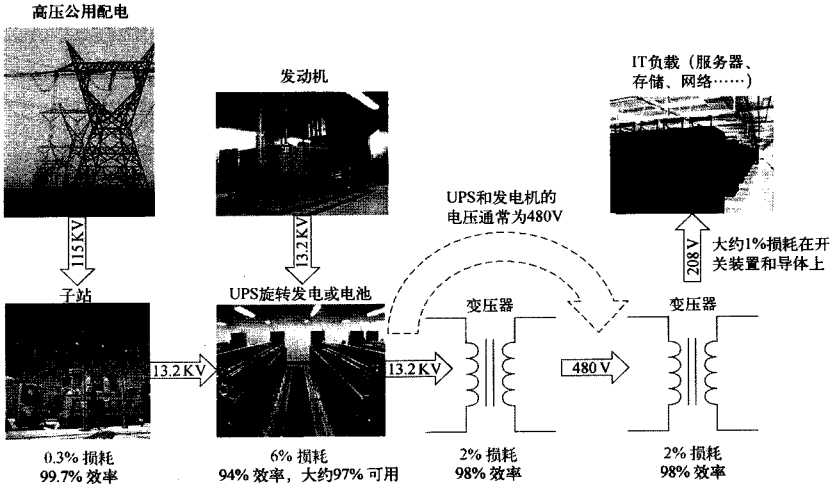


图 6-5　配电与发生损耗的地方。注意，最佳改进值为 11%（据 Hamilton[2010]。）

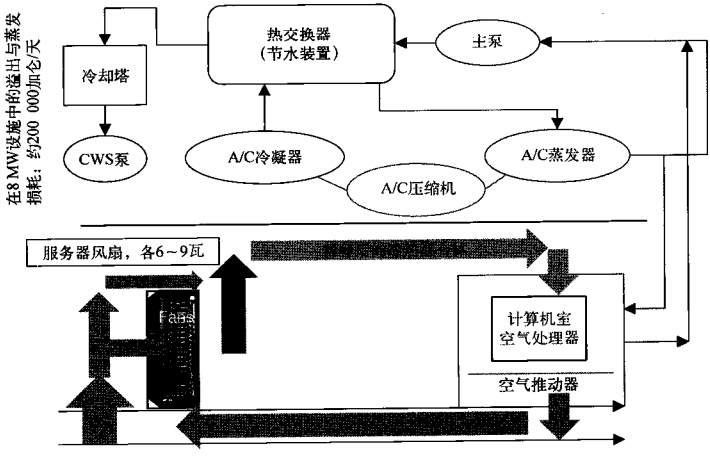


图 6-6　制冷系统的机制设计。CWS 表示循环水系统（据 Hamilton[2010]。）

(3) 系统中的下一个组件是配电单元（PDU），它将转换为 480 伏的三相内部低压电源。这一转换效率为 98%。典型的 PDU 可以承载 75～225 千瓦的负载，相当于 10 个机架的负载。

(4) 还有另外一个下变换步骤，将其转换为服务器可以使用的 208 伏两相电源，其效率也是 98%。（在服务器内部，还有另外一些步骤，将电压降到芯片可以使用的级别；见 6.7 节。）

447

(5) 连接器、断路器和至服务器电气连接的整体效率为 99%。

北美之外的 WSC 使用不同的变换值，但整体设计是类似的。

总而言之，将来自公共用电的 115 千伏电源转换为服务器可以使用的 208 伏电源，其效率为 89%：

$$99.7\% \times 94\% \times 98\% \times 98\% \times 99\% = 89\%$$

这一总效率仅留下 10%多一点的改进空间，但后面将会看到，工程师们仍然在尝试锦上添花。

制冷基础设施中的改进机会要大得多。计算机室空调（CRAC）单元使用冷却水来冷却服务器室内的空间，类似于冰箱通过向外部释放热量来降低温度。当液体吸收热量时，它会蒸发。相反，当液体释放热量时，它会冷凝。空调机将液体注入低压螺旋管中，使其蒸发并吸收热量，然后再将其发送到外部冷凝器，在这里释放热量。因此，在 CRAC 单元中，风扇吹动热空气，穿过一组装有冷水的旋管，水泵将加热后的水移到外部冷凝器进行冷却。服务器的冷却空气通常介于 64℉与 71℉之间（18℃和 22℃之间。）图 6-6 显示了一大组风扇和水泵，使空气和水在整个系统中流动。 448

显然，提高能耗效率的最简单方法之一就是让 IT 设备在较高温度下运行，降低冷却空气的需求。一些 WSC 在远高于 71（22℃）的温度下运行自己的设备。

除了冷凝器之外，在一些数据中心中还会利用冷却塔，先充分利用外部的较冷空气对水进行冷却，然后再将初步冷却后的水发送给冷凝器。真正要紧的温度称为**湿球温度**。在测量湿球温度时，会向一个温度计沾有水的球形末端吹动空气。这是通过空气流动蒸发水分所能达到的最低温度。

温水流过冷却塔的一个大面积表面，通过蒸发将热传递给外部空气，从而使水冷却。这种技术被称为**经济化供风**。一种替换方法是使用冷水，而不是冷空气。Google 公司在比利时的 WSC 使用一种“水到水”中间冷却器，从工业导管取得冷水，用于冷却来自 WSC 内部的热水。

IT 设备本身的空气流进行了仔细规划，有些设计甚至还使用了空气流仿真器。高效的空气流设计可以减少冷空气与热空气的混合机会，从而保持冷空气的温度。例如，在机架上摆放服务器时，可以让相邻行中的服务器朝向相反方向，使排出的高温废气吹向相反方向，从而使 WSC 的热空气通道和冷空气通道交替存在。

除了能量损耗之外，冷却系统还会因为蒸发或溢入下水道的原因而消耗大量水。例如，8 兆瓦设施每天可能使用 70 000 到 200 000 加仑的水。

在典型数据中心中，制冷系统与 IT 设备的相对功率成本[Barroso 和 Hölzle 2009]如下：

- 冷凝器占 IT 设备功率的 30%～50%；
- CRAC 占 IT 设备功率的 10%～20%，大多消耗在风扇上。

让人奇怪的是，在减去配电设备与制冷系统的开销之后，仍然不能很清楚地看出一个 WSC 可以支持多少服务器。服务器制造商提供的所谓**铭牌额定功率**总是很保守的；它是一个服务器可能消耗的最大功率。因此，第一步就是在 WSC 中可能部署的各种工作负载下对一台服务器进行测量。（联网设备通常占总功耗的 5%，在开始时可以忽略。）

为了确定 WSC 的服务器数目，可以将 IT 的可用功率除以测得的服务器功率；但是，根据 Fan、Weber 和 Barroso[2007]所说，这同样太过保守了。他们发现，由于很少有真正的工作负载可以让数千台服务器同时工作于峰值状态，所以数千台服务器理论上在最糟情况所做的工作与它们实际所做的工作之间有很大的差距。他们发现，根据单个服务器的功率，他们可以放心地将服务器数目超额认购 40%。他们建议，WSC 架构师应当做一些工作，提高 WSC 内部的平均 449

功率利用率；但是，他们还建议使用大量的监视软件和安全机制，当工作负载偏移时可以取消某些较低优先级的任务。

Barroso 和 Hölzle[2009]对 IT 设备自身内部的功率利用进行了分解，据此给出了在 2007 年部署的 Google WSC 的数据，如下所示：

- 33%的功率用于处理器；
- 30%用于 DRAM；
- 10%用于磁盘；
- 5%用于联网；
- 22%用于其他原因（服务器内部）。

6.4.1　测量 WSC 的效率

有一种广泛使用的简单度量可以用来评估一个数据中心或 WSC 的效率，称为功率利用效率（或 PUE）：

PUE=（总设施功率）/（IT 设备功率）

因此，PUE 总是大于或等于 1，PUE 越大，WSC 的效率就越低。

Greenberg 等人[2006]报告了 19 个数据中心的 PUE，以及制冷基础设施开销所占的比例。图 6-7 展示了他们的研究成果，按照 PUE 从最高效到最低效排列。PUE 的中值为 1.69，制冷基础设施使用的功率超过服务器本身的一半，平均起来，1.69 中有 0.55 用于制冷。注意，这些都是平均 PUE，可能会根据工作负载、甚至外部空气温度而发生日常变化，稍后将会看到这一点。

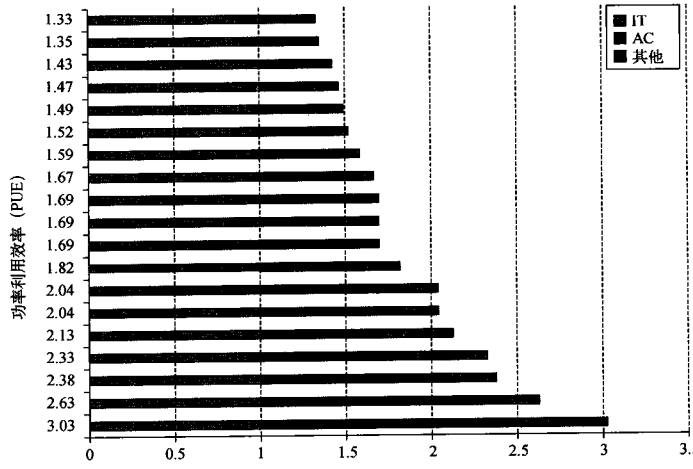


图 6-7　2006 年 19 个数据中心的功率利用效率[Greenberg 等人，2006]。在计算 PUE 时，以 IT 设备的功率为基准，对空调（AC）及其他应用（比如配电）的功率进行了归一化。因此，IT 设备的功率必然为 1.0，AC 的功率为 IT 设备功率的 0.3 ~ 1.4 倍。“其他”功率为 IT 设备的 0.05 ~ 0.6 倍

由于最终度量是每美元实现的性能，所以仍然需要测试性能。如前面的图 6-3 所示，距离

数据越远，带宽越低，延迟越大。在 WSC 中，服务器内部的 DRAM 带宽为机架内带宽的 200 倍，而后者又是阵列内带宽的 10 倍。因此，在 WSC 内部放置数据和程序时，还需要考虑另一种局域性。

WSC 的设计人员经常关注带宽，而为 WSC 开发应用程序的程序员还会关注延迟，因为这些延迟会让用户感受到。用户的满意度和生产效率都与服务的响应时间紧密联系在一起。在分时时代的几项研究表明：用户生产效率与交互时间成反比，这一交互时间通常被分解为人员输入时间、系统响应时间、人们在输入下一项时考虑回应的时间。试验结果表明：将系统响应时间削减 30%可以将交互时间减少 70%。这一令人难以相信的结果可以用人类自身的特性来解释：人们得到响应的速度越快，需要思考的时间越短，因为在这种情况下不太容易分神，一直保持“高速运转”。

表 6-5 给出了对 Bing 搜索引擎进行这一试验的结果，在此试验中，搜索服务器端插入了 50 ms 至 2000 ms 的延迟。和前面研究中的预测一样，到下一次单击之前的时间差不多是这一延迟的两倍；也就是说，服务器端延迟为 200 ms 时，下一次单击之前的时间会增加 500 ms。收入随着延迟的增加而线性下降，用户满意度也是如此。对 Google 搜索引擎进行的另一项研究发现，这些影响在试验结束 4 周之后还没有消失。五个星期之后，当用户体验到的延迟为 200 ms 时，每天的搜索人员会减少 0.1%，当用户体验到的延迟为 400 ms 时，每天的搜索人员会减少 0.2%。考虑到搜索产生的经济效益，即使如此之小的变化也是令人不安的。事实上，这些结果的负面影响非常严重，以致这项试验提前结束[Schurman 和 Brutlag 2009]。 450

表6-5 Bing搜索服务器的延迟对用户行为的负面影响（Schurman和Brutlag [2009]）

服务器延迟(ms)	下一次单击之前的增加时间（ms）	查询/用户	任意单击/用户	用户满意度	收益/用户
50	—	—	—	—	—
200	500	—	−0.3%	−0.4%	—
500	1200	—	−1.0%	−0.9%	−1.2%
1000	1900	−0.7%	−1.9%	−1.6%	−2.8%
2000	3100	−1.8%	−4.4%	−3.8%	−4.3%

451

由于因特网服务极端看重所有用户的满意度，所以通常在指定性能目标时，不是提供一个平均延迟目标，而是给出一个很高的百分比，要求至少有如此比例的请求低于某一延迟门限。这种门限目标被称为**服务级目标**（SLO）或者**服务级协议**（SLA）。某一 SLO 可能是要求 99% 的请求都必须低于 100 ms。因此，Amazon Dynamo 键值存储系统的设计者决定：为使服务能够在 Dynamo 的顶层提供好的延迟性能，他们的存储系统必须在 99.9%的时间内实现其延迟目标[DeCandia 等人 2007]。例如，Dynamo 的一项改进对第 99.9 个百分点的帮助要远多于平均情景，这反映了他们的侧重点。

6.4.2 WSC的成本

在引言中曾经提到，WSC 的设计者与大多数架构师不同，他们要同时关注 WSC 的**运行成本**和**构建成本**。会计部门将前一成本记为**运行性支出**（OPEX），后一成本记为**资本性支出**（CAPEX）。

为了正确合理地看待能源成本，Hamilton[2010]通过一项案例研究了 WSC 的成本。他确定 8 MW 设施的 CAPEX 为 8800 万美元，大约 46 000 台服务器和相应的网络设备向 WSC 的 CAPEX

另外增加了7900万美元。表6-6给出了这一案例研究的其他假定。

表6-6　WSC的案例研究，据Hamilton[2010]，四舍五入到最接近的5000美元

设施规模（临界负载：瓦）	8 000 000
平均功率利用率（%）	80%
功率利用效率	1.45
电力成本（美元/千瓦时）	0.07美元
电力和制冷基础设施百分比（占总设施成本的百分比）	82%
设施的CAPEX（不包括IT设备）	88 000 000美元
服务器数	45 978
成本/服务器	1450美元
服务器的CAPEX	66 700 000美元
机架交换机数	1150
成本/机架交换机	4800美元
阵列交换机数	22
成本/阵列交换机	300 000美元
L3交换机数	2
成本/L3交换机	500 000美元
边界路由器数目	2
成本/边界路由器	144 800美元
联网设备的CAPEX	12 810 000美元
WSC的总CAPEX	167 510 000美元
服务器分摊时间	3年
网络设备分摊时间	4年
设施分摊时间	10年
借款的年度利率	5%

* 因特网带宽成本随应用程序变化，所以这里未包含在内。设施CAPEX的其余18%包括购买知识产权和机房的建设成本。我们在表6-7中加入了安全和设施管理的人力成本，此案例研究没有包括这一部分。注意，Hamilton的评估是在他加入Amazon之前完成的，这些评估值并非以特定公司的WSC为基础。

由于美国会计规则允许我们将CAPEX转换为OPEX，所以我们现在可以为总能耗成本定价了。我们只需要将CAPEX分摊到设备有效寿命内的每个月份，使其为一个固定值。表6-6对这一案例研究的月度OPEX进行了分解。注意，不同设备的分摊率有很大不同，设施的分摊期限为10年，网络设备为4年，服务器为3年。因此，WSC设施持续一个年代，但需要每3年将服务器更换一次，每4年将网络设备更换一次。通过分摊CAPEX，Hamilton得到了月度OPEX，包括借钱支付WSC款项的利率（每年5%）。月度OPEX为380万美元，大约为CAPEX的2%。

利用此表，可以算出一个很便捷的准则，在决定使用哪些与能源有关的组件时，一定要记住这一准则。在一个WSC中，每年每瓦的全额成本（包括分摊电力和制冷基础设施的成本）为：

$$\frac{\text{基础设施的月度成本+电力的月度成本}}{\text{设施规模（单位：瓦）}}\times 12=\frac{76.5\text{万美元}+47.5\text{万美元}}{800\text{万}}\times 12=1.86\text{美元}$$

此成本大约是2美元/(瓦·年)。因此，要想通过节省能源来降低成本，花费不应当超过2美元/(瓦·年)（见6.8节）。

452　注意，有超过三分之一的OPEX是与电力有关的，当服务器成本随时间下降时，这一部分

反而会上升。网络设备的成本也很高，占总 OPEX 的 8%，服务器 CAPEX 的 19%，网络设备成本不会像服务器成本那样快速下降。特别是对于那些在联网层次结构中高于机架的交换机，这一点尤其正确，大多数联网成本都花费在这些交换机上（见 6.6 节）。关于安全与设施管理的人力成本大约为 OPEX 的 2%。将表 6-7 中的 OPEX 除以服务器的数目及每个月的小时数，可以得到其成本大约为每小时 0.11 美元/服务器。

453

表6-7 表6-6的月度OPEX，四舍五入到最接近的5000美元

费用（占总费用的百分比）	类别	月度成本（美元）	月度成本百分比
分摊CAPEX（85%）	服务器	2 000 000	53%
	网络设备	290 000	8%
	电力与制冷基础设施	765 000	20%
	其他基础设施	170 000	4%
OPEX(15%)	月度用电成本	475 000	13%
	月度人员薪金与津贴	85 000	2%
	总OPEX	3 800 000	100%

* 注意，服务器的 3 年分摊期意味着需要每 3 年购买一次新服务器，而设施的分摊期为 10 年。因此，服务器的分摊构建成本大约是设施的 3 倍。人力成本包括 3 个安保岗位，每天连续 24 小时，每年 365 天，每人每小时为 20 美元；1 个设施人员，每天连续 24 小时，每年 365 天，每人每小时为 30 美元。津贴为薪金的 30%。这一计算没有包含因特网网络带宽成本，因为它是随应用程序的变化而变化的，也没有包含供应商维护费用，因为它们是随设备与协议的变化而变化的。

例题 美国不同地区的电力成本变化范围为 0.03 ~ 0.15 美元/千瓦时。这两种极端费率对每小时的服务器成本有什么样的影响？

答案 我们将 8 兆瓦的临界负载乘以表 6-6 中的 PUE 和平均功率利用率，以计算平均功率使用率：

$$8 \times 1.45 \times 80\% = 9.28 \text{ 兆瓦}$$

于是，若费率为 0.03 美元/千瓦时，月度电力成本从表 6-7 的 475 000 美元变为 205 000 美元，若费用为 0.15 美元/千瓦时，则变为 1 015 000 美元。电力成本的这些变化使每小时的服务器成本分别由 0.11 美元变为 0.10 美元和 0.13 美元。

例题 如果将所有分摊时间都变为相同的（比如）5 年，那月度成本会发生什么变化？每台服务器每小时成本会发生什么变化？

解答 可以从 http://mvdirona.com/jrh/TalksAndPapers/PerspectivesDataCenterCostAndPower.xls 下载该电子表格。将分摊时间改为 5 年会将表 6-7 的前 4 行变为：

服务器	1 260 000美元	37%
网络设备	242 000美元	7%
电力与制冷基础设施	1 115 000美元	33%
其他基础设施	245 000美元	7%

454

总月度 OPEX 为 3 422 000 美元。如果我们每 5 年更换所有东西，成本将为每小时 0.103 美元/服务器，分摊成本的主体现在是设施，而不是服务器，如表 6-7 所示。

每小时0.11美元/服务器的费率远低于许多公司拥有和运行自有（较小）传统数据中心的成本。WSC的成本优势导致大型因特网公司都将计算功能当作一种公用设施来提供，和电力一样，你只需要为自己使用的那一部分付费即可。今天，公用计算有一个更好的名字——云计算。

6.5　云计算：公用计算的回报

> 如果未来的计算机就是我所倡导的计算机类型，那可能会有那么一天，计算功能也被当作一种公用设施，就像电话系统是一种公用设施一样……计算机公共设施可能成为一种新的重要行业的基础。
>
> ——John McCarthy，
> 《MIT百年庆典》（*MIT Centennial celebration*）（1961）

由于用户数目不断增大，受其需求的推动，诸如Amazon、Google和微软等因特网公司用大众化组件构建了日益增大的仓库级计算机。这一需求导致了系统软件的革新，以能够在这一规模正常工作，这些系统软件包括Bigtable、Dynamo、GFS和MapReduce。它还要求改进运行技术，尽管存在组件故障和安全攻击，也要使所提供的服务能够保证至少在99.99%的时间内可供使用。这些技术的示例包括故障转移、防火墙、虚拟机和防御分布式拒绝服务攻击的保护。有了提供扩展能力的软件和专业知识，再加上日益增长的客户需求证明了投资的合理性，拥有50 000到100 000台服务器的WSC已经在2011年变得很常见。

随着规模的增大，规模经济的好处也日益明显。2006年的一项研究对比了WSC和仅有1000台服务器的数据中心，根据这一研究，Hamilton[2010]报告了WSC的以下优势。

- **存储成本缩减为数据中心的17.7%**——WSC用于磁盘存储的费用为每年4.6美元/GB，而数据中心则为26美元/GB。
- **管理成本缩减为数据中心的14.0%**——WSC的服务器与管理员之比超过1000，而数据中心仅为140。
- **联网成本缩减为数据中心的13.7%**——WSC的因特网带宽成本为每月13美元/Mbps，而数据中心为95美元。不难想到，在协商带宽价格时，订购1000 Mbps的单位Mbps价格肯定可以远低于订购10 Mbps的价格。

455

规模经济在采购期间也得到了体现。大规模的采购可以获得服务器与网络设备的大额折扣价格。它可以对供应链进行优化。Dell、IBM和SGI都是在一周内完成有关WSC的订单，而不会拖到4～6个月之后。提交时间的缩短大大简化了这一公共设施的扩展过程，能够更好地满足需求。

规模经济同样适用于运行成本。我们在上一节看到，许多数据中心的运行PUE为2.0。大型公司可以雇用机械工程师和电力工程师对WSC进行改进，使其PUE更低，降至1.2左右（见6.7节）。

出于对可靠性和降低延迟的考虑（特别是对于国际市场），因特网服务需要分布在多个WSC上。由于这一原因，所有大型公司都采用多个WSC。各个公司在世界各地创建多个小型数据中心的成本要远高于在公司总部创建单个数据中心。

最后，出于6.1节给出的原因，数据中心中服务器的利用时间往往只有总时间的10%～20%。将WSC推向公众使用之后，来自不同客户的不相关峰值可以将平均利用率提高到50%以上。

因此，因为WSC的几种组件，可以使WSC的规模经济因数达到5～7，整个WSC的规模经济因数为1.5～2。

尽管有多家云计算服务提供商，但我们这里以 Amazon Web Services（AWS）为例，一方面是因为它的普及性，另一方面是因为这些服务的抽象层级较低，更为灵活。Google App Engine 和 Microsoft Azure 将抽象层级提高到托管运行时，提供自动扩展服务，这样可以更好地适应一些客户，但不像 AWS 那样更适于作为本书中的介绍材料。

Amazon Web Services

公用计算可以追溯到 20 世纪 60 年代和 70 年代的商业分时系统，甚至是批处理系统，当时公司只需要预先支付终端和电话线路的费用，然后再根据它们所使用的计算量来计费。在分时时代结束之后，人们进行了许多尝试，希望再为服务提供这种支付模式，但经常遭遇失败。

当 Amazon 在 2006 年开始通过 Amazon 简单存储服务（Amazon S3）和后来的 Amazon 弹性计算机云来提供公用计算时，它作出了几项不同寻常的技术与商业决策。

- **虚拟机**。利用运行 Linux 操作系统和 Xen 虚拟机的 x86 大众化计算来构建 WSC，解决了几个问题。第一，通过这一方法可以让 Amazon 为用户提供保护，使他们免受其他用户的伤害。第二，简化了 WSC 中的软件分布，因为客户只需要安装一个映像，AWS 之后会自动将它发布到所使用的全部实例上。第三，能够可靠终止虚拟机的功能使 Amazon 和客户能够轻松地控制资源利用情况。第四，由于虚拟机可以限制它们利用物理处理器、磁盘和网络的速率和使用主存储器的数量，从而为 AWS 提供了多种不同价格选择：将多个虚拟核心挤在单个服务器上，价格最低；独占使用所有机器资源，价格最高；还有介于两者之间的几个中间选择。第五，虚拟机隐藏了较旧硬件的身份，使 AWS 可以继续出售旧机器的工作时间，如果让客户知道了这些机器的年龄，很可能就会失去对他们的吸引力。最后，虚拟机允许 AWS 引入新的更快硬件，具体方式可以是在每个服务器上运行更多的虚拟核心，也可以是提供一些在每个虚拟核心上拥有更高性能的实例；虚拟化的使用，意味着所提供的性能不一定是硬件性能的整数倍。 456
- **极低成本**。当 AWS 在 2006 年宣布每个实例 0.10 美元/小时的费率时，这个数字低得让人吃惊。一个实例就是一个虚拟机，以每小时 0.10 美元的价格，AWS 可以在一个多核心处理器的每个核心上运行两个实例。因此，一个 EC2 计算机单元等价于那个时代的 1.0 至 1.2 GHz AMD Opteron 或 Intel Xeon。
- **对开源软件的（初始）依赖**。一些开源软件的质量很高，而且没有版权问题，在数百乃至数千台服务器上运行时也没有相关成本，利用这些软件大幅提高了公用计算的经济性，对 Amazon 和客户都是如此。最近，AWS 开始提供一些包含商用第三方软件的实例，其价格要高一些。
- **没有（最初的）服务保证**。Amazon 最初仅承诺尽最大努力。极具吸引力的成本让许多服务在没有保证的情况下也能生存下来。今天，AWS 对一些服务（比如 Amazon EC2 和 Amazon S3）提供了高达 99.95%的可用性 SLA。此外，Amazon S3 的设计目标是达到 99.999999999%的持久性，具体方法是在多个位置保存每一对象的多个副本。也就是说，永久丢失一个对象的机率是 1 万亿分之一。AWS 还提供了一种**服务健康仪表板**，它可以实时给出每个 AWS 服务器的当前运行状态，使 AWS 的工作时间与性能做到完全透明。
- **不需要合同**。部分原因是其成本如此之低，只需要有一张信用卡就能开始使用 EC2。

表 6-8 给出了 2011 年许多 EC2 实例每小时的价格。除了计算之外，EC2 还负责长期存储和

因特网通信流量。（AWS 区域内部的网络通信流量没有成本。）弹性块存储的成本为每月 0.10 美元/GB、0.10 美元/百万次 I/O 请求。进入 EC2 的因特网通信流量的成本为 0.10 美元/GB，离开 EC2 的成本为 0.08～0.15 美元/GB，具体取决于流量大小。从历史的角度来看，每月支付 100 美元所能使用的容量相当于 1960 年生产的全部磁盘容量总和！

457

表6-8　2011年1月美国维吉尼亚地区EC2实例按需计费的价格和性能指标

实　例	每小时的费用（美元）	与小型实例的比值	计算单元	虚拟核心	计算单元/核心	存储器（GB）	磁盘（GB）	地址大小（位）
微型	0.020	0.5～2.0	0.5～2.0	1	0.5～2.0	0.6	EBS	32/64
标准小型	0.085	1.0	1.0	1	1.00	1.7	160	32
标准大型	0.340	4.0	4.0	2	2.00	7.5	850	64
标准超大型	0.680	8.0	8.0	4	2.00	15.0	1690	64
高存储器超大型	0.500	5.9	6.5	2	3.25	17.1	420	64
高存储器双超大型	1.000	11.8	13.0	4	3.25	34.2	850	64
高存储器四倍超大型	2.000	23.5	26.0	8	3.25	68.4	1690	64
高CPU中型	0.170	2.0	5.0	2	2.50	1.7	350	32
高CPU超大型	0.680	8.0	20.0	8	2.50	7.0	1690	64
集群四倍超大型	1.600	18.8	33.5	8	4.20	23.0	1690	64

* 微型实例是最新、最便宜的类别，它们提供最多两个计算单元的短时簇发，每小时仅 0.02 美元。客户报告，微型实例平均仅 0.5 个计算单元。最后一行的集群计算实例（AWS 确认该实例为专用双插槽 Intel Xeon X5570 服务器，每个插槽有四个核心，工作频率为 2.93 GHz）提供 10Gbit/s 的网络。它们是为 HPC 应用准备的。AWS 还以低得多的价格提供了"现货实例"，在这种情况下，用户设定自己希望支付的价格以及希望运行的实例数目，当"现货价格"降低到用户的设置水平时，AWS 将会运行这些服务，并将一直持续下去，直到用户停止这些服务，或者现货价格超出用户设置的限度为止。在 2011 年 1 月日间的一次取样表明：现货价格的下降因数为 2.3～3.1，具体取决于实例类型。有些客户知道自己在一年的大多数时间内都会使用某一实例，AWS 为这种情况提供了"保留实例"。用户可以支付每个实例的年度费用，然后再按小时支付其使用费用，该费率大约是第 1 列的 30%。如果需要整年使用保留实例，每小时的平均成本包括了年度费用的分摊值，大约为第一列费率的 65%。与表 6-6 和表 6-7 中等价的服务器为标准超大型或高 CPU 超大型实例，我们为其算出的成本为 0.11 美元/小时。

例题　在 EC2 上运行表 6-2 中的平均 MapReduce 作业，计算其成本。假定作业数非常充足，所以不存在为获得整数小时数进行舍入而带来的大量额外成本。忽略月度存储成本，但包含 AWS 弹性块存储（EBS）执行磁盘 I/O 的成本。接下来计算运行所有 MapReduce 作业所需要的年度成本。

解答　第一个问题是：多大规模的实例能与 Google 的典型服务器相匹配？6.7 节的图 6-11 显示：2007 年，一个典型 Google 服务器有四个核心，运行频率为 2.2 GHz，存储器为 8 GB。由于单个实例就是一个虚拟核心，等价于 1 到 1.2 GHz AMD Opteron，所以与表 6-8 最匹配的是拥有 8 个虚拟核心、7.0 GB 存储器的高 CPU 超大型实例。为简单起见，我们假定平均 EBS 存储访问为 64 KB，以计算 I/O 数目。

458

表 6-9 计算了在 EC2 上运行 Google MapReduce 工作负载的平均年度成本及总年度成本。2009 年在 EC2 上运行平均 MapReduce 作业的成本略低于 40 美元，2009 年的总工作负载将在 AWS 上花费 1.33 亿美元。注意，这些作业的 EBS 访问大约占总成本的 1%。

表 6-9 按照 2011 年的 AWS ECS、EBS 价格（表 6-8）运行 Google MapReduce 工作负载（表 6-2）时的估测成本

	8月4日	3月6日	9月7日	9月9日
平均完成时间（小时）	0.15	0.21	0.10	0.11
每项作业的平均服务器数目	157	268	394	488
EC2高CPU XL实例每小时的成本	0.68（美元）	0.68（美元）	0.68（美元）	0.68（美元）
每项MapReduce作业的平均EC2成本	16.35（美元）	38.47（美元）	25.56（美元）	38.07（美元）
EBS I/O请求平均数（百万个）	2.34	5.80	3.26	3.19
每百万I/O请求的EBS成本	0.10（美元）	0.10（美元）	0.10（美元）	0.10（美元）
每项MapReduce作业的平均EBSI/O成本	0.23（美元）	0.58（美元）	0.33（美元）	0.32（美元）
每项MapReduce作业的平均总成本	16.58（美元）	39.05（美元）	25.89（美元）	38.39（美元）
MapReduce作业的年度数目	29 000	171 000	2 217 000	3 467 000
EC2/EBS上MapReduce作业的总成本	480 910(美元)	6 678 011(美元)	57 394 985(美元)	133 107 414(美元)

* 由于使用的是 2011 年价格，所以前几年估测值的准确性低于近几年的估测值。

例题 由于 MapReduce 作业的成本正在增长，已经超过每年 1 亿美元，假想老板希望你研究一些降低成本的方法。两种可供选择的低成本选项是 AWS 保留实例或 AWS 现货实例。你推荐哪一种？

解答 AWS 保留实例收取固定的年度费用，再加上按小时计算的使用费用。2011 年，高 CPU 超大型实例的年度成本为 1820 美元，每小时的费率为 0.24 美元。由于无论我们是否使用这些保留实例都要为其付费，所以假定保留实例的平均利用率为 80%。则每小时的平均价格变为：

$$\frac{\text{年度价格}}{\text{每年的小时数}}+\text{每小时的价格}=\frac{\frac{1820\text{美元}}{8760}+0.24\text{美元}}{80\%}=(0.21+0.241)\times1.25=0.56\text{美元}$$

使用率

因此，对于 2009 年 MapReduce 工作负载，使用保留实例大约可以节省 17%，也就是 2300 万美元。

对 2011 年 1 月份的一些日子进行采样，得出高 CPU 超大型现货实例的每小时成本平均为 0.235 美元。由于这是使用一台服务器的最低价格，而且我们通常希望顺利地完成任务，所以平均成本不可能是最低价格。我们假定为了完成大型 MapReduce 作业，需要支付的价格为最低价格的两倍。对于 2009 年工作负载，使用现货实例节省的成本大约为 31%，即 4100 万美元。

459

因此，你试探性地向老板推荐现货实例，因为它的预付金较低，而且可能节省更多资金。但是，你告诉老板，需要尝试在现货示例上运行 MapReduce 作业，以了解最终采用哪种方式来确保作业可以顺利完成，并确保确实有数百个高 CPU 超大型实例可用于每天运行这些作业。

除了公用计算的低成本与按需付费模型之外，另一个强烈吸引云计算用户的因素是云计算提供商承担了过度供应或供应不足的风险。风险规避对刚起步的公司来说是一件天赐之物，因为随便一个错误都可能是致命的。如果在产品得到广泛应用之前，将过多的宝贵资源花费在服务器上，那公司可能会耗尽自己的资金。如果这种服务突然变得流行起来，但又没有足够的服务器来满足需求，那公司就会给自己希望拼命发展的潜在新客户留下很坏的印象。

这种情景的一个典型代表是 Zynga 的 FarmVille，它是 Facebook 上的一个社交网络游戏。在发布 FarmVille 之前，最大型的社交游戏大约有 500 万个每日玩家。FarmVille 在发行之后的 4 天拥有 100 万个玩家，60 天之后拥有 1000 万个玩家。在 270 天之后，它拥有 2800 万个每日玩家和 7500 万个每月玩家。由于它们部署在 AWS 上，所以能够随着用户的数目无缝增长。此外，它还根据用户需求来降低负载。

有更多的知名公司也正在充分利用云的可伸缩性。Netflix 将其网站和流视频服务由传统的数据中心移植到 AWS 上。Netflix 的目标是让用户能够在回家的路上在（比如）自己的手机上观看电影，到家后无缝切换到电视机上，继续观看还没有看完的电影。这种工作量涉及一些批处理操作，以便将新电影转换为无数种格式，在向手机、平板电脑、笔记本电脑、台式机、游戏主机和数字视频录像机上发送这些电影时需要这些格式。这些 AWS 批处理作业可能需要数千台机器花费几周的时间来完成这些转换。这个事务流的后端在 AWS 完成，编码文件的提供通过“内容分发网络”完成（比如 Akamai 和 Level 3）。这种联机服务的价格比邮寄 DVD 便宜得多。这种低成本使这一新服务变得非常普及。一项研究认为 Netflix 占据了美国晚上峰值时段 30%的因特网下载流量。（与之相对，YouTube 仅占晚上 8 点至 10 点同一时间段的 10%。）事实上，其平均占到因特网流量的 22%，北美最大部分的互联网流量都是由 Netflix 一家公司产生的。尽管 Netflix 订阅账户的增长速度还在增加，Netflix 数据中心的增长速度已经停止，所有容量扩展都是通过 AWS 完成的。

云计算已经可以让每个人都能享受到 WSC 带来的好处。云计算向用户提供了成本相关性（cost associativity），人们可以想象它拥有无限的可伸缩性，但却没有任何额外成本：1000 台服务器工作 1 小时的成本不大于 1 台服务器工作 1000 小时的成本。由云计算提供者来负责确保有充足的服务器、存储和因特网带宽来满足需求。上面提到的优化供应链（这一供应链将新计算机的交付时间缩短到一周）可以帮供应商一个大忙，让他们能够在不至破产的前提下实现这一愿景。这种风险的转移、成本相关性和按需付款的定价方式，是各种不同规模的公司使用云计算的强有力依据。

460

有两个交叉问题影响到了 WSC 的性价比，从而影响了云计算的性价比，即 WSC 网络和服务器软硬件的效率。

6.6　交叉问题

网络设备是数据中心的 SUV。

——James Hamilton（2009）

6.6.1　成为瓶颈的WSC网络

6.4 节表明，机架交换机上层的网络设备占去 WSC 成本中很大的一部分。在完全配置时，Juniper 出品的一个 128 端口 1 Gbit 数据中心交换机的定价是 716 000 美元，其中不包括光纤接口，若包括这些接口则为 908 000 美元。（这些定价有很大的打折幅度，但它们的成本仍然是机架交换机的 50 倍。）这些交换机往往还是耗电大户。例如，EX8216 消耗大约 19 200 瓦，是 WSC 中一台交换机的 500 ~ 1000 倍。此外，这些大型交换机大多是手动配置的，而且非常脆弱。由于其价格原因，所以在 WSC 中很难承受用这些大型交换机来实现双重冗余，这就限制了对容错性的选择[Hamilton 2009]。

但是，对交换机的真正影响是超额认购如何影响 WSC 中软件的设计和服务器与数据的布

置。理想的 WSC 网络可能是一个黑盒，人们对其拓扑和带宽不感兴趣，因为它们不存在限制：我们可以在任意地方运行任意工作负载，针对服务器利用情况进行优化，而不是针对网络通信流量的局域性优化。这些 WSC 网络瓶颈现在包含数据布置，而它又使 WSC 软件变得更为复杂。由于这个软件是 WSC 公司最有价值的资产，增加复杂性带来的成本可能是非常庞大的。

如果读者想了解有关交换机设计的更多知识，可以参阅附录 F，其中介绍了设计互联网络时涉及的问题。此外，Thacker [2007]建议借用超级计算的网络技术来克服价格和性能问题。Vahdat 等人[2010]做得也非常出色，提出了一种网络基础设计，可以扩展到 100 000 个端口和 1 Pbps 的对分带宽。这些新颖数据中心交换机的主要好处就是简化了由于超额认购导致的软件问题。 461

6.6.2 在服务器内部高效利用能量

PUE 是 WSC 效率的度量，但没有哪个指标来描述进入 IT 设备自身内部的能耗情况。因此，还有另外一个可能导致电气效率不佳的因素（未在图 6-5 中给出），那就是服务器内部的电源，它将输入的 208 伏或 110 伏转换为芯片和磁盘使用的电压，通常为 3.3 伏、5 伏和 12 伏。根据微处理器和存储器的要求，再将 12 伏电压变换为主板上的 1.2 至 1.8 伏。2007 年，许多电源的效率为 60%～80%，也就是说，服务器内部的损耗要大于从公共电力塔的高压线经过许多步骤和电压转换直到服务器低压电线的损失。造成这一结果的一个原因是它们事先不知道主板上都有哪些设备，必须为芯片和磁盘提供各种不同电压。第二个原因是电源的功率数通常大于主板所需要的功率。另外，当带有 25%或更低负载时，这些电源的效率通常为最差状态，而图 6-1 显示，许多 WSC 服务器都工作在这一范围。计算机主板也有一些电压调节模块（VRM），它们的效率也比较低。

为了提高当前技术，表 6-10 给出了计算产业拯救气候计划标准（2007），用于评定电源的等级及其目标。注意，这一标准除了指定 100%满载情况下的要求之外，还给出了在 20%和 50%负载情况下的要求。

表6-10 计算产业拯救气候计划中对电源不同时间的效率评估与目标

带载条件	基础	Bronze（2008年6月）	Silver（2009年6月）	Gold（2010年6月）
20%	80%	82%	85%	87%
50%	80%	85%	88%	90%
100%	80%	82%	85%	87%

* 这些评估值是针对多输出电源单元的，是指非冗余系统中的台式机与服务器电源。针对单输出 PSU 有一个略高一点的标准，通常用于冗余配置中（1U/2U 单、双、四插槽及刀片服务器）。

除了电源之外，Barroso 和 Hölzle [2007]还指出，整个服务器的目标应当符合能耗比例；也就是说，服务器消耗的能量应当与其执行的工作量成比例。图 6-8 使用 SPECpower 表明我们距离这一理想目标的实现还有多远，SPECpower 是一种服务器基准测试，它测量在不同性能级别消耗的能量（见第 1 章）。图中显示了一种服务器的实际电能使用情况（截至 2010 年 7 月，这一服务器是运行 SPECpower 的最高效服务器），并在图中添加了能耗比例线。大多数服务器都达不到这样的效率；这一最高效系统的指标大约优于该年度其他被测系统达 2.5 倍，在后来的基准测试比赛中，经常为赢得基准测试而对系统进行特殊设置，它们通常不是该领域的典型系统。例如，最佳 SPECpower 服务器使用了固态磁盘，其容量要小于主存储器！即使如此，这一非常高效的系统在空闲时也消耗了全部功率的 30%，在仅有 10%的负载时，消耗了全部功率的 50%。因此，能耗比例仍然是一个有待实现的远大目标，而不是令人自豪的现实。 462

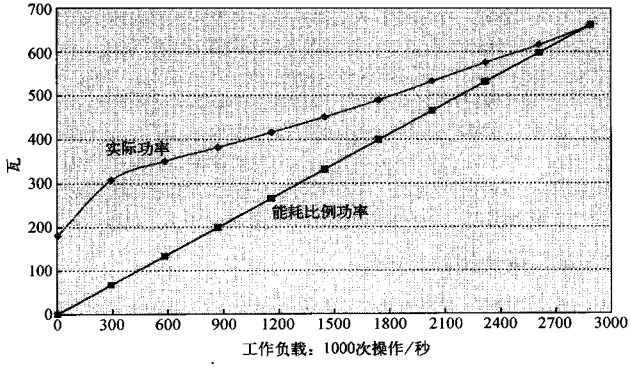


图6-8　2010年7月的最佳SPECpower结果与理想能耗比例特性的对比。此系统为HP ProLiant SL2x170z G6，它使用了由4个双插槽Intel Xeon L5640组成的集群，每个插槽有6个核心，时钟频率为2.27 GHz。这个系统的DRAM为64 GB，辅助存储很小，为60 GB SSD。（主存储器大于磁盘容量这一事实暗示出这一系统是为此基准测试定制的。）所用软件为IBM Java Virtual Machine V9和Windows Server 2008企业版

系统软件的设计思路是利用所有可用资源，只要这样可能提升性能即可，并不关心能耗方面的影响。例如，操作系统为程序数据或文件缓存使用所有存储器，并不考虑许多数据可能根本就用不到。在未来的设计中，软件架构师需要在考虑性能的同时也考虑能耗[Carter和Rajamani 2010]。

例题　根据图6-8所示类型的数据，五个利用率为10%的服务器与一个利用率为50%的服务器相比，可以节省多少功率？

解答　对于单个服务器，负载为10%时为308瓦，负载为50%时为451瓦。因此，节省的功率为：

$$5 \times 308/451 = (1540/451) \approx 3.4$$

如果希望WSC非常环保，必须在利用率下降时对服务器进行整合，购买更符合能耗比例特性的服务器，或者寻找其他一些措施，改进服务器在活跃性较低时的运行情况。

463

有了前面6节的背景知识，我们现在可以欣赏一下Google WSC架构师的作品了。

6.7　融会贯通：Google仓库级计算机

由于许多拥有WSC的公司在市场上的竞争非常激烈，所以直到最近，它们也不愿意与公众（及其他公司）分享自己的最新技术。2009年，Google描述了2005年技术水平的WSC。Google慷慨地提供了2007年对其WSC的更新，从而使这一节成为Google WSC的最新介绍[Clidaras，Johnson和Felderman 2010]。最近，Facebook在http://opencomputer.org介绍了其最新的数据中心。

6.7.1　集装箱

Google和Microsoft都使用货运集装箱来构建WSC。用集装箱来构建WSC的思想是实现

WSC设计的模块化。每个集装箱是独立的，仅有的外部连接是网络、电源和供水。这些集装箱又向放在其内部的服务器提供联网、电源和制冷，所以 WSC 的任务就是向这些集装箱提供联网、电力和冷却水，并将产生的热水抽至外部的冷却塔和冷凝器。

我们正在研究的Google WSC包含45个长40英尺的集装箱，占用300英尺×250英尺的面积，即75 000平方英尺（大约7000平方米）。为了放置在仓库中，30个此种容器两两摞在一起，构成15对堆叠集装箱。尽管其所在位置还没有公开，但当Google在俄勒冈州达尔斯开发WSC时已经兴建了这一场所，这个位置的气候适宜，接近便宜的水力发电厂和因特网骨干光纤。这个WSC在先前的12个月内提供了10兆瓦，PUE为1.23。在0.23的PUE开销中，85%属于制冷损耗（0.195 PUE），15%（0.035）为电力损耗。这一系统于2005年11月建成，本节描述其在2007年的状态。

一个 Google 集装箱可以处理高达 250 千瓦的容量。这意味着这个容器可以处理 780 瓦/平方英尺（0.09平方米），或者说133瓦/平方英尺，在整个75 000平方英尺的空间中放置40个容器。但是，这个WSC中的集装箱平均值只有222千瓦。

图6-9是Google集装箱的一个剖面图。集装箱容纳多达1160台服务器，所以45个集装箱可以容纳52 200台服务器。（这个WSC拥有大约40 000台服务器。）这些服务器堆叠在机架中，高度为20个服务器，构成很长的两行，每行29个机架（也称为分区），各位于集装箱的一侧。机架交换机为48端口1Gbps的以太网交换机，每隔一个机架放置一台机架交换机。 464

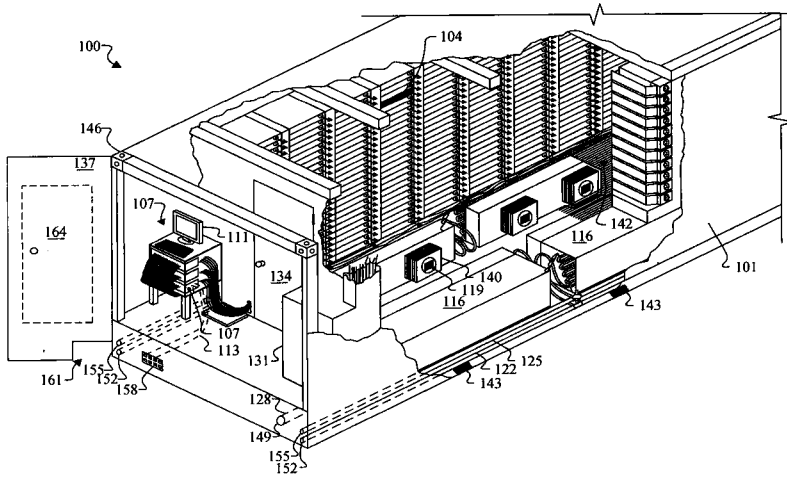


图6-9 Google定制了一个标准的1AAA集装箱：40×8×9.5英尺（12.2×2.4×2.9米）。这些服务器在机架内堆叠在一起，高度为20个服务器，构成很长的两行，每行29个机架，各位于集装箱的一侧。冷却通道在集装箱的中部下行，热空气回行管位于外部。悬挂式机架结构便于修理制冷系统，不用移动服务器。为便于集装箱内的人员修理组件，它包含了用于火灾探测与细雾式灭火、紧急出口与照明和紧急切断供电等安全系统。集装箱内还有许多传感器：温度、气流压力、气体泄露检测和移动感应照明。在 http://www.google.com /corporate /green/datacenters/summit.html 可以找到数据中心的一个视频漫游。微软、雅虎和其他许多公司现在都基于这些思想构建了模块化数据中心，但由于尺寸不太方便，所以他们已经停止使用ISO标准集装箱

6.7.2 Google WSC中的冷却与供电

图 6-10 是集装箱的一个横截面，显示了空气流动情况。计算机机架被固定在集装箱的天花板上。集装箱的地面被升高，制冷系统就位于地面下方，向机架之间的通道中吹送冷风。热空气从机架的后面返回。集装箱中的有限空间防止冷热空间相混合，提高了冷却效率。变速风扇以冷却机架所需的最低转速旋转，而不是一直恒定运转。

"冷"空气保持在 81℉（27℃），与许多传统数据中心的温度相比，这一温度要温和得多。传统数据中心的运行温度之所以非常低，不是因为 IT 设备的原因，而是因为为了保证数据中心的热区不会导致隔离问题。通过精心控制空气流可以避免热区的产生，集装箱的正常运行温度可以高出许多。

465

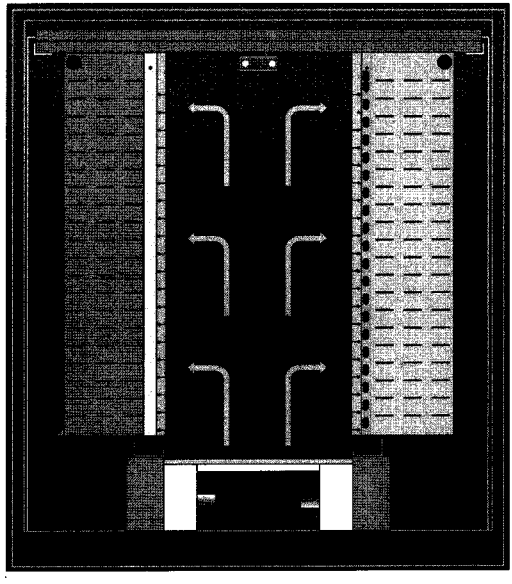


图 6-10　图 6-9 所示集装箱中的空气流。这个横截图给出了两个各位于集装箱一侧的机架。冷空气吹入集装箱中部的通道，然后被吸入服务器。热空气在服务器的边缘返回。这一设计将冷热空气流隔离开来

外部冷凝器有一些断路器，在天气合适时，只需要室外冷却塔对水进行降温。如果离开冷却塔的水温不高于 70℉（21℃），则跳过冷凝器。

注意，如果外部温度过低，冷却塔需要加热器，以防止结冰。将 WSC 放在达尔斯的一个好处就是这里的年度湿球温度范围为 15℉到 66℉（-9℃到 19℃），平均为 41℉（5℃），所以冷凝器可以经常关闭。与之相对，内华达州的拉斯维加斯的变化范围为-42℉到 62℉（-41℃到 17℃，平均为 29℉（-2℃）。此外，由于只需要将集装箱中的温度冷却到 81℉（27℃），所以很

466

有可能仅靠大自然就可以完成水的冷却任务了。

图 6-11 展示了 Google 为这个 WSC 设计的服务器。为了提高电源的效率，它仅向主板提供 12 伏电压，主板仅为板上的若干磁盘提供足够的电源。（便携式电脑采用类似方式为其磁盘供电。）服务器标准是直接提供磁盘和芯片需要的许多种电平。这种简化意味着 2007 年的电源能够以 92%的效率工作，远高于 2010 年黄金级电源的效率（见表 6-10）。

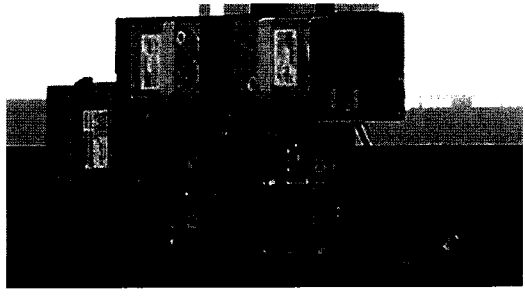


图 6-11 Google WSC 的服务器。电源在左侧，两块磁盘位于顶部。低于左侧磁盘的两个风扇覆盖了 AMD Barcelona 微处理器的两个插槽，每个处理器有两个核心，工作频率为 2.2 GHz。右下方有 8 个 DIMM，各自容量为 1 GB，总数为 8 GB。没有更多的金属片，服务器插入电池中，在机架内为每个服务器准备一个独立空间，用于帮助控制气流。在一定程度上受电池高度的影响，一个机架内装载了 20 台服务器

Google 工程师认识到，采用 12 伏电压的电源，每个架子上的标准电池就可以作为 UPS。因此，它没有采用独立的电池室（图 6-9 中显示其效率为 94%），而是由每个服务器拥有自己的铅酸电池，其效率为 99.99%。这种“分布式 UPS”随每台机器递增部署，这就意味着没有因为超容量配置而花费资金和电力。他们使用标准的开箱即用式 UPS 单元来保护网络交换机。

使用第 1 章介绍的动态电压频率调节（DVFS）来节省功率会怎么样呢？在这一系列的计算机中没有部署 DVFS，这是因为它对延迟的影响过大，对于联机负载，仅在活跃性非常低的情况下才是可行的，即使在这些情况下，系统范围的节省量也是非常小的。因此，部署 DVFS 所需要的复杂管理控制环可能是得不偿失的。

467

要使 PUE 达到 1.23，关键之一就是将测量设备（称为电流转换器）放在集装箱内各个地方的所有电路内以及 WSC 内的其他位置，以测量实际电能利用情况。这些测量允许 Google 在不同时间调整 WSC 的设计。

Google 每季度公布其 WSC 的 PUE。图 6-12 绘制了 10 个 Google WSC 从 2007 年第三季度到 2010 年第二季度的 PUE 曲线；这一节描述标识为 Google A 的 WSC。Google E 的 PUE 为 1.16，冷却系统仅为 0.105，这是因为其工作温度较高，而且拥有冷凝器断流器。配电系统仅为 0.039，这是因为采用了分布式 UPS 和单电压电源。最好的 WSC 结果为 1.12，Google A 为 1.23。在 2009 年 3 月，后面 12 个月的数据用所有数据中心的利用率进行加权之后，其平均值为 1.19。

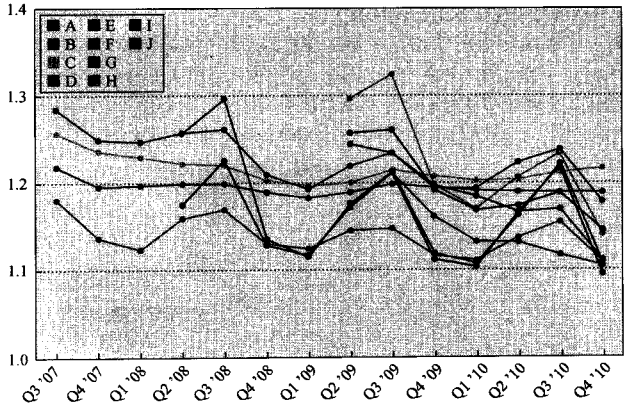


图 6-12　10 个 Google WSC 随时间变化的功率利用效率（PUE）。Google A 是本节描述的 WSC。它是 2007 年第 3 季度和 2010 年第 2 季度的最高线。（据 www.google.com/corporate/green/datacenters/measuring.htm。）Facebook 最近宣布了一个新的数据中心，它给出了一个给人们留下深刻印象的 PUE，仅为 1.07（见 http://opencompute.org/）。Prineville Oregon Facility 没有空调，没有冷却水。它完全依靠外部空气：从建筑物的一侧引入外部空气，通过喷雾器进行过滤和冷却，再送入 IT 设备的各个位置，然后通过排风扇送出机房。此外，服务器使用了一种定制电源，允许配电系统跳过图 6-5 的一个电压变换步骤

6.7.3　Google WSC中的服务器

图 6-11 中的服务器有两个插槽，各包含一个双核心 AMD Opteron 处理器，工作频率为 2.2 GHz。相片中显示 8 个 DIMM，这些服务器通常配有 8 GB 的 DDR2 DRAM。一种新颖的特征就是降低了存储器总线的时钟频率，由标准的 666 MHz 降至 533 MHz，这是因为速度较慢的总线对于性能没什么影响，但对功率的影响却非常显著。

468

基准设计是使用一块网络接口卡（NIC）来实现 1 Gbps 的以太网链接。尽管图 6-11 中的相片显示了两个 SATA 磁盘驱动器，但基准服务器只有一个。基准设计的峰值功率大约为 160 瓦，空闲功率为 85 瓦。

对这个基准节点进行了补充，以提供存储（或“磁盘满”）节点。第一，向服务器连接一个包含 10 个 SATA 磁盘的第二盘塔。为了再多一块磁盘，在主板上的空插槽中放置了一个第二磁盘，使存储节点达到 12 块 SATA 磁盘。最后，由于一个存储节点就可能使单个 1Gbps 以太网达到饱和，所以添加了第二块以太网 NIC。存储节点的峰值功率大约为 300 瓦，空闲功率为 198 瓦。

注意，这个存储节点占去了机架的两个插槽，这也是 Google 在 45 个集装箱中部署 40 000 台而非 52 200 台服务器的原因之一。在这一设施中，对于每个存储节点，两个计算节点大约都是这一比值，但在 Google 的不同 WSC 中，这一比值的变化范围很广。因此，Google A 在 2007 年大约拥有 190 000 块磁盘，平均每台服务器差不多有 5 块磁盘。

6.7.4　Google WSC中的联网

40 000 台服务器被分为三个阵列，每个阵列中有超过 10 000 台服务器。（在 Google 术语中，

阵列被称为集群。）48端口的机架交换机使用40个端口连接到服务器，另外8个端口留给阵列交换机的上行链路使用。

阵列交换机的配置最多支持480个1 Gbps以太网链接和一些10 Gbps端口。这些1 Gbit端口用于连接这些机架交换机，每个机架交换机与每个阵列交换机之间有单一链路。10 Gbps端口分别连接到两个数据中心路由器，它将所有阵列路由器汇集在一起，提供与外部世界的连接性。WSC使用两个数据中心路由器，以保证可靠性，所以单个数据中心路由器故障并不会使整个WSC宕机。

每个机架交换机所用上行链路端口的数目不同，最小为2，最大为8。在双端口情况下，机架交换机工作时的超额认购率为20：1。即，交换机内部的网络带宽为离开交换机带宽的20倍。如果应用程序有大量超出机架之外的通信流量需求，那就容易受到网络性能不佳的困扰。因此，对于有更多通信流量需求的阵列，采用了8端口上行链接设计（它的超额认购率要低得多，仅有5：1）。

6.7.5 Google WSC的监控与修复

对于一位要负责1000多台服务器的操作员来说，需要一种比较全面的监控基础设施和一定程度的自动化，以帮助调度事件。

469

Google部署监测软件来跟踪所有服务器和网络设备的健康状况。诊断程序无时无刻不在运行。当系统发生故障时，许多可能出现的问题都有简单的自动化解决方案。在这种情况下，下一步是重启计算机，然后尝试重新安装软件组件。因此，这一过程会处理大多数故障。

经过前面几步未能修复的机器被添加到待修机队列中。问题的诊断情况与故障机的ID一起被放在队列中。

为了分摊修复费用，发生故障的机器由修理技师分批解决。当诊断软件确信自己的评估时，该组件部分会立即被替换，无须经过人工诊断过程。例如，如果诊断软件说某个存储节点的第3号磁盘损坏，则立即更换该磁盘。对于没有诊断信息或诊断信息可信度较低的故障机器，需要进行人工诊断。

目标是在任意时间内，人工修理队列中的节点数低于全部节点的1%。节点在修理队列中停留的平均时间为一周，尽管由修理工人修复它所需要的时间要短得多。较长的延迟时间体现了修理吞吐量的重要性，它影响到了操作成本。注意，自动化修复的第一步骤需要几分钟来重新启动或重新安装，也可能需要几个小时来运行直接压力测试，以确保该机器能够真正胜任工作。

这些延迟没有考虑使故障服务器空闲下来的时间。原因在于：节点中的状态量是一个很大的变数。如果在替换一个节点之前需要先将它的数据清空，那一个无状态节点所需要的时间要比存储节点短得多。

6.7.6 小结

到2007年，Google已经演示了几种用于提高其WSC能耗效率的创新技术，使Google A的PUE达到1.23。

- 除了提供一些廉价的外壳来封装服务器之外，经过改装的货运集装箱还将冷热空气通风道隔离开来，可以帮助降低服务器进风口的温度扰动。由于最糟糕情况下的热区没有那么严重，所以允许冷空气温度稍高。
- 这些集装箱还缩短了空气循环流通的距离，减少了移动空气的能耗。

- 让服务器运行在较高温度下，意味着只需要将空气冷却到 81℉（27℃），而不是通常的 64℉至 71℉（18℃至 22℃）。
- 冷空气的目标温度较高，有助于更经常地将设施保持在通过蒸发冷却系统（冷却塔）所能维持的温度范围内，比传统的冷凝器更具能耗效率。
- 将 WSC 部署在适宜气候下，可以在一年中的较长时间内仅利用蒸发式冷却系统。
- 470 部署全面的监控软硬件，测量 PUE 的实际值与设计值，以提高运行效率。
- 实际运行的服务器要多于配电系统在最差情况下所能支持的服务器，这意味着：由于从统计学的角度来看，不太可能有数千台服务器同时处于高度繁忙状态，而是依靠监测系统，在它们不可能完成任务时卸载其中一部分[Fan，Weber 和 Barroso 2007]、[Ranganathan 等人 2006]。PUE 提高是因为此设施的工作状态接近于其满载设计容量，由于服务器和冷却系统不具备能耗比例特性，所以这种状态是最高效状态。这样提高利用率后，就可以降低对新服务器和新 WSC 的需求。
- 将主板设计为仅需要一个 12 伏的电源，这样只需要每个服务器的标准电池就可以提供 UPS 功能，不再需要电池间，从而降低了成本，减少了 WSC 中的一个配电低效源。
- 细致周到地设计服务器主板，提高其能耗效率。例如，降低这些微处理器前端总线的时钟频率，在没有明显性能影响的情况下降低能耗。（注意，这些优化不会影响 PUE，但可以降低总 WSC 能耗。）

这些年来 WSC 设计肯定有了改进，Google 最佳 WSC 的 PUE 已经从 Google A 的 1.23 降至 1.12。Facebook 在 2011 年宣布，在它们的新数据中心中已经将 PUE 降至 1.07（见 http://opencompute.org/）。了解哪些创新仍然在进一步提高 WSC 效率是一件很有意义的事情，以使我们成为优秀的环境保护者。也许在将来，我们还会考虑 WSC 内部设备的制造能耗成本[Chang 等人 2010]。

6.8 谬论与易犯错误

尽管 WSC 还不到 10 岁，但诸如 Google 的 WSC 架构师已经发现了有关 WSC 的许多谬论和易犯错误，通常会把人们引领到一条艰难的道路上。我们在引言中已经说过，WSC 架构师就是当今的 Seymour Crays。

谬论 *云计算提供商在赔钱。*

关于云计算有一个很常见的问题：以如此之低的价格，能否赢利呢？

根据表 6-8 的 AWS 定价，我们可能要花费每个服务器 0.68 美元/小时的计算费用。（0.085 美元/小时的价格等价于一个 EC2 计算单元的虚拟机的价格，不是整个服务器。）如果我们可以销售 50%的服务器运行时间，那每台服务器每小时就可以产生 0.34 美元的收入。（注意，即使客户很少使用他们占用的服务器，也要支付费用，所以销售 50%的服务器运行时间并不一定意味着平均服务器利用率为 50%。）

471 另一种计算收入的方法是利用 AWS **保留实例**，在这种方案中，客户支付一定的年费来保留一个实例，然后再按小时来支付使用费用，其费率通常要低一些。将这些收费结合起来，在一个整年中，AWS 的每台服务器在每个小时内将产生 0.45 美元的收入。

如果我们能够以 AWS 价格为每台服务器售出 750 GB 的存储容量，那除了计算收入之外，每台服务器每个月还会另外产生 75 美元的收入，也就是 0.1 美元/小时。

这些数字意味着：每台服务器每小时的平均收入为 0.44 美元（通过按需实例方式）至 0.55 美元/小时（通过保留实例方式）。根据表 6-6，我们计算出 6.4 节 WSC 中每台服务器的费用为 0.11 美元/小时。尽管表 6-6 中的成本只是一些估计值，并非基于 AWS 的实际成本，而且销售 50%的处理器处理时间以及每台服务器存储达到 750 GB 的利用率，这些都只是例子，但这些假设仍然能够间接表明毛利润在 75%至 80%之间。假定这些计算是合理的，那就意味着云计算是可赢利的，对于服务企业尤为如此。

谬论 *WSC设施的构建成本高于它所容纳的服务器。*

尽管粗略浏览表 6-6 可能会让你得到上面的结论，但它忽略了整个 WSC 中每一部分的成本分摊时间。整个设施会持续 10 至 15 年，而服务器每 3 至 4 年就需要重新购买一次。分别利用表 6-6 中 10 年和 13 年的分摊时间，整个设施在 10 年间的资本支出为 7200 万美元，而服务器则为 3.3 ×6700 万美元，即 2 亿 2100 万美元。因此，WSC 中服务器在 10 年中的构建费用要比 WSC 设施高 3 倍。

易犯错误 *尝试通过非活跃低功率模式使耗用功率低于活跃低功率模式。*

图 6-1 显示服务器的平均利用率介于 10% ~ 50%之间。在关注 6.4 节 WSC 的运行成本时，你可能会认为低功率模式也许会很有帮助。

第 1 章曾经提到，我们无法在这种非活跃低功率模式中访问 DRAM 或磁盘，因此，无论读写速度多么缓慢，都必须返回完全活跃模式才能完成。这一易犯错误在于：返回完全活跃模式所需要的时间和能耗降低了非活跃低功率模式的吸引力。图 6-1 显示，几乎所有服务器的平均利用率都至少为 10%，所以我们可以预测它长期处于低活跃状态，而不是长时期的非活跃状态。

作为对比，处理器仍然以低功率模式运行，其速率是常规速率的微小倍数，所以活跃低功率模式更容易使用。注意，处理器返回完全活跃模式的时间也是以微秒来测量的，所以活跃低功率模式还解决了有关低功率模式的延迟问题。

易犯错误 *在尝试提高WSC性价比时使用功能较差的处理器。*

Amdahl 定律对 WSC 仍然适用，这是因为对于每个请求都会有某一连续作业，如果它在缓慢的服务器上运行，可能会增大请求延迟[Hölzle 2010]，[Lim 等人 2008]。如果这一连续作业增大了延迟，那么使用功能较差的处理器时，其成本中必须要增加一部分软件开发成本，以对代码进行优化，将其延迟降至原来的较低水平。许多缓慢服务器上有更多的线程，也更难以进行调度和实现负载均衡，因此，线程性能的可变性可能导致更长时间的延迟。如果只有 10 项任务，那千分之一的不良调度几率可能不会造成什么问题，但如果有 1000 项任务，那就的确是一个问题了，你必须等待那个耗时最长的任务。许多较小的服务器也可能会导致利用率较低，显然，需要调度的内容越少，调度起来越轻松。最后，如果问题的划分过于精细，那即使一些并行算法的效率也可能会很低。Google 的经验方法是使用低端服务器类计算机[Barroso 和 Hölzle 2009]。

472

作为一个具体例子，Reddi 等人[2010]对比了运行 Bing 搜索引擎的嵌入微处理器（Atom）和服务器微处理器（Nehalem Xeon）。他们发现，在 Atom 上运行查询的延迟大约是 Xeon 上的 3 倍。此外，Xeon 更可靠一些。当 Xeon 上的负载增加时，服务质量会逐渐、适度下降。而 Atom 会因为尝试吸收增加的负载，而快速偏离其服务质量目标。

这一特性直接影响了搜索质量。因为延迟对用户的重要性，所以如表 6-5 所示，如果查询

延迟还没有超过截止延迟，Bing 搜索引擎会使用多种策略对搜索结果进行精炼。更大型 Xeon 节点的延迟较低，这意味着它们可以用更多的时间来精炼搜索结果。因此，即使在几乎没有负载时，Atom 在 1%的查询中会给出弱于 Xeon 的搜索结果。而在正常负载时，会有 2%的搜索结果弱于 Xeon。

谬论　*由于DRAM可靠性及WSC软件容错性的提高，所以不需要在WSC的ECC存储器上投入更多。*

DRAM 的测试结果已经宣称其每兆位的故障率为 1000 至 5000 FIT（每 10 亿工作小时的故障数），而采用 ECC 会向每 64 位 DRAM 中添加 8 位，所以去除纠错码（ECC）有可能节省九分之一的 DRAM 成本[Tezzaron Semiconductor 2004]。

Schroeder、Pinheiro 和 Weber [2009]在两年半的时间里对带有 ECC 保护的 DRAM 进行了测试研究，这一工作主要在 Google 的 WSC 上完成，其中包含有数十万台服务器。他们发现测得的 FIT 率要比公布的错误率高 15～25 倍，即每兆位 25 000 到 70 000 次故障。这些故障影响到超过 8%的 DIMM，DIMM 每年平均有 4000 次可纠正错误和 0.2 次不可纠正错误。在服务器上的测量结果是，每年大约有三分之一的服务器遭遇过 DRAM 错误，平均 22 000 次可纠正错误和 1 次不可纠正错误。也就是说，对于三分之一的服务器，每 2.5 小时纠正一次存储器错误。注意，这些系统使用功能更强大的 CHIPKILL 代码，而不是简单的 SECDED 代码。如果使用更简单的机制，不可能纠正的错误率会提高至 4～10 倍。

在一个仅有奇偶校验位错误保护的 WSC 中，每发生一个存储器奇偶校验位错误，服务器都必须重新启动。如果重启时间为 5 分钟，那三分之一的机器会将 20%的时间花费在重启上！这一特性会将这个价值 1.5 亿美元的设施性能降低 6%。此外，这些系统可能会遇到许多不可纠正的错误，而操作人员根本就没有注意到这些错误的发生。

473

在前些年里，Google 使用了甚至没有奇偶校验保护的 DRAM。2000 年，在交付下一版搜索索引之前的测试过程中，它开始建议采用随机文档来回应文本查询[[Barroso 和 Hölzle 2009]。其原因是一些 DRAM 中的“固定 0”故障，它会损坏新的索引。Google 添加了一致性检验，以检测将来的此类错误。随着 WSC 规模的增大、ECC DIMM 价格变得更容易承受，ECC 已经成为 Google WSC 的标准。ECC 还有另外一个好处：可以在修复期间更轻松地找到损坏的 DIMM。

这些数据间接地表明了 Fermi GPU（第 4 章）为什么要向其存储器中添加 ECC，它的前辈甚至连奇偶位保护也没有。此外，这些 FIT 率使人们开始关注在 WSC 中使用 Intel Atom 处理器的努力（因为它的功率效率有所提高），2011 年的芯片组不支持 ECC DRAM。

谬论　*在低活跃期间关闭硬件可以提高WSC的性价比。*

表 6-7 显示，配电与冷却基础设施的分摊成本比每个月的全部电费高 50%。因此，压缩工作负载、关闭空闲机器当然可以节省一些费用，不过，即使你能节省一半的电能，也只能将每月的运行费用降低 7%。由于大范围的 WSC 监控基础设施需要能够探测设备，并查看其反应，所以还存在一些需要克服的实际问题。能耗比例与活跃低功率模式的另一个好处是它们与 WSC 监控基础设施兼容，使一位操作员能够负责 1000 多台服务器。

传统的 WSC 精髓是在低活动期间运行其他重要任务，以便弥补配电和冷却方面的投入。一个主要例子是创建搜索索引的批处理 MapReduce 作业。从低利用率中获取价值的另一个例子是 AWS 的现货定价，表 6-8 的标题中对其进行了介绍。如果 AWS 用户执行任务的时间比较灵

活，那就可以使用现货实例，让 AWS 更灵活地调度这些任务，比如安排在 WSC 利用率较低的时间执行，这样就可以节省计算费用，降低因数在 2.7 ~ 3 之间。

谬论 *用闪存替换所有磁盘可以提高WSC的性价比。*

对于一些 WSC 工作负载，比如进行大量随机读写的负载，闪存的速度要比磁盘快得多。例如，Fackbook 在其 WSC 中部署了封装为固态磁盘（SSD）形式的闪存，作为写回缓存，称为 Flashcache，它是 WSC 文件系统的一个组成部分。将访问频繁的文件存储在 Flash 中，访问较少的文件放在磁盘上。但是，由于 WSC 中的所有性能改进都必须从性价比的角度来评价，在用 SSD 替换所有磁盘之前，我们关注的是每美元的实际 I/O 数／秒，以及每美元的存储容量。在第 2 章已经看到，每 GB 闪存的成本至少比磁盘高 20 倍：2 美元/GB 比 0.09 美元/GB。 474

Narayanan 等人[2009]通过模拟小型与大型数据中心的工作负载轨迹，研究了从磁盘向 SSD 迁移工作负载的效果。他们的结论是：由于 SSD 每美元的存储容量很低，所以对于任何工作负载的成本效率都很低。为了达到平衡点，闪存存储设备需要将每美元的容量数提高 3 ~ 3000 倍，具体取决于工作负载。

即使将功率代入公式中，对于那些不经常访问的数据，也很难证明用闪存代替磁盘是合理的。根据 6.4 节每年 2 美元/瓦的经验值，一个 1 TB 的磁盘耗电大约 10 瓦，通过降低能耗，每块磁盘每年最多可以节省 20 元。但是，2011 年 1 TB 闪存的 CAPEX 成本为 2000 美元，而磁盘仅为 90 美元。

6.9 结语

WSC 的计算机架构师继承了构建世界上最大计算机的头衔，未来的 IT 设备将使移动客户端变得完善，而 WSC 的架构师正在设计其中的一大部分。我们中有许多人一天会多次使用 WSC，每天使用 WSC 的次数、使用 WSC 的人数在接下来的 10 年中一定会增长。这个星球上的近 70 亿人中，已经有超过一半的人拥有移动电话。这些设备都已经为使用因特网做好了准备，将有更多来自世界各地的人们能够从 WSC 中获益。

此外，通过 WSC 实现的规模经济已经实现了长久以来将计算作为公用设施的目标。云计算意味着任何一个想出好主意、好业务模型的人可以在任何地方使用数千台服务器，几乎在瞬间展现出其想像力。当然，关于标准、隐私和因特网带宽增长速度等方面还有一些重要的障碍会限制云计算的发展，但我们预测它们将会得到解决，使云计算进入繁荣期。

由于每个芯片上核心数目的增长（见第 5 章），集群将会增长到包含数千个核心。我们相信为运行 WSC 而开发的技术对于集群也是有用的，所以集群将会运行那些为 WSC 开发的虚拟机和系统软件。一个好处是可以很轻松地支持“混合式”数据中心，在这种数据中心中，可以很容易地将工作负载交给云中进行处理，然后再收缩回来，仅依靠本地计算进行处理。

云计算有许多富有吸引力的特性，其中之一就是从经济的角度来鼓励人们节省能源。对于给定的基础设施投入成本，很难说服云计算供应商关闭未使用设备而节省能量，不过，要说服云计算使用者放弃空闲实例是很轻松的，因为无论他们是不是在做有用的事情，都要为所使用的实例支付费用。与此类似，按使用情况收费还会鼓励程序员高效地利用计算、通信和存储，如果没有一种容易理解的定价方案，是很难有很大动力的。由于这种明确的定价方案使成本的测量现在变得很容易，而且测量结果也是可信的，所以研究人员就有可能从性价比的角度对创 475

新技术进行评估，而不只是评估其性能。最后，云计算意味着研究人员可以在数千台计算机的规模上评估自己的想法，这在过去只有大型公司才能承受得起。

我们相信 WSC 正在改变服务器设计的目标和原理，就像移动客户端的需求正在改变处理器设计的目标和原理一样。这两者也都对软件行业产生革命性的影响。每美元实现的性能和每焦耳实现的性能推动着移动客户端硬件和 WSC 硬件的发展，并行是实现这些目标的关键。

未来世界是令人兴奋的，而架构师在它的各个方面都将扮演至关重要的角色。我们期待着见证和使用将要到来的一切。

6.10　历史回顾与参考文献

附录 L.8 节介绍了集群的发展，它是 WSC 和公用计算的基础。（希望了解更多知识的读者可以先阅读 Barroso 和 Hölzle [2009]的参考文献，以及 James Hamilton 的博客与访谈：http://perspectives.mvdirona.com）。

案例研究与练习（Parthasarathy Ranganathan 设计）

案例研究 1：影响仓库级计算机设计决策的总拥有成本

本案例研究说明的概念

- 总拥有成本（TCO）
- 服务器成本与功率对整个 WSC 的影响
- 低功率服务器的优缺点

总拥有成本是衡量仓库级计算机（WSC）有效性的一个重要度量。TCO 包含 6.4 节介绍的 CAPEX 和 OPEX，反映了整个数据中心为获得特定性能级别所需要的拥有成本。考虑到不同的服务器、网络和存储体系结构，数据中心的拥有者经常把 TCO 作为一个重要的对比度量指标，以判断哪些选项是最佳的；但是，TCO 是一个考虑了许多不同因素的多维计算。这一案例研究的目标是详细研究 WSC、不同的体系结构如何影响 TCO、TCO 如何推动经营者的决策。本案例研究将使用来自表 6-6 和 6.4 节的数字，并假定所述 WSC 达到了经营者的目标性能级别。TCO 经常用于对比拥有多种规模的不同服务器选项。本案例研究中的练习如何在 WSC 的上下文中进行此种对比，以及在作出此类决策时所涉及的复杂性。

476

6.1　[5/5/10] <6.2、6.4>这一章已经讨论了数据级并行，将其作为 WSC 针对大型问题实现高性能的一种方法。可以想到，利用高端服务器可以获得更高的性能；但是，更高性能的服务器通常伴随着以非线性增长的价格。

a. [5] <6.4>假定服务器在同等利用率下快 10%，但贵 20%，WSC 的 CAPEX 为多少？

b. [5] <6.4>如果这些服务器使用的电力还要多出 15%，OPEX 为多少？

c. [10] <6.2、6.4>给定以上速度与电力的增加比例，新服务器的成本必须为多少才能与原集群具有可比性？（提示：根据这一 TCO 模型，可能需要改变这一设施的关键负载。）

6.2　[5/10] <6.4、6.8>为了获得较低的 OPEX，一种富有吸引力的替代方法是使用服务器的低功率版本，以减少运行服务器所需要的总电能；但是，与高端服务器类似，高端组件的低功率版本也会有非线性权衡。

a. [5] <6.4、6.8>如果低版本服务器选项在提供相同性能的情况下可以降低 15%的功率，但要贵 20%，这是不是一种好的权衡？

b. [10] <6.4、6.8>这些服务器的成本为多少时才能与原集群具有可比性？如果电能的价格加倍，又应为多少？

6.3 [5/10/15] <6.4、6.6>具有不同运行模式的服务器提供了在集群中动态运行不同配置的可能性，以匹配工作负载的使用。对于一种给定的低功率服务器，使用表 6-11 中关于功率/性能模式的数据。

a. [5] <6.4、6.6>如果服务器操作人员决定以中等性能运行服务器来节省电力成本，需要多少服务器才能实现相同级别的性能？ 477

b. [10] <6.4、6.6>这种配置的 CAPEX 和 OPEX 为多少？

c. [15] <6.4、6.6>如果有一种替代方法，一台便宜 20%的服务器，但速度更慢、消耗的电能更少，给出性能-功率曲线，使其提供与基准服务器具有可比性的 TCO。

表6-11 低功率服务器的功率性能模式

模　式	性　能	功　率
高	100%	100%
中	75%	60%
低	59%	38%

6.4 [讨论] <6.4>讨论练习 6.3 中两种选项的权衡与好处，假定在这些服务器上运行的工作负载是恒定的。

6.5 [讨论] <6.2、6.4>WSC 与高性能计算（HPC）集群不同，经常会在一天之内经历大幅的工作负载波动。讨论练习 6.3 中两个选项的折中和好处，这一次假定工作负载是变化的。

6.6 [讨论] <6.4、6.7>到目前为止，我们对给出的 TCO 模型进行了一些抽象，省略了大量低级细节。讨论这些抽象对 TCO 模型整体准确度的影响。什么时候进行这些抽象是安全的？在哪些情况下，提供更多的细节会给出明显不同的答案？

案例研究 2：WSC中的资源分配与TCO

本案例研究说明的概念

- WSC 中的服务器与电力供给
- 工作负载的时变特性
- 这些变化对 TCO 的影响

在部署高效 WSC 时的一些关键挑战包括：正确供给资源、在最大限度上利用这些资源。由于 WSC 的规模以及所运行工作负载的潜在变化，这一问题可能非常复杂。本案例研究中的练习说明对资源的不同应用会如何影响 TCO。

6.7 [5/5/10] <6.4>为 WSC 供给资源时的一个挑战是：在给定设施规模的前提下，正确确定功率负载。如本章所述，铭牌功率经常是很少遇到的峰值。

a. [5] <6.4>如果铭牌功率为 200 瓦，成本为 3000 美元，估计每台服务器的 TCO 如何变化。

b. [5] <6.4>另考虑一种功率更高，但价格更低的选项，其功率为 300 瓦，成本为 3000 美元。

c. [10] <6.4>如果服务器的实际平均功率只是铭牌功率的 70%，每台服务器的 TCO 如何变化？

6.8 [15/10] <6.2、6.4>TCO 模型中有一个假定：此设施的临界负载是固定的，服务器的数量与这一临界负载相吻合。实际上，由于服务器功率会根据负载发生变化，所以设施使用的临界功率可能在任意给定时间发生变化。经营者最初必须根据其临界功率资源以及数据中心组件所用功率量的估计值来为数据中心提供供给。 478

a. [15] <6.2、6.4>扩展此 TCO 模型，以根据铭牌功率为 300 瓦的服务器为 WSC 提供初始供给，并计算所使用的实际月度临界功率和 TCO，假定服务器的平均利用率为 40%，消耗 225 瓦。未使用的容量为多少？

b. [10] <6.2、6.4>用铭牌功率为 500 瓦的服务器、平均利用率为 20%、消耗 300 瓦来重复这一练习。

6.9 [10] <6.4、6.5>6.5 节曾经提到，在使用 WSC 时经常与终端用户进行交互。这种交互性应用经常会导致一天之内不同时间的波动，其峰值与特定的时间段相关。例如，对于 Netflix 租用来说，在晚上 8 ~ 10 点会有一个峰值；一天之内不同时间造成的这些影响在整体上是非常显著的。对比数据中心在拥有不同容量时每台服务器的 TCO，一种容量值与早晨 4 点的利用率相对应，另一种容量值与晚上 9 点的利用率相匹配。

6.10 [讨论/15] <6.4、6.5>讨论一些选项，以便在非峰值时间内更好地利用多余的服务器，或者讨论一些可以节省成本的选项。由于 WSC 的交互性本质，为了积极地缩减电能的使用，会面对哪些挑战？

6.11 [讨论/25] <6.4、6.6>给出一种可行方法，通过专门降低服务器功率来提高 TCO。在评估这一方法时会有哪些挑战？根据你的提议，估计 TCO 的增长。其优缺点有哪些？

练习

6.12 [10/10/10] <6.1>推动 WSC 发展的一个重要因素就是有丰富的请求级并行，而不是指令级或线程级并行。这个问题探讨不同类型的并行对计算机体系结构和系统设计有着什么样的影响。

a. [10] <6.1>讨论一些情景，其中提高指令级或线程线并行所得到的好处可大于通过请求级并行所能实现的好处。

b. [10] <6.1>增大请求级并行对软件设计有什么样的影响？

c. [10] <6.1>提高请求级并行可能存在哪些缺点？

6.13 [讨论/15/15] <6.2>当一个云计算服务提供商接到一些包含多个虚拟机（VM）的作业时（比如 MapReduce 作业），会有许多调度选项。可以用轮询方式来调度这些 VM，将其分散在所有可用处理器和处理器上，也可以对它们进行整合，以尽量减少所使用的处理器数目。利用这些调度选项，如果提交了一项拥有 24 个 VM 的作业，并且云中有 30 个处理器可供使用（每个处理器最多可以运行 3 个 VM），轮询过程将使用 24 个处理器，合并后的调度过程将使用 8 个处理器。调度程序还可以在不同范围内寻找可用处理器核心，即：插槽、服务器、机架和机架阵列。

479

a. [讨论] <6.2>假定所提交的作业都是计算密集型工作负载，可能会有不同的存储器带宽需求，就电力与冷却成本、性能和可靠性而言，轮询与合并调度的优缺点都有哪些？

b. [15] <6.2>假定所提交的作业都是 I/O 密集型工作负载，轮询与合并调度在不同范围内的优缺点有哪些？

c. [15] <6.2>假定所提交的作业都是网络密集型工作负载，轮询与合并调度在不同范围内的优缺点有哪些？

6.14 [15/15/10/10] <6.2、6.3>MapReduce 在多个节点上运行与数据无关的任务，从而开发出大量并行，通常使用的是大众化商品硬件；但是，对于并行级别也存在一些限制。例如，为实现冗余，MapReduce 会将数据块写到多个节点，占用磁盘，还可能占用网络带宽。假定数据集的总大小为 300 GB，网络带宽为 1 Gbps，10 s/GB 映射速率、20 s/GB 约简率。还假定必须从远程节点读取 30%的数据，每个输出文件被写到其他两个节点，以实现冗余。所有其他参数采用表 6-4 中的数据。

a. [15] <6.2、6.3>假定所有节点都在同一机架内。采用 5 个节点时的预期运行时间为多少？10 个节点、100 个节点、1000 个节点呢？讨论每种节点大小的瓶颈。

b. [15] <6.2、6.3>假定每个机架有 40 个节点，任意远程读取/写入进入任意节点的机会相等。100 个节点的预期运行时间为多少？1000 个节点呢？

c. [10] <6.2、6.3>一个重要的考虑因素是尽可能减少数据移动。在从本地到机架、到阵列的访问速度大幅减缓时，必须对软件进行有效优化，尽量提高局域特性。假定每个机架有 40 个

节点，在 MapReduce 作业中使用了 1000 个节点。如果远程访问在 20%的时间都不超出同一机架，则运行时间为多少？50%的时间呢？80%的时间呢？

d. [10] <6.2、6.3>给定 6.2 节中的简单 MapReduce 程序，讨论一些可能的优化方法，使工作负载的局域性达到最大。

6.15 [20/20/10/20/20/20] <6.2>WSC 程序员经常使用数据复制来克服软件中的故障。比如，Hadoop HDFS 采用三路复制（一个本地副本、机架内的一个远程副本、另一机架内的一个远程副本），但值得研究一下何时需要这些复制。

a. [20] <6.2>Hadoop World 2010 参与者调查表明：超过半数 Hadoop 集群的节点数不超过 10 个，数据集大小不超过 10 TB。使用表 6-1 中的故障频率数据，在采用一路、两路和三路复制时，10 节点 Hadoop 集群的可用性如何？ 480

b. [20] <6.2>假定有表 6-1 中的故障数据和一个 1000 节点的 Hadoop 集群，在采用一路、两路和三路复制时，它的可用性如何？

c. [10] <6.2>复制的相对开销随每个本地计算机在每个小时内写入的数据量变化。对于一个 1000 节点、对 1 PB 数据进行排序的 Hadoop 作业，计算其额外的 I/O 通信流量和网络流量（机架内和跨机架），其中数据混洗的中间结果被写到 HDFS。

d. [20] <6.2>利用表 6-4，计算两路与三路复制的时间开销。使用表 6-1 所示的故障率，对比在没有复制与两路及三路复制时的预测执行时间。

e. [20] <6.2>现在考虑一个向日志应用复制操作的数据库系统，假定每个事务平均访问一次硬盘，生成 1 KB 的日志数据。计算两路与三路复制的时间开销。如果该事务在存储器内执行，耗用 10 μs，结果又会如何？

f. [20] <6.2>现在考虑一个采用 ACID 一致性的数据库系统，它需要两次网络往返进行两阶段确认。为保持一致性和进行复制所需要的时间开销为多少？

6.16 [15/15/20/15/] <6.1、6.2、6.8>尽管请求级并行允许许多计算机并行处理同一问题，从而可以实现更高的整体性能，但它面对的一个挑战就是避免将问题划分得过于精细。如果在服务级协议（SLA）的上下文中来研究这一问题，通过进一步划分来降低问题规模，可能需要更多的工作量才能实现目标 SLA。假定一个 SAL 要求 95%的请求在 0.5 秒或更短时间内得到响应，那类似于 MapReduce 的并行体系结构可以启动多个冗余作业，以获得相同结果。对于以下问题，假定查询响应时间曲线如图 6-13 所示。此曲线根据每秒执行的查询数目，显示了基准服务器以及使用缓慢处理器模型的“小型”服务器的响应延迟。

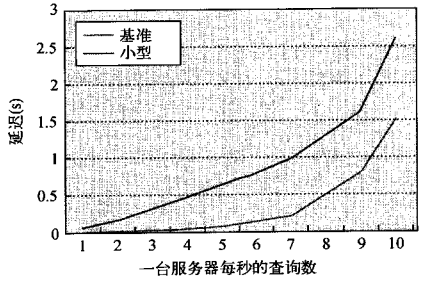


图 6-13 查询响应时间曲线

a. [15] <6.1、6.2、6.8>假定 WSC 每秒接收 30 000 个查询，查询响应时间曲线如图 6-13 所示，那么需要多少个服务器来实现该 SLA？给定这一响应时间概率曲线，需要多少个“小型”服

务器来实现这一 SLA？如果仅关注服务器成本，对于目标 SLA 而言，“廉价”服务器必须比正常服务器便宜多少才能实现该成本优势？

b. [15] <6.1、6.2、6.8>由于采用了更便宜的组件，所以“小型”服务器的可靠性通常更差一些。使用图 6-1 中的数字，假定由于计算机廉价、存储器不良所导致的事件数提高 30%。现在需要多少“小型”服务器？这些服务器必须比标准服务器便宜多少？

481

c. [20] <6.1、6.2、6.8>现在假定有一个批处理环境。“小型”服务器提供的总性能为常规服务器的 30%。仍然假定练习 6.15 b 部分中的可靠性数字，需要多少“廉价”节点才能提供一个 2400 节点标准服务器阵列的同等预期吞吐量？假定阵列性能与节点规模之间具有完美的线性扩展关系，每个节点的平均任务长度为 10 分钟。如果此扩展为 80%呢？60%呢？

d. [15] <6.1、6.2、6.8>这一扩展通常不是线性函数，而是对数函数。一个很自然的想法可能是购买更大型的节点，使每个节点拥有更强的计算能力，从而使阵列大小降至最低。讨论这一体系结构的一些优缺点。

6.17 [10/10/15] <6.3、6.8>高端服务器中的一个趋势是在存储器层次结构中包含非易失性闪存，或者是通过固态磁盘（SSD）形式，或者是通过 PCI Express 卡。典型 SSD 的带宽为 250 MB/s，延迟为 75 μs，而 PCIe 卡的带宽为 600 MB/s，延迟为 35 μs。

a. [10]根据图 6-3，并在本地服务器层次结构中包含这些点。假定在不同层次级别可以实现与 DRAM 相同的性能扩展因数，当跨机架访问时，这些闪存装置的性能如何？如果是跨阵列呢？

b. [10]讨论一些基于软件的优化方式，以利用存储器层次结构的这个新级别。

c. [25]重复（a）部分，但这次假定每个节点有一个 32 GB 的 PCIe 卡，能够缓存 50%的全部磁盘访问。

d. [15]根据“谬论与易犯错误”（见 6.8 节）中的讨论，用 SSD 代替所有磁盘并不一定是一种具有高成本效率的策略。假定有 WSC 经营者使用 SSD 来提供云服务。讨论一些利用 SSD 或其他闪存会有所帮助的情景。

6.18 [20/20/讨论] <6.3>**存储器层次结构**：在某些 WSC 设计中广泛使用缓存来降低延迟，有许多缓存选项可用于满足不同的访问模式和需求。

a. [20]考虑一些设计选项，用于以流式获取来自 Web（例如，Netflix）的丰富媒体。首先，我们需要估计电影数、每部电影的编码格式数、当前正在观看的用户数。2010 年，Netflix 有 12 000 部线上媒体，每部至少有四种编码格式（分别为 500、1000、1600 和 2200 kbps）。我们假定整个网站同时有 100 000 位观看者，每部电影平均长 1 个小时。估计总存储容量、I/O 与网络带宽以及与视频流相关的计算需求。

b. [20]每位用户、每部电影以及所有电影的访问模式与引用局域性特性如何？（提示：是随机还是顺序，时域与空域局域性是好还是差，工作集的大小是较小还是较大。）

482

c. [讨论]利用 DRAM、SSD 和硬盘，存在哪些电影存储选项？对比它们的性能和 TCO。

6.19 [10/20/20/讨论/讨论] <6.3>考虑一个社交网站，有 1 亿活跃用户张贴有关自己的更新（以文本和图片形式），他们在社交网络上浏览更新并进行互动。为了降低延迟，Facebook 和许多其他网站都使用了 memcached 作为缓存层，放在后端存储/数据库层之前。

a. [10] 估计每位用户和整个网站的数据生成与请求率。

b. [20] 对于这里讨论的社交网站，需要多少 DRAM 来托管其工作集？使用各拥有 96 GB DRAM 的存储器，估计需要多少本地、远程存储器访问来生成一位用户的主页？

c. [20] 现在考虑两种备选的 memcached 服务器设计，一种使用传统的 Xeon 处理器，另一种使用较小的核心，比如 Atom 处理器。假定 memcached 需要大容量的物理存储器，但 CPU 利用率很低，这两种设计有哪些优缺点？

d. [讨论]今天，存储器模块和处理器紧密耦合在一起，通常需要增加 CPU 插槽数，以支持更大容量的存储器。请列举其他一些设计，能够提供大容量物理存储器，但不会按比例增大服务

器中的插槽数目。对比这些设计的性能、功率、成本和可靠性。

e. [讨论]同一用户的信息可以存储在 memcached 服务器和存储服务器中，可以采用不同方式对这些服务器进行物理托管。讨论 WSC 中以下服务器布局方式的优缺点：(1) memcached 服务器与存储服务器是同一服务器；(2) memcached 服务器和存储服务器位于同一机架的不同节点上；(3) memcached 服务器位于同一机架上，存储器服务器位于其他机架上。 483

6.20 [5/5/10/10/讨论/讨论] <6.3、6.6>**数据中心联网**：MapReduce 和 WSC 是一种功能强大的组合方式，可以应对大规模的数据处理；例如，2008 年，Google 使用 4000 台服务器和 48 000 块硬盘对 1 PB 记录进行排序，只用了 6 小时稍多一点的时间。

a. [5]从表 6-1 和相关文本中推断磁盘带宽，需要多少秒的时间将数据读入主存储器并写回排序后的结果？

b. [5]假定每台服务器有两个 1 Gbps 的以太网接口卡（NIC），WSC 交换机基础设施的超额认购系数为 4，需要多少秒的时间才能将 4000 台服务器上的整个数据集混洗完毕？

c. [10]假定网络传输是 PB 级排序的性能瓶颈，能否估计 Google 数据中心中的超额认购比？

d. [10]现在让我们研究拥有 10 Gbps 以太网（没有超额认购）的好处，比如使用 48 端口 10 Gbps 以太网（2010 年 Indy 排序基础测试获胜者 TritonSort 就是采用这一配置）。需要多少时间才能将 1 PB 数据混洗完毕？

e. [讨论]对比下面两种方法：(1) 采用高网络超额认购比的大规模扩展方法；(2) 采用高带宽网络的小规模系统。它们的潜在瓶颈是什么？就可伸缩性和 TCO 而言，它们有哪些优势和劣势？

f. [讨论]排序和许多重要的科学计算工作负载都是计算密集的，而许多其他工作负载则并非如此。列举三种不会从高速联网中获益的工作负载示例。对于这两类工作负载，你建议使用哪种 EC2 实例？

6.21 [10/25/讨论] <6.4、6.6>由于 WSC 的超大规模，根据需要运行的工作负载恰当地分配网络资源是极为重要的。不同的分配方法可能会对性能和总拥有成本产生严重影响。

a. [10]利用表 6-6 中给出的具体数字，每个访问层交换机的超额认购率为多少？如果超额认购率折半，对 TCO 有什么影响？如果加倍呢？

b. [25]如果工作负载受网络限制，那降低超额认购率可能会提高性能。假定一项 MapReduce 作业使用 120 台服务器，读取 5 TB 数据。假定如表 6-2 中 2009 年 9 月的读取/中间/输出数据比，并使用表 6-4 来确定存储器层次结构的带宽。关于数据读取，假定有 50%的数据是从远程磁盘读取的；其中，80%是从机架内读取，20%是从阵列中读取。对于中间数据和输出数据，假定 30%的数据使用远程磁盘；其中 90%在机架范围内，10%在阵列范围内。将超额认购率折半时，整体性能提高多少？如果超额认购率加倍，性能又变为多少？计算每种情况下的 TCO。 484

c. [讨论]我们看到每个系统正在向更多个核心发展的趋势。我们还看到光纤通信的应用越来越多（其带宽可能更高，能耗效率也有改进）。你认为这些及其他一些新兴技术趋势将会如何影响未来 WSC 的设计？

6.22 [5/15/15/20/25] <6.5>**认识 Amazon Web 服务的容量**：设想你是一家 Alexa.com 顶级网站的网站运行与基础设施管理员，正在考虑使用 Amazon Web 服务（AWS）。在决定是否迁移到 AWS 之前，需要考虑哪些因素？要使用哪些服务和实例类型，以及将会节省多少成本？可以使用 Alexa 和网站通信流量信息（例如，维基百科上提供了页面查看数据），以估计一个顶级网站接收到通信流量数目，或者可以从 Web 上选择一些具体示例，比如下面来自 2010 年 DrupalCon San Francisco 的示例：http://2bits.com/sites/2bits.com/files/drupal-single-server-2.8-million-page-views-a-day.pdf。这些幻灯片描述了一个 Alexa #3400 网站，它每天接收 280 万个页面视图，使用单个服务器。这台服务器有两个四核 Xeon 2.5 GHz 处理器，8 GB DRAM 和三个采用 RAID1 配置的 15 K RPM SAS 硬盘，每个月大约消耗 400 美元。这个网站大量使用缓存，CPU 利用率的变化范围为 50%～250%（大约有 0.5 至 2.5 个核心处于繁忙状态）。

a. [5] 查看可用的 EC2 实例（http://aws.amazon.com/ec2/instance-types/），哪些实例类型可以匹配或超出当前的服务器配置？

b. [15]查看 EC2 定价信息（http://aws.amazon.com/ec2/pricing/），选择最具成本效率的 EC2 实例（允许采用组合方式），在 AWS 上托管此网站。EC2 每个月的费用是多少？

c. [15]现在向公式中添加 IP 地址和网络通信流量的成本，并假定网站每天在因特网上传输的流量为 100 GB。该网站现在每月的费用是多少？

d. [20]AWS 还向新客户提供 1 年期的免费微实例，提供 15 GB 带宽，供进出 AWS 的通信流量使用。根据你对自己部门 Web 服务器的峰值、平均通信流量的估计，能否将它免费托管在 AWS 上？

e. [25]Netfix.com 是一个大得多的网站，它也已经将自己的流化基础设施和编码基础设施迁移到 AWS。根据它们的服务特性，Netflix 可以使用哪些 AWS 服务？用于何种目的？

6.23 [讨论/讨论/20/20/讨论] <6.4>表 6-5 给出了用户感受的响应时间对收入的影响，激发了在保持低延迟的情况下实现高吞吐量的需求。

485

a. [讨论]以 Web 搜索为例，有哪些可能缩短查询延迟的方法？

b. [讨论]可以收集哪些监控统计数字，以帮助理解时间花费在哪里？你计划怎样来实现这样一种监控工具？

c. [20]假定每个查询的磁盘访问数符合正态分布，其均值为 2，标准差为 3，需要哪种磁盘访问延迟使 95%的查询的延迟 SLA 为 0.1 秒？

d. [20]在存储器中进行缓存可以减少长延迟事件（比如，访问硬盘）的频率。假定稳态命中率为 4%，命中延迟为 0.05 秒，缺失延迟为 0.2 秒，进行缓存是否有助于满足 95%查询的延迟 SLA 为 0.1 秒的要求？

e. [讨论]缓存内容什么时候过时，甚至变得不一致？发生这种情况的频繁程度如何？可以怎样检测这种内容并使其失效？

6.24 [15/15/20] <6.4>典型电源单元（PSU）的效率随负载的变化而变化。例如，在带有 40%的负载时（例如，从 100 瓦 PSU 输出 40 瓦），PSU 效率大约为 80%，当负载介于 20%与 40%之间时，此效率为 75%，当负载低于 20%时，效率为 65%。

a. [15]假定有一个功率比例的服务器，其功率与 CPU 的利用率成比例，利用率曲线如图 6-1 所示。平均 PSU 效率为多少？

b. [15]假定此服务器为 PSU 采用了 2*N*冗余（即，将 PSU 数目加倍），以确保一个 PSU 发生故障时能够提供稳定电源。平均 PSU 效率为多少？

c. [20]刀片服务器供应商使用一种共享的 PSU 池，不仅用于提供冗余，还可以使 PSU 的数量与服务器的实际功率相匹配。HP c7000 封装为总共 16 个服务器使用了多达 6 个 PSU。在这种情况下，对于具有相同利用率曲线的服务器封装，平均 PSU 效率为多少？

6.25 [5/讨论/10/15/讨论/讨论/讨论] <6.4>**功率闲置**（Power stranding）一词是指提供给数据中心但未被使用的功率容量。考虑图 6-13 针对不同机器组给出的数据[Fan、Weber 和 Barroso 2007]。（注意，这篇论文中所说的“集群”就是本章所说的“阵列”。）

a. [5]在(1) 机架级别，(2) 配电单元级别和(3) 阵列（集群）级别的“闲置”功率为多少？在更大型的机器组中，功率容量超额认购的趋势如何？

b. [讨论]你认为是什么导致不同机器组之间的闲置功率有所不同？

c. [10]考虑一组阵列级别的机器，其中全部机器消耗的功率绝对不超过功率总和的 72%（有时将其称为“总和的峰值”和“峰值的总和”之比）。利用本案例研究的成本模型，通过对比一个针对峰值容量进行供电的数据中心和针对实际使用进行供电的数据中心，计算可以节省多少成本。

486

d. [15]假定数据中心设计者选择在阵列级别包含更多服务器，以充分利用闲置功率。利用图 6-14a 部分中的示例配置和假定，对于所提供的同等总功率，计算现在可以在此仓库级计算机中多包含多少服务器。

e. [讨论]在现实部署中，需要怎样才能使本题(d)部分中的优化生效？（提示：在很罕见的情况下，阵列中的所有服务器都耗用峰值功率，考虑在此情况下需要怎样才能划定功率上限。）

f. [讨论]可以设想两种策略来管理功率上限[Ranganathan 等人 2006]：(1) 先发式策略，功率预算为事先确定（“不要假定自己可以使用更多的功率，在此之前先行申请！”）或(2) 反应式策略，在超出功率预算时调整功率预算（“得到禁令之前，尽可根据需要任意耗用功率！”）。讨论这些方法的优缺点，以及在什么情况下使用哪一种方法。

g. [讨论]如果提高了系统的能耗比例特性（假定工作负载与表 6-3 中的工作负载相似），总闲置功率会有什么变化？

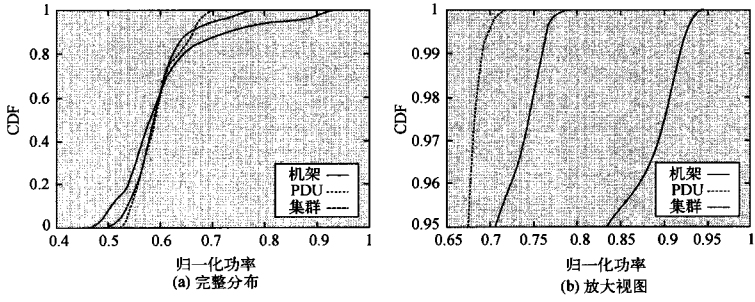


图 6-14 真实数据中心的累积分布函数（CDF）

6.26 [5/20/讨论] <6.4、6.7> 6.7 节讨论了在 Google 设计中使用各服务器的电池作为电源。让我们来研究这一设计的结果。

a. [5]假定利用电池作为微服务器级别 UPS 的效率为 99.99%，不再需要在整个设施范围内使用 UPS，后者的效率仅为 92%。假定分站交换的效率为 99.7%，PDU、递减变换及其他电力断路器的效率分别为 98%、98%和 99%。请计算为各个服务器使用电池备份使整个电力基础设计的效率提高了多少。

b. [20]假定 UPS 的成本占 IT 设备的 10%。其余假定采用本案例研究成本模型中的数据，电池的平衡点为多少（以其占单个服务器成本的比例表示），达到此平衡点时，采用电池解决方案时的总拥有成本优于设施范围内 UPS 的拥有成本。

c. [讨论]这两种方法之间进行了哪些其他折中？具体来说，你认为这两种不同设计的可管理性和故障模型会如何变化？

6.27 [5/5/讨论] <6.4>对于本练习，考虑一个用于计算 WSC 总运行功率的简化公式，如下所示：总运行功率=（1+冷却低效乘数）×IT 设备功率。

a. [5]假定一个 8 MW 数据中心的功率利用率为 80%，电力费用为 0.1 美元/千瓦时，冷却低效乘数为 0.8。对比以下优化方式所能节省的成本：(1) 将冷却效率提高 20%，(2) 将 IT 设备的能耗效率提高 20%。

b. [5]IT 设备能耗效率提高百分之多少才能与冷却效率提高 20%节省的成本相匹配？

c. [讨论/10]从服务器能耗效率和冷却系统能耗效率来看，关于这些优化的相对重要性可以得出哪些结论？

6.28 [5/5/讨论] <6.4>如本章中的讨论，WSC 中的冷却设备本身会消耗许多能量。通过主动控制温度，可以降低冷却成本。人们已经提出一种优化方式，在安排工作负载时对温度加以考虑，以控制温度，降低冷却成本。其思想是确定给定房间中的冷却分布图，将较热的系统与较冷的区域相

487

对应，从而降低了WSC级别的整体冷却需求。

a. [5] CRAC单元的性能系数（COP）定义为所清除的热量（Q）与清除该热量所做的功（W）之比。CRAC单元压入通风系统的空气温度升高时，CRAC单元的COP也随之增大。如果返回CRAC单元的空气温度为20摄氏度，当COP为1.9时，要清除10 KW的热量，需要在CRAC单元中消耗多少能量？如果要对同等数量的空气进行冷却，但这次返回温度为25摄氏度，取COP为3.1，则现在要在CRAC单元中消耗多少能量？

b. [5]假定某种工作负载分配算法能够将热工作负载很好地与冷区域相匹配，使计算机房空调（CRAC）单元能够运行在较高温度，以提高上面练习中的冷却效率。上述两种情况相比可以节省多少功率？

c. [讨论]鉴于WSC系统的规模，功率管理问题可能非常复杂，涉及多个方面。为提高能耗效率所采取的改进可以在硬件和软件中实施，可以在系统级、集群级针对IT设备或冷却设备实施，等等。在为WSC设计整体功耗效率解决方案时，考虑这些交互作用是非常重要的。考虑一种整合算法，它查看服务器利用率，并整合同一计算机上的不同工作负载类别，以提高服务器利用率。（如果系统不具备能耗比例特性，这一做法可能会使系统运行时具有较高的能耗效率。）这一优化如何与尝试使用不同功率状态的并发算法交互？请参见ACPI（高级配置功率接口）中的一些例子。关于WSC中可能相互冲突的多种优化方式，你能想到其他哪些例子？如何解决这一问题？

488

6.29 [5/10/15/20] <6.2>能耗比例特性（有时也称为“能耗比例缩减”）是系统的一种属性，当系统空闲时不消耗功率，当活跃性与所完成的工作量增加时，所消耗的功率会逐渐增加，并与其完成的工作量成比例。在这个练习中，我们将研究能量消耗对不同能耗比例模型的敏感性。在下面的练习中，除非另行指出，否则默认使用表6-3中的数据。

a. [5]推导能耗比例特性的一种简单方法是假定活跃性与所使用的功率之间存在线性关系。只要使用表6-3中的峰值功率和空闲功率，并利用线性插值，就可以画出当活跃性变化时的能耗效率趋势。（能耗效率表示为“性能/瓦”。）当空闲功率（即活跃度为0%时）为表6-3中所示数据的一半时，会出现什么情况？如果空闲功率为0，又会发生什么情况？

b. [10] 绘制在活跃性变化时的能耗效率趋势，使用表6-3中第3列的功率变化数据。假定空闲功率（仅这一指标）为表6-3所示数据的一半，绘制能耗效率。将这些曲线与上一练习中的线性模型进行对比。关于仅关注空闲功率的后果，可以得到哪些结论？

c. [15]假定如表6-3第7列的混合系统利用率数据。为简便起见，假定1000台服务器的利用率为离散分布，109台的利用率为0%，80台服务器的利用率为10%，等等。利用（a）部分和（b）部分的假定，计算对于这一混合工作负载的总性能和总能耗。

d. [20]有人可能设计一种系统，当负载级别介于0%～50%之间时，其功率与负载的关系满足亚线性。这种系统的能耗效率曲线可能在较低利用率时达到峰值（为高利用率为代价）。给出表6-3中第3列的一个新版本，用以展示这一能耗效率曲线。假定采用表6-3中第7列的混合系统利用率数据。为简单起见，假定1000台服务器的利用率为离散分布，109台的利用率为0%，80台服务器的利用率为10%，等等。计算对于这一混合工作负载的总性能和总能耗。

489

6.30 [15/20/20] <6.2、6.6>这个练习说明了能耗比例模型之间的一些相互作用，这些模型采用了一些优化方法，比如服务器整合与高能耗效率的服务器设计。考虑表6-12和表6-13所示的情景。

a. [15]考虑两台服务器，其功率分布如表6-12所示：情景A（图6-4中考虑的服务器）和情景B（其能耗比例特性低于情景A，但服务器的能耗效率高于情景A）。假定有表6-3中第7列的混合系统利用率。为简单起见，假定1000台服务器的利用率为离散分布，109台的利用率为0%，80台服务器的利用率为10%，等等，如表6-13第1行所示。假定其性能变化如表6-3第2列所示。对比这两种服务器类型在采用这一混合工作负载时的总性能和总能耗。

b. [20]考虑一个由1000台服务器组成的集群，其数据类似于表6-3所示数据（汇总在表6-12和

表 6-13 的第一行内）。根据这些假定，在此混合工作负载情况下的总性能和总能耗为多少？现在假定我们能够整合工作负载，对情景 C 中所示的分布进行建模（表 6-13 的第二行）。现在的总性能和总能耗是多少？某一系统具有线性能耗比例模型，其空闲功率为 0 瓦，峰值功率为 662 瓦，与这一系统相比，前面计算的总能耗如何？

c. [20]重复(b) 部分，但这一次采用服务器 B 的功率模型，并与(a) 部分的结果进行对比。

表6-12 两台服务器的功率分布

活跃度（%）	0	10	20	30	40	50	60	70	80	90	100
功率、情景A（W）	181	308	351	382	416	451	490	533	576	617	662
功率、情景B（W）	250	275	325	340	395	405	415	425	440	445	450

表6-13 集群中在无、有合并情景下的利用率分布

活跃度（%）	0	10	20	30	40	50	60	70	80	90	100
无、服务器、情景A和B	109	80	153	246	191	115	51	21	15	12	8
无、服务器、情景C	504	6	8	11	26	57	95	123	76	40	54

6.31 [10/讨论] <6.2、6.4、6.6>系统级能耗比例趋势。考虑一台服务器功率消耗的如下平衡点：

CPU，50%；存储器，23%；磁盘，11%；网络/其他，16%

CPU，33%；存储器，30%；磁盘，10%；网络/其他，27%

a. [10]假定一个 CPU 的动态功率范围为 3.0 倍（即，CPU 在空闲时的功率消耗是其峰值工作负载时功率消耗的三分之一。）假定上述存储器系统、磁盘和网络/其他类别的动态范围分别为 2.0、3.0 和 1.2 倍。这两种情况下，总系统的整体动态范围为多少？ 490

b. [讨论/10]从(a) 部分的结果中能够学到些什么？如何在系统级别实现更好的能耗比例特性？（提示：仅通过 CPU 优化不能实现系统级的能耗比例特性，而是需要对所有组件进行改进。）

6.32 [30] <6.4>Pitt Turner IV 等人[2008]对数据中心层的分类进行了很好的综述。层分类确定了网站基础设施的性能。为简单起见，考虑如表 6-14 所示的关键差别（摘自 Pitt Turner IV 等人[2008]）。使用此案例研究中的 TCO 模型，对比所显示的不同层对成本的影响。

表6-14 数据中心层分类概述

第1层	用于配电和冷却的单一路径，没有冗余组件	99.0%
第2层	（N+1）冗余=两个配电与冷却路径	99.7%
第3层	（N+2）冗余=三条配电与冷却路径，即使在维护期间也能正常工作	99.98%
第4层	两个活跃配电与冷却路径，每条路径有冗余组件，以容忍任何单一设备故障，不对负载产生影响	99.9995%

* 摘自 Pitt Turner IV 等人[2008]。

6.33 [讨论] <6.4>根据表 6-6 中的观察结果，关于宕机时间的收入损失和正常工作所需要的成本，可以定性地说点什么？

6.34 [15/讨论] <6.4>最近的一些研究定义了名为 TPUE 的度量，表示“真正 PUE”（true PUE）或“总 PUE”（total PUE）的意思。TPUE 定义为 PUE × SPUE。PUE 表示功率利用效率，在第 6.4 节中定义，为总设施功率与总 IT 设备功率之比。SPUE（服务器 PUE）是一个与 PUE 类似的新度量，但它不适用于计算设备，而是定义为总服务器输入功率与其有用功率之比，其中有用功率定义为直接参与计算的电子组件所消耗的功率，这些电子组件包括：主板、磁盘、CPU、DRAM、I/O 卡，等等。换句话说，SPUE 度量衡量的是一台服务器中所包含的电源、电压调节器和风扇等器件的效率不佳程度。

a. [15] <6.4>考虑一种设计，它为 CRAC 单元提供的空气温度较高。CRAC 单元的效率大约与

温度的4次方成正比，这一设计从而提高了整体PUE，假定提高了7%（假定基准PUE为1.7。）但是，服务器级别的较高温度会触发板载风扇控制器，使其控制的风扇转速高出许多。风扇功率是速度的三次函数，提高风扇速度会导致SPUE的下降。假定风扇的功率模型为：

491

$$风扇功率=284\times ns\times ns\times ns-75\times ns\times ns$$

其中，ns是归一化风扇转速=风扇转速/18 000（rps）。

基准服务器功率为350W。如果风扇转速(1)从10 000 r/min增加到12 500 r/min，(2)从10 000 r/min增加到18 000 r/min，计算SPUE。对比这两种情况下的PUE和TPUE。（为简单起见，忽略在SPUE模型中功率输出的低效程度。）

b. [讨论] (a)部分说明：尽管PUE对于衡量设备开销是一个出色的度量，但它并不能衡量IT设备本身的低效程度。你能否指出另外一种设计，其TPUE可能低于PUE？（提示：参见练习6.26。）

6.35 [讨论/30/讨论] <6.2>要衡量服务器中的能耗效率，以最近发布的两个基准测试作为起点是非常合适的：SPECpower_ssj 2008基准测试（可从http://www.spec.org/power_ssj2008/获取）和JouleSort度量（可从http://sortbenchmark.org/获取）。

a. [讨论] <6.2>查看关于这两个基准测试的描述。它们有哪些相似之处？有哪些不同之处？为了改进这两个基准测试，以更好地与提高WSC能耗效率的目标保持一致，可以做些什么？

b. [30] <6.2>JouleSort测量全部系统能耗，以执行核心外的排序，并尝试推导出一种度量，用来对比从嵌入式设备到超级计算机的各种不同系统。查看http://sortbenchmark.org中关于JouleSort的说明。下载该排序算法的公共使用版本，并在不同类型的计算机上运行它，比如便携式计算机、PC和移动电话等，或者采用不同配置运行该算法。从不同设置的JouleSort评价结果中可以学到些什么？

c. [讨论] <6.2>考虑在上述试验中获得最佳JouleSort评级的系统。如何提高能耗效率呢？例如，尝试重写该排序代码，以提高JouleSort评级。

6.36 [10/10/15] <6.1、6.2>表6-1是一个服务器阵列的停运列表。在应对大规模的WSC时，做到集群设计与软件体系结构的平衡是非常重要的，以在不增加大量成本的情况下实现所需要的正常运行时间。这个问题研究仅通过硬件来实现可用性的含义。

a. [10] <6.1、6.2>假定经营者希望通过仅改进服务器硬件来实现95%的可用性，必须减少多少各类事件？现在，假定通过冗余机器完全解决了个别服务器崩溃问题。

b. [10] <6.1、6.2>如果50%的时间内是通过冗余处理了个别服务器崩溃问题，(a)部分的答案将如何变化？20%的时间呢？0%的时间呢？

492

c. [15] <6.1、6.2>讨论软件冗余性对于实现高级可用性的重要性。如果WSC经营者考虑购买一些机器，这些机器略便宜一些，但可靠性也降低10%，这对软件体系结构可能有哪些影响？与软件冗余相关联的挑战有哪些？

6.37 [15] <6.1、6.8>查看标准DDR3 DRAM与带有纠错码（ECC）的DDR3 DRAM的当前价格。为了实现ECC所提供的更高可靠性，每比特的价格会提高多少？仅使用这些DRAM价格以及6.8节提供的数据，对于采用非ECC DRAM与有ECC DRAM的WSC，每美元的工作时间是多少？

6.38 [5/讨论] <6.1>WSC可靠性和可管理性方面的考虑因素。

a. [5]考虑一个服务器的集群，其中每台服务器的成本为2000美元。假定年度故障率为5%，每次修复的平均服务时间为1小时，每次故障更换零件时需要系统成本的10%，每台服务器的年度维护费用为多少？假定一个服务技师每小时收费为100美元。

b. [讨论]解释这一可管理性模型与传统企业数据中心可管理性模型的差别，在这一传统企业数据中心中，有大量中小型应用程序，分别运行在自己的专用硬件基础设施上。

493

A

指令集基本原理

An　将存储位置 n 中的数字加到累加器中。

En　如果累加器中的数字大于或等于零，则执行位于存储位置 n 的命令；否则顺序执行。

Z　停止运行计算机，并发出警告铃声。

——Wilkes 和 Renwick，
选自 EDSAC 的 18 条机器指令列表（1949）

A.1 引言

本附录主要介绍指令集体系结构——程序员或编译器编写人员能够看到的计算机部分。对于本书读者来说，这一材料中的大多数内容应当都是一种回顾；我们主要作为背景知识提供这些内容。本附录中介绍了指令集体系结构的各种设计选项。具体来说，我们主要关注四个主题。第一，对各种指令集进行了分类，并对各种方法的优势和劣势进行某种量化评估。第二，给出一些指令集测量数据，并对其进行分析，这些数据大体与特定的指令集无关。第三，讨论语言与编译器问题以及它们对指令集体系结构的影响。最后，“融会贯通”一节展示这些思想如何在 MIPS 指令集中得到反映，MIPS 指令集是一种典型的 RISC 体系结构。在附录的末尾是有关指令集设计的一些谬论和易犯错误。

为了进一步说明这些基本原理，附录 K 还给出了 4 种通用 RISC 体系结构的例子（MIPS、PowerPC、Precision Architecture、SPARC）、4 种嵌入式 RISC 处理器示例（ARM、Hitachi SH、MIPS 16、Thumb）和 3 种较早的体系结构（80x86、IBM 360/370 和 VAX）。在讨论如何对体系结构分类之前，需要先来谈谈指令集的测量。

在这个附录中，我们将研究大量体系结构方面的测试结果。显然，这些测量结果依赖于被测程序和用于进行这些测量的编译器。这些结果不应被认为是绝对的，如果用不同编译器或一组不同程序来进行测试，那可能会得到不同数据。我们相信本附录给出的测量值可以合理地代表一类典型的应用程序。许多测量是使用一小组基准测试给出的，以便适度地显示数据，并可能看出程序之间的差别。架构师在设计新计算机时，可能希望预先分析大量程序，再来作出体系结构方面的决策。所示测量通常是动态的——对一个被测事件的频率进行了加权，权重就是这一事件在被测程序执行期间出现的次数。

在开始讨论一般性基本原理之前，先来回顾一下第 1 章介绍过的三种应用领域。**桌面式计算**强调涉及整数、浮点类型的程序的性能，很少考虑程序的规模。例如，在五代 SPEC 基准测试中从来没有报告过代码规模。今天的**服务器**主要用于数据库、文件服务器和 Web 应用程序，再加上一些针对许多用户的时分应用程序。因此，浮点性能的重要性要远低于整数与字符字符串的重要性，不过几乎所有服务器处理器都仍然包含浮点指令。**个人移动设备**和**嵌入式应用**看重成本和能耗，所以代码规模非常重要，这是因为减少存储器就意味着可以降低价格和能耗，为了降低芯片成本，某些类型的指令（比如浮点）可能成为可选选项。

A-2

因此，所有这三种应用的指令集都是非常类似的。事实上，作为本附录主体内容的 MIPS 体系结构已经成功地运用在台式机、服务器和嵌入式应用中。

一种与 RISC 有很大不同的成功体系结构是 80x86（参见附录 K）。令人惊奇的是，它的成功并不一定与 RISC 指令集的优势相矛盾。由于与 PC 软件保持二进制兼容的重要性，再加上摩尔定律提供了充足的晶体管，使 Intel 在内部使用 RISC 指令集，而在外部支持 80x86 指令集。后来的 80x86 微处理器（比如 Pentium 4）使用硬件将 80x86 指令转换为类 RISC 的指令，然后在芯片内部执行经过转换的操作。它们仍然向程序员展现 80x86 体系结构的幻像，同时允许计算机设计人员实现 RISC 类型的处理器，以提高性能。

既然已经确定了背景，现在可以开始研究如何对指令集体系结构进行分类了。

A.2 指令集体系结构的分类

处理器中的内部存储类型是最基本的区别，所以在这一节，我们将主要关注体系结构中这

一部分的各种选项。主要选项包括栈、累加器或寄存器组。操作数可以显式命名，也可以隐式命名：在**栈体系结构**中，操作数隐式位于栈的顶部，而在**累加器体系结构**中，操作数隐式为累加器。**通用寄存器体系结构**只有显式操作数，或者为寄存器，或者为存储器位置。图 A-1 显示了此类体系结构的框图，表 A-1 显示了代码序列 C=A+B 在这三类指令集中通常是如何显示的。显式操作数也许可以直接从存储器访问，也可能需要首先加载到临时存储中，具体取决于体系结构的类别及特定指令的选择。

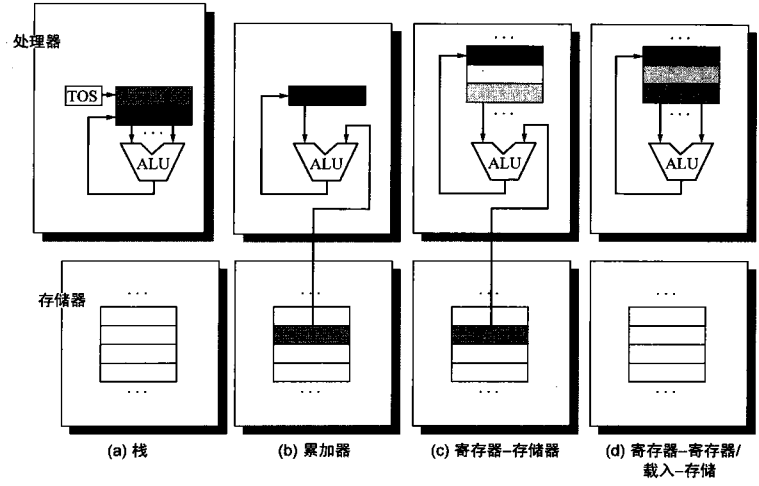


图 A-1 4 类指令集体系结构的操作数位置。这些箭头指示操作数是算术逻辑单元（ALU）操作的输入还是结果，或者既是输入又是结果。较浅的阴影表示输入，深色阴影表示结果。在(a)中，栈顶寄存器（TOS）指向顶部输入操作数，它与下面的操作数合并在一起。第一个操作数从栈中移走，结果占据第二个操作数的位置，对 TOS 进行更新，以指向结果值。所有操作数都是隐式的。在(b)中，累加器既是隐式输入操作数，也是结果。在(c)中，一个输入操作数是寄存器，一个在存储器中，结果保存在寄存器中。在(d)中，所有操作数都是寄存器，而且和栈体系结构类似，只能通过独立指令将其传送到存储器中：在(a)中为 push 或 pop，在(d)中为 load 或 store

表A-1 四类指令集中C=A+B的代码序列

栈	累加器	寄存器（寄存器-存储器）	寄存器（载入-存储）
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R3, R1, B	Load R2, B
Add	Store C	Store R3, C	Add R3, R1, R2
Pop C			Store R3, C

* 注意，对于栈和累加器体系结构，Add 指令拥有隐式操作数，对于寄存器体系结构拥有显式操作数。假定 A、B 和 C 都属于存储器，A和 B 的值不能被销毁。图 A-1 显示了针对每类体系结构的 Add 运算。

如上面的图和表所示，实际上有两种类型的寄存器计算机。一类可以用任意指令来访问存储器，称为**寄存器-存储器**体系结构，另一类则只能用载入和存储指令来访问存储器，称为**载入**

-存储体系结构。第三类（还没有出现在今天交付的计算机中）将所有操作数都保存在存储器中，称为存储器-存储器体系结构。一些指令集体系结构的寄存器要多于单个累加器，但对这些特殊寄存器的使用设置了一些限制。此类体系结构有时被称为扩展累加器或专用寄存器计算机。

A-3 ~ A-4

尽管大多数早期计算机都使用栈或累加器类型的体系结构，但在1980年之后的几乎所有新体系结构都使用了载入-存储寄存器体系结构。通用寄存器（GPR）计算机之所以会出现，其主要原因有两个。第一，寄存器（类似于处理器内部其他形式的存储器）快于存储器。第二，对编译器来说，使用寄存器要比使用其他内部存储形式的效率更高。例如，在寄存器计算机中，在对表达式(A×B) - (B×C) - (A×D)求值时，可以按任意顺序执行乘法计算，这种做法的效率更高一些，可能是操作数位置的原因，也可能是流水线因素的原因（见第3章）。不过，在栈计算机上，硬件只能按唯一的顺序对表达式进行求值，这是因为操作数是隐藏在栈中的，它必须多次载入操作数。

更重要的是，寄存器可用于保存变量。当变量被分配到寄存器中时，可以降低存储器通信流量、加快程序速度（由于寄存器的速度快于存储器），提高代码密度（由于寄存器的名称位数少于存储器位置的名称位数）。

在A.8节将会解释，编译器编写人员希望所有寄存器都是等价的、无保留的。较早的计算机在满足这一期望方面打了折扣，将一些寄存器专门用于一些特殊应用，显著降低了通用寄存器的数量。如果真正通用寄存器的数目过小，那尝试将变量分配到寄存器中就没有什么好处。编译器将所有未确认用途的寄存器保留给表达式求值使用。

有多少个寄存器才是足够的呢？其答案当然取决于编译器的有效性。大多数编译器会为表达式求值保留一些寄存器，为参数传递使用一些，其余寄存器可用于保存变量。现代编译器技术能够有效地使用大量寄存器，从而增加了最新体系结构的寄存器数目。

有两个重要指令集特性可以用来区分GPR体系结构。这两个特性都关注一个典型算术或逻辑指令（ALU指令）操作的本质。第一个特性关注一个ALU指令是有两个还是三个操作数。在三操作数格式中，指令包含一个结果操作数和两个源操作数。在两操作数格式中，操作数之一既是运算的源操作数，又是运算的结果操作数。GPR体系结构的第二个区别是考虑ALU指令中可能有多少个操作数是存储器地址。一个典型ALU指令所支持的存储器操作数数量可以是0~3个。表A-2给出了这两种特性的组合及其计算机示例。尽管共有7种可能组合方式，但其中3种就可以对几乎全部现有计算机进行分类。前面曾经提到，这3种是载入-存储（也称为寄存器-寄存器）、寄存器-存储器和存储器-存储器。

表A-2　存储器操作数与每条典型ALU指令中总操作数的典型组合方式，并给出计算机示例

存储器地址的数目	所允许的最大操作数个数	体系结构的类型	示　例
0	3	载入-存储	Alpha、ARM、MIPS、PowerPC、SPARC、SuperH、TM32
1	2	寄存器-存储器	IBM 360/370、Intel 80x86、Motorola 68000、TI TMS320C54x
2	2	存储器-存储器	VAX（还有三操作数格式）
3	3	存储器-存储器	VAX（还有两操作数格式）

* ALU指令中没有存储器引用的计算机称为载入-存储或寄存器-寄存器计算机。每条典型ALU指令中有多个存储器操作数的指令称为寄存器-存储器或存储器-存储器，具体取决于它们是拥有一个还是一个以上的存储器操作数。

表 A-3 显示了每一类型的优势和劣势。当然，这些优势和劣势不是绝对的：它们是定性的，它们的实际影响取决于编译器和实现策略。采用存储器-存储器运算的 GPR 计算机很容易被编译器忽略，用作一种载入-存储计算机。体系结构方面最普遍的影响之一是对指令编码和执行一项任务所需指令数的影响。在附录 C 和第 3 章中可以了解这些不同体系结构对实现方法的影响。 A-5

表A-3 3种最常见通用寄存器计算机的优势和劣势

类 型	优 势	劣 势
寄存器-寄存器（0，3）	简单、固定长度指令编码。简单代码生成模型。指令执行所需要的时钟数相似（参见附录C）	指令数目高于指令中有存储器引用的体系结构。指令多、指令密度低，增大了程序的规模
寄存器-存储器（1，2）	无需独立的载入指令就可以访问数据。指令格式易于编码，可以得到很好的指令密度	由于在二元运算中，源操作数会被销毁，所以操作数是不等价的。在每条指令中对寄存器数目和存储器地址进行编码可能会限制寄存器的个数。每条指令的时钟数会随操作数的位置变化
存储器-存储器（2，2）或(3，3)	最紧凑。没有为临时值浪费寄存器	指令规模变化很大，特别是对于三操作数指令。此外，每条指令的工作也有很大变化。存储器访问会造成存储器瓶颈（现在未使用）

* 符号（m,n）表示存储器操作数有 m 个，共有 n 个操作数。一般来说，可选项较少的计算机简化了编译器的任务，因为编译器需要作出的决策较少（见 A.8 节）。具有大量灵活指令格式的计算机减少了程序编码需要的位数。寄存器的数目对指令大小也有所影响，因为对于指令中的每个寄存器分类符，需要 $\log_2$（寄存器个数）。因此，对于寄存器-寄存器体系结构而言，寄存器个数加倍需要增加 3 个位，大约是 32 位指令的 10%。

小结：指令集体系结构分类

在这里和 A.3 节到 A.8 节的末尾，我们总结了希望在新指令集体系结构中出现的特性，为 A.9 节介绍的 MIPS 体系结构奠定基础。从这一节开始，我们应当期待通用寄存器的使用。表 A-3 再结合关于流水线的附录 C，给出了关于通用寄存器体系结构载入-存储版本的一些展望。 A-6

在介绍了体系结构分类之后，下一个主题将介绍操作数。

A.3 存储器寻址

一个体系结构，无论是载入-存储式，还是允许任何操作数都是存储器引用，它都必须定义如何解释存储器地址以及如何指定这些地址。这里给出的测量值大体与计算机无关，但并非绝对如此。在某些情况下，这些测量值受编译器技术的影响很大。由于编译器技术扮演着至关重要的角色，所以这些测量都是使用一种优化编译器测得的。

A.3.1 解释存储器地址

如何解释一个存储器地址呢？也就是说，根据地址和长度会访问到什么对象呢？本书中讨论的所有指令集都是字节寻址的，提供对字节（8 位）、半字（16 位）和字（32 位）的访问方式。大多数计算机还提供了对双字（64 位）的访问。

关于如何对一个较大对象中的字节进行排序，有两种不同的约定方式。小端字节顺序将地址为“x...x000”的字节放在双字的最低有效位置（小端）。字节的编号为：

7	6	5	4	3	2	1	0

大端字节顺序将地址为“x...x000”的字节放在双字的最高有效位置（大端）。字节的编号为：

0	1	2	3	4	5	6	7

在同一台计算机内部进行操作时，字节顺序通常不会引起人们的注意——只有那些将相同位置同时作为字和字节进行访问的程序才会注意到这一区别。但是，在采用不同排序方式的计算机之间交换数据时，字节顺序就会成为一个问题。在对比字符串时，小端排序也不能与字的正常排序方式相匹配。字符串在寄存器中是反向表示的，如 backwards 显示为“SDRAWKCAB”。

第二个存储器问题是：在许多计算机中，对大于一字节的对象进行寻址时都必须是对齐的。大小为 s 字节的对象，字节地址为 A，如果 $A \bmod s=0$，则对该对象的寻址是对齐的。图 A-2 显示了寻址为对齐和不对齐时的地址。

	字节地址的低3位取值							
对象宽度	0	1	2	3	4	5	6	7
1字节（字节）	对齐	对齐	对齐	对齐	对齐	对齐	对齐	对齐
2字节（半字）	对齐		对齐		对齐		对齐	
2字节（半字）								
4字节（字）	对齐				对齐			
4字节（字）								
4字节（字）								
4字节（字）								
8字节（双字）	对齐							
8字节（双字）								
8字节（双字）								
8字节（双字）								
8字节（双字）								
8字节（双字）								
8字节（双字）								
8字节（双字）								

图 A-2　在字节寻址计算机中，字节、半字、字和双字对象的对齐与未对齐地址。对于每种未对齐示例，一些对象需要两次存储器访问才能完成。每个对齐对象总是可以在一次存储器访问中完成，只要存储器与对象的宽度相同即可。本图显示的存储器宽度为 8 个字节。标记各列的字节偏移指定了该地址的低 3 位

为什么有人要设计一种带有对齐限制的计算机呢？由于存储器通常与一个字或双字的倍数边界对齐，所以非对齐寻址会增加硬件复杂性。一个非对齐存储器寻址可能需要多个对齐的存储器引用。因此，即使在允许非对齐寻址的计算机中，采用对齐寻址的程序也可以运行得更快一些。

A-7

即使数据是对齐的，要支持字节、半字和字寻址也需要一个对齐网络来对齐 64 位寄存器中的字节、半字和字。例如，在表 A-4 中，假定从低 3 位取值为 4 的地址中读取一个字节。我们需要右移 3 个字节，对准 64 位寄存器中正确位置的字节。根据具体指令，计算机可能还需要对这个量进行符号扩展。存储过程很容易：只有存储器的寻址字节可被修改。在某些计算机中，字节、半字和字操作不会影响到寄存器的上半部分。尽管本书中讨论的所有计算机都允许对存

储器进行字节、半字和字访问，但只有 IBM 360/370、Intel 80x86 和 VAX 支持对不足完整宽度的寄存器操作数进行 ALU 运算。

既然已经讨论了存储器寻址的各种解释方法，现在可以讨论指令用来指定地址的方式了，这些方式称为**寻址方式**。 A-8

A.3.2 寻址方式

给定地址后，我们就知道了去访问存储器的哪些字节。在这一小节中，我们将研究寻址方式——体系结构如何指定要访问对象的地址。除了存储器中的位置之外，寻址方式还指定常量和寄存器。在使用存储器位置时，由寻址方式指定的实际存储器地址称为**有效地址**。

表 A-4 显示了在最近计算机中用到的所有数据寻址方式。立即数或直接操作数寻址通常被看作存储器寻址模式（即使它们访问的值位于指令流中也是如此），不过，由于寄存器通常没有存储器地址，所以我们将它们分离出来。我们已经将那些依赖于程序计数器的寻址模式（称为 PC 相对寻址）分离出来。PC 相对寻址主要用于在控制转移指令中指定代码地址，A.6 节将对此进行讨论。 A-9

表A-4 寻址方式的选择，并给出示例、含义及用法

寻址方式	指令举例	含 义	使用时机
寄存器寻址	Add R4,R3	Regs[R4] ← Regs[R4]+ Regs[R3]	当一个值在寄存器中
立即数寻址	Add R4,#3	Regs[R4] ← Regs[R4] + 3	对于常量
位移量寻址	Add R4,100(R1)	Regs[R4] ← Regs[R4]+ Mem[100 + Regs[R1]]	访问本地变量（+模拟寄存器间接、直接寻址方式）
寄存器间接寻址	Add R4,(R1)	Regs[R4] ← Regs[R4] + Mem[Regs[R1]]	使用指针或计算得出的地址寻址
索引寻址	Add R3,(R1 + R2)	Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]	有时用于数组寻址：R1为数组的基址，R2为索引值
直接或绝对寻址	Add R1,(1001)	Regs[R1] ← Regs[R1] + Mem[1001]	有时用于访问静态数据；地址常量可能很大
存储器间接寻址	Add R1,@(R3)	Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]	如果R3为指针*p*的地址，则此方式生成**p*
自动递增寻址	Add R1,(R2)+	Regs[R1] ← Regs[R1] + Mem[Regs[R2]] Regs[R2] ← Regs[R2] + *d*	用于在循环内部逐步遍历数组。R2指向数组的开始位置；每次引用时都会将R2的值增大一个元素的大小*d*
自动递减寻址	Add R1, -(R2)	Regs[R2] ← Regs[R2] − *d* Regs[R1] ← Regs[R1] + Mem[Regs[R2]]	与自动递增的用途相同。自动递增/递减也可以用作push/pop，以实现栈
比例寻址	Add R1,100(R2)[R3]	Regs[R1] ← Regs[R1]+ Mem[100 + Regs[R2]+ Regs[R3] * *d*]	用于索引数组。在某些计算机中，可用于任何索引寻址方式

* 在自动递增/递减和比例寻址方式中，变量 *d* 指定所访问数据项的大小（即，该指令访问的是 1、2、4 或 8 字节中的哪一种）。只有当被访问元素位于存储器中的连续位置时，这些寻址方式才有用。RISC 计算机使用位移量寻址来模拟寄存器间接寻址（地址为 0）和直接寻址（基址寄存器中为 0）。在我们的测量结果中，使用为每种模式显示的第一个名称。在 A.9.5 节定义了用作硬件描述的 C 语言扩展。

表 A-4 给出了这些寻址方式的最常见名称，当然，这些名称在不同体系结构中是有差别的。

在本表及整本书中，我们将使用 C 编程语言的扩展作为硬件描述符号。在本表中，只使用了一个非 C 特征：用左箭头（←）表示赋值。我们还使用数组 Mem 作为主存储器的名称，使用数组 Regs 表示寄存器。因此，Mem[Regs[R1]]是指存储器位置的内容，这一位置的地址由寄存器 1（R1）的内容给出。对于小于一个字的数据，我们将在后面介绍用于访问和转移此类数据的扩展。

寻址模式能够大幅减少指令数目，它们也会增加构建计算机的复杂度，对于实施这些方式的计算机，还可能增加每条指令的平均时钟周期数目（CPI）。因此，各种寻址模式的使用对于帮助架构师选择包含哪些功能是十分重要的。

图 A-3 中给出在 VAX 体系结构上对 3 个程序中寻址方式使用样式的测量结果。在这个附录中，我们使用较旧的 VAX 体系结构进行一些测量，这是因为它拥有最丰富的寻址方式，对存储器寻址的限制最少。例如，表 A-2 给出了 VAX 支持的所有方式。但是，本附录中的大多数测量将使用最近的寄存器-寄存器体系结构，以显示程序如何使用当前计算机的指令集。

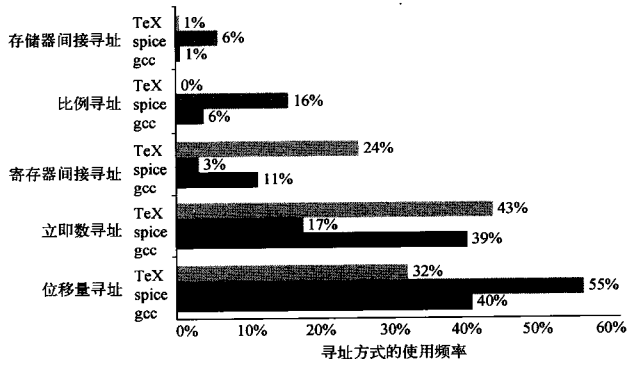


图 A-3　存储器寻址方式（包括立即数）的用法小结。几乎所有存储器访问都采用这几种主要寻址方式（可能只有 0%～3%的例外）。一半的操作数引用采用寄存器寻址方式（没有计算在前面的比例之中），而另一半则采用存储器寻址方式（包括立即数）。当然，编译器会影响到选用哪种寻址模式；参见 A.8 节。VAX 上的存储器间接寻址方式可使用位移量、自动递增或自动递减来形成初始存储器地址；在这些程序中，几乎所有存储器间接引用都以位移量寻址方式为基准方式。位移量寻址方式包括了所有位移量长度（8、16 和 32 位）。PC 相关寻址方式（几乎专用于分支）未包含在内。图中仅给出了平均使用频率超过 1%的寻址方式

如图 A-3 所示，位移量寻址和立即数寻址成为主导寻址方式。让我们来看看这两种广泛应用的寻址方式的一些特性。

A.3.3　位移量寻址方式

在使用位移量类型的寻址方式时，一个主要问题就是所用位移量的范围。根据所使用的各种位移量大小，可以决定支持哪些位移量大小。由于位移量字段的大小直接影响到指令的长度，所以其选择非常重要。图 A-4 是利用基准测试程序对载入-存储体系结构中数据访问进行的测量结果。我们将在 A.6 节研究分支偏移——数据访问样式与分支是不同的；将它们两者结合起来并没有什么收益，但在实际中，为简单起见，一般会使立即数的大小相同。

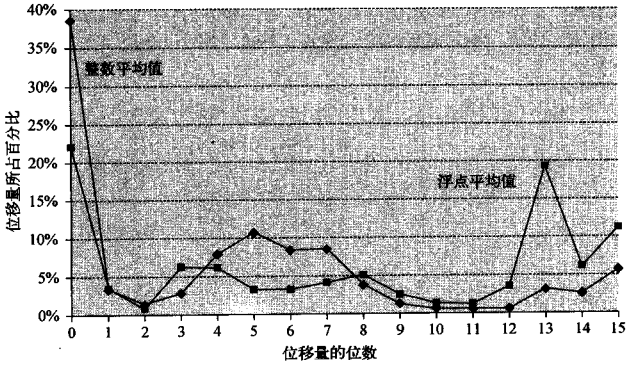


图 A-4 位移值的分布非常广泛。既存在大量小数值，又有相当数量的大数值。位移值的广泛分布是由于变量有多个存储区域，而且访问它们的位移量不同（见 A.8 节），而且编译器使用的总寻址机制也各不相同。x 轴是位移量以 2 为底的对数值，即表示该位移量所需要的字节大小。x 轴上的零表示位移值 0 的百分比。该曲线没有包含符号位，存储布局对它会有严重影响。大多数位移值是正数，但最大的位移值（14 位以上）为负值。由于这些数据是在一个采用 16 位位移量的计算机上收集的，所以无法从中了解更长位移量的信息。这些数据是在 Alpha 体系结构上测得的，对 SPEC CPU2000 进行了全面优化（见 A.8 节），给出了整数程序（CINT2000）的平均值和浮点程序（CFP2000）的平均值

A.3.4 立即数或直接操作数寻址方式

在进行算术运算、比较（主要用于分支）和移动时，如果希望将常量放在寄存器中，可以使用立即数。后一种情景可用于写在代码中的常量（这种常量较小）和地址常量（这种常量可能很大）。对于立即数的使用，重点是要知道是需要对所有运算都支持立即数，还是仅对一部分运算支持立即数。图 A-5 显示了在一个指令集中，立即数在一般类型的整数和浮点运算中的使用频率。 A-10

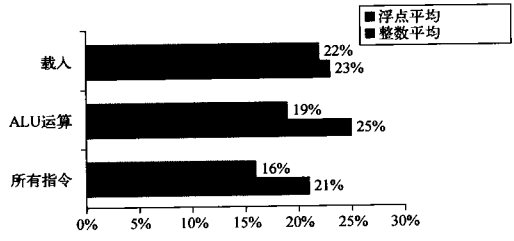


图 A-5 大约有四分之一的数据传送和 ALU 运算拥有立即操作数。下面的长条表示整数程序在大约五分之一的指令中使用立即数，而浮点程序在大约六分之一的指令中使用立即数。对于载入操作，载入立即数指令将 16 位载入一个 32 位寄存器的任一半中。载入立即数并不是严格意义上的载入，因为它们并不访问存储器。偶尔会使用一对载入立即数来载入 32 位常量，但这种情况很少见。（对于 ALU 运算，还包含移动常数位，作为带有立即操作数的运算。）用于收集这些统计数字的程序和计算机与图 A-4 相同

另一个重要指令集测量是立即数的取值范围。与位移值相似，立即数取值的大小也影响到指令长度。如图 A-6 所示，小立即数的应用最多。不过，有时也会使用大型立即数，更多的是用在地址计算中。

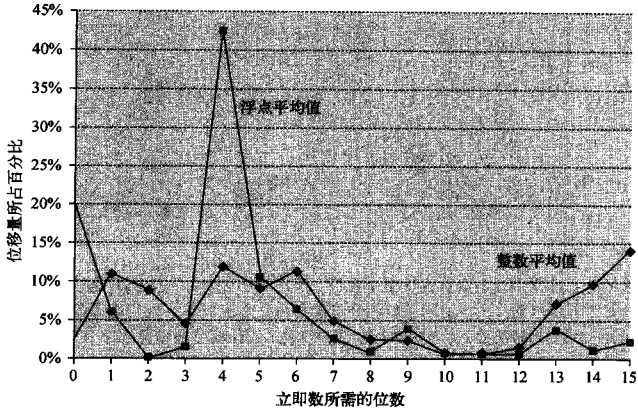


图 A-6　立即数值的分布。x 轴给出表示立即值取值大小所需要的位数——0 表示立即数字段值为 0。大多数立即数取值为正数。对于 CINT2000，大约 20%为负数，对于 CFP2000，大约 30%为负数。这些测量是在 Alpha 上执行的，其中最大立即数为 16 位，被测程序与图 A-4 中相同。对 VAX 进行的类似测量（它支持 32 位立即数）表明：大约 20%~25%的立即数长于 16 位。因此，16 位的长度覆盖大约 80%，8 位可以覆盖大约 50%

A.3.5　小结：存储器寻址

首先，我们预测一种新的体系结构至少支持以下寻址方式：位移量寻址、立即数寻址和寄存器间接寻址，这主要是因为它们非常普及。图 A-3 显示它们代表了我们测量中所使用的 75%~99%的寻址方式。其次，我们预测位移量寻址方式中的地址大小至少为 12~16 位，根据图 A-4 的图题，这些大小将占到位移量的 75%~99%。最后，我们预测立即数字段的大小至少为 8~16 位。这一说法在它提到的图题中没有得到证实。

A-11 ~ A-12

我们已经介绍了指令集分类并决定采用寄存器-寄存器体系结构，再加上前面关于数据寻址模式的建议，下面将介绍数据的大小与意义。

A.4　操作数的类型与大小

如何指定操作数的类型呢？通常，通过在操作码中进行编码来指定操作数的类型——这是最常用的方法。或者，用一些可以被硬件解读的标签对数据进行标记。这些标签指定操作数的类型，并相应选择操作。但是，带有标记数据的计算机只能在计算机博物馆里找到了。

我们首先从台式机和服务器体系结构开始。通常，操作数的类型（整数、单精度浮点、字符等）有效地确定了其大小。常见操作数类型包括字符（8 位）、半字（16 位）、字（32 位）、单

A-13

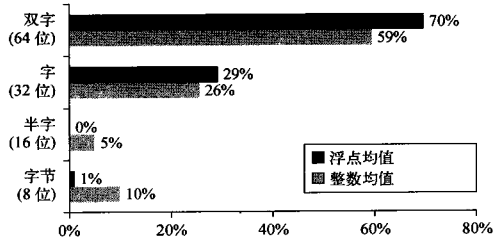
精度浮点（也是1个字）和双精度浮点（2个字）。整数几乎都是用二进制补码数字表示的。字符通常用ASCII表示，但随着计算机的国际化，16位Unicode（在Java中使用）也正在普及。直到20世纪80年代早期，大多数计算机制造商还在选择自己的浮点表示法。从那之后，几乎所有的计算机都遵循了相同的浮点标准——IEEE标准754。IEEE浮点标准在附录J中详细讨论。

一些体系结构提供了对字符串的操作，不过这些操作通常都十分有限，将字符串中的每个字符都看作单个字符。支持对字符串执行的典型操作包括比较和移动。

对于商务应用程序，一些体系结构支持二进制格式，通常称为**压缩十进制**或**二进制编码十进制**——用4个位对0至9的数值进行编码，两个十进制数位被压缩到两个字节中。数值字符串有时称为**非压缩十进制**，通常提供在被称为**压缩**和**解压缩**的操作之间来回转换。

使用十进制操作数的一个理由是获得与二进制数字完全匹配的结果，这是因为一些十进制小数无法用二进制准确表示。例如，0.10_{10}在十进制中是一个很简单的小数，但在二进制中，它需要无限个重复数位来表示：$0.0001100110011\cdots_2$。因此，十进制中的准确计算在二进制中可能十分接近但并非完全准确，对于金融事务，这可能会成为一个问题。（如需了解有关精确算术的更多知识，请参阅附录J。）

我们的SPEC基准测试使用字节或字符、半字（短整数）、字（整数）、双字（长整数）和浮点数据类型。图A-7给出了为这些程序引用存储器对象的大小动态分布。对不同数据类型的访问频率有助于确定哪些类型最为重要，应当加以高效支持。计算机是否应当拥有64位访问路径，或者用两个时钟周期来访问一个双字是否可行？我们前面曾经看到，字节访问需要一个对齐网络：将字节作为基元类型提供支持有多么重要？图A-7使用存储器引用来查看被访问数据的类型。



图A-7 对于基准测试程序，所访问数据的大小分布。双字数据类型用于表示浮点程序中的双精度浮点值，还用于表示地址，这是因为该计算机使用64位地址。在采用32位地址的计算机上，64位地址将被32位地址代替，所以整数程序中的几乎所有双字访问都变为单字访问

在一些体系结构中，寄存器中的对象可以作为字节或半字进行访问。但是，这种访问是非常罕见的——在VAX上，不超过12%的寄存器引用采用这种方式，也就是这些程序中所有操作数访问的大约6%。

A.5 指令集中的操作

大多数指令集体系结构支持的操作符可以如表A-5那样进行分类。关于所有体系结构的一

A-14 条经验规律是：执行最多的指令是一个指令集中的简单操作。例如，表 A-6 给出了 10 种简单指令，对于一组在流行 80x86 上运行的整数程序，这 10 种简单指令占所执行指令的 96%。因为它们很常见，所以这些指令的实现应当确保它们能够快速完成。

表A-5　指令操作符的分类与示例。所有计算机通常都提供所有前三类运算

操作符类型	示　例
算术与逻辑	整数算术与逻辑运算：加、减、与、或、乘、除
数据传送	载入-存储（在采用存储器寻址的计算机上为move指令）
控制	分支、跳转、过程调用与返回、陷阱
系统	操作系统调用、虚拟存储器管理指令
浮点	浮点运算：加、乘、除、比较
十进制	十进制加、十进制乘、二进制到字符的转换
字符串	字符串移动、字符串比较、字符串搜索
图形	像素与顶点操作、压缩/解压缩操作

* 在不同体系结构中，指令集中对系统功能的支持有所不同，但所有计算机都必须对一些基本的系统功能提供指令支持。指令集对后 4 类的支持数量可能为 0，也可能包含大量特殊指令。在任何计算机中都提供浮点指令，供那些大量使用浮点数的应用程序使用。这些指令有时是可选指令集的一部分。十进制和字符串指令有时是基元类型，比如在 VAX 或 IBM 360 中，也可能是由编译器使用更简单的指令合成的。图形指令通常会对许多较小的数据项进行并行操作——例如，对 2 个 64 位操作数执行 8 个 8 位加法。

前面曾经提到，表 A-6 中的指令在每一台计算机的每个应用程序（台式机、服务器和嵌入式）中都可以找到，会对表 A-5 中的操作进行一些变化，而这主要取决于该指令集包含哪些数据类型。

表A-6　80x86中执行最多的前10类指令

排　位	80x86指令	整数均值（占所执行指令总数的百分比）
1	载入	22%
2	条件分支	20%
3	比较	16%
4	存储	12%
5	加	8%
6	与	6%
7	减	5%
8	寄存器之间的移动	4%
9	调用	1%
10	返回	1%
总计		96%

* 简单指令是这个列表的主体，占所执行指令的 96%。这些百分数是 5 个 SPECint92 程序的均值。

A.6　控制流指令

由于分支与跳转行为的测量在相当程度上与其他测量值和程序无关，所以我们现在研究控

制流指令的使用，它与上一节的操作之间没有什么共同点。

关于改变控制流的指令，没有非常一致的术语。在 20 世纪 50 年代，它们通常被称为**转移**（transfer）。在 20 世纪 60 年代，开始使用**分支**（branch）一词。后来计算机还另外引入了一些名称。在本书中，当控制中的改变是无条件时，我们使用**跳转**（jump），当改变是有条件时，使用分支。

我们可以区分 4 种不同类型的控制流变化：

- 条件分支；
- 跳转；
- 过程调用；
- 过程返回。

A-15 ~ A-16

由于每个事件都是不同的，可能使用不同指令，可能拥有不同行为，所以我们希望知道这些事件的相对频率。图 A-8 中给出了这些控制流指令在一个载入-存储计算机上的出现频率，我们就是在这种计算机上运行基准测试的。

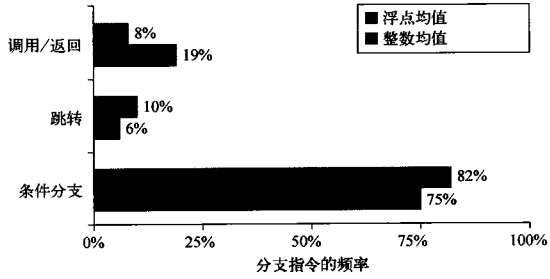


图 A-8 将控制流指令分为三类：调用或返回、跳转和条件分支。条件分支显然占绝大多数。每种类型的计数分别用三个长条来显示。用于收集这些统计数字的程序和计算机与图 A-3 中的数字相同

A.6.1 控制流指令的寻址方式

控制流指令中的目标地址在任何情况下都必须指定。在绝大多数情况下，这个目标是在指令中明确指定的，但过程返回是一个重要例外，这是因为在编译时无法知道要返回的目标位置。指定目标的最常见方法是提供一个将被加到**程序计数器**（PC）的位移量。这类控制流指令被称为 **PC 相对指令**。由于目标位置通常与当前指令的距离较近，而且，在指定相对于当前 PC 的位置时，需要的位数较少，所以 PC 相对分支或跳转具备一些优势。采用 PC 相对寻址还可以使代码的运行不受装载位置的影响。这一特性称为**位置无关**，可以在链接程序时减少一些工作，而且对于在执行期间进行动态链接的程序也比较有用。 A-17

如果在编译时不知道目标位置，为了实现返回和间接跳转，需要一种不同于 PC 相对寻址的方法。这时，必须有一种动态指定目标的方法，使目标能够在运行时发生变化。这种动态寻址可能非常简单，只需要给出包含目标地址的寄存器名称即可；跳转可能允许使用任意寻址方式来提供目标地址。

这些寄存器间接跳转对于其他 4 种重要功能也是有用的。

- **case 或 switch 语句**，大多数编程语言中都会有这些语句（用于选择几种候选项之一）。
- **虚拟函数或虚拟方法**，存在于诸如 C++或 Java 之类的面向对象式语言中（允许根据参数类型调用不同例程）。
- **高阶函数或函数指针**，存在于诸如 C 或 C++等语言中（它允许以参数方式传递一些函数，提供面向对象编程的一种好处）。
- **动态共享库**（允许仅当程序实际调用一个库时才在运行时加载和链接，而不是在运行程序之前进行静态加载和链接）。

在所有这 4 种情况下，目标地址在编译时都是未知的，因此，通常是在寄存器间接跳转之前从存储器加载到寄存器中。

由于分支通常使用 PC 相对寻址来指定其目标，一个重要的问题就是关注分支目标距离分支有多远。了解这些位移量的分布有助于选择支持哪些分支偏移量，从而会影响到指令长度和编码。图 A-9 给出了指令中 PC 相对分支的位移量分布。这些分支中大约有 75%是正向的。

A-18

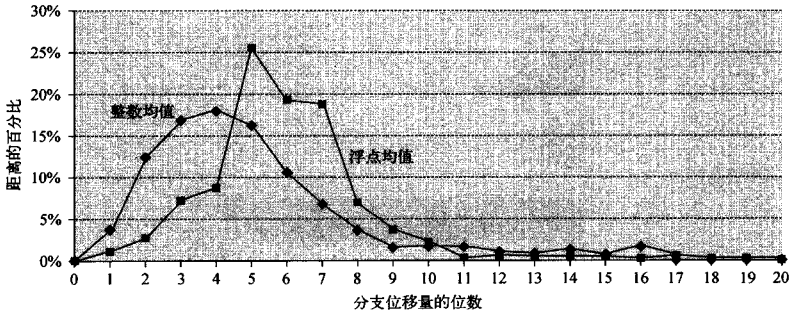


图 A-9 分支距离（以目标与分支指令之间的指令数来表示）。整数程序中的最常见分支是转向可以用 4~8 位编码的目标地址。这一结果告诉我们，短位移量字段对于分支指令通常足够了，有了较小分支位移量的较短指令，设计者可以提高编码密度。这些测量结果是在载入-存储计算机（Alpha 体系结构）上测得的，所有指令都与字边界对齐。对于同一程序，如果体系结构需要的指令较少（比如 VAX），那分支距离就较短。但是，如果计算机的指令长度是变化的，可以与任意字节连接对齐，则表示该位移量所需要的位数可能会增加。收集这些统计数字的程序和计算机与图 A-4 中相同

A.6.2 条件分支选项

由于大多数控制流改变都是分支，所以决定如何指定分支条件是很重要的。表 A-7 列出了当前使用的 3 种主要技术及其优缺点。

分支的最明显特性之一是大量比较都是简单的测试，其中有很多是与 0 进行比较。因此，一些体系结构选择将这些比较当作特殊情景进行处理，特别是在使用**比较并分支**指令中。图 A-10 给出了条件分支中用到的各种不同比较的频率。

表A-7 对分支条件进行求值的主要方法及其优缺点

名 称	示 例	如何测试条件	优 点	缺 点
条件代码(CC)	80x86、ARM、PowerPC、SPARC、SuperH	测试由ALU运算设定的特殊位，可能受程序的控制	有时条件设置比较自由	CC是一种额外状态。由于条件代码是将来自一条指令的信息传送给一个分支，所以它们限制了指令的顺序
条件寄存器	Alpha、MIPS	用比较结果测试任意寄存器	简单	占用一个寄存器
比较与分支	PA-RISC、VAX	比较是分支的一部分。比较范围通常仅限于子集	分支需要一条指令，而不是两条	对于流水线执行来说，每个指令要完成的工作可能过多

* 尽管条件代码可以由ALU运算设定（用于其他目的），但对程序的测量显示，这种情况会很少发生。当条件代码由一大组指令或一组偶然选定的指令设定，而不是由指令中的一个比特来设定时，就会出现条件代码的重要实现问题。拥有比较和分支指令的计算机通常会限制比较范围，使用条件寄存器进行更复杂的比较。通常，对于根据浮点比较进行的分支和基于整数比较进行的分支会采用不同技术。由于根据浮点比较执行的分支数目要远小于根据整数比较进行的分支数目，所以这种做法是合理的。

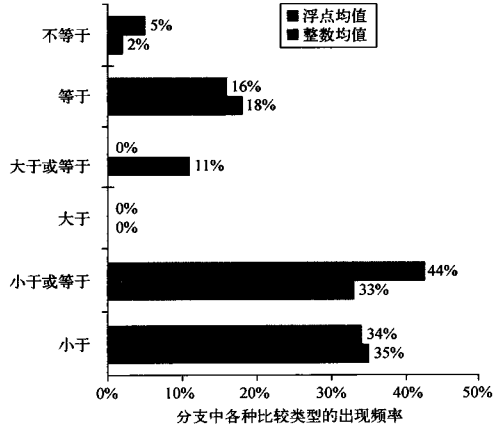


图 A-10 条件分支中不同比较类型的使用频率。编译器与体系结构的这种组合中，小于（或等于）分支占主导地位。这些测量值包含了分支中的整数和浮点比较。用于收集这些统计数字的程度和计算机与图 A-4 中相同

A.6.3 过程调用选项

过程调用和返回包括控制转移，还可能涉及一些状态保存过程；至少必须将返回地址保存在某个地方，有时保存在特殊的链接寄存器中，有时只是保存在 GPR 中。一些较早的体系结构提供了一种用于保存许多寄存器的机制，而较新的体系结构需要编译器为所存储和恢复的每个寄存器生成存储和载入操作。

在保存寄存器时，有两种基本约定：要么保存在调用位置，要么保存在被调用的过程内部。

调用者保存是指发出调用的过程必须保存它希望在调用之后进行访问的寄存器，因此，被调用的过程不需要为寄存器操心。**被调用者保存**与之相反：被调用过程必须保存它希望使用的寄存器，而调用者不受限制。在某些时间必须使用调用者保存方法，其原因在于两种不同过程中对全局可见变量的访问样式。例如，假定我们有一个过程 P1，它调用过程 P2，这两个过程都对全局变量 x 进行处理。如果 P1 已经将 x 分配给一个寄存器，它必须确保在调用 P2 之前将 x 保存到 P2 知晓的一个位置。编译器希望知道被调用过程可能在什么时候访问寄存器分配量，由于不同进程可能是分别编译的，所以增加了获知这一信息的难度。假定 P2 可能不会触及 x，但可能调用另一个可能访问 x 的进程 P3，而 P2 和 P3 是分别编译的。由于这些复杂性的存在，大多数编译器会采用比较稳健的方式，由调用者将所有可能在调用期间访问的变量都保存起来。

A-19

在可以采用任一约定的情况下，有些程序更适于采用被调用者保存，有些程序更适于采用调用者保存。结果，今天的大多数实际系统都采用这两种机制的组合方式。这一约定在应用程序二进制接口（ABI）中指定，它确定了一些基本规则，指出哪些寄存器应当由调用者保存，哪些应当由被调用者保存。在本附录的后面，我们将研究用于自动保存寄存器的高级指令与编译需求之间的不一致之处。

A.6.4 小结：控制流指令

控制流指令属于执行频率最高的一部分指令。尽管条件分支有许多选项，但我们希望新体系结构中的分支寻址能够跳转到分支指令之前或之后数百条指令处。这一要求意味着 PC 相对分支位移量至少为 8 位。我们还希望看到跳转指令采用寄存器间接和 PC 相对寻址，来支持过程返回和当前系统的许多其他功能。

A-20

我们现在已经从汇编语言程序员或编译器编写人员的层次，完成了对指令体系结构的浏览。我们介绍了一种采用位移量、立即数和寄存器间接寻址方式的载入-存储体系结构。所介绍的数据为 8 位、16 位、32 位和 64 位整数，还有 32 位和 64 位浮点数。指令包括简单操作、PC 相对条件分支、用于过程调用的跳转和链接指令，还有用于过程返回的寄存器间接跳转（还有其他一些应用）。

现在我们需要选择如何采用一种便于硬件执行的方式来表示这一体系结构。

A.7 指令集编码

显然，上述选择会影响到如何对这些指令进行编码，表示为供计算机执行的二进制形式。这种表示形式不仅影响到程序经过编译后的大小，还会影响到处理器的实现，处理器必须对这种表示形式进行译码，快速找出操作和操作数。操作通常在一个称为**操作码**的字段中指定。后面将会看到，一个重要决定是如何通过编码将寻址方式与操作结合在一起。

这一决定取决于寻址方式的范围和操作码与寻址方式之间的独立程序。一些较早的计算机有 1~5 个操作数，每个操作数有 10 种寻址方式（见表 A-4）。对于如此大量的组合情况，通常需要为每个操作数使用独立**地址标识符**：地址标识符告诉使用哪种寻址方式来访问该操作数。另一个极端是仅有一个存储器操作数、仅有一或两种寻址方式的载入-存储计算机；显然，在这种情况下，可以将寻址方式作为操作码的一部分进行编码。

在对指令进行编码时，由于寄存器字段和寻址方式字段可能在一条指令中出现许多次，所以寄存器数目和寻址方式的数目都对指令大小有显著影响。事实上，对于大多数指令，对寻址

方式字段和寄存器字体进行编码时所占用的位数，要远多于指定操作码所占用的位数。在对指令集进行编码时，架构师必须平衡以下几种竞争力量。

(1) 希望允许尽可能多的寄存器和寻址方式。

(2) 寄存器字段和寻址方式字段的大小对平均指令大小存在影响，从而对平均程序大小产生影响。

(3) 希望编码后的指令长度易于以流水线实施方式处理。（易解码指令的价值在附录 C 和第 3 章讨论。）至少，架构师希望指令长度为字节的总数，而不是任意位长度。许多台式机和服务器架构师已经选择使用固定长度的指令来获得实现方面的好处，不过这样是以牺牲平均代码规模为代价的。 A-21

图 A-11 给出三种常见的指令集编码选择。第一种称为**变长编码**，这是因为它几乎允许对所有操作使用所有寻址方式。当存在许多寻址方式和操作时，这是最佳选择。第二种选择称为**定长编码**，因为它将操作和寻址方式合并到操作码中。通常，采用定长编码时，所有指令的大小都相同；当寻址方式与操作数较少时，其效果最好。变长编码与定长编码之间进行的权衡是程序大小与处理器译码的难易程度。变长编码在表示程序时尽力减少所用位数，各条指令的大小和要执行的工作量可能会有很大变化。

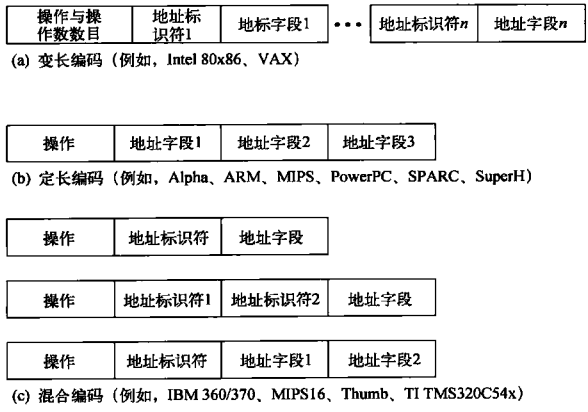


图 A-11 指令编码的三种基本变体：变长编码、定长编码、混合编码。变长格式可以支持任何数目的操作数，每个地址标识符确定操作数的寻址方式和标识符的长度。这种方式的代码表示长度通常是最短的，因为不会包含没有使用的字段。定长格式中的操作数个数总是相同的，寻址方式（如果存在选项的话）作为操作码的一部分进行指定。它生成的代码规模通常是最大的。尽管字段的位置不会变化，但不同指令会将其用于不同目的。混合编码方法拥有多种由操作码指定的格式，添加了一到两个字段来指定寻址方式，还有一到两个字段来指定操作数地址

让我们看一条 80x86 指令，作为变长编码的一个例子。

```
add EAX,1000(EBX)
```

add 是指一条有两个操作数的 32 位整数加法指令，这个操作码占 1 个字节。80x86 地址标识符为 1 或 2 个字节，指定源/目标寄存器（EAX）和第二个操作数的寻址方式（在这个例子中为位移 A-22

量）与基址寄存器（EBX）。这一组合占用1个字节来指定操作数。在32位模式中（参见附录K），地址字段的大小为1或4个字节。由于1000大于2^8，所以这条指令的长度是

1+1+4=6个字节

80x86指令的长度介于1~17个字节之间。80x86程序通常短于RISC体系结构，后者使用定长格式（参见附录K）。

有了变长编码和定长编码这两种指令集设计的极端情况之后，立即就可以想到第三种选择：降低变长体系结构中指令大小与任务的变化程度，但提供多种指令长度，以缩小代码尺寸。这种混合式方法是第三种编码选择，稍后将会看到其示例。

A.7.1 RISC中的精简代码

嵌入式应用程序中的成本和缩短代码非常重要，随着RISC计算机开始进入这一领域，32位定长格式已经成为一种负担。为应对这一情况，几家制造商提供了其RISC指令集的一种新混合版本，同时拥有16位和32位指令。这些较窄的指令支持较少的运算、较小的地址与立即数字段、较少的寄存器和两地址格式，而不是RISC计算机的典型三地址格式。附录K给出两个示例——ARM Thumb和MIPS MIPS 16，这两个都声称代码规模最多可以缩减40%。

与这些指令集扩展相对，IBM只是对其标准指令集进行了压缩，然后添加一些硬件，在因为发生指令缓存缺失而从存储器中提取指令时，对其进行解压缩。因此，指令缓存中包含完整的32位指令，但在主存储器、ROM和磁盘中保存的是压缩后的代码。MIPS 16和Thumb的优点是指令缓存的工作方式使其好像增大了大约25%。而IBM的CodePack意味着不需要为了处理不同指令集而对编译器进行修改，指令译码仍然非常简单。

CodePack首先对任意PowerPC程序进行游程编码压缩，然后将所得到的压缩表载入芯片上一个大小为2KB的表中。因此，每个程序都有其自己独特的编码。为了处理分支（不再与字边界对齐），PowerPC在存储器中生成一个散列表，将压缩前后的地址对应起来。它像TLB一样（见第2章），缓存了最近使用过的地址映射，用来减少存储器访问的次数。IBM宣称整体性能成本为10%，代码规模缩减35%~40%。

而日立则为嵌入式应用程序发明了一种采用固定16位格式的RISC指令集，称为SuperH（见附录K）。它有16个而不是32个寄存器，使其与较窄的格式和较少指令相吻合，而其他方面看起来与经典RISC体系结构一致。

A-23

A.7.2 小结：指令集编码

前几节讨论了指令集设计时所作的决策，这些决策决定了架构师是否能够在变长指令编码和定长指令编码之间进行选择。给定选择之后，那些看重代码规模多于性能的架构师会选择变长编码，而看重性能多于看重代码规模的架构师则会选择定长编码。附录E给出了架构师的13个选择结果示例。在第3章和附录C中进一步讨论这种变化对处理器性能的影响。

我们几乎已经为将在A.9节介绍的MIPS指令集体系结构奠定了基础。但在开始介绍之前，先来简要地了解一下编译器技术及其对程序特性的影响，这也会有所帮助。

A.8 交叉问题：编译器的角色

今天，几乎所有的台式机和服务器应用程序都是用高级语言编写的。这种开发意味着：由

于所执行的大多数指令都是编译器的输出，所以指令集体系结构基本上就是编译器目标。在这些应用程序的早期，在体系结构方面作出的决策经常是为了简化汇编语言编程，或者是针对特定内核。由于编译器会显著影响到计算机的性能，所以理解今天的编译器技术对于设计、高效实现指令集是至关重要的。

曾经有一种很流行的做法：试图将编译器技术及其对硬件性能的影响与体系结构及其性能隔离开来，就像过去经常尝试将体系结构与其实现隔离开来一样。对于今天的台式机编译器和计算机，这种隔离基本上是不可能的。体系结构方面的选择会影响到为一台计算机生成的代码质量和为其构造优良编译器的复杂性，这种影响可能是正面的，也可能是负面的。

在这一节，我们主要从编译器的视角来讨论指令集的关键目标。首先回顾对当前编译器的剖析。接下来讨论编译器技术如何影响架构师的决策，还有架构师如何增大或降低编译器生成良好代码的难度。最后回顾编译器和多媒体处理，很遗憾，这是编译器编写人员与架构师之间协作不佳的一个示例。

A.8.1 目前编译器的结构

首先让我们看看今天的最佳编译器是什么样的。图 A-12 显示了目前编译器的结构。 A-24

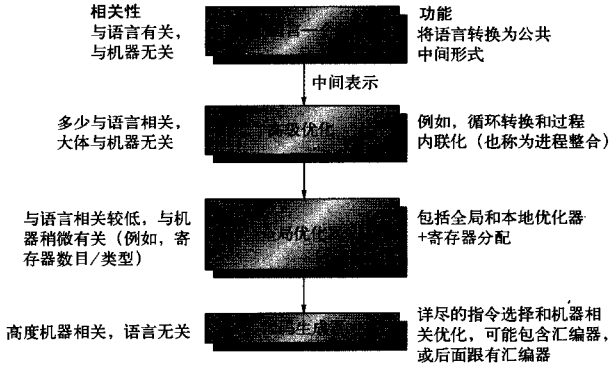


图 A-12 编译器通常包括 2～4 遍扫描（pass），一些优化程度更高的编译器会有更多遍扫描。当输入相同时，以不同优化级别编译的程序应当给出相同结果，图上的结构将这种可能性增至最大。优化扫描的设计是希望获得最优代码，如果希望加快编译速度，并且可以接受较低质量的代码，那就可以跳过优化扫描。扫描就是编译器读取和转换整个程序的一个阶段（phase）。（阶段一词经常与扫描互换使用。）由于优化扫描是独立的，所以有多种语言使用了相同的优化和代码生成扫描。一种新的语言只需要一个新的前端即可

编译器编写人员的首要目标是正确性——所有有效程序的编译结果都必须正确。第二个目标通常是编译后的代码速度。通常，还有一整套目标排在这两个目标之后，包括快速编译、支持调试、语言之间的互操作性。正常情况下，编译器中的各次扫描将更抽象的高级表示转换为逐渐降低层级的表示方式。最后到达指令集级别。这种结构可以帮助控制转换的复杂度，更容易编写出没有错误的编译器。

正确编写编译器是一件很复杂的事情，而所能完成的优化程度主要受这一复杂度的限制。尽管采用多遍扫描结构可以帮助降低编译器的复杂性，但它也意味着编译器必须进行排序，某些转换必须在其他转换之前完成。在图 A-12 所示的优化编译器框图中可以看出，某些高级优化要在执行很久之后，才可能知道最终代码会是什么样子。一旦执行这种转换，编译器就不太可能再返回并重新审视所有步骤，甚至撤消这些转换，这样的成本太过高昂。无论是编译时间还是复杂性，都不允许进行这种迭代。因此，编译器假定最后的几个步骤有能力处理特殊的问题。例如，在知道被调用过程的确切大小之前，编译器通常就必须选择对哪些进程调用进行内联展开。编译器编写人员将这一问题称为**阶段排序问题**。

A-25

这种转换排序如何与指令集体系结构互动呢？一种名为**全局公共子表达式消去法**的优化提供了一个很好的例子。这种优化找出一个表达式计算相同取值的两个实例，并将第一次计算的结果值保存在临时存储位置。然后利用这个临时值，清除这一公共表达式的第二次计算。

为使这一优化发挥显著效用，必须将临时值分配到寄存器中。否则，先将临时值存储在存储器中，之后再重新载入它，其成本将会抵消因为不用重复计算该表达式所节省的成本。的确存在一些情况：如果没有将临时值保存到寄存器中，这一优化会减缓代码的运行速度。寄存器分配通常是全局优化扫描即将结束、马上要生成代码时进行的，所以阶段排序使上述问题变得复杂。因此，执行这一优化的优化程序必须假定寄存器分配器会将这一临时值分配到寄存器中。

根据转移类型，可以将现代编译器执行的优化进行如下分类：

- **高级优化**一般对源代码执行，并将输出结果传送给之后的优化扫描；
- **本地优化**仅对直行代码段（编译器设计者称为**基本块**）内的代码进行优化；
- **全局优化**将本地优化扩展到分支范围之外，并引入了一组专为优化循环的转换；
- **寄存器分配**将寄存器与操作数关联在一起；
- **与处理器相关的优化**尝试充分利用特定的体系结构知识。

A.8.2 寄存器分配

鉴于寄存器分配在加快代码速度和使其他优化发挥效用方面扮演的角色，可以说它是最重要的优化之一（甚至就是最重要的那一个）。今天的寄存器分配算法以一种称为**图形着色**的技术为基础。这种图形着色技术背后的基本思想是构造一幅图，用来表示可能执行的寄存器分配方案，然后利用这个图来分配寄存器。大致来说，问题在于如何使用有限种颜色，使相关图中两个相邻节点的颜色都不相同。这种方法的重点是将活跃变量全部分配到寄存器中。图形着色问题的求解时间通常是图形大小的指数函数（NP 完全问题）。不过，有一些启发式算法在实际中的应用效果很好，生成分配结果的时间近似与图形大小成线性关系。

A-26

当至少有 16 个通用寄存器（多多益善）可用于为整数变量进行全局分配，而且有其他寄存器分配为浮点变量时，图形着色方法的效果最好。遗憾的是，如果寄存器的数目很少，图形着色的启发式算法很可能会失败，所以图形着色的效果不是太好。

A.8.3 优化对性能的影响

有时很难将一些较简单的优化（本地优化和与处理器相关的优化）与代码生成器中完成的转换隔离开来。典型优化的示例在表 A-8 中给出。表 A-8 的最后一列指明对源程序执行所列优化转换的频率。

表A-8 主要优化类型及每种类型的示例。这些数据告诉我们各种优化技术的相对使用频率

优化名称	解 释	优化转换总数的百分比
高级	**在源代码级别或接近该级别；与处理器无关**	
过程整合	用过程主体代替过程调用	未测量
本地	**在直行代码范围内**	
公共子表达式消除法	用单一副本代替同一计算的两个示例	18%
常量传播	对于一个被赋值为常量的变量，用该常量代替其所有实例	22%
降低栈高度	重新排列表达式树，以最大限度地减少表达式求值所需要的资源	未测量
全局	**跨越分支**	
全局公共子表达式消去法	与本地优化相同，但这一版本跨越了分支范围	13%
副本传播	对于一个已经被赋值为X的变量A（即$A=X$），用X代替变量A的所有实例	11%
代码移动	如果在循环的每次迭代中，其中一些代码总是计算相同值，则从该循环中移除该代码	16%
消去归纳变量	简化/消去循环内的数组寻址计算	2%
与处理器相关	**依赖于处理器知识**	
降低强度	许多示例，比如用加法和移位来代替与常量的乘法	未测量
流水线调度	重新排列指令顺序，以提高流水线性能	未测量
分支偏移优化	选择能够到达目标的最短分支位移	未测量

* 第三列给出一些常见优化在一组 12 个小型 Fortran 和 Pascal 程序中的静态应用频率。在测量过程中，编译器共完成了 9 个本地与全局优化。图中给出了这些优化中的 6 种，剩下 3 种的总静态频率占 18%。“未测量”是指没有测量该优化方法的使用次数。与处理器相关的优化通常是在代码生成器中完成，所有这些优化都未在此次试验中测量。所示百分比是特定类型的静态优化所占的比例。数据来自 Chow[1983]（使用 Standford UCODE 编译器收集）

图 A-13 显示了对两个程序的指令进行各种优化的效果。在这个例子中，与未经优化的程序相比，已优化程序执行的指令数会减少大约 25%~90%。该图表明在提议新指令集功能之前首先浏览已优化代码的重要性，因为编译器可能会将架构师正在尝试改进的指令完全清除。

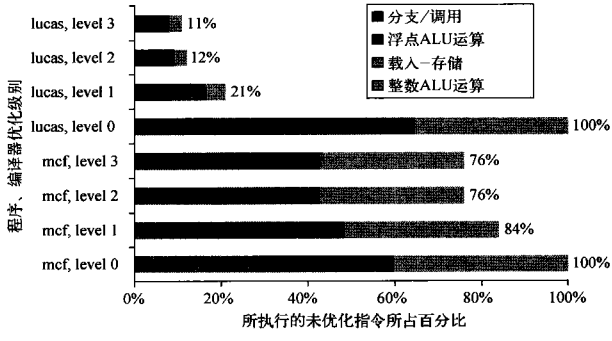


图 A-13 当编译器优化级别变化时，SPEC2000 中 lucas 和 mcf 程序中指令数目的变化。第 0 级表示未优化代码。第 1 级包含本地优化、代码调度和本地寄存器分配。第 2 级包括全局优化、循环转换（软件流水线）和全局寄存器分配。第 3 级增加了过程整合。这些试验是在 Alpha 编译器上执行的

A.8.4 编译器技术对架构师决策的影响

编译器与高级语言之间的互动显著影响着程序利用指令集体系结构的方式。这里有两个重要问题：如何对变量进行分配和寻址？需要多少个寄存器才能对变量进行适当分配？为了回答这些问题，必须看看当前高级语言用来保存数据的三个独立区域。

- 栈用于分配本地变量。栈会在进程调用与返回时相应增大或缩小。栈内的对象是相对于栈指针进行寻址的，这些对象主要是标量（单变量），而不是数组。栈用于活动记录，而不是用于表达式求值。因此，几乎从来不会在栈中压入或弹出数值。
- 全局数据区用于静态分配所声明的对象，比如全局变量和常量。这些对象中有很大一部分都是数组或者其他聚合数据结构。
- 堆用于分配那些不符合栈规则的动态对象。堆中的对象用指针访问，通常不是标量。

对于分配到栈中的对象，寄存器分配的处理效率要远高于对全局变量的处理效率，而寄存器分配对于分配到堆中的对象基本上不可能实现，因为它们是用指针访问的。全局变量和一些栈变量也不可能分配，因为它们具有别名，也就是说有多种方法可以引用变量的地址，从而不能合法地将其放到寄存器中。（在当今的编译器技术中，大多数堆变量实际上都拥有别名。）

A-27 ~ A-28

例如，考虑以下代码序列，其中&返回变量的地址，*取得指针所指向的对象：

```
p = &a-- 将 a 的地址放入 p 中
a = ...-- 直接为 a 赋值
*p = ...-- 使用 p 为 a 赋值
...a...-- 访问 a
```

不可能跨过对*p的赋值而对变量a进行寄存器分配，同时还不生成错误代码。在使用别名时，通常很难甚至不可能判断指针可能指向哪些对象，所以会导致一个实质性问题。编译器必然为保守的，如果有指针可能指向过程中的多个本地变量之一，某些编译器就不会在寄存器中分配该过程的任意本地变量。

A.8.5 架构师如何帮助编译器编写人员

今天，编译器的复杂性并非来自对诸如 A=B+C 等简单语句的转换。大多数程序都具有局部简单性，简单转换的效果很好。之所以会有这种复杂性，是因为程序规模庞大而且其全局互动非常复杂，还因为编译器的结构决定了在判定哪种代码序列最佳时，一次只能判断一步。

编译器编写人员在工作时，通常会遵循他们自己对一条体系结构基础原理的推论：加快常见情况的速度、保证少见情况的正确性。也就是说，如果我们知道哪些情况是常见的，哪些情况是少见的，而且为这两种情况生成代码都是比较轻松的事情，那少见情况的代码质量可能就不是非常重要——但必须正确！

A-29

一些指令集特性可以为编译器编写人员提供帮助。这些特性不应被看作需要严格执行的规则，而应当看到一种指南，便于编写出生成高效、正确代码的编译器。

- 提供正则性——只要可能，指令集的三个要素——操作、数据类型和寻址方式，就应当是正交的。如果体系结构的两个方面互不影响，就说它们是正交的。以操作和寻址方式为例，如果对于任何一个可以应用寻址方式的操作，都可以向其应用所有寻址方式，那就说操作和寻址是正交的。这种正则性有助于简化代码生成过程，如果在决定生成何种代码时，需要分散在编译器的两遍扫描中做出决策，那这一特性尤为重要。这一特性的一个典型反例是：限制可供特定指令类型使用的寄存器。针对专用寄存器体系结构的编译器通常会陷入这种两难境地。因为这一限制，编译器可能会发现有许多空闲寄存器，但却都不适用！

- **提供原型而非解决方案**——与一种语言构造或内核功能“相匹配”的特殊功能通常是无法使用的。为支持高级语言所做的尝试可能仅对一种语言有效，也可能与该种语言的正确、高效实现相偏离，可能有点过头，也可能有所不及。A.10 节给出一个例子，说明这些尝试是如何失败的。
- **简化候选项之间的权衡**——对于编译器编写人员来说，最艰巨的任务之一就是对于所出现的任何一段代码，都能指出哪种指令序列最为适合。在早些日子里，指令数或总代码规模可能是好的度量标准，但我们在第 1 章已经看到，现在已经并非如此了。设计者所做的任何事情，只要能够帮助编译器编写人员了解替代代码序列的成本，就能帮助改进代码。在进行这种复杂权衡时，最困难的情景之一发生在寄存器-存储器体系结构中，就是判断一个变量的引用次数达到多大数值以后，将其载入寄存器的成本才会更低一些。这一阈值很难计算，事实上，在同一体系结构的不同模型之间也可能发生变化。
- **提供一些指令，将编译时的已知量绑定为常量**——编译器编写人员特别讨厌处理器在运行时费力解读一个在编译时就已经知晓的取值。有些指令需要解读在编译时就已经固定的取值，这就是以上原则的绝佳反例之一。例如，VAX 进程调用指令（calls）会动态解释一个掩码，这个掩码说明在进行调用时要保存哪些寄存器，但它在编译时就已经固定下来了（见 A.10 节）。

A-30

A.8.6 编译器对多媒体指令的支持（或支持不足）

唉！SIMD 指令（见 4.3 节）的设计者基本上会忽略上一小节的内容。这些指令往往就是解决方案，而不是原型；它们缺少寄存器；数据类型与现有编程语言不匹配。架构师希望找出一种能够帮助某些用户的廉价解决方案，但通常只有很少一些低级图形库例程使用它们。

SIMD 指令实际是一种出色体系结构的简化版本，它拥有自己的编译器技术。在 4.2 节进行了解释，**向量体系结构**对数据的向量进行操作。多媒体内核最初是为科学代码发明的，通常也是可以向量化的，当然通常是处理较短的向量。4.3 节曾经提及，我们可以将 Intel 的 MMX 和 SSE 或者 PowerPC 的 AltiVec 看作简单的短向量计算机：MMX 的向量可以有 8 个 8 位元素、4 个 16 位元素或 2 个 32 位元素，AltiVec 的向量长度是以上长度的两倍。它们被实现为宽寄存器中的相邻窄元素。

这些微处理器体系结构将向量寄存器大小设定到体系结构内部：对于 MMX，元素大小的总和限制为 64 位，AltiVec 限制为 128 位。当 Intel 决定扩展到 128 位向量时，它添加了一整套新指令，名为流式 SIMD 扩展（SSE）。

向量计算机的一个主要优势是：一次载入许多元素，然后将执行与数据传输重叠起来，从而隐藏存储器访问的延迟。向量寻址方式的目标是收集散布在存储器中的数据，以紧凑方式放置它们，便于对其进行高效处理，然后再将处理结果放回所属位置。

向量计算机包括**步幅寻址**和**集中/分散**寻址（见 4.2 节），以提高可向量化程序的数目。步幅寻址在每次访问之间跳过数量固定的一些字，所以顺序寻址经常被称为**单位步幅寻址**。集中与分散寻址在另一个向量寄存器中查找其地址：将其看作向量计算机的寄存器间接寻址。与之相对，从向量的角度来看，这些短向量 SIMD 计算机仅支持单位步幅访问：存储器访问一次从单个宽存储器位置载入或存储所有元素。由于多媒体应用程序的数据经常是一些流，起始点和终止点都在存储器中，步幅寻址方式和集中/分散寻址方式是成功实现向量化的必备条件（见 4.7 节）。

举例 下面的例子将一个向量计算机与 MMX 进行对比，将像素的色彩表示方式由 RGB（红、绿、蓝）转换为 YUV（发光度色度），每个像素用 3 个字节表示。这种转换只需要三行 C 代码，放在循环中即可：

```
Y = (9798*R + 19235*G + 3736*B) / 32768;
U = (-4784*R - 9437*G + 4221*B) / 32768 + 128;
V = (20218*R - 16941*G - 3277*B) / 32768 + 128;
```

A-31

宽度为 64 位的向量计算机可以同时计算 8 个像素。一个采用步幅寻址的媒体向量计算机将执行以下操作：

- 3 次向量载入（以获得 RGB）；
- 3 次向量相乘（转换 R）；
- 6 次向量乘加（转换 G 和 B）；
- 3 次向量移位（除以 32 768）；
- 2 次向量加（加上 128）；
- 3 次向量存储（存储 YUV）。

总共有 20 条指令用于执行前面 C 代码中转换 8 个像素的 20 个操作[Kozyrakis 2000]。（由于向量可能有 32 个 64 位元素，这一代码实际上可以转换最多 32×8=256 个像素。）

与之相对，Intel 网站显示一个对 8 个像素执行相同计算的库例程使用了 116 条 MMX 指令和 6 个 80x86 指令[Intel 2001]。指令数之所以会增加到 6 倍是因为没有步幅存储器访问，需要大量的指令来载入 RGB 像素并解包，然后再打包并存储 YUV 像素。

采用受体系结构限制的短向量，而且没有多少寄存器和简单的存储器寻址方式，就很难利用向量化编译器技术。因此，这些 SIMD 指令更可能出现于硬编码库中，而不是编译后的代码中。

A.8.7 小结：编译器的角色

这一节给出了几点建议。第一，我们希望一种新的指令集体系结构中至少拥有 16 个通用寄存器（另外用于浮点数的寄存器不计在内），以简化使用图形着色的寄存器分配。关于正交性的建议意味着所支持的全部寻址方式都适用于所有传送数据的指令。最后的三点建议（提供原型而非解决方案、简化候选项之间的权衡、不要在运行时绑定常量）都意味着注重简单性是最稳妥的。换句话说，要想清楚，在指令集设计中，少就是多。SIMD 扩展是一个出色营销的例子，而不是软硬件协调设计的杰出成果。

A.9 融会贯通：MIPS 体系结构

在这一节，我们介绍一种名为 MIPS 的简单 64 位载入-存储体系结构。MIPS 和 RISC 系列的指令集体系结构的基础与前几节讨论的内容相似。（附录 L.3 节讨论了这些体系结构如何、为何变得流行。）下面回顾一下我们在每一节对桌面应用程序的期望。

A-32

- A.2 节——以载入-存储体系结构使用通用寄存器。
- A.3 节——支持以下寻址方式：位移量（地址偏移大小为 12~16 位）、立即数（大小为 8~16 位）和寄存器间接寻址。

- A.4 节——支持以下数据大小和类型：8 位、16 位、32 位和 64 位整数以及 64 位 IEEE 754 浮点数。
- A.5 节——支持以下简单指令（它们占所执行指令的绝大多数）：载入、存储、加、减、移动寄存器和移位。
- A.6 节——相等、不相等、小于、分支（长度至少为 8 位的 PC 相对地址）、跳转、调用和返回。
- A.7 节——如果关注性能则使用定长指令编码，如果关注代码规模则使用变长指令编码。
- A.8 节——至少提供 16 个通用寄存器，确保所有寻址模式可应用于所有数据传送指令，希望获得最小规模的指令集。这一节并没有包含浮点程序，但它们经常使用独立的浮点寄存器。其理由是增大寄存器的总数，但不会在指令格式或通用寄存器堆的速度方面产生问题。不过，这两个方面并非相互独立。

我们在介绍 MIPS 时，将展示它是如何遵循这些建议的。与最近的大多数计算机类似，MIPS 强调：

- 简单的载入-存储指令集；
- 针对流水线效率的设计（在附录 C 中讨论），包括定长指令集编码；
- 编译器目标的效率。

MIPS 提供了一种非常适合研究的体系结构模型，不仅是因为这种处理器非常普及，还因为它是一种非常易于理解的体系结构。我们已经在第 3 章并将在附录 C 中再次使用这一体系结构，它是大量练习和编程项目的基础。

从 1985 年诞生第一个 MIPS 处理器以来，已经发布了 MIPS 的许多版本（见附录 K）。我们将使用其中一个现在被称为 MIPS64 的子集，它经常被简写为 MIPS，这一完整指令集可以在附录 K 中找到。 A-33

A.9.1 MIPS的寄存器

MIPS64 有 32 个 64 位通用寄存器（GPR），即 R0、R1……R31。GPR 有时也称为**整数寄存器**。此外，还有一组 32 位浮点寄存器（FPR），即 F0、F1……F31，它可以保存 32 个单精度（32 位）值或 32 位双精度（64 位）值。（在保存一个单精度数时，另一半 FPR 没有使用。）它提供了单精度和双精度浮点运算（32 位和 64 位）。MIPS 还包括在单个 64 位浮点寄存器中对两个单精度操作数进行运算的指令。

R0 的值总是 0。我们稍后将会看到如何使用这个寄存器将简单指令集合成各种有用操作。

一些特殊寄存器可以与通用寄存器进行相互转换。其中一个例子就是浮点状态寄存器，用于保存有关浮点运算结果的信息。还有一些指令用于在 FPR 和 GPR 之间移动数据。

A.9.2 MIPS的数据类型

MIPS 的数据类型包括 8 位字节、16 位半字、32 位字和 64 位双字整型数据和 32 位单精度与 64 位双精度浮点数据。添加半字是因为它们在诸如 C 之类的语言中存在这一类型，而且在一些关注数据结构大小的程序（比如操作系统）中非常普遍。如果 Unicode 的应用变得更为广泛，它们也会变得更流行。出于类似原因添加了单精度浮点操作数。（别忘了前面给出的警告：在设计一种指令集之前应当测量大量程序。）

MIPS64 操作对 64 位整数和 32 位或 64 位浮点数进行操作。字节、半字和字被载入通用寄

存器中，并通过重复 0 或符号位来填充 GPR 的 64 个位。一旦载入之后，就可以用 64 位整数运算对其进行操作。

A.9.3　MIPS的数据传输的寻址方式

仅有的数据寻址方式就是立即数寻址和位移寻址，均采用 16 位字段。寄存器间接寻址通过在 16 位位移字段中放置 0 来实现，而采用 16 位字段的寻址则是以寄存器 0 为基址寄存器来完成的。尽管这种体系结构中仅支持两种寻址方式，但通过包含 0 提供了四种有效方式。

MIPS 存储器可以用 64 位地址进行字节寻址。它有一个方式位，允许软件选择大端或小端。由于它是一个载入-存储体系结构，介于存储器与 GPS 或 FRP 之间的所有引用都是通过载入或存储完成的。通过支持上述数据类型，涉及 GPR 的存储器访问可以是字节、半字、字或双字。FPR 可以载入和存储单精度或双精度数。所有存储器访问都必须对齐。

A-34

A.9.4　MIPS指令格式

由于 MIPS 只有两种寻址方式，所以能把它们编码到操作码中。为便于处理器实现流水线和译码，所有指令的长度都是 32 位，其中有一个 6 位的主操作码。图 A-14 显示了指令布局。这些格式非常简单，提供 16 位字段用于位移量寻址、立即数常量寻址或 PC 相对分支寻址。

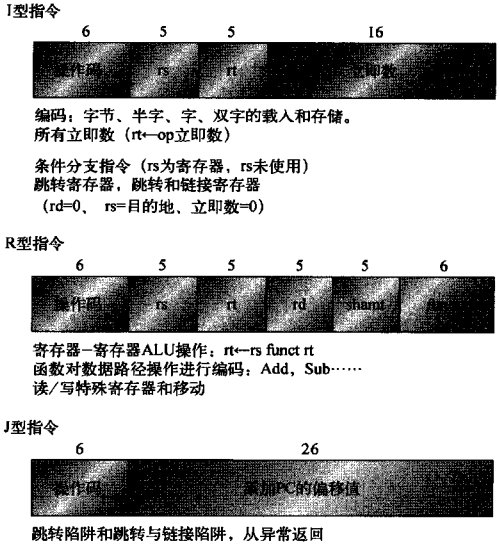


图 A-14　MIPS 的指令布局。所有指令均采用三种类型之一进行编码，在每种格式的相同位置有公共字段

附录 K 给出 MIPS 的一个变体（称为 MIPS 16），它拥有 16 位和 32 位指令来提高嵌入式应用程序的编码密度。本书中一直使用传统的 32 位格式。

A.9.5 MIPS 操作

MIPS 支持前面推荐的简单操作及其他一些操作。共有四大类指令：载入与存储、ALU 运算、分支与跳转、浮点运算。 A-35

任意通用或浮点寄存器都可以载入或存储，只是载入 R0 没有任何效果。表 A-9 给出载入与存储指令的一些例子。单精度浮点数占据浮点寄存器的一半。单、双精度之间的转换必须显式完成。浮点格式为 IEEE 754（参见附录 J）。表 A-12 中列出了这一部分给出的全部 MIPS 指令。

表A-9 MIPS中的载入和存储指令

指令举例	指令名称	含 义
LD R1,30(R2)	载入双字	Regs[R1]←$_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	载入双字	Regs[R1]←$_{64}$ Mem[1000+0]
LW R1,60(R2)	载入字	Regs[R1]←$_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	载入字节	Regs[R1]←$_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	载入无符号字节	Regs[R1]←$_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	载入半字	Regs[R1]←$_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ##Mem[40+Regs[R3]] ## Mem[41+Regs[R3]]
L.S F0,50(R3)	载入单精度浮点数	Regs[F0]←$_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	载入双精度浮点数	Regs[F0]←$_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	存储双精度字	Mem[500+Regs[R4]]←$_{64}$ Regs[R3]
SW R3,500(R4)	存储字	Mem[500+Regs[R4]]←$_{32}$ Regs[R3]$_{32..63}$
S.S F0,40(R3)	存储单精度浮点数	Mem[40+Regs[R3]]←$_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	存储双精度浮点数	Mem[40+Regs[R3]]←$_{64}$ Regs[F0]
SH R3,502(R2)	存储半字	Mem[502+Regs[R2]]←$_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	存储字节	Mem[41+Regs[R3]]←$_{8}$ Regs[R2]$_{56..63}$

* 均使用单一寻址方式，需要存储器值对齐。当然，载入和存储都可用于全部所示数据类型。

为理解这些图形，需要对最初在 A.3.2 节使用的 C 描述语言再进行一些扩展。

- 只要所传送数据的长度不够明确，则向符号←附加一个下标。因此，$\leftarrow_n$表示传送一个 n 位量。我们使用 x，$y \leftarrow z$ 表示应当将 z 传送给 x 和 y。
- 使用一个下标来表示选择字段中的某一位。在标记字段中的各个位时，最高有效位从 0 开始。下标可能是单个数位（例如，Regs[R4]$_0$给出 R4 的符号位），也可能是一个子范围（例如，Regs[R3]$_{56..63}$给出 R3 的最低有效字节）。
- 变量 Mem 用作一个表示主存储器的数组，它是按字节地址索引，可以传送任意数目的字节。
- 使用一个上标来表示复制字段（例如，0^{48}给出长度为 48 位的全零字段）。
- 使用符号##将两个字段串联在一起，可能出现在数据传输的任一端。 A-36

例如，假定 R8 和 R10 为 64 位寄存器：

Regs[R10]$_{32..63}$ ← $_{32}$(Mem[Regs[R8]]$_0$)^{24} ## Mem[Regs[R8]]

上式的含义是对某一存储器位置的字节进行符号扩展（该存储器位置由寄存器 R8 的内容寻址），构成一个 32 位量，存储在寄存器 R10 的低位。（R10 的高位不变。）

所有 ALU 指令都是寄存器-寄存器指令。表 A-10 给出了算术/逻辑指令的一些例子。这些操作包括简单的算术和逻辑运算：加、减、AND、OR、XOR 和移位。所有这些指令的立即数形式都是用 16 位符号扩展立即数提供的。操作 LUI（加载高位立即数）加载一个寄存器的第 32~47 位，并将该寄存器的其他位设置为 0。LUI 允许在两条指令中内置一个 32 位常数，也可以在一个额外指令中使用任意常数 32 位地址进行数据传送。

表A-10　MIPS的算术/逻辑指令的示例，有、无立即数

指令举例	指令名称	含　义
DADDU R1,R2,R3	加上无符号数	Regs[R1]←Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	加上无符号立即数	Regs[R1]←Regs[R2]+3
LUI R1,#42	载入高位立即数	Regs[R1]←0^{32}##42##0^{16}
DSLL R1,R2,#5	逻辑左移	Regs[R1]←Regs[R2]<<5
SLT R1,R2,R3	若小于则置位	if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0

前面曾经提到，R0 用于合并常见操作。载入常数的操作，其实就是执行一个源操作数为 R0 的加立即数指令，寄存器-寄存器移动就是源操作数之一为 R0 的加法。我们有时会使用助记符 LI（表示载入立即数，load immediate）来表示前者，用助记符 MOV 表示后者。

A.9.6　MIPS控制流指令

MIPS 提供比较指令，它比较两个寄存器，查看第一个寄存器是否小于第二个寄存器。如果该条件为真，则这些指令在目标寄存器中放入 1（表示真）；否则，放入数值 0。因为这些操作会对寄存器进行“置位”，所以它们被称为“相等置位”、“不相等置位”、“小于置位”，等等。这些比较指令还有立即数形式。

控制是通过一组跳转指令和一组分支指令处理的。表 A-11 给出了一些典型的分支与跳转指定。通过两种指定目标地址的方法以及是否设定链接来区分四种跳转指令。两种跳转指令使用一个进行两位移位的 26 位偏移量，然后替代程序计数器的低 28 位，以确定目标地址（程序计数器是指该跳转指令后面指令的程序计数器）。另两种跳转指令指定了包含目标地址的寄存器。共有两类跳转：单纯跳转和跳转并链接（用于过程调用）。后者将返回地址（下一条顺序指令的地址）放在 R31 中。

A-37

表A-11　MIPS中的典型控制流指令。所有控制指令（跳转到寄存器中的地址除外）都是PC相对的

指令举例	指令名称	含　义
J　name	跳转	$PC_{36...63}$←name
JAL　name	跳转并链接	Regs[R31]←PC+8; $PC_{36...63}$←name; ((PC + 4)-2^{27}) ≤ name < ((PC+4)+2^{27})
JALR R2	跳转并链接寄存器	Regs[R31]←PC+8; PC←Regs[R2]
JR　R3	跳转寄存器	PC←Regs[R3]
BEQZ R4,name	等于零分支	if (Regs[R4]==0) PC←name; ((PC+4)-2^{17}) ≤ name < ((PC+4)+2^{17})
BNE　R3,R4,name	不等于零分支	if (Regs[R3]!= Regs[R4]) PC←name; ((PC+4)-2^{17}) ≤ name < ((PC+4)+2^{17})
MOVZ R1,R2,R3	等于零时的条件移动	if (Regs[R3]==0) Regs[R1]←Regs[R2]

* 注意，分支距离大于地址字段建议的距离；由于 MIPS 指令的长度都是 32 位，所以将字节分支地址乘以 4，以获得更长的距离。

所有分支都是有条件的。分支条件由指令指定，它可以测试寄存器源操作数是否为 0；寄存器可以包含一个数据值或比较的结果。还有一些条件分支指令用于判断一个寄存器是否为负数，以及两个寄存器是否相等。分支目标地址由一个 16 位有符号偏移量指定，该偏移量被左移两位，然后加到指向下一顺序指令的程序计数器。还有一个分支用来关注浮点条件分支的浮点状态寄存器，如下文所述。

附录 C 和第 3 章表明：条件分支是流水线执行的主要挑战；因此，许多体系结构增加了用于将简单分支转换为条件算术指令的指令。MIPS 包含在等于零或不等于零时执行的条件移动。目标寄存器的值或者保持不变，或者由源寄存器之一的副本替代，具体取决于其他源寄存器的值是否为零。

A.9.7 MIPS 浮点运算

浮点运算操控浮点寄存器，并指出运算是以单精度还是双精度执行。操作 MOV.S 和 MOV.D 将一个单精度（MOV.S）或双精度（MOV.D）浮点寄存器复制到另一个同种类型的寄存器中。操作 MFC1、MTC1、DMFC1 和 DMTC1 在单精度或双精度寄存器与整数寄存器之间移动数据，而且还提供了整数向浮点的转换，反之亦然。 A-38

浮点运算为加、减、乘、除；后缀 D 表示双精度，后缀 S 表示单精度（例如，ADD.D、ADD.S、SUB.D、SUB.S、MUL.D、MUL.S、DIV.D、DIV.S）。浮点比较指令会对特殊浮点状态寄存器中的一个位进行置位，有一对分支指令可以检测这个位，即：BC1T 和 BC1F，也就是浮点真分支和浮点假分支。

为进一步提高图形例程的性能，MIPS64 中提供了一些可以针对 64 位高低位执行 2 个 32 位浮点操作的指令。这些成对单精度操作包括：ADD.PS、SUB.PS、MUL.PS 和 DIV.PS。（它们使用双精度载入和存储指令进行载入和存储。）

MIPS64 认识到多媒体应用程序的重要性，还引入了整数与浮点乘加指令：MADD、MADD.S、MADD.D 和 MADD.PS。在这些合并操作中，寄存器的宽度都相同。表 A-12 给出了一部分 MIPS64 操作的清单及其含义。

表A-12 MIPS64的部分指令

指令类型/操作码	指令含义
数据传送	**在寄存器和存储器之间，或者在整数和FP或特殊寄存器之间移动数据；唯一的存储器寻址模式是16位位移量加上GPR的内容**
LB,LBU,SB	载入字节、载入无符号字节、存储字节（至/自整数寄存器）
LH,LHU,SH	载入半字、载入无符号半字、存储半字（至/自整数寄存器）
LW,LWU,SW	载入字、载入无符号字、存储字（至/自整数寄存器）
LD,SD	载入双字、存储双字（至/自整数寄存器）
L.S,L.D,S.S,S.D	载入SP浮点、载入DP浮点、存储SP浮点、存储DP浮点
MFC0,MTC0	在GPR与特殊寄存器之间复制数据
MOV.S,MOV.D	将一个SP或DP FP寄存器复制到另一个FP寄存器
MFC1,MTC1	在FP寄存器与整数寄存器之间复制32位
算术/逻辑	**对GPR中的整数或逻辑数据进行操作；带符号算术运算溢出时进行陷阱捕获**
DADD,DADDI,DADDU,DADDIU	加，加立即数（所有立即数为16位）；有符号和无符号

（续）

指令类型/操作码	指令含义
DSUB,DSUBU	减，有符号和无符号
DMUL,DMULU,DDIV, DDIVU, MADD	乘和除，有符号和无符号；乘-加；所有运算的操作数和结果都是64位数值
AND,ANDI	与，和立即数相与
OR,ORI,XOR,XORI	或，和立即数求或，异或，和立即数求异或
LUI	载入高位立即数；将立即数载入到寄存器的32－47位，然后进行符号扩展
DSLL,DSRL,DSRA,DSLLV,DSRLV, DSRAV	移位：立即数形式（DS__）和变量形式（DS__V）；移位为左逻辑移位、右逻辑移位、右算术移位
SLT,SLTI,SLTU,SLTIU	若小于操作数则置位、若小于立即数则置位、有符号和无符号
控制	**控制分支和跳转，相对于PC寄存器或通过寄存器控制**
BEQZ,BNEZ	GPR等于/不等于0时转移、相对于PC+4偏移16位偏移量
BEQ,BNE	GPR相等/不相等时转移、相对于PC+4转移16位偏移量
BC1T,BC1F	测试FP状态寄存器中的对比位，并转移；相对于PC+4转移16位偏移量
MOVN,MOVZ	如果第三个GPR为负数/零，则将第一个GPR复制到第二个GPR
J,JR	跳转至与PC+4偏移26位偏移量的位置（J）、跳转至寄存器中的目标位置（JR）
JAL,JALR	跳转和链接：将PC+4保存在R31中，目标为相对于PC（JAL）或寄存器（JALR）
TRAP	转移到操作系统的一个向量地址
ERET	从异常中返回用户代码，恢复用户模式
浮点	**对DP和SP格式执行FP操作**
ADD.D,ADD.S,ADD.PS	DP、SP数相加，一对SP数相加
SUB.D,SUB.S,SUB.PS	DP、SP数相减，一对SP数相减
MUL.D,MUL.S,MUL.PS	DP、SP浮点数相乘，一对SP数相乘
MADD.D,MADD.S,MADD.PS	DP、SP浮点数相乘加，一对SP数相乘加
DIV.D,DIV.S,DIV.PS	DP、SP浮点数相除，一对SP数相除
CVT._._	转换指令：CVT.x.y从类型x转换为类型y，其中x和y为L（64位整数），W（32位整数），D（DP）或S（SP）。两个操作数都是FRP
C.__.D,C.__.S	DP和SP对比："__"=LT，GT，LE，GE，EQ，NE；在FP状态寄存器中置位

* 图 A-14 列出了这些指令的格式。

A.9.8　MIPS 指令集的使用

为使读者了解哪些指令使用得更为频繁，表 A-13 中给出了 5 个 SPECint2000 程序中各指令及指令类别的使用频率，表 A-14 给出了对于 5 个 SPECfp2000 程序的相同数据。

表A-13　5个SPECint2000程序的MIPS动态指令比例

指　令	gap	gcc	gzip	mcf	perlbmk	整数均值
载入	26.5%	25.1%	20.1%	30.3%	28.7%	26%
存储	10.3%	13.2%	5.1%	4.3%	16.2%	10%
加	21.1%	19.0%	26.9%	10.1%	16.7%	19%
减	1.7%	2.2%	5.1%	3.7%	2.5%	3%
乘	1.4%	0.1%				0%

（续）

指　令	gap	gcc	gzip	mcf	perlbmk	整数均值
比较	2.8%	6.1%	6.6%	6.3%	3.8%	5%
载入立即数	4.8%	2.5%	1.5%	0.1%	1.7%	2%
条件分支	9.3%	12.1%	11.0%	17.5%	10.9%	12%
条件移动	0.4%	0.6%	1.1%	0.1%	1.9%	1%
跳转	0.8%	0.7%	0.8%	0.7%	1.7%	1%
调用	1.6%	0.6%	0.4%	3.2%	1.1%	1%
返回	1.6%	0.6%	0.4%	3.2%	1.1%	1%
移位	3.8%	1.1%	2.1%	1.1%	0.5%	2%
与	4.3%	4.6%	9.4%	0.2%	1.2%	4%
或	7.9%	8.5%	4.8%	17.6%	8.7%	9%
异或	1.8%	2.1%	4.4%	1.5%	2.8%	3%
其他逻辑	0.1%	0.4%	0.1%	0.1%	0.3%	0%
载入浮点						0%
存储浮点						0%
加浮点						0%
减浮点						0%
乘浮点						0%
除浮点						0%
移动寄存器-寄存器浮点						0%
比较浮点						0%
条件移动浮点						0%
其他浮点						0%

* 注意整数寄存器-寄存器移动指令包含在“或”指令中。空白项表示取值为0.0%。

表A-14　SPECfp2000中5个程序的MIPS动态指令比例

指　令	applu	art	equake	lucas	swim	浮点均值
载入	13.8%	18.1%	22.3%	10.6%	9.1%	15%
存储	2.9%		0.8%	3.4%	1.3%	2%
加	30.4%	30.1%	17.4%	11.1%	24.4%	23%
减	2.5%		0.1%	2.1%	3.8%	2%
乘	2.3%			1.2%		1%
比较		7.4%	2.1%			2%
载入立即数	13.7%		1.0%	1.8%	9.4%	5%
条件分支	2.5%	11.5%	2.9%	0.6%	1.3%	4%
条件移动		0.3%	0.1%			0%
跳转			0.1%			0%
调用			0.7%			0%
返回			0.7%			0%

（续）

指　令	Applu	Art	Equake	Lucas	Swim	浮点均值
移位	0.7%		0.2%	1.9%		1%
与			0.2%	1.8%		0%
或	0.8%	1.1%	2.3%	1.0%	7.2%	2%
异或		3.2%	0.1%			1%
其他逻辑			0.1%			0%
载入浮点	11.4%	12.0%	19.7%	16.2%	16.8%	15%
存储浮点	4.2%	4.5%	2.7%	18.2%	5.0%	7%
加浮点	2.3%	4.5%	9.8%	8.2%	9.0%	7%
减浮点	2.9%		1.3%	7.6%	4.7%	3%
乘浮点	8.6%	4.1%	12.9%	9.4%	6.9%	8%
除浮点	0.3%	0.6%	0.5%		0.3%	0%
移动寄存器-寄存器浮点	0.7%	0.9%	1.2%	1.8%	0.9%	1%
比较浮点		0.9%	0.6%	0.8%		0%
条件移动浮点		0.6%		0.8%		0%
其他浮点				1.6%		0%

* 注意整数寄存器-寄存器移动指令包含在“或”指令中。空白项表示取值为 0.0%。

A.10 谬论和易犯错误

架构师经常会抱有一些常见但却错误的观点。我们将在本节研究其中几个。

易犯错误 设计专门支持高级语言结构的“高级”指令集功能。

架构师试图在指令集中整合高级语言功能，从而提供功能强大、极具灵活性的指令。但是，这些指令所完成的工作通常会超出常见情景下的需求，或者不能与某些语言的需求完全匹配。许多此类努力的目的是消除 20 世纪 70 年代所说的语义鸿沟。尽管其思路是对指令集进行一些补充，使硬件能够实现语言级的功能，但这些补充可能会生成 Wulf、Levin 和 Harbison [1981] 所说的语义冲突：

A-39 ~ A-40

> ……计算机设计者赋予指令过多的语义内容之后，也许只能在非常有限的上下文中使用这些指令[P43]。

更经常出现的情况是，这些指令的功能过于强大——对最常见情景来说，它们太具一般性，做了一些无用功，减缓了指令的执行速度。VAX CALLS 又是一个好例子。CALLS 采用了“被调用者保存策略”（要保存的寄存器由被调用者指定），但是，保存过程是由调用方的调用指令完成的。CALLS 指令首先将参数压入栈，然后执行以下步骤。

A-41

(1) 必要时对齐栈。

(2) 将参数数目压入栈。

(3) 将过程调用屏蔽码指定的寄存器保存到栈中（见 A.8 节）。这个屏蔽码保存在被调用过程的代码中，这样，即使采用分离编译，也能允许被调用者指定要由调用者保存的寄存器。

A-42

(4) 将返回地址压入栈，然后压入栈底、栈顶指针（对于活动记录）。

(5) 清除条件代码，这样会将陷阱使能设置为已知状态。

(6) 在栈中压入表示状态信息的字和零字。

(7) 更新两处栈指针。

(8) 分支转移到过程的第一条指令。

实际程序中的大量调用并不需要这么多开销。大多数过程知道它们的参数个数，可以建立一条更快速的链接协定，使用寄存器而不是存储器栈来传递参数。此外，CALLS 指令强制为链接使用两个寄存器，而许多语言只需要一个链接寄存器。很多希望支持过程调用和活动栈管理的努力都失败了，要么是因为不满足语言要求，要么是通用性太强，使用成本过于高昂。

VAX 设计者提供了一种更简单的指令 JSB，由于它仅在栈中压入返回 PC 就跳至进程，所以执行速度要快得多。但是，大多数 VAX 编译器使用成本更高的 CALLS 指令。这些调用指令包含在体系结构中，用于实现过程链接约定的标准化。其他计算机通过在编译器编写人员之间达成一致而实现其调用约定的标准化，不再需要非常通用的复杂过程调用指令的开销。

谬论 *存在典型程序这样一种东西。*

许多人愿意相信存在一种用于设计最优指令集的单一“典型”程序。例如，参见第 1 章讨论的合成基准测试。本附录中的数据清楚地表明：各个程序对指令集的使用方式存在显著不同。例如，图 A-15 给出了 4 个 SPEC2000 程序中数据传送大小所占的比例：很难说这 4 个程序中哪个才是典型的。对于专门支持某一类应用的指令集，这种变化可能更大，比如其他应用就不会使用十进制指令。

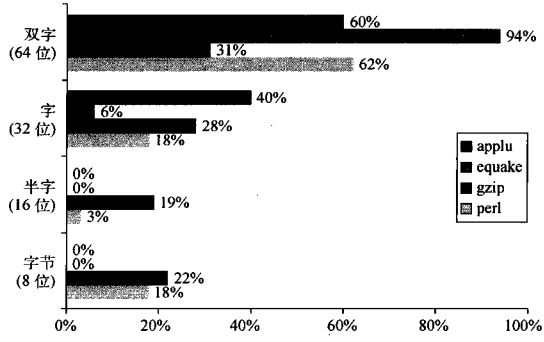


图 A-15 4 种 SPEC2000 程序的数据引用大小。尽管可以计算平均大小，但很难声称这个平均值就是程序的典型值

易犯错误 *不考虑编译器，仅通过指令集体系结构的创新来缩小代码规模。*

表 A-15 显示了 MIPS 指令集四种编译器的相对代码规模。架构师一直在为了将代码规模缩小 30%~40%而努力，而不同编译器策略对代码规模的影响则要大得多。和性能优化技术相似，架构师应当首先考虑编译器所能生成的最紧凑代码，然后再考虑通过硬件创新来节省空间。 A-43

表A-15 EEMBC基准测试Telecom应用程序相对于Apogee Software Version 4.1 C的代码规模

编 译 器	Apogee Software Version 4.1	Green Hills Multi2000 Version 2.0	Algorithmics SDE4.0B	IDT/c 7.2.1
体系结构	MIPS IV	MIPS IV	MIPS 32	MIPS 32
处理器	NEC VR5432	NEC VR5000	IDT 32334	IDT 79RC32364
自动相关内核	1.0	2.1	1.1	2.7
卷积编码器内核	1.0	1.9	1.2	2.4
定点位分配内核	1.0	2.0	1.2	2.3
定点复数FFT内核	1.0	1.1	2.7	1.8
维特比GSM解码器内核	1.0	1.7	0.8	1.1
5种内核的几何平均	1.0	1.7	1.4	2.0

* 这些指令集体系结构几乎相同，而代码规模却相差一倍。这些结果于 2000 年 2 月~6 月报告。

谬论 有缺陷的体系结构不可能获得成功。

80x86 提供了一个很生动的例子：可能只有其创造者才喜爱这一指令集体系结构（见附录 K）。后来的 Intel 工程师已经尝试纠正在设计 80x86 时作出的一些不受欢迎的体系结构决策。例如，80x86 支持段式存储，而所有其他体系结构都选择了页式存储；它为整型数据使用扩展累加器，而其他处理器则使用通用寄存器；它为浮点数使用栈，而其他所有人在很久之前就放弃了执行栈。

A-44

尽管存在这些重大问题，80x86 体系结构仍然取得了巨大成功。其原因有三个方面：第一，最初的 IBM PC 选择它作为微处理器，使 80x86 的二进制兼容性变得极为重要。第二，摩尔定律提供了足够的资源，供 80x86 微处理器先转换为内部 RISC 指令集，然后再执行类似于 RISC 的指令。这种混合方式既能保证与宝贵的 PC 软件基础保持二进制兼容，又能实现与 RISC 处理器相当的性能。第三，PC 微处理器的销售量如此之高，使 Intel 可以很轻松地支付不断增加的硬件转换设计成本。此外，高销售量使制造商能够抬高学习曲线，降低生产成本。

为进行转换而增大晶粒大小、提高功率，对于嵌入式应用程序来说可能会成为一种负担，而对于桌面应用来说则极具经济意义。而且，它在桌面应用方面的成本性能也吸引了服务器领域的关注，对服务器而言，它的主要弱点是采用了 32 位地址，AMD64 中用 64 位地址克服了这一缺点（见第 2 章）。

谬论 可以设计一种没有缺陷的体系结构。

所有体系结构设计都需要在一组软硬件技术之间进行折中。随着时间的流逝，这些技术可能会发生变化，原来曾经正确的决定可能看起来像是一些错误决定。例如，1975 年，VAX 设计人员强调代码规模效率的重要性，低估了译码与流水化的简易程度在 5 年之后的重要性。RISC 阵营的一个例子是延迟分支（参见附录 K）。对于五级流水线来说，控制流水线冒险是很简单的事情，但如果处理器的流水线更长，在每个时钟周期发射多条指令，那就是一个挑战了。此外，几乎所有体系结构最终都会屈从于缺少足够地址空间这一问题。

总体来说，避免体系结构出现这些长期问题可能意味着会影响到它的短期效率，这是非常危险的，因为一种新的指令集体系结构必须挣扎着经历最初几年的考验。

A.11 结语

最早体系结构的指令集受到当时硬件技术的限制。只要硬件技术允许，计算机架构师就会探索支持高级语言的方式。因为这一探索，在三个不同时期内，人们关于如何高效支持程序有着截然不同的考虑。20 世纪 60 年代，栈体系结构变得非常流行。人们认为它们与高级语言非常匹配，根据当时的编译器技术，可能也确实如此。20 世纪 70 年代，架构师主要关注如何降低软件成本。其解决方案主要是用硬件代替软件，或者提供能够简化软件设计人员任务的高级体系结构。其结果就是高级语言计算机体系结构和诸如 VAX 之类的强大体系结构，这种体系结构有大量的寻址方式、多种数据类型和高度正交的体系结构。20 世纪 80 年代，一些更高级的编译器技术和对处理器性能的再度重视见证了简单体系结构的回归，其主要基础就是载入-存储型计算机。 A-45

20 世纪 90 年代，指令集体系结构发生了以下变化。

- **地址大小加倍**——大多数台式与服务器处理器的 32 位地址指令集被扩展到 64 位地址，寄存器的宽度（及其他相关项目）被扩展到 64 位。附录 K 给出三个已经从 32 位扩展到 64 位的体系结构实例。
- **通过条件执行优化条件分支**——在第 3 章，我们看到条件分支可能限制积极计算机设计的性能。因此，人们愿意将条件分支代替操作的条件执行，比如条件移动（见附录 H），大多数指令集中都添加了这一指令。
- **通过预取优化缓存性能**——第 2 章曾经解释过，由于一些计算机中发生缓存缺失时所消耗的指令时间可能与页面错误在早期计算机上消耗的时间一样多，所以存储器层次结构在计算机性能中扮演着更为重要的角色。因此添加了预取指令，以尝试通过预取来隐藏缓存缺失成本（见第 2 章）。
- **支持多媒体**——大多数台式和嵌入式指令集都进行了扩展，为多媒体应用程序提供支持。
- **浮点运算速度更快**——附录 J 描述了为提高浮点性能所添加的操作，比如执行乘加和成对单次执行的操作。（我们在 MIPS 中包含了此类操作。）

在 1970 年至 1985 年期间，许多人认为计算机架构师的主要任务是设计指令集。结果，当时的教科书都在强调指令集设计，就像 20 世纪 50 年代、60 年代的计算机体系结构教科书强调计算机算术运算一样。人们希望经过专业训练的架构师能够深刻地了解流行计算机的优势与缺点，特别是对后者的了解。在指令集设计的改进过程中，许多研究人员和教科书编写人员都没有认识到二进制兼容性的重要性，给人们留下一种印象：许多架构师都有机会设计一种指令集。

今天的计算机体系结构定义已经进行了扩展，包含整个计算机系统的设计与评估，而不只是指令集的定义，也不只是处理器的定义，因此，架构师有大量主题需要研究。事实上，本附录中的材料是本书 1990 年第一次出版时的中心内容，但现在它被放在附录中，主要作为参考材料！

附录 K 可以满足那些对指令集体系结构感兴趣的读者；其中介绍了各种指令集，这些指令集要么在今天的市场上非常重要，要么从历史的角度来说非常重要，它将 9 种流行的载入-存储计算机与 MIPS 进行了对比。 A-46

A.12 历史回顾与参考文献

附录 L.4 节讨论了指令集的演进，并给出了一些参考文献，供深入阅读和探讨相关主题。

练习（Gregory D. Peterson设计）

A.1 [15] <A.9>使用表 A-13 计算 MIPS 的实际 CPI。假定我们已经对各指令类型的平均 CPI 进行了以下测量：

指　令	时钟周期
所有ALU指令	1.0
载入-存储	1.4
条件分支	
选中	2.0
未选中	1.5
跳转	1.2

假定有 60%的条件分支被选中，表 A-13 中“其他”类别的所有指令都是 ALU 指令。对 gap 和 gcc 的指令频率求平均值，以获得指令比例。

A.2 [15] <A.9>利用表 A-13 和上表计算 MIPS 的实际 CPI。对 gzip 和 perlbmk 的指令频率求平均值，以获得指令比例。

A.3 [20] <A.9>使用表 A-14 计算 MIPS 的实际 CPI。假定我们已经对各指令类型的平均 CPI 进行了以下测量：

指　令	时钟周期
所有ALU指令	1.0
载入-存储	1.4
条件分支	
选中	2.0
未选中	1.5
跳转	1.2
浮点乘	6.0
浮点加	4.0
浮点除	20.0
载入-存储浮点	1.5
其他浮点	2.0

A-47 假定有 60%的条件分支被选中，表 A-14 中“其他”类别的所有指令都是 ALU 指令。对 lucas 和 swim 的指令频率求平均值，以获得指令比例。

A.4 [20] <A.9>利用表 A-14 和上表计算 MIPS 的实际 CPI。对 applu 和 art 的指令频率求平均值，以获得指令比例。

A.5 [10] <A.8>考虑下面这个由 3 行语句组成的高级代码序列：

```
A = B + C;
B = A + C;
D = A - B;
```

使用复制传播技术（见表 A-8）将此代码序列转换为所有操作数都不是计算值的情况。注意这种转换减少了语句计算任务的实例，以及加重这些任务的情况。为什么这意味着在试图满足优化编译器的期望时所面对的技术挑战？

A.6 [30] <A.8>编译器优化可能会改进代码大小及/或性能。考虑 SPEC CPU2006 套件中的一或多个基准测试程序。使用可以使用的处理器和 GNU C 编译器，采用无优化、-O1、-O2 和-O3 来优化程序。对比所得程序的性能和规模。并将所得结果与图 A-13 进行对比。

A.7 [20/20] <A.2、A.9>考虑以下 C 代码段：

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

假定 A 和 B 是 64 位整数的数组，C 和 i 是 64 位整数。假定所有数据值及其地址都保存在存储器中（A、B、C、i 分别位于地址 1000、3000、5000、7000 处），但在对其进行操作时例外。假定寄存器中的值在该循环的各次迭代之间丢失。

a. [20] <A.2、A.9>写出 MIPS 的代码。动态需要多少条指令？将执行多少次存储器数据引用？代码大小为多少字节？

b. [20] <A.2>写出 x86 的代码。动态需要多少条指令？将执行多少次存储器数据引用？代码大小为多少字节？

A.8 [10/10/10] <A.2、A.7>对于以下练习，考虑针对指令集体系结构的指令编码。 A-48

a. [10] <A.2、A.7>考虑以下情景：处理器的指令长度为 12 位，有 32 个通用寄存器，所以地址字段的大小为 5 位。是否有可能拥有如下指令编码？

- 3 个两地址指令
- 30 个单地址指令
- 45 个零地址指令

b. [10] <A.2、A.7>假定指令长度和地址段大小如上所述，判断是否有可能拥有以下指令编码。

- 3 个两地址指令
- 31 个单地址指令
- 35 个零地址指令

解释你的回答。

c. [10] <A.2、A.7>假定指令长度和地址段大小如上所述。进一步假定已经拥有 3 个两地址指令和 24 个零地址指令。最多可以为这一处理器编码多少个单地址指令？

A.9 [10/15] <A.2>对于以下练习，假定 A、B、C、D、E 和 F 驻存在存储器中。另外假定指令操作码以 8 位表示，存储器地址为 64 位，寄存器地址为 6 位。

a. [10] <A.2>对于表 A-1 所示的每个指令集体系结构，对于计算 C=A+B 的代码，每条指令中出现多少个地址或名称？总代码大小为多少？

b. [15] <A.2>表 A-1 中的一些指令集体系结构会在计算过程中销毁操作数。这种在处理器器内部存储中丢失数据值的情况会造成性能影响。对于表 A-1 中的每种体系结构，编写代码序列，以计算：

```
C = A + B
D = A - E
F = C + D
```

在代码中，标出所有将在执行期间被销毁的操作数，有些指令的存在只是为了应对处理器内部存储器的数据丢失，也请标出所有这些“开销”指令。对于每段代码序列，总代码大小、向（自）存储器移动的指令与数据的字节数、开销指令的数量、开销数据字节的数目各为多少？

A.10 [20] <A.2、A.7、A.9>MIPS 的设计提供了 32 个通用寄存器和 32 个浮点寄存器。如果这些寄存

A-49 器表现良好，那包含更多寄存器是不是更好一些？当指令集体系结构设计人员在确定是否要增加 MIPS 寄存器、应当增加多少时，必需考虑一些折中，请尽你所能列出并讨论这些折中。

A.11 [5] <A.3>考虑包含以下成员的 C 结构：

```
struct foo {
        char a;
        bool b;
        int c;
        double d;
        short e;
        float f;
        double g;
        char * cptr;
        float * fptr;
        int x;
};
```

对于 32 位机器，foo 结构的大小为多少？假定可以任何安排结构成员的顺序，这一结构最小为多少？对于 64 位机器呢？

A.12 [30] <A.7>许多计算机制造商现在在处理器中包含了一些工具或模拟器，可以用来测量用户程序的指令集使用情况。所使用的方法包括机器模拟、硬件支持的陷阱中断，还有一种编译器技术，通过插入计数器来测量目标代码模块。找一种自己能够使用并包含此种工具的处理器。用它来测量每个 SPEC CPU2006 基准测试的指令集比例。将测量结果与本章所示结果进行对比。

A.13 [30] <A.8>诸如 Intel 的 i7 Sandy Bridge 等新处理器都支持 AVX 向量/多媒体指令。编写一个使用单精度值的密集矩阵相乘函数，并将用不同编译器和优化标志进行编译。使用基础线性代码子例程（BLAS）（比如 SGEMM）的线性代数代码包含了密集矩阵乘法的优化版本。对比你所编写代码与 BLAS SGEMM 代码的大小和性能。探讨在使用双精度值和 DGEMM 时的情景。

A.14 [30] <A.8>对于上面为 i7 处理器开发的 SGEMM 代码，使用 AVX 内建函数来提高性能。具体来说，尝试实现代码的向量化，以更好地利用 AVX 硬件。将其代码大小和性能与原代码进行对比。

A.15 [30] <A.7、A.9>SPIM 是一种用于模拟 MIPS 处理器的流行模拟器。使用 SPIM 来测量一些 SPEC CPU2006 基准测试程序的指令集比例。

A.16 [35/35/35/35] <A.2–A.8>gcc 的测试目标是最现代的指令集体系结构（参见 www.gnu.org/software/gcc/）。针对你能够使用的几种体系结构（例如，x86、MIPS、PowerPC 和 ARM）创建 gcc 的一个版本。

a. [35] <A.2–A.8>编译 SPEC CPU2006 整数基准测试的一部分，并绘制一个表格，列出其代码大小。对于每个程序，哪种体系结构的运行效果最佳？

A-50

b. [35] <A.2–A.8>编译 SPEC CPU2006 浮点基准测试的一部分，并绘制一个表格，列出其代码大小。对于每个程序，哪种体系结构的运行效果最佳？

c. [35] <A.2–A.8>编译 EEMBC AutoBench 基准测试的一部分（参见 www.eembc.org/home.php），并绘制一个表格，列出其代码大小。对于每个程序，哪种体系结构的运行效果最佳？

d. [35] <A.2–A.8>编译 EEMBC FPBench 浮点基准测试的一部分，并绘制一个表格，列出其代码大小。对于每个程序，哪种体系结构的运行效果最佳？

A.17 [40] <A.2–A.8>对于现代处理器，特别是嵌入式系统，功率效率已经变得非常重要。为你能使用的两种体系结构（比如 x86、MIPS、PowerPC 和 ARM）创建一个 gcc 版本。编译一部分 EEMBC 基准测试，并使用 EnergyBench 来测试执行期间的能量使用情况。对比针对不同处理器的代码大小、性能和能量使用。对于每个程序，哪种处理器的运行效果最佳？

A.18 [20/15/15/20]你的任务是对比 4 种不同指令集体系结构的存储器效率。这 4 种体系结构类型如下所述。

- **累加器**——所有操作都在单个寄存器和存储器位置之间进行。
- **存储器–存储器**——所有指令地址仅引用存储器位置。

- **栈**——所有操作都在栈顶部执行。只有压入栈和弹出栈指令会访问存储器；所有其他指令都删除其在栈中的操作数，代以结果。这种实施方式仅为顶部两个栈项使用硬连线栈，这样可以使处理器电路保持小型化，并降低其成本。其他栈位置保存在存储器位置中，对这些栈位置的访问需要进行存储器引用。
- **载入-存储**——所有操作都是在寄存器中进行，每个寄存器至寄存器指令有 3 个寄存器名称。

为了测量存储器效率，对所有 4 种指令集作出以下假定。

- 所有指令的长度都是整数个字节。
- 操作码的长度均为一个字节（8 位）。
- 存储器访问使用直接寻址或绝对寻址。
- 变量 A、B、C 和 D 最初在存储器中。

a. [20] <A.2、A.3>创建自己的汇编语言标记（表 A-1 提供了一个可供推广的有用示例），针对每种体系结构为以下高级语言代码序列编写最为等价的汇编语言代码： A-51

```
A = B + C;
B = A + C;
D = A - B;
```

b. [15] <A.3>对于（a）部分中编写的汇编代码，如果有一个值已经被载入一次后再次从存储器中载入，请标出此类实例。如果一条指令的结果被作为操作数传递给另一条指令，也请标出，并对这些情况进行分类，说明哪些涉及处理器内部的存储，哪些涉及存储器内部的存储。

c. [15] <A.7>假定给定的代码序列来自一个小型的嵌入式应用程序，比如微波炉控制器，它的存储器地址和数据操作数为 16 位。如果使用载入-存储体系结构，则假定它有 16 个通用寄存器。对于每种体系结构，回答以下问题：提取了多少个指令字节？向（从）寄存器传送多少个数据字节？根据测得的总存储器通信流量（代码+数据），哪种体系结构的效率最高？

d. [20] <A.7>现在假定采用存储器地址和数据操作数为 64 位的处理器。针对每种体系结构，回答（c）部分的问题。对于选定的度量，这些体系结构的相对优势有哪些变化？

A.19 [30] <A.2、A.3>使用上面给出的 4 种不同指令集体系结构，但假定所支持的存储器操作包括寄存器间接寻址和直接寻址。创建自己的汇编语言标记（表 A-1 提供了一个可供推广的有用示例），针对每种体系结构为以下 C 代码段编写最为等价的汇编语言代码：

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

假定 A 和 B 是 64 位整数的数组，C 和 i 是 64 位整数。

第二列和第三列分别包含数据引用和分支的累积百分比，可以在位移量中用相应的位数来提供这些引用和分支。这是图 A-4 和图 A-9 中所有整数与浮点程序的平均距离。

A.20 [20/20/20] <A.3>我们正在为一种载入-存储体系结构设计指令集格式，并试图判断是否值得为分支和存储器引用包含多种偏移长度。一条指令的长度等于（16+偏移长度）位，所以 ALU 指令的长度为 16 位。表 A-16 包含了 Alpha 体系结构的偏移量大小数据，其中针对 SPEC CPU2000 进行了完全优化。关于指令集频率，使用表 A-13 中 5 种基准测试针对载入-存储机器测得的 MIPS 平均数据。假定其他指令均为仅使用寄存器的 ALU 指令。 A-52

表A-16 Alpha体系结构的偏移量大小数据，对SPEC CPU2000进行了全面优化

偏移量的位数	累积数据引用	累积分支
0	30.4%	0.1%
1	33.5%	2.8%
2	35.0%	10.5%
3	40.0%	22.9%

（续）

偏移量的位数	累积数据引用	累积分支
4	47.3%	36.5%
5	54.5%	57.4%
6	60.4%	72.4%
7	66.9%	85.2%
8	71.6%	90.5%
9	73.3%	93.1%
10	74.2%	95.1%
11	74.9%	96.0%
12	76.6%	96.8%
13	87.9%	97.4%
14	91.9%	98.1%
15	100%	98.5%
16	100%	99.5%
17	100%	99.8%
18	100%	99.9%
19	100%	100%
20	100%	100%
21	100%	100%

a. [20] <A.3>假定允许偏移量的长度为 0、8、16 或 24 位，包括符号位在内。所执行指令的平均长度是多少？

b. [20] <A.3>假定我们希望有固定长度的指令，并选择指令长度为 24 位（所有指令均如此，包括 ALU 指令）。对于每个大于 8 位的偏移量，需要增加指令。判断在这种采用固定指令长度的机器中所提取的指令字节数，并对比(a) 部分定义的变长指令中所提取的字节数。

c. [20] <A.3>现在假定我们使用 24 位的固定偏移量长度，所以不需要额外指令。需要多少指令字节数？将这一结果与(b) 部分的答案进行对比。

A.21 [20/20] <A.3、A.6、A.9>位移量寻址方式或 PC 相对寻址方式的位移值大小可以从编译后的应用程序中获取。对于针对 MIPS 处理器编译的一或多个 SPEC CPU2006 基准测试使用反汇编程序。

A-53

a. [20] <A.3、A.9>对于每条使用位移量寻址的指令，记录所使用的位移量值。生成位移量值的一个直方图。将这里所示的结果与图 A-4 中所示的结果进行对比。

b. [20] <A.6、A.9>对于每条使用 PC 相对寻址的指令，记录所使用的位移量值。生成位移量值的一个直方图。将这里所示的结果与图 A-9 中所示的结果进行对比。

A.22 [15/15/10/10] <A.3>一数值用十六进制数字 434F 4D50 5554 4552 表示，将其存储在 64 位对齐双字中。

a. [15] <A.3>使用图 A-2 中第一行的物理排列，用大端字节顺序写出要存储的值。接下来，将每个字节解读为 ASCII 字符，在每个字节下面写出相应的字符，形成将要以大端顺序存储的字符串。

b. [15] <A.3>使用(a) 部分中的相同物理排列，用小端字节顺序写出要存储的址，并在每个字节下面写出相应的 ASCII 字节。

c. [10] <A.3>当以大端字节顺序存储时，可以从给定 64 位双字中读取的所有非对齐 2 字节字的十六进制取值为多少？

d. [10] <A.3>当以小端字节顺序存储时，可以从给定 64 位双字中读取的所有非对齐 4 字节字的十六进制取值为多少？

A.23 [讨论] <A.2–A.12>考虑桌面、服务器、云和嵌入式计算的典型应用程序。对于面向这些市场的计算机而言，指令集体系结构将会受到什么样的影响？

A-54

B

存储器层次结构回顾

缓存：隐藏或存储东西的安全位置。

《韦氏新世界美语词典》
学院版第 2 版（1976）

B.1 引言

本附录对存储器层次结构进行了快速回顾，包括缓存与虚拟存储器的基础知识、性能公式、简单优化。第一节介绍下面 36 个术语。

缓存（cache）	全相联（fully associative）	写入分派（write allocate）
虚拟存储器（virtual memory）	重写（脏）位（dirty bit）	统一缓存（unified cache）
存储器停顿周期（memory stall cycles）	块偏移（block offset）	每条指令缺失数（misses per instruction）
直接映射（direct mapped）	写回（write back）	块（block）
有效位（valid bit）	数据缓存（data cache）	局域性（locality）
块地址（block address）	命中时间（hit time）	地址跟踪（address trace）
直写（write through）	缓存缺失（cache miss）	组（set）
指令缓存（instruction cache）	页面错误（page fault）	随机替换（random replacement）
存储器平均访问时间（average memory access time）	缺失率（miss rate）	索引字段（index field）
缓存命中（cache hit）	*n*路组相联（*n*-way set associative）	无写入分派（no-write allocate）
页（page）	最近最少使用（least-recently used）	写入缓冲区（write buffer）
缺失代价（miss penalty）	标志字段（tag field）	写停顿（write stall）

如果这一部分回顾内容过于简略，读者可以查看《计算机组成与设计》一书的第 7 章，那是我们专为初学者编写的。

缓存是指地址离开处理器后遇到的最高级或第一级存储器层次结构。由于局域性原理适用于许多级别，而且充分利用局域性来提高性能的做法非常普遍，所以现在只要利用缓冲方法来重复使用常见项目，就可以使用**缓存**一词，**文件缓存**、**名称缓存**等都是一些实例。

如果处理器在缓存中找到了所需要的数据项，就说发生了**缓存命中**。如果处理器没有在缓存中找到所需要的数据项，就是发生了**缓存缺失**。从主存储器中提取固定大小且包含所需字的数据集，并将其放在缓存中，这个数据集称为**块**。**时域局域性**告诉我们：我们很可能会在不远的将来再用到这个字，所以把它放在缓存中是有用的，在这里可以快速访问它。由于**空域局域性**，马上用到这个块中其他数据的可能性也很高。

缓存缺失需要的时间取决于存储器的延迟和带宽。延迟决定了提取块中第一个字的时间，带宽决定了提取这个块中其他内容的时间。缓存缺失由硬件处理，会导致采用循序执行方式的处理器暂停或停顿，直到数据可用为止。在采用乱序执行方式时，需要使用该结果的指令仍然必须等待，但其他指令可以在缺失期间继续进行。

B-2

与此类似，程序引用的所有对象不一定都要驻存在主存储器中。**虚拟存储器**意味着一些对象可以驻存在磁盘上。地址空间通常被分为固定大小的块，称为页。在任何时候，每个页要么在主存储器中，要么在磁盘上。当处理器引用一个页中既不在缓存中也不在主存储器中的数据项时，就会发生**页错误**，并把整个页从磁盘移到主存储器中。由于页错误消耗的时间太长，所以它们由软件处理，处理器不会停顿。在进行磁盘访问时，处理器通常会切换到其他某一任务。从更高级别来看，缓存和主存储器在对引用局域性的依赖性方面、在大小和单个位成本等方面的关系，类似于主存储器与磁盘的相应关系。

表 B-1 给出了各种计算机（从高端台式机到低端服务器）每一级存储器层次结构的大小与访问时间范围。

表B-1 大型工作站或小型服务器存储器层次结构的典型级别，在远离处理器时，相应级别的速度减缓、容量变大

级别	1	2	3	4
名称	寄存器	缓存	主存储器	磁盘存储
典型大小	< 1 KB	32 KB~8 MB	< 512 GB	>1 TB
实现技术	具有多个端口的定制存储器、CMOS	片上CMOS SRAM	CMOS DRAM	磁盘
访问时间（ns）	0.15~0.30	0.5~15	30~200	5 000 000
带宽（MB/s）	100 000-1 000 000	10 000-40 000	5000-20 000	50-500
管理者	编译器	硬件	操作系统	操作系统/操作员
支持者	缓存	主存储器	磁盘	其他磁盘和DVD

* 嵌入式计算机可能没有磁盘存储，存储器和缓存也要小得多。在移向层次结构的更低级别时，访问时间延长，从而有可能以较低的反应速度来管理数据传输。实现技术显示了这些功能所用的典型技术。表中给出的访问时间数据是2006年的典型值，单位为纳秒；这些时间数据将随时间的推移而缩小。存储器层次结构各级之间的带宽以 MB/s 单位给出。磁盘存储的带宽包括介质和缓冲接口的带宽。

B.1.1 缓存性能回顾

由于局域性的原因，再加上存储器越小其速度越快，所以存储器层次结构可以显著提高性能。评价缓存性能的一种方法是扩展第 1 章给出的处理器执行时间公式。我们现在考虑处理器在等待存储器访问而停顿时的周期数目，称为**存储器停顿周期**。性能结果为处理器周期与存储器停顿周期之和与时钟周期时间的乘积： B-3

$$\text{CPU 执行时间} = (\text{CPU 时钟周期} + \text{存储器停顿周期}) \times \text{时钟周期时间}$$

此公式假定 CPU 时钟周期包括处理缓存命中的时间，并假定处理器在发生缓存缺失时停顿。B.2 节重新检视了这一简化假定。

存储器停顿周期数取决于缺失数目和每次缺失的成本，后者称为**缺失代价**：

$$\begin{aligned}\text{存储器停顿周期} &= \text{缺失数} \times \text{缺失代价}\\ &= \text{IC} \times \frac{\text{缺失}}{\text{指令}} \times \text{缺失代价}\\ &= \text{IC} \times \frac{\text{存储器访问}}{\text{指令}} \times \text{缺失率} \times \text{缺失代价}\end{aligned}$$

最后一种形式的优点在于其各个分量容易测量。我们已经知道如何测量指令数（IC）。（对于推测处理器，只计算提交的指令数。）可以采用同一方式来测量每条指令的存储器引用数；每条指令需要一次指令访问，很容易判断它是否还需要数据访问。

注意，我们计算出缺失代价，作为平均值，但下面将其作为常数使用。在发生缺失时，缓存后面的存储器可能因为先前的存储器请求或存储器刷新而处于繁忙状态。在处理器、总线和存储器的不同时钟之间进行交互时，时钟周期数也可能发生变化。所以请记住，为缺失代价使用单一数值是一种简化。

缺失率分量就是缓存访问中导致缺失的访问比例（即，导致缺失的访问数除以总访问数）。

缺失率可以用缓存模拟器测量，它会取得指令与数据引用的地址跟踪，模拟该缓存行为，以判断哪些引用命中，哪些缺失，然后汇报命中与缺失总数。今天的许多微处理器提供了用于计算缺失与存储器引用数的硬件，这一缺失率测量方式要容易得多、快得多。

由于读取和写入操作的缺失率和缺失代码通常是不同的，所以上面的公式得出的是一个近似值。存储器停顿时钟周期可以用每条指令的存储器访问次数、读写操作的缺失代价（以时钟周期为单位）、读写操作的缺失率来定义：

$$\text{存储器停顿时钟周期} = \text{IC} \times \text{每条指令的读取操作} \times \text{读取缺失率} \times \text{读取缺失代价} + \text{IC} \times \text{每条指令的写入操作} \times \text{写入缺失率} \times \text{写入缺失代价}$$

B-4 我们通常会合并读写操作，求出读取与写入操作的平均缺失度与缺失代价，以简化上面的完整公式：

$$\text{存储器停顿时钟周期} = \text{IC} \times \frac{\text{存储器访问}}{\text{指令}} \times \text{缺失率} \times \text{缺失代价}$$

缺失率是缓存设计中最重要的度量之一，但在后面各节将会看到，它不是唯一的度量标准。

例题 假定有一个计算机，当所有存储器访问都在缓存中命中时，其每条指令的周期数（CPI）为 1.0。仅有的数据访问就是载入和存储，占总指令数的 50%。如果缺失代价为 25 个时钟周期，缺失率为 2%，当所有指令都在缓存中命中时，计算机可以加快多少？

解答 首先计算计算机总是命中时的性能：

$$\begin{aligned}\text{CPU 执行时间} &= (\text{CPU 时钟周期}+\text{存储器停顿周期}) \times \text{时钟周期}\\ &= (\text{IC} \times \text{CPI}+0) \times \text{时间周期}\\ &= \text{IC} \times 1.0 \times \text{时间周期}\end{aligned}$$

现在，对于采用实际缓存的计算机，首先计算存储器停顿周期：

$$\begin{aligned}\text{存储器停顿周期} &= \text{IC} \times \frac{\text{存储器访问数}}{\text{指令}} \times \text{缺失率} \times \text{缺失代价}\\ &= \text{IC} \times (1+0.5) \times 0.02 \times 25\\ &= \text{IC} \times 0.75\end{aligned}$$

式中，中间项（1+0.5）表示每条指令有 1 次指令访问和 0.5 次数据访问。总性能为：

$$\begin{aligned}\text{CPU 执行时间}_{\text{缓存}} &= (\text{IC} \times 1.0+\text{IC} \times 0.75) \times \text{时间周期}\\ &= 1.75 \times \text{IC} \times \text{时钟周期}\end{aligned}$$

性能比是执行时间的倒数：

$$\begin{aligned}\frac{\text{CPU执行时间}_{\text{缓存}}}{\text{CPU执行时间}} &= \frac{1.75 \times \text{IC} \times \text{时钟周期}}{1.0 \times \text{IC} \times \text{时钟周期}}\\ &= 1.75\end{aligned}$$

没有缓存缺失时，计算机的速度为有缺失时的 1.75 倍。

一些设计师在测量缺失率时更愿意表示为**每条指令的缺失数**，而不是每次存储器引用的缺失数。这两者的关系为：

$$\frac{\text{缺失数}}{\text{指令数}}=\frac{\text{缺失数}\times\text{存储器访问数}}{\text{指令数}}=\text{缺失数}\times\frac{\text{存储器访问数}}{\text{指令数}}$$

如果知道每条指令的平均存储器访问数，后面一个公式是有用的，因为这样可以将缺失率转换为每条指令的缺失数，反之亦然。例如，我们可以将上面示例中每次存储器引用的缺失率转换为每条指令的缺失： B-5

$$\frac{\text{缺失数}}{\text{指令数}}=\text{缺失数}\times\frac{\text{存储器访问数}}{\text{指令数}}=0.02\times1.5=0.030$$

顺便说一下，每条指令的缺失数经常以每千条指令的缺失数给出，以显示整数而非小数。因此，上面的答案也可以表示为每 1000 条指令发生 30 次缺失。

表示为“每条指令的缺失数”的好处在于它与硬件实现无关。例如，推测处理器提取的指令数大约是实际提交指令数的两倍，如果测量每次存储器引用而非每条指令的缺失数，那么就可以人为地降低缺失率。其缺点在于每条指令的缺失数与体系结构相关；例如，对于 80x86 与 MIPS，每条指令的存储器访问平均数可能会有很大不同。因此，对于仅使用单一计算机系列的架构师，最常使用的是每条指令的缺失数，不过与 RISC 体系结构的相似性也可以让人们深入理解其他体系结构。

例题 为了展示这两个缺失率公式的等价性，让我们重做上面的例题，这一次假定每千条指令的缺失率为 30。根据指令个数，存储器停顿时间为多少？

解答 重新计算存储器停顿周期：

$$\begin{aligned}\text{存储器停顿周期}&=\text{缺失数}\times\text{缺失代价}\\&=\text{IC}\times\frac{\text{缺失数}}{\text{指令数}}\times\text{缺失代价}\\&=\text{IC}/1000\times\frac{\text{缺失数}}{\text{指令数}\times1000}\times\text{缺失代价}\\&=\text{IC}/1000\times30\times25\\&=\text{IC}/1000\times750\\&=\text{IC}\times0.75\end{aligned}$$

得到的答案与前面的例题相同，表明这两个公式的等价性。

B.1.2 4 个存储器层次结构问题

我们通过回答有关存储器层次结构第一级的 4 个常见问题来继续对缓存的介绍。

问题 1：一个块可以放在上一级的什么位置？（**块的放置**）

问题 2：如果一个块在上一级中，如何找到它？（**块的识别**）

问题 3：在缺失时应当替换哪个块？（**块的替换**）

问题 4：在写入时会发生什么？（**写入策略**） B-6

这些问题的答案可以帮助我们理解存储器在层次结构不同级别所做的不同折中；因此，我们对每个示例都会问这四个问题。

问题 1：一个块可以放在缓存中的什么位置？

图 B-1 显示，根据对块放置位置的限制，可以将缓存组织方式分为以下三类。

- 如果每个块只能出现在缓存中的一个位置，就说这一缓存是**直接映射**的。这种映射通常是：

  （块地址）MOD（缓存中的块数）

- 如果一个块可以放在缓存中的任意位置，就说该缓存是**全相联**的。
- 如果一个块可以放在缓存中由有限个位置组成的组（set）内，则该缓存是**组相联**的。组就是缓存中的一组块。块首先映射到组，然后这个块可以放在这个组中的任意位置。通常以**位选择方式**来选定组；即：

  （块地址）MOD（缓存中的组数）

  如果组中有 n 个块，则称该缓存放置为 n **路组相联**。

从直接映射到全相联的缓存范围实际上就是组相联的一个统一体。直接映射就是一路组相联，拥有 m 个块的全相联缓存可以称为“m 路组相联”。同样，直接映射可以看作是拥有 m 个组，全相联可以看作拥有一个组。

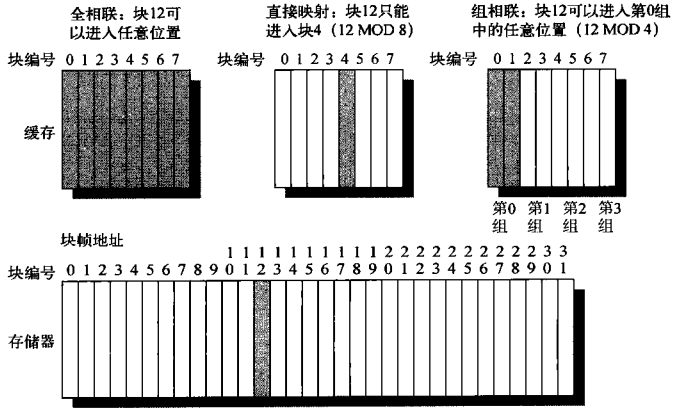


图 B-1　这一缓存示例有 8 个块帧，存储器有 32 个块。三种缓存选项由左向右给出。在全相联中，来自较低层级的块 12 可以进入该缓存 8 个块帧的任意一个。采用直接映射时，块 12 只能放在块帧 4（12 Mod 8）中。组相联拥有这两者的一些共同特性，允许这个块放在第 0 组的任意位置（12 Mod 4）。由于每个组中有两个块，所以这意味着块 12 可以放在缓存的块 0 或块 1 中。实际缓存包含数千个块帧，实际存储器包含数百万个块。拥有四个组、每组两个块的组相联组织形式称为**两路组相联**。假定缓存中没有内容，而且所关心的块地址确认了较低级别的块 12

今天的绝大多数处理器缓存为直接映射、两路组相联或四路组相联，其原因将在稍后介绍。

问题 2：如果一个块就在缓存中，如何找到它？

缓存中拥有每个块帧的地址标志，给出块地址。每个缓存块的标志中可能包含所需要的信息，会对这些标志进行查看，以了解它是否与来自处理器的块地址匹配。由于速度非常重要，所以会对所有可能标志进行并行扫描，这是一条规则。

必须存在一种方法来获知缓存块中不包含有效信息。最常见的过程是向标志中添加一个**有效位**，表明这一项是否包含有效地址。如果没有对这个位进行置位，那就不可能存在对这一地址的匹配。

在继续讨论下一问题之前，先来研究一个处理器地址与缓存的关系。图 B-2 显示了址是如何划分的。第一次划分是在**块地址**和**块偏移**之间，然后再将块帧地址进一步分为**标志字段**和**索引字段**。块偏移字段从块中选择期望数据，索引字段选择组，通过对比标志字段来判断是否命中。尽管可以对标志之外的更多地址位进行对比，但并不需要如此，原因如下所述。 B-7

- 在对比中不应使用偏移量，因为整个块或者存在或者不存在，因此，根据定义，所有块偏移都会导致匹配。
- 核对索引是多余的，因为它是用来选择待核对组的。例如，存储在第 0 组的地址，其索引字段必须为 0，否则就不能存储在第 0 组中；第 1 组的索引值必须为 1，以此类推。这一优化通过缩小缓存标志的宽度来节省硬件和功率。 B-8

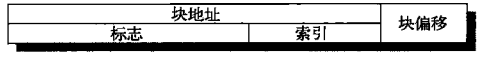


图 B-2 组相联或直接映射缓存中地址的三个组成部分。标志用于检查组中的所有块，索引用于选择该组。块偏移是块中所需数据的地址。全相联缓存没有索引字段

如果总缓存大小保持不变，增大相联度将提高每个组中的块数，从而降低索引的大小、增大标志的大小。即，图 B-2 中的标志索引边界因为相联度增大而向右移动，到端点处就是全相关缓存，没有索引字段。

问题 3：在缓存缺失时应当替换哪个块？

当发生缺失时，缓存控制器必须选择一个用期望数据替换的块。直接映射布置方式的好处就是简化了硬件判决——事实上，简单到没有选择了：只会查看一个块帧，以确定是否命中，而且只有这个块可被替换。对于全相联或组相联布置方式，在发生缺失时会有许多块可供选择。主要有以下三种策略用来选择替换哪个块。

- **随机——为进行均匀分配**，候选块是随机选择的。一些系统生成伪随机块编号，以实现可重复的行为，这在调试硬件时有一定的用处。
- **最近最少使用（LRU）**——为尽量避免抛弃不久就会用到的信息，会记录下对数据块的访问。依靠过去行为来预测未来，将替换掉未使用时间最久的块。LRU 依赖于局域性的一条推论：如果最近用过的块很可能被再次用到，那么放弃最近最少使用的块是一种不错的选择。
- **先入先出（FIFO）**——因为 LRU 的计算可能非常复杂，所以这一策略是通过确定最早的块来近似 LRU，而不是直接确定 LRU。

随机替换的一个好处是易于用硬件实现。随着要跟踪块数的增加，LRU 的成本也变得越得来高，通常只能采用近似法。一种常见的近似方法（经常称为“伪 LRU”）是为缓存中的每个组设定一组比特，每个比特应于缓存中的一路，一路就是组相联缓存中的条（bank）；四路组相联缓存中有四路。在访问一组时开启一个特定比特，这一比特与包含所需块的路相对应；如果与一个组相关联的所有比特都被开启，除最近刚被开启的比特之外，将所有其他比特关闭。在必须替换一个块时，处理器从相应被关闭的路中选择一个块，如果有多种选择，则随机选定。这种方法会给出近似 LRU，这是因为自上次访问组中的所有块之后，被替换块再没有被访问过。表 B-2 给出了 LUR、随机和 FIFO 替换方式中的缺失率之差。 B-9

表B-2　每千条指令的数据缓存缺失，比较了几种不同大小和相联度的最近最少使用、随机和先入先出替换方式

	相联度								
	两路			四路			八路		
大小	LRU	随机	FIFO	LRU	随机	FIFO	LRU	随机	FIFO
16 KB	114.1	1117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

* 对于最大的缓存，LRU和随机方式之间没有什么差别，当缓存较小时，LUR胜过其他几种方式。当缓存较小时，FIFO通常优于随机方式。这些数据是使用10个SPEC2000基准测试采用64字节的块大小，针对Alpha体系结构测得的。其中5个基准测试来自SPECint2000（gap、gcc、gzip、mcf和perl），5个来自SPECfp2000（applu、art、equake、lucas和swim）。在本附录的大多数图形中都将使用这一计算机和这些基准测试程序。

问题4：在写入时发生什么？

大多数处理器缓存访问都是读取操作。所有指令访问都是读取，大多数指令不会向存储器写入数据。附录A中的表A-27和表A-28表明，MIPS程序中存储指令占10%，载入指令占26%，总存储器通信流量中，写入占10%/（100%+26%+10%），大约为7%。在数据缓存通信流量中，写入操作占10%/（26%+10%），大约为28%。要加快常见情景的执行速度，就意味着要针对读取操作对缓存进行优化，尤其是处理器通常会等待读取的完成，而不会等待写入操作。但Amdahl定律（见1.9节）提醒我们，高性能设计不能忽视写入操作的速度。

幸运的是，常见情景也是容易提升速度的情景。可以在读取和比对标志的同时从缓存中读取块，所以只要有了块地址就开始读取块。如果读取命中，则立即将块中所需部分传送给处理器。如果读取缺失，那就没有什么好处——但除了桌面和服务器计算机增加一点功耗之外，也没有坏处；只需忽略所读值即可。

不能对写入操作应用这一优化。要想修改一个块，必须先核对标志，以查看该地址是否命中。由于标志核对不能并行执行，所以写入操作需要的时间通常要长于读取。另一种复杂性在于处理器还指定写入的大小，通常介于1~8个字节之间；只能改变一个块的相应部分。而读取则与之不同，可以毫无顾虑地访问超出所需的更多字节。

写入策略通常可以用来区分缓存设计。在写入缓存时，有下面两种基本选项。

- **直写**——信息被写入缓存中的块和低一级存储器中的块。
- **写回**——信息仅被写到缓存中的块。修改后的缓存块仅在被替换时才被写到主存储器。

为减少在替换时写回块的频率，通常会使用一种称为重写（脏）位的功能。这一状态位表示一个块是脏的（在缓存中经历了修改）还是干净的（未被修改）。如果它是干净的，则在缺失时不会写回该块，因为在低级存储器中可以看到缓存中的相同信息。

写回和直写策略都有自己的优势。采用写回策略时，写入操作的速度与缓存存储器的速度相同，一个块中的多个写入操作只需要对低一级存储器进行一次写入。由于一些写入内容不会进入存储器，所以写回方式使用的存储器带宽较少，使写回策略对多处理器更具吸引力。由于写回策略对存储器层次结构其余部分及存储器互连的使用少于直写，所以它还可以节省功耗，对于嵌入式应用极具吸引力。

相对于写回策略，直写策略更容易实现。缓存总是清洁的，所以它与写回策略不同，读取缺失永远不会导致对低一级存储器的写入操作。直写策略还有一个好处：下一级存储器中拥有

数据的最新副本，从而简化了数据一致性。数据一致性对于多处理器和 I/O 来说非常重要，在第 4 章和附录 D 中对此进行了研究。多级缓存使直写策略更适于高一级缓存，这是因为写入操作只需要传播到下一个较低级别，而不需要传播到所有主存储器。

稍后将会看到，I/O 和多处理器有些反复无常：它们希望为处理器缓存使用写回策略，以减少存储器通信流量，又希望使用直写策略，以与低级存储器层次结构保持缓存一致。

如果处理器在直写期间必须等待写入操作的完成，则说该处理器处于写入停顿状态。减少写入停顿的常见优化方法是写入缓冲区，利用这一优化，数据被写入缓冲区之后，处理器就可以立即继续执行，从而将处理器执行与存储器更新重叠起来。稍后将会看到，即使有了写入缓冲区也会发生写入停顿。

由于在写入时并不需要数据，所以在发生写入缺失时共有以下两种选项。

- **写入分派**——在发生写入缺失时将该块读到缓存中，随后对其执行写入命中操作。在这一很自然的选项中，写入缺失与读取缺失类似。
- **无写入分派**——这显然是一种不太寻常的选项，写入缺失不会影响到缓存。而是仅修改低一级存储器中的块。

B-11

因此，在采用无写入分派策略时，在程序尝试读取块之前，这些块一直都在缓存之外，但在采用写入分派策略时，即使那些仅被写入的块也会保存在缓存中。让我们看一个例子。

例题 假定一个拥有许多缓存项的全相联写回缓存，在开始时为空。下面是由 5 个存储器操作组成的序列（地址放在中括号内）：

```
Write Mem[100];
Write Mem[100];
Read  Mem[200];
Write Mem[200];
Write Mem[100].
```

在使用无写入分派和写入分派时，命中数和缺失数为多少？

解答 对于无写入分派策略，地址 100 不在缓存中，在写入时不进行分派，所以前两个写入操作将导致缺失。地址 200 也不在缓存中，所以该读取操作也会导致缺失。接下来对地址 200 进行的写入将会命中。最后一个对地址 100 的写入操作仍然是缺失。所以对无写入分派策略，其结果是 4 次缺失和 1 次命中。

对于写入分派策略，前面对地址 100 和地址 200 的访问导致缺失，由于地址 100 和地址 200 都可以在缓存中找到，所以其他写入操作将会命中。因此，采用写入分派时，其结果为 2 次缺失和 3 次命中。

任何一种写入缺失策略都可以与直写或写回策略一起使用。通常，写回缓存采用写入分派策略，希望对该块的后续写入能够被缓存捕获。直写缓存通常使用无写入分派策略。其原因在于：即使存在对该块的后续写入操作，这些写入操作仍然必须进入低一级存储器，那还有什么好处呢？

B.1.3 举例：Opteron数据缓存

为了展示这些思想的本质，图 B-3 给出了 AMD Opteron 微处理器中数据缓存的组织方式。该缓存包含 65 536（64KB）字节的数据，块大小为 64 字节，采用两路组相联布置方式、最近使用最少替代策略、写缺失时采用写入分派。

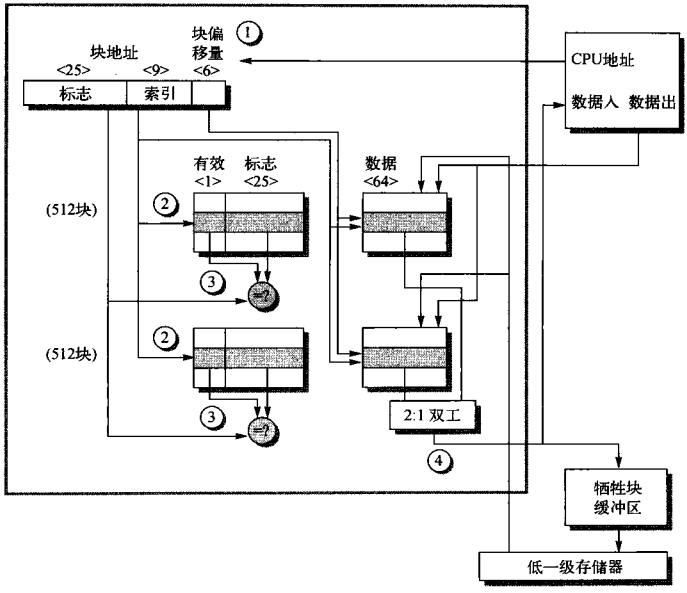


图 B-3　Opteron 微处理器中数据缓存的组织方式。这个 64 KB 的缓存为两路组相联，块大小为 64 字节。长为 9 位的索引从 512 个组中进行选择。一次读取命中的四个步骤（按发生顺序表示为带圆圈的数字）标记了这一组织方式。块偏移量的三位加上索引，提供了 RAM 地址，恰好选择 8 个字节。因此，该缓存保存了由 4096 个 64 位字组成的群组，每个群组包含 512 个组的一半。从低级存储器至缓存的线路用于在发生缺失时载入缓存，不过未在这一示例中展示。离开处理器的地址大小为 40 位，这是因为它是物理地址而不是虚拟地址。图 B-14 解释了 Opteron 如何从虚拟地址映射到物理地址，以进行缓存访问

我们通过图 B-3 中标注的命中步骤来跟踪一次缓存命中的过程。（这 4 个步骤用带圆圈的数字表示。）如 B.5 节所述，Opteron 向缓存提供 48 位虚拟地址进行标志比对，它将同时被翻译为 40 位物理地址。

Opteron 之所以没有利用虚拟地址的所有 64 位，是因为它的设计者认为还没有人会需要那么大的虚拟地址空间，而较小的空间可以简化 Opteron 虚拟地址的映射。设计者计划在未来的微处理器中增大此虚拟地址。

B-12

进入缓存的物理地址被分为两个字段：34 位块地址和 6 位块偏移量（$64=2^6$，$34+6=40$）。块地址进一步分为地址标志和缓存索引。第 1 步显示了这一划分。

缓存索引选择要测试的标志，以查看所需块是否在此缓存中。索引大小取决于缓存大小、块大小和级相联度。Opteron 缓存的组相联度被设置为 2，索引计算如下：

B-13

$$2^{索引}=\frac{缓存大小}{块大小\times组相联度}=\frac{65\,536}{64\times 2}=512=2^9$$

因此，索引宽 9 位，标志宽 34–9=25 位。尽管这是选择正确块所需要的索引，但 64 个字节远多于处理器希望一次使用的数目。因此，将缓存存储器的数据部分安排为宽 8 个字节更有意义一些，这是 64 位 Opteron 处理器的自然数据字。因此，除了用于索引正确缓存块的 9 个位之外，还使用来自块偏移量的另外 3 个位来索引恰好 8 个字节。索引选择是图 B-3 中的第 2 步。

在从缓存中读取这两个标志之后，将它们与处理器所提供块地址的标志部分进行对比。这一对比是图中第 3 步。为了确保标志中包含有效信息，必须设置有效位，否则，对比结果将被忽略。

假定有一个标志匹配，最后一步是通知处理器，使用 2 选 1 多工器的获胜输入从缓存中载入正确数据。Opteron 可以在 2 个时钟周期内完成这四个步骤，因此，如果后面 2 个时钟周期中的指令需要使用载入结果，那就得等待。

在 Opteron 中，写入操作的处理要比读取操作更复杂，这一点与在任何缓存中都是一致的。如果要写入的字在缓存中，前三步相同。由于 Opteron 是乱序执行的，所以只有在它发出指令已提交而且缓存标志比对结果显示命中的信号之后，才会将数据写到缓存中。

到目前为止，我们假定的是缓存命中的常见情景。那在缺失时会发生什么情况呢？在读取缺失时，缓存会向处理器发出信号，告诉它数据还不可用，并从下一级层级结构中读取 64 个字节。对于该块的前 8 个字节，延迟为 7 个时钟周期，对于块的其余部分，为每 8 个字节需要 2 个时钟周期。由于数据缓存是组相联的，所以需要选择替换哪个块。Opteron 使用 LRU（选择在最早之前被引用的块），所以每次访问都必须更新 LRU 位。替换一个块意味着更新数据、地址标志、有效位和 LRU 位。

由于 Opteron 使用写回策略，旧的数据块可能已经被修改，所以不能简单地放弃它。Opteron 为每个块保存 1 个脏位，以记录该块是否被写入。如果“牺牲块”被修改，它的数据和地址就被发送给牺牲块缓冲区。（这种结构类似于其他计算机中的**写入缓冲区**。）Opteron 有 8 个牺牲块的空间。它会将牺牲块写到低一级层次结构，这一操作与其他缓存操作并行执行。如果牺牲块缓冲区已满，缓存就必须等待。

由于 Opteron 在读取缺失和写入缺失时都会分派一个块，所以写入缺失与读取缺失非常类似。 B-14

我们已经看到数据缓存是如何工作的，但它不可能提供处理器所需要的所有存储器：处理器还需要指令。尽管可以尝试用一个缓存来提供数据、指令两种缓存，但这样可能会成为瓶颈。例如，在执行载入或存储指令时，流水化处理器将会同时请求数据字和指令字。因此，单个缓存会表现为载入与存储的结构性冒险，从而导致停顿。解决这一问题的一种简单方法是分开它：一个缓存专门用于指令，另一个缓存专门用于数据。最近的处理器中都使用了独立缓存，包括 Opteron 在内。因此，它有一个 64 KB 的指令缓存和 64 KB 的数据缓存。

处理器知道它是在发射一个指令地址，还是一个数据地址，所以可能存在用于这两者的独立端口，从而使存储器层次结构和处理器之间的带宽加倍。采用分离缓存还提供了分别优化每个缓存的机会：采用不同的容量、块大小和相联度可能会得到更佳性能。（与 Opteron 中的指令缓存和数据缓存相对，**统一**或**混合**等词适用于可能包含指令或数据的缓存。）

表 B-3 显示指令缓存的缺失率低于数据缓存。指令与数据缓存分离，消除了因为指令块和数据块冲突所导致的缺失，但这种分离固定了每种类型所能使用的缓存空间。与缺失率相比，哪个更重要呢？要公平地对比指令数据分离缓存和统一缓存，需要总缓存大小相同。例如，分离的 16 KB 指令缓存和 16 KB 数据缓存应当与 32 KB 统一缓存相对比。要计算分离指令与数据缓存的平均缺失率，需要知道对每种缓存的存储器引用百分比。从附录 A 中的数据可以找到：指令引用为 100%（100%+26%+10%），大约为 74%；数据引用为（26%+10%）（100%+26%+10%），

大约为26%。稍后将会看到，分割对性能的影响并非仅限于缺失率的变化。

表B-3　对于不同大小的指令、数据与统一缓存，每千条指令的缺失数

大小（KB）	指令缓存	数据缓存	统一缓存
8	8.16	44.0	63.0
16	3.82	40.9	51.0
32	1.36	38.4	43.3
64	0.61	36.9	39.4
128	0.30	35.3	36.2
256	0.02	32.6	32.9

B-15 * 指令引用所占百分比大约为74%。此数据的收集采用与表B-2中相同的计算机和基准测试，两路相联缓存，块大小为64字节。

B.2　缓存性能

由于指令的数目与硬件无关，所以用这个数值来评价处理器性能是很有诱惑力的。这种间接性能度量已经让一位又一位计算机设计师栽了跟头。由于缺失率也与硬件的速度无关，所以评价存储器层次结构性能的相应焦点就主要集中在缺失率上。后面将会看到，缺失率可能与指令数目一样产生误导。存储器层次结构性能的一个更好度量标准是**存储器平均访问时间**：

存储器平均访问时间 = 命中时间+缺失时间 × 缺失代价

式中，**命中时间**是指在缓存中命中的时间；其他两项已经在前面看到过。平均访问时间的各个分量可以用绝对时间衡量，比如，一次命中的时间为0.25~1.0 ns，也可以用处理器等待该存储器的时间周期数来衡量，比如一次缺失代价为150~200个时钟周期。注意，存储器平均访问时间仍然是性能的间接度量；尽管它优于缺失率，但并不能替代执行时间。

这个公式可以帮助我们决定是选择分离缓存还是统一缓存。

例题　16 KB指令缓存加上16 KB数据缓存相对于一个32 KB统一缓存，哪一种的缺失率较低？利用表B-3中的缺失率数据来帮助计算正确答案，假定36%的指令为数据传输指令。假定一次命中需要1个时钟周期，缺失代价为100个时钟周期。对于统一缓存，如果仅有一个缓存端口来满足两个同时请求，一次载入或存储命中另需要一个时钟周期。利用第3章的流水线技术，统一缓存会导致结构性冒险。每种情况下的存储器平均访问时间为多少？假定采用具有写入缓冲区的直写缓存，忽略由于写入缓冲区导致的停顿。

解答　首先让我们将每千条指令的缺失数转换为缺失率。求解上面的一般公式，缺失率为：

$$\text{缺失率} = \frac{\dfrac{\text{缺失率}}{1000\text{条指令}}/1000}{\dfrac{\text{存储器访问数}}{\text{指令数}}}$$

由于每次指令访问都正好有一次存储器访问来提取指令，所以指令缺失率为：

$$\text{缺失率}_{16\,\text{KB指令}} = \frac{3.82/1000}{1.00} = 0.004$$

由于36%的指令为数据传输，所以数据缺失率为：

$$缺失率_{16\,KB数据}=\frac{40.9/1000}{0.36}=0.114$$

B-16

统一缺失率需要考虑指令和数据访问：

$$缺失率_{32\,KB统一}=\frac{43.3/1000}{1.00+0.36}=0.0318$$

如上所述，大约74%的存储器访问为指令引用。因此，分离缓存的总缺失率为：

$$(74\%\times0.004)+(26\%\times0.114)=0.0326$$

因此，32 KB统一缓存的实际缺失率略低于两个16 KB缓存。

存储器平均访问时间公式可分为指令访问和数据访问：

存储器平均访问时间
= 指令百分比 × (命中时间+指令缺失率 × 缺失代价)
+数据百分比 × (命中时间+数据缺失率 × 缺失代价)

因此，每种组织方式的时间为：

存储器平均访问时间$_{分离}$
$= 74\%\times(1+0.004\times200)+26\%\times(1+0.114\times200)$
$= (74\%\times1.80)+(26\%\times23.80)=1.332+6.188=7.52$

存储器平均访问时间$_{统一}$
$= 74\%\times(1+0.0318\times200)+26\%\times(1+1+0.0318\times200)$
$= (74\%\times7.36)+(26\%\times8.36)=5.446+2.174=7.62$

因此，在这个示例中，尽管分离缓存（每时钟周期提供两个存储器端口，从而避免了结构性冒险）的实际缺失率较差，但其存储器平均访问时间要优于单端口统一缓存。

B.2.1 存储器平均访问时间与处理器性能

一个显而易见的问题是：因缓存缺失导致的存储器平均访问时间能否预测处理器性能。

首先，还有其他原因会导致停顿，比如使用存储器的I/O设备产生争用。由于存储器层次结构导致的停顿远多于其他原因导致的停顿，所以设计人员经常假定所有存储器停顿都是由于缓存缺失导致的。我们这里也采用这一简化假定，但在计算最终性能时，一定要考虑所有存储器停顿。

第二，上述问题的回答也受处理器的影响。采用循序执行处理器（见第3章），那回答基本上就是肯定的。处理器会在缺失期间停顿，存储器停顿时间与存储器平均访问时间存在很强的相关性。现在假定采用循序执行，但下一小节会返回来讨论乱序处理器。 B-17

如上一节所述，可以为CPU时间建立如下模型：

CPU时间 = (CPU执行时钟周期+存储器停顿时钟周期) × 时钟周期时间

这个公式会产生一个问题：一次缓存命中的时钟周期应看作CPU执行时钟周期的一部分，还是存储器停顿时钟周期的一部分？尽管每一种约定都有自己的正当理由，但最为人们广泛接受的是将命中时钟周期包含在CPU执行时钟周期中。

我们现在可以研究缓存对性能的影响了。

例题 让我们对第一个示例使用循序执行计算机。假定缓存缺失代价为 200 个时钟周期，所有指令通常都占用 1.0 个时钟周期（忽略存储器停顿）。假定平均缺失率为 2%，每条指令平均有 1.5 次存储器引用，每千条指令的平均缓存缺失数为 30。如果考虑缓存的行为特性，对性能的影响如何？使用每条指令的缺失数及缺失率来计算此影响。

解答

$$\text{CPU时间} = \text{IC} \times \left(\text{CPI}_{\text{执行}} + \frac{\text{存储器停顿时钟周期}}{\text{指令数}}\right) \times \text{时钟周期时间}$$

其性能（包括缓存缺失）为：

$$\begin{aligned}\text{CPU时间}_{\text{包括缓存}} &= \text{IC} \times [1.0+(30/1000 \times 200)] \times \text{周期时钟时间} \\ &= \text{IC} \times 7.00 \times \text{时钟周期时间}\end{aligned}$$

现在使用缺失率计算性能：

$$\text{CPU时间} = \text{IC} \times \left(\text{CPU}_{\text{执行}} + \text{缺失率} \times \frac{\text{存储器访问}}{\text{指令}} \times \text{缺失代价}\right) \times \text{时钟周期时间}$$

$$\begin{aligned}\text{CPU时间}_{\text{包括缓存}} &= \text{IC} \times [1.0 \times (1.5 \times 2\% \times 200)] \times \text{时钟周期时间} \\ &= \text{IC} \times 7.00 \times \text{时钟周期时间}\end{aligned}$$

在有、无缓存情况下，时钟周期时间和指令数均相同。因此，CPU 时间提高至 7 倍，CPI 从“完美缓存”的 1.00 增加到可能产生缺失的缓存的 7.00。在根本没有任何存储器层次结构时，CPU 将再次升高到 1.0+200 × 1.5=301，比带有缓存的系统长出 40 多倍。

如上例所示，缓存特性可能会对性能产生巨大影响。此外，对于低 CPI、高时钟频率的处理器，缓存缺失会产生双重影响。

(1) $\text{CPI}_{\text{执行}}$越低，固定数目的缓存缺失时钟周期产生的相对影响越高。

(2) 在计算 CPI 时，一次缺失的缓存缺失代价是以处理器时钟周期进行计算的。因此，即使两个计算机的存储器层次结构相同，时钟频率较高的处理器在每次缺失时会占用较多的时钟周期，CPI 的存储器部分也相应较高。

B-18

对于低 CPI、高时钟频率的处理器，缓存的重要性更高，因此，如果在评估此类计算机的性能时忽略缓存行为，其危险性更大。Amdahl 定律再次发挥威力！

尽管将存储器平均访问时间降至最低是一个合理的目标（在本附录中大多使用这一目标），但请记住，最终目标是缩短处理器执行时间。下面的例子说明如何区分这两者。

例题 两种不同缓存组织方式对处理器性能的影响如何？假定完美缓存的 CPI 为 1.6，时钟周期时间为 0.35 ns，每条指令有 1.4 次存储器引用，两个缓存的大小都是 128 KB，两者的块大小都是 64 字节。一个缓存为直接映射，另一个为两路组相联。图 B-3 显示，对于组相联缓存，必须添加一个多工器，以根据标志匹配在组中的块之间作出选择。由于处理器的速度直接与缓存命中的速度联系在一起，所以假定必须将处理器时钟周期时间扩展 1.35 倍，才能与组相联缓存的选择多工器相适应。对于一级近似，每一种缓存组织方式的缓存缺失代价都是 65 纳秒。（在实践中，通常会舍入为整数个时钟周期。）首先，计算存储器平均访问时间，然后再计算处理器性能。

假定命中时间为 1 个时钟周期，128 KB 直接映射缓存的缺失率为 2.1%，同等大小的两路组相联缓存的缺失率为 1.9%。

解答 存储器平均访问时间为：

$$\text{存储器平均访问时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

因此，每种组织方式的时间为：

$$\text{存储器平均访问时间}_{\text{一路}} = 0.35 + (0021 \times 65) = 1.72\text{ns}$$

$$\text{存储器平均访问时间}_{\text{两路}} = 0.35 \times 1.35 + (0019 \times 65) = 1.71\text{ns}$$

这一存储器平均访问时间优于两路组相联缓存。

处理器性能为：

$$\begin{aligned}\text{CPU时间} &= \text{IC} \times \left(\text{CPI}_{\text{执行}} + \frac{\text{缺失数}}{\text{指令数}} \times \text{缺失代价}\right) \times \text{时钟周期时间} \\ &= \text{IC} \times \left[\left(\text{CPI}_{\text{执行}} \times \text{时钟周期时间}\right) + \left(\text{缺失率} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失代价} \times \text{时钟周期时间}\right)\right]\end{aligned}$$

B-19

将（缺失代价 × 时钟周期时间）代以 65ns，可得每种缓存组织方式的性能为：

$$\text{CPU 时间}_{\text{一路}} = \text{IC} \times [1.6 \times 0.35 + (0.021 \times 1.4 \times 65)] = 2.47 \times \text{IC}$$

$$\text{CPU 时间}_{\text{两路}} = \text{IC} \times [1.6 \times 0.35 \times 1.35 + (0.019 \times 1.4 \times 65)] = 2.49 \times \text{IC}$$

相对性能为：

$$\frac{\text{CPU时间}_{\text{两路}}}{\text{CPU时间}_{\text{一路}}} = \frac{2.49 \times \text{指令数}}{2.47 \times \text{指令数}} = \frac{2.49}{2.47} = 1.01$$

与存储器平均访问时间的对比结果相反，直接映射缓存的平均性能略好一些，这是因为尽管两组组相联的缺失数较少，但针对所有指令扩展了时钟周期。由于 CPU 时间是我们的基本评估，而且由于直接映射的构建更简单一些，所以本示例中的优选缓存为直接映射。

B.2.2 缺失代价与乱序执行处理器

对于乱序执行处理器，如何定义“缺失代价”呢？是存储器缺失的全部延迟，还是仅考虑处理器必须停顿时的“暴露”延迟或无重叠延迟？对于那些在完成数据缺失之前必须停顿的处理器，不存在这一问题。

让我们重新定义存储器停顿，得到缺失代价的一种新定义，将其表示为非重叠延迟：

$$\frac{\text{存储器停顿周期}}{\text{指令数}} = \frac{\text{缺失数}}{\text{指令数}} \times (\text{总缺失代价} - \text{重叠缺失延迟})$$

与此类似，由于一些乱序处理器会拉长命中时间，所以性能公式的这一部分可以除以总命中延迟减去重叠命中延迟之差。可以对这一公式进一步扩展，将总缺失延迟分解为没有争用时的延迟和因为争用导致的延迟，以考虑乱序处理器中的存储器资源。我们仅关注缺失延迟。

我们现在必须决定以下各项。

- **存储器延迟长度**——在乱序处理器中如何确定存储器操作的起止时刻。
- **延迟重叠的长度**——如何确定与处理器相重叠的起始时刻（或者说，在什么时刻我们说存储器操作使处理器停顿）。

由于乱序执行处理器的复杂性，所以不存在单一的准确定义。

B-20 由于在流水线退出阶段只能看到已提交的操作，所以我们说：如果处理器在一个时钟周期内没有退出（retire）最大可能数目的指令，它就在该时钟周期内停顿。我们将这一停顿记在第一条未退出指令的账上。这一定义绝不像看上去那么简单。例如，为缩短特定停顿时间而应用某一种优化，并不一定总能缩短执行时间，这是因为此时可能会暴露出另一种类型的停顿（原本隐藏在所关注的停顿背后）。

关于延迟，我们可以从存储器指令在指令窗口中排队的时刻开始测量，也可以从生成地址的时刻开始，还可以从指令被实际发送给存储器系统的时刻开始。只要保持一致，任何一种选项都是可以的。

例题 让我们重做上面的例题，但这一次假定具有较长时钟周期时间的处理器支持乱序技术，但仍采用直接映射缓存。假定 65 ns 的缺失代价中有 30%可以重叠，也就是说，CPU 存储器平均停顿时间现在为 45.5 ns。

解答 乱序计算机的存储器访问时间为：

$$\text{存储器平均访问时间}_{\text{一路、乱序}}=0.35\times1.35+(0.021\times45.5)=1.43\ \text{ns}$$

乱序缓存的性能为：

$$\text{CPU 时间}_{\text{一路、乱序}}=\text{IC}\times[1.6\times0.35\times1.35+(0.021\times1.4\times45.5)]=2.09\times\text{IC}$$

因此，尽管乱序计算机的时钟周期时间要慢得多，直接映射缓存的缺失率也更高一些，但如果它能隐藏 30%的缺失代价，那仍然可以稍快一些。

总而言之，尽管乱序处理器存储器停顿的定义和测量比较复杂，但由于它们会严重影响性能，所以应当了解这些问题。这一复杂性的出现是因为乱序处理器容忍由缓存缺失导致一定的延迟，不会对性能造成伤害。因此，设计师在评估存储器层次结构的权衡时，通常使用乱序处理器与存储器的模拟器，以确保一项帮助缩短平均存储器延迟的改进能够真的有助于提高程序性能。

为了帮助总结本节内容，同时也作为一个方便使用的参考，图 B-4 列出了本附录中的缓存公式。

B-21

$$2^{\text{索引}}=\frac{\text{缓存大小}}{\text{块大小}\times\text{组相联度}}$$

CPU 执行时间=（CPU 时钟周期+存储器停顿周期）×时钟周期时间

存储器停顿周期=缺失数×缺失代价

$$\text{存储器停顿周期}=\text{IC}\times\frac{\text{缺失数}}{\text{指令}}\times\text{缺失代价}$$

$$\frac{\text{缺失数}}{\text{指令}}=\text{缺失率}\times\frac{\text{存储器访问}}{\text{指令}}$$

存储器平均访问时间=命中时间+缺失率×缺失代价

$$\text{CPU执行时间}=\text{IC}\times\left(\text{CPI}_{\text{执行}}+\frac{\text{存储器停顿时钟周期}}{\text{指令数}}\right)\times\text{时钟周期时间}$$

图 B-4 **本附录中的性能公式汇总。第一个公式计算缓存索引大小，其余公式帮助评估性能。**后两个公式处理多级缓存，在下一节的开头部分会对其进行介绍。将它们包含在此处，使本图成为有用的参考资料

$$\text{CPU执行时间} = \text{IC} \times \left(\text{CPI}_{\text{执行}} + \frac{\text{缺失数}}{\text{指令数}} \right) \times \text{缺失代价} \times \text{时钟周期时间}$$

$$\text{CPU执行时间} = \text{IC} \times \left(\text{CPI}_{\text{执行}} + \text{缺失率} \times \frac{\text{缺失访问}}{\text{指令数}} \times \text{缺失代价} \right) \times \text{时钟周期时间}$$

$$\frac{\text{存储器停顿周期}}{\text{指令}} = \frac{\text{缺失数}}{\text{指令数}} \times (\text{总缺失延迟} - \text{重叠缺失延迟})$$

$$\text{存储器平均访问时间} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2})$$

$$\frac{\text{存储器停顿周期}}{\text{指令}} = \frac{\text{缺失数}_{L1}}{\text{指令数}} \times \text{命中时间}_{L2} + \frac{\text{缺失数}_{L2}}{\text{指令数}} \times \text{缺失代价}_{L2}$$

图 B-4（续）

B.3 6种基本的缓存优化

存储器平均访问时间公式为我们提供了一个框架，用于展示提高缓存性能的缓存优化方法：

存储器平均访问时间 = 命中时间+缺失率 × 缺失代价

因此，我们将6种缓存优化分为以下3类。

- 降低缺失率——较大的块、较大的缓存、较高的关联度。
- 降低缺失代价——多级缓存，为读取操作设定高于写入操作的优先级。
- 缩短在缓存中命中的时间——在索引缓存时避免地址转换。

表 B-8 汇总了这6种技术的实现复杂度和性能优势，作为本节的总结。 B-22

改进缓存特性的经典方法是降低缺失率，我们给出3种实现技术。为了更好地理解导致缺失的原因，首先介绍一个模型，将所有缺失分为3个简单类别。

- **强制缺失**（Compulsory）——在第一次访问某个块时，它不可能会在缓存中，所以必须将其读到缓存中。这种缺失也被称为冷启动缺失或首次引用缺失。
- **容量缺失**（Capacity）——如果缓存无法容纳程序执行期间所需要的全部块，由于一些块会被放弃，过后再另行提取，所以会（在强制缺失之外）发生容量缺失。
- **冲突缺失**（Conflict）——如果块布置策略为组相联或直接映射，则会（在强制缺失和容量缺失之外）发生冲突缺失，这是因为如果有太多块被映射到一个组中，这个组中的某个块可能会被放弃，过后再另行提取。这种缺失也被称为碰撞缺失。其要点就是：由于对某些常用组的请求数超过 n，所以本来在全相联缓存中命中的情景会在 n 路组相联缓存中变为缺失。

（第5章增加了第四个 C，即一致性（Coherency）缺失，它是因为要在多处理器中保持多个缓存一致而进行缓存刷新所导致的；这里不讨论此类缺失。）

表 B-4 显示了根据 3C 分类后的缓存缺失相对频率。强制缺失在无限缓存中发生，容量缺失在全相联缓存中发生。冲突缺失在从全相联变为八路相联、四路相联……时发生。图 B-5 以图形方式展示相同数据。上图显示绝对缺失率，下图绘制了当缓存大小变化时，各类缺失占总缺失数的百分比曲线。

表B-4　每种缓存大小的总缺失率，及根据3C划分的每种缺失所占百分比

缓存大小（KB）	相联度	总缺失率	缺失率组成（相对百分比）（总和=总缺失率的百分之百）					
			强制缺失		容量缺失		冲突缺失	
4	一路	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	两路	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	四路	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	八路	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8	一路	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8	两路	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8	四路	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8	八路	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16	一路	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	两路	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	四路	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	八路	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32	一路	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32	两路	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32	四路	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32	八路	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64	一路	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	两路	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	四路	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	八路	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128	一路	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128	两路	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	四路	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	八路	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256	一路	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	两路	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	四路	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	八路	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512	一路	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512	两路	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512	四路	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512	八路	0.006	0.0001	1.1%	0.005	95%	0.000	4%

* 强制缺失与缓存大小无关，而容量缺失随容量的增加而降低，冲突缺失随相联度的增大而降低。图 B-5 以图形方式显示了相同数据。注意，在不超过 128 KB 时，大小为 N 的直接映射缓存的缺失率大约与大小为 $N/2$ 的两路组相联缓存的缺失率相同。大于 128 KB 的缓存不符合这一规则。注意，“容量”列给出的也是全相联缺失率。数据的收集方式与表 B-2 中一样，也是使用 LRU 替换方式收集的。

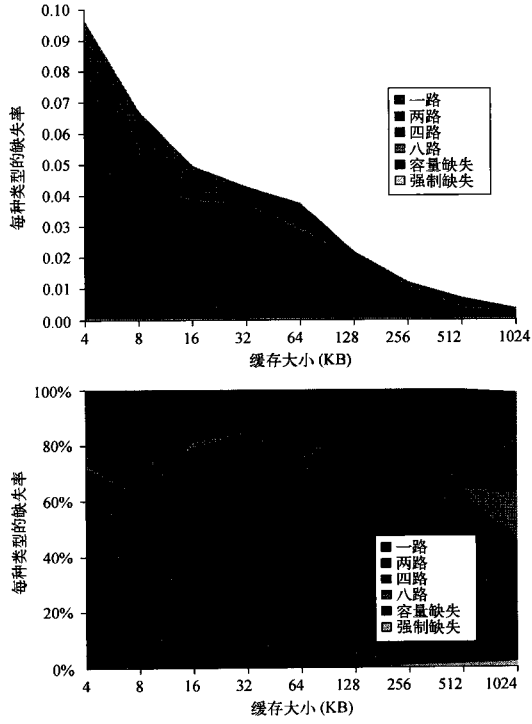


图 B-5 根据表 B-4 中的 3C 数据，每种不同缓存大小的总缺失率（上）和缺失率分布（下）。上图显示实际数据缓存缺失率，下图显示每个类别的百分比（与表 B-4 中所能容纳的缓存大小相比，本图中的空间使这些图形能够多显示一种缓存大小。）

为了展示相联度的好处，将冲突缺失划分为每次相联度下降时所导致的缺失。一共有 4 类冲突缺失，其计算方式如下所示。

- **八路**——从全相联（无冲突）到八路相联时产生的冲突缺失。
- **四路**——从八路相联到四路相联时产生的冲突缺失。
- **两路**——从四路相联到两路相联时产生的冲突缺失。
- **一路**——从两路相联到一路相联（直接映射）时产生的冲突缺失。

从图中可以看出，SPEC2000 程序的强制缺失率非常低，对许多长时间运行的程序都是如此。

在确认这 3C 缺失率之后，计算机设计师可以针对它们做点什么呢？从概念上来讲，冲突缺失是最容易避免的：全相联布置策略就可以避免所有冲突缺失。但是，全相联的硬件实现成本非常高昂，可能会降低处理器时间频率（见 B.3.3 节的示例），从而降低整体性能。 B-23 ~ B-24

除了增大缓存之外，针对容量缺失没有什么办法了。如果上一级存储器远小于程序所需要的容量，那就会有相当一部分时间用于在层次结构的两级之间移动数据，我们说这种存储器层次结构将会**摆动**。由于需要太多的替换操作，所以摆动意味着计算机的运行速度接近于低级存储器的速度，甚至会因为缺失开销变得更慢。

B-25

另外一种降低 3C 缺失的方法是增大块的大小，以降低强制缺失数，但稍后将会看到，大型块可能会增加其他类型的缺失。

3C 分类使我们可以更深入地了解导致缺失的原因，但这个简单的模型也有它的局限性；它让我们深入地了解了平均性能，但不能解释个体缺失。例如，由于较大的缓存可以将引用扩展到更多个块中，所以改变缓存大小会改变冲突缺失和容量缺失。因此，当缓存大小变化时，一个缺失可能会由容量缺失变为冲突缺失。注意，3C 分类还忽略了替换策略，一方面是因为其难以建模，另一方面是因为它总体来说不太重要。但在具体环境中，替换策略可能会实际导致异常行为，比如，在大相联度下得到较低的缺失率，这与 3C 模型的结果矛盾。（有人提议使用地址跟踪来确定存储器中的最优放置策略，以避免 3C 模型中的放置缺失；我们在这里没有采纳这一建议。）

遗憾的是，许多降低缺失率的技术也会增加命中时间或缺失代价。在使用 3 种优化方法降低缺失率时，必须综合考虑提高整体系统速度的目标，使两者达到平衡。第一个例子显示了平衡观点的重要性。

B.3.1 第一种优化方法：增大块大小以降低缺失率

降低缺失率的最简单方法是增大块大小。图 B-6 针对一组程序及缓存大小，给出了块大小与缺失率的折中。较大的块大小也会降低强制缺失。这一降低是因为局域性原理分为两个部分：时间局域性和空间局域性。较大的块充分利用了空间局域性的优势。

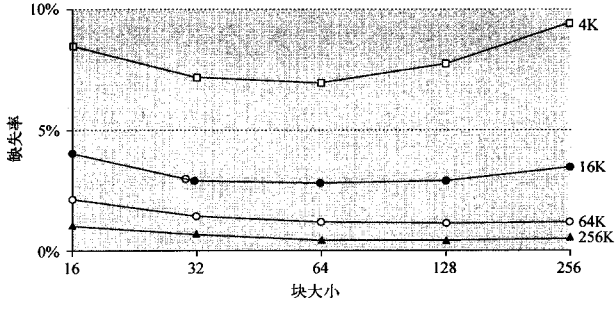


图 B-6 对于 5 种不同大小的缓存，缺失率与块大小的相互关系。注意，如果与缓存大小相比，块大小过大，则缺失率实际上会上升。每条曲线表示一个不同大小的缓存。表 B-5 给出绘制这些曲线的数据。遗憾的是，如果包含块大小因素，SPEC2000 跟踪所需要的时间过长，所以这些数据是在 DECstation 5000 上运行 SPEC92 获得的[Gee 等人，1993]

同时，较大的块也会增加缺失代价。由于它们降低了缓存中的块数，所以较大块可能会增大冲突缺失，如果缓存很小，甚至还会增加容量缺失。显然，没有理由要将块大小增大到会升

高缺失率的程度。如果它会增加存储器平均访问时间，那降低缺失率也没有什么好处。缺失代价的增加会超过缺失率的下降。

表B-5 图B-6中5种不同大小缓存的实际缺失率随块大小的变化

块大小	缓存大小			
	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

* 注意，对于4 KB缓存，块大小为256字节时的缺失率高于32字节。在本例中，缓存大小必须为256 KB，以使块大小为256字节时能够降低缺失。

例题 表B-5显示了图B-6中绘制的实际缺失率。假定存储器系统的开销为80个时钟周期，然后每2个时钟周期提交16个字节。因此，它可以在82个时钟周期内提供16个字节，在84个时钟周期内提供32个字节，以此类推。对于表B-5中的每种缓存大小，哪种缓存大小的存储器平均时间最短？

解答 存储器平均访问时间为：

存储器平均访问时间 = 命中时间+缺失率 × 缺失代价 B-26

如果我们假定命中时间为1个时钟周期，与块大小无关，那么在4 KB缓存中，对16字节块的访问时间为：

存储器平均访问时间 = 1+(8.57% × 82) = 8.027 时钟周期

在256 KB缓存中，对256字节块的存储器平均访问时间为：

存储器平均访问时间 = 1+(0.49% × 112) = 1.549 时钟周期 B-27

表B-6显示了这两个极端值之间所有块与缓存大小的存储器平均访问时间。粗体项目表示对于给定缓存大小能够实现最快访问的块大小：若缓存大小为4 KB，则块大小为32字节时的访问速度最快；若缓存大小大于4 KB，则块大小应为64字节。事实上，这些数值也正是当前处理器缓存的常见块大小。

表B-6 图B-10中5种不同大小缓存的存储器平均访问时间随块大小的变化

块大小	缺失代价	缓存大小			
		4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

* 注意，绝大多数的块大小为32字节和64字节。每种缓存大小的最短平均访问时间用黑体标出。

在所有这些技术中，缓存设计者都在尝试尽可能同时降低缺失率和缺失代价。块大小的选择有赖于低级存储器的延迟和带宽。高延迟和高带宽鼓励采用大块，因为缓存在每次缺失时能够获取的字节可以多出许多，而缺失代价却很少增加。相反，低延迟和低带宽则鼓励采用小块，

因为这种情况下采用较大块不会节省多少时间。例如，一个小块的两倍缺失代价可能接近一个两倍大小块的缺失代价。更多的小块还可能减少冲突缺失。注意，图 B-6 和表 B-6 给出了基于缺失率最低、存储器平均时间最短选择块大小时的差别。

在了解了较大块对强制缺失和容量缺失的正面与负面影响之后，下面两小节将研究较高容量和较高相联度的可能性。

B.3.2　第二种优化方法：增大缓存以降低缺失率

降低表 B-4 和图 B-5 中容量缺失的最明显方法是增加缓存的容量，其明显的缺点是可能延长命中时间、增加成本和功耗。这一技术在片外缓存中尤其常用。

B.3.3　第三种优化方法：提高相联度以降低缺失率

B-28 表 B-4 和图 B-5 显示了缺失率是如何随着相联度的增大而得以改善的。从中可以看出两个一般性的经验规律。第一条规律是：对于这些特定大小的缓存，从实际降低缺失数的功效来说，八路组相联与全相联是一样有效的。通过对比表 B-4 中的八路项目与容量缺失列可以看出这一差别，其中的容量缺失是使用全相联缓存计算得出的。

从图中观察得到的第二条规则称为 **2：1 缓存经验规律**：大小为 N 的直接映射缓存与大小为 $N/2$ 的两路组相联缓存具有大体相同的缺失率。这一规律对 3C 图形中小于 128KB 的缓存也是成立的。

与许多此类示例类似，要改善存储器平均访问时间的一个方面，可能会导致另一方面的恶化。增大块大小可以降低缺失率，但会提高缺失代价；增大相联度可能会延长命中时间。因此，加快处理器时钟速度的压力鼓励使用简单的缓存设计，但提高相联度的回报是提高缺失代价，如下例所示。

例题　假定提高相联度将会延长时钟周期时间，如下所示：

$$\text{时钟周期时间}_{\text{两路}} = 1.36 \times \text{时钟周期时间}_{\text{一路}}$$
$$\text{时钟周期时间}_{\text{四路}} = 1.44 \times \text{时钟周期时间}_{\text{一路}}$$
$$\text{时钟周期时间}_{\text{八路}} = 1.52 \times \text{时钟周期时间}_{\text{一路}}$$

假定命中时间为 1 个时钟周期，直接映射情景的缺失代价为到达第 2 级缓存的 25 个时钟周期（见下一小节），在第 2 级缓存中绝对不会缺失，还假定不需要将缺失代价舍入为整数个时钟周期。利用表 B-4 中的缺失率，对于哪种缓存大小来说，这三种表述是正确的？

$$\text{存储器平均访问时间}_{\text{八路}} < \text{存储器平均访问时间}_{\text{四路}}$$
$$\text{存储器平均访问时间}_{\text{四路}} < \text{存储器平均访问时间}_{\text{两路}}$$
$$\text{存储器平均访问时间}_{\text{两路}} < \text{存储器平均访问时间}_{\text{一路}}$$

解答　每种相联度的存储器平均访问时间为：

$$\text{存储器平均访问时间}_{\text{八路}} = \text{命中时间}_{\text{八路}} + \text{缺失率}_{\text{八路}} \times \text{缺失代价}_{\text{八路}}$$
$$= 1.52 + \text{缺失率}_{\text{八路}} \times 25$$
$$\text{存储器平均访问时间}_{\text{四路}} = 1.44 + \text{缺失率}_{\text{四路}} \times 25$$
$$\text{存储器平均访问时间}_{\text{两路}} = 1.36 + \text{缺失率}_{\text{两路}} \times 25$$

$$存储器平均访问时间_{一路}=1.00+缺失率_{一路}\times 25$$ B-29

每种情况下的缺失代价相同，所以我们使其保持25个时钟周期。例如，对于一个4 KB的直接映射缓存，存储器平均访问时间为：

$$存储器平均访问时间_{一路}=1.00+(0.098\times 25)=3.44$$

对于512 KB八路组相联缓存，该时间为：

$$存储器平均访问时间_{八路}=1.52+(0.006\times 25)=1.66$$

利用这些公式及表B-4中的缺失率，表B-7给出了每种缓存和相联度时的存储器平均访问时间。该表显示，对于不大于8 KB、不超过四路相联度的缓存，本例中的公式成立。从16 KB开始，较大相联度的较长命中时间超过了因为缺失降低所节省的时间。

表B-7 以表B-4中的缺失率作为本例中参数得出的存储器平均访问时间

缓存大小（KB）	相联度			
	一路	两路	四路	八路
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

* 粗体类型意味着这一时间高于左侧的数字，即较高的相联度延长了存储器平均访问时间。

注意，在本例中，我们没有考虑较慢时钟频率对程序其余部分的影响，因此低估了直接映射缓存的收益。

B.3.4 第四种优化方法：采用多级缓存降低缺失代价

降低缓存缺失已经成为缓存研究的传统焦点，但缓存性能公式告诉我们：通过降低缺失代价同样可以获得降低缺失率所带来的好处。此外，图2-2显示的技术趋势表明：处理器的速度增长快于DRAM，从而使缺失代价的相对成本随时间的推移而升高。

处理器与存储器之间的性能差距让架构师开始思考这样一个问题：是应当加快缓存速度以与处理器速度相匹配呢？还是让缓存更大一些，以避免加宽处理器与主存储器之间的鸿沟？

一个回答是：两者都要。在原缓存与存储器之间再添加一级缓存可以简化这一决定。第一级缓存可以小到足以与快速处理器的时钟周期时间相匹配。而第二级缓存则大到足以捕获本来可能进入主存储器的访问，从而降低实际缺失代价。

尽管再添加一级层次结构的思路非常简单，但它增加了性能分析的复杂程度。第二级缓存的定义也并非总是那么简单。首先让我们为一个二级缓存定义存储器平均访问时间。用下标L1和L2分别指代第一级、第二级缓存，原公式为： B-30

$$存储器平均访问时间=命中时间_{L1}+缺失率_{L1}\times 缺失代价_{L1}$$

及

$$缺失代价_{L1}=命中时间_{L2}+缺失率_{L2}\times 缺失代价_{L2}$$

得

$$存储器平均访问时间=命中时间_{L1}+缺失率_{L1}\times(命中时间_{L2}+缺失率_{L2}\times缺失代价_{L2})$$

在这个公式中，第二级缺失率是针对第一级缓存未能找到的内容进行测量的。为了避免模糊，对二级缓存系统采用以下术语。

- **局部缺失率**——此比值即是缓存中的缺失数除以对该缓存进行存储器访问的总数。可以想到，对于第一级缓存，它等于**缺失率**$_{L1}$，对于第二级缓存，它等于**缺失率**$_{L2}$。
- **全局缺失率**——缓存中的缺失数除以处理器生成的存储器访问总数。利用以上术语，第一级缓存的全局缺失率仍然为**缺失率**$_{L1}$，但对于第二级缓存则为**缺失率**$_{L1}\times$**缺失率**$_{L2}$。

第二级缓存的这一局部缺失率很大，这是因为第一级缓存已经提前解决了存储器访问中便于实现的部分。这就是为什么说全局缺失率是一个更有用的度量标准：它指出在处理器发出的存储器访问中，有多大比例指向了存储器。

这是一个让**每条指令缺失数**度量闪光的地方。利用这一度量标准，不用再担心局部缺失率或全局缺失率的混淆问题，只需要扩展每条指令的存储器停顿，以增加第二级缓存的影响。

$$每条指令的平均存储器停顿时间=每条指令的缺失数_{L1}\times命中时间_{L2}+每条指令的缺失数_{L2}\times缺失代价_{L2}$$

例题 假定在1000次存储器引用中，第一级缓存中有40次缺失，第二级缓存中有20次缺失。各缺失率等于多少？假定L2缓存到存储器的缺失代价为200个时钟周期，L2缓存的命中时间为10个时钟周期，L1的命中时间为1个时钟周期，每条指令共有1.5次存储器引用。每条指令的存储器平均访问时间和平均停顿周期为多少？忽略写入操作的影响。

解答 第一级缓存的缺失率（局部缺失率或全局缺失率）为40/1000=4%。第二级缓存的局部缺失率为20/40=50%。第二级缓存的全局缺失率为20/1000=2%。则：

$$\begin{aligned}存储器平均访问时间&=命中时间_{L1}+缺失率_{L1}\times(命中时间_{L2}+缺失率_{L2}\times缺失代价_{L2})\\&=1+4\%\times(10+50\%\times200)\\&=1+4\%\times110=5.4个时钟周期\end{aligned}$$

B-31

为了知道每条指令会有多少次缺失，我们将1000次存储器引用除以每条指令的1.5次存储器引用，得到667条指令。因此，我们需要将缺失数乘以1.5，得到每千条指令的缺失数。于是得到每千条指令的L1缺失数为40×1.5=60次，L2缺失数为20×1.5=30次。关于每条指令的平均存储器停顿，假定缺失数在指令与数据之间是均匀分布的：

$$\begin{aligned}每条指令的平均存储器停顿&=每条缺失的缺失数_{L1}\times命中时间_{L2}+每条指令的缺失数_{L2}\times缺失代价_{L2}\\&=(60/1000)\times10+(30/1000)\times200\\&=0.060\times10+0.030\times200=6.6个时钟周期\end{aligned}$$

如果从存储器平均访问时间（AMAT）中减去L1命中时间，然后再乘以每条指令的平均存储器引用数，则可以得到每条指令平均存储器停顿值：

$$(5.4-1.0)\times1.5=4.4\times1.5=6.6个时钟周期$$

如本例所示，与缺失率相比，使用每条指令的缺失数进行计算可能减少多级缓存造成的混淆。

注意，这些公式是针对混合式读取与写入操作的，假定采用写回第一级缓存。显然，直写第一级缓存会将所有写入操作都发往第二级，而不是仅限于缺失，而且还可能使用写入缓冲区。

图 B-7 和图 B-8 显示了一个设计中的缺失率和相对执行时间是如何随着第二级缓存的大小而变化的。从两个图中可以有两点体悟。第一，如果第二级缓存远大于第一级缓存，则全局缓存缺失率与第二级缓存的单一缓存缺失率非常类似。第二，局部缓存缺失率不是第二级缓存的良好度量标准；它是第一级缓存缺失率的函数，因此可以通过改变第一级缓存而变化。所以，在评估第二级缓存时，应当使用全局缓存缺失率。

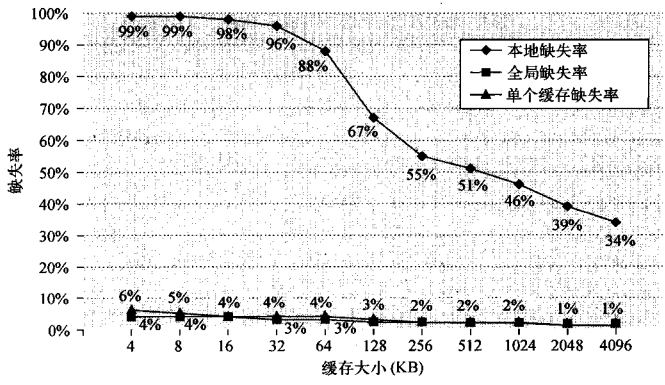


图 B-7 多级缓存的缺失率随缓存大小的变化。小于两个 64 KB 一级缓存总和的第二级缓存没有什么意义，从其高缺失率中可以反映出这一点。大于 256 KB 之后，单个缓存在全局缺失率在 10%以内。单级缓存的缺失率随大小的变化是根据第二级缓存的局部缺失率和全局缺失率绘制的，采用的是 32 KB 一级缓存。L2 缓存（统一缓存）为两路组相联，采用替换策略。分别有独立的 L2 指令与数据缓存，它们都是 64 KB 两路组相联，采用 LRU 替换策略。L1 与 L2 缓存的块大小均为 64 字节。数据的收集条件与图表 B-2 相同

有了这些定义，我们可以考虑第二级缓存的参数。两级缓存之间的首要区别就是第一级缓存的速度影响着处理器的时钟频率，而第二级缓存的速度仅影响第一级缓存的缺失代价。因此，我们可以在第二级缓存中考虑许多不能用于第一级缓存的替代选项。在设计第二级缓存时，主要有两个问题：是否要降低 CPI 的存储器平均访问时间部分？其成本有多高？

首先要决定的是第二级缓存的大小。由于第一级缓存中的所有内容都可能在第二级缓存中，所以第二级缓存应当远大于第一级缓存。如果第二级缓存只是稍大一点，那局部缺失率将会很高。这一观察结果激励了巨型第二级缓存的设计——其大小达到较早期计算机主存储器的规模！

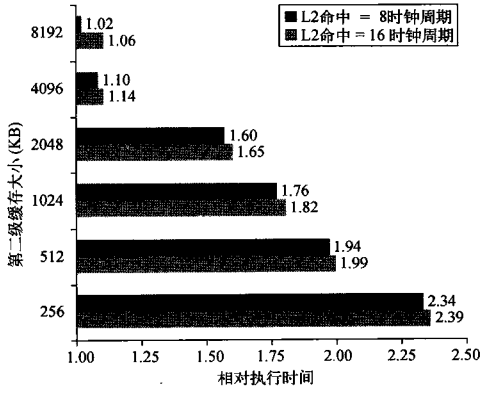


图 B-8 相对执行时间与第二级缓存大小的关系。图中的每两个长条表示一次 L2 缓存命中的不同时钟周期。引用执行时间 1.00 是指一个 8192 KB 第二级缓存在第二级命中的延迟为 1 个时钟周期。这些数据的收集方式与图 B-7 相同，使用模拟器模拟了 Alpha 21264

有一个问题是：组相联对于第二级缓存是否有意义？

例题 给定以下数据，第二级缓存相联度对其缺失代价的影响如何？

- 直接映射的命中时间 $_{L2}$ 为 10 个时钟周期。
- 两路组相联将命中时间增加 0.1 个时钟周期，达到 10.1 个时钟周期。
- 直接映射的局部缺失率 $_{L2}$ 为 25%。
- 缺失代价 $_{L2}$ 为 200 个时钟周期。
- 两路组相联直接映射的局部缺失率 $_{L2}$ 为 20%。

解答 对于直接映射第二级缓存，第一级缓存缺失代价为：

B-33

$$缺失代价_{一路L2} = 10+25\% \times 200 = 60.0 个时钟周期$$

加上相联度的成本仅使命中成本增加 0.1 个时钟周期，由此得到新的第一级缓存缺失代价为：

$$缺失代价_{两路L2} = 10.1+20\% \times 200 = 50.1 个时钟周期$$

事实上，第二级缓存几乎总是与第一级缓存、处理器同步。相应地，第二级命中时间必须为整数个时钟周期。如果幸运的话，我们可以将第二级命中时间缩短到 10 个时钟周期；如果不够幸运，则会舍入到 11 个周期。相对于直接映射第二级缓存，任一选项都是一种改进：

$$缺失代价_{两路L2} = 10+20\% \times 200 = 50.0 个时钟周期$$

$$缺失代价_{两路L2} = 11+20\% \times 200 = 51.0 个时钟周期$$

现在我们可以通过降低第二级缓存的缺失率来降低缺失代价了。

另一条关注事项涉及第一级缓存中的数据是否在第二级缓存中。**多级包含**是存储器层次结构的一种自然策略：L1 数据总是出现在 L2 中。这种包含性是我们所希望的，因为仅通过检查

B-34

第二级缓存就能确定 I/O 与缓存之间（或多处理器中的缓存之间）的一致性。

包含性的一个缺点就是：测量结果可能提议对较小的第一级缓存使用较小的块，对较大的第二级缓存使用较大的块。例如，Pentium 4 的 L1 缓存中的块为 64 个字节，L2 缓存中的块为 128 个字节。包含性仍然能够得到保持，但在第二级缺失时要做更多工作。如果一级块所映射的二级块将被替换，则第二级缓存必须使所有此类一级块失效，从而会略微提高第一级缺失率。为了避免此类问题，许多缓存设计师使所有各级缓存的块大小保持一致。

但是，如果设计师只能承受略大于 L1 缓存的 L2 缓存呢？它是不是有很大一部分空间要被用作 L1 缓存的冗余副本？在此种情况下，可以使用一种明显相反的策略：**多级互斥**，L1 中的数据绝对不会出现在 L2 缓存中。典型情况下，在采用互斥策略时，L1 中的缓存缺失将会导致 L1 与 L2 的块交换，而不是用 L2 块来替代 L1 块。这一策略防止了 L2 缓存中的空间浪费。例如，AMD Opteron 芯片使用两个 64 KB L1 缓存和 1 MB L1 缓存来执行互斥策略。

这些问题表明，尽管一些新手可能会独立地设计第一级和第二级缓存，但在给定一个兼容第二级缓存时，第一级缓存设计师的任务要简单一些。比如，如果下一级有写回缓存来为重复写入操作提供支持，而且使用了多级包含，那使用直写的风险就会小一些。

所有缓存设计的基础都是在加速命中和减少缺失之间实现平衡。对于第二级缓存，命中数要比第一级缓存中少得多，所以重心更多地偏向减少缺失。因为这一认知，人们开始采用大得多的缓存和降低缺失率的技术，比如更高的相联度和更大的块。

B.3.5 第五种优化方法：使读取缺失的优先级高于写入缺失，以降低缺失代价

这一优化方法在完成写入操作之前就可以为读取操作提供服务。我们首先看一下写入缓冲区的复杂性。

采用直写缓存时，最重要的改进就是一个大小合适的写入缓冲区。但是，由于写入缓冲区可能包含读取缺失时所需要的更新值，所以它们的确会使存储器访问变得复杂。

例题 看以下序列：

```
SW R3, 512(R0)     ;M[512] ← R3     (cache index 0)
LW R1, 1024(R0)    ;R1 ← M[1024]    (cache index 0)
LW R2, 512(R0)     ;R2 ← M[512]     (cache index 0)
```

假定有一个直接映射直写缓存，它将 512 和 1024 映射到同一块中，假定有一个四字写入缓存区，在读取缺失时不会进行检查。R2 中的值是否总等于 R3 中的值？ B-35

解答 使用第 2 章的术语，这是存储器中的一个“写后读”数据冒险。我们通过跟踪一次缓存访问来了解这种危险性。R3 的数据在存储之后被放在写入缓冲区中。随后的载入操作使用相同的缓存索引，因此产生一次缺失。第二条载入指令尝试将位置 512 处的值放到寄存器 R2 中，这样也会导致一次缺失。如果写入缓冲区还没有完成向存储器中位置 512 的写入，对位置 512 的读取就会将错误的旧值放到缓存块中，然后再放入 R2 中。如果没有事先防范，R3 是不等于 R2 的！

摆脱这一两难境地的最简单方法是让读取缺失一直等待到写入缓冲区为空为止。一种替代方法是在发生读取缺失时检查写入缓冲区的内容，如果没有冲突而且存储器系统可用，则让读取缺失继续。几乎所有桌面与服务器处理器都使用后一方法，使读取操作的优先级高于写入操作。

处理器在写回缓存中的写入成本也可以降低。假定一次读取缺失将替换一个脏服务器块。

我们不是将这个脏块写到存储器中，然后再读取存储器，而是将这个脏块复制到缓冲区中，然后读存储器，然后再写存储器。这样，处理器的读取操作将会很快结束（处理器可能正在等待这一操作的完成）。和前一种情况类似，如果发生了读取缺失，处理器或者停顿到缓冲区为空，或者检查缓冲区中各个字的地址，以了解是否存在冲突。

我们已经介绍了五种用于降低缓存缺失代价或缺失率的优化方法，现在该来研究一下如何降低存储器平均访问时间的最后一个分量。命中时间会影响到处理器的时钟频率，所以它是至关重要的；在今天的许多处理器中，缓存访问时间限制都限制了时钟频率，即使那些使用多个时钟周期来访问缓存的处理器也是如此。因此，缩短命中时间可以对各个方面提供帮助，从而具有多重重要性，超出了存储器平均访问时间公式的限制。

B.3.6 第六种优化方法：避免在索引缓存期间进行地址转换，以缩短命中时间

即使一个小而简单的缓存也必须能够将来自处理器的虚拟地址转换为用以访问存储器的物理地址。如 B.4 节所述，处理器就是将主存储器看作另一级存储器层次结构，因此，必须将存在于磁盘上的虚拟存储器地址映射到主存储器。

根据“加快常见情景速度”这一指导原则，我们为缓存使用虚拟地址，因为命中的出现频率当然远高于缺失。这种缓存被称为**虚拟缓存**，而**物理缓存**用于表示使用物理地址的传统缓存。稍后将会看到，区别以下两项任务是非常重要的：索引缓存、对比地址。因此，问题是：在索引缓存中应当使用虚拟地址还是使用物理地址，在标志对比中应当使用虚拟地址还是使用物理地址。如果对索引和标志都完全采用虚拟寻址，那在缓存命中时就可以省掉地址转换的时间。那为什么不是所有体系结构都构建虚拟寻址的缓存呢？

B-36

一个原因是要提供保护。在将虚拟地址转换为物理地址时，无论如何都必须检查页级保护。一种解决方案是在缺失时从 TLB 复制保护信息，添加一个字段来保存这一信息，然后在每次访问虚拟寻址缓存时进行核对。

另一个原因是：在每次切换进程时，虚拟地址会指向不同的物理地址，需要对缓存进行刷新。图 B-9 显示了这一刷新对缺失率的影响。一种解决方案是增大缓存地址标志的宽度，增加一个**进程识别符标志**（PID）。如果操作系统将这些标志指定给进程，那么只需要在 PID 被回收时才刷新缓存；也就是说，PID 可以区分缓存中的数据是不是为此这个程序准备的。图 B-9 显示了通过 PID 避免缓存刷新而对缺失率的改善。

B-37

虚拟缓存没有更加普及的第三个原因是操作系统和用户程序可能为同一物理地址使用两种不同的虚拟地址。这些重复地址称为**同义**地址或**别名**地址，可能会在虚拟缓存中生成同一数据的两个副本；如果其中一个被修改了，另一个就会包含错误值。而采用物理缓存是不可能发生这种情况的，因为这些访问将会首先被转换为相同的物理缓存块。

同义地址问题的硬件解决方案称为别名消去，保证每个缓存块都拥有一个独一无二的物理地址。例如，AMD Opteron 使用两路组相联的 64 KB 指令缓存，页大小为 4 KB；因此，硬件必须处理组索引中三个虚拟地址位所涉及的别名问题。它避免别名的方法就是在发生缺失时检查所有 8 种可能地址（4 个组中各有 2 个块），以确保它们都与被提取数据的物理地址相匹配。如果发现匹配，则使其失效，所以在向缓存中载入新数据时，就能保证它们的物理地址是独一无二的。

软件可以强制这些别名共享某些地址位，从而大大简化了这一问题。比如 Sun Microsystems 出品的一个较早 UNIX 版本，它要求所有别名地址的后面 18 位都必须相同；这一限制称为页面

着色。注意，页面着色就是向虚拟存储器应用的组相联映射：使用 64（2^6）个组来映射 4 KB（2^{12}）个页面，确保物理地址和虚拟地址的后 18 位匹配。这一限制意味着不大于 2^{18}（256 K）字节直接映射缓存绝对不会为块使用重复的物理地址。从缓存的角度来看，页着色有效地增大了页偏移，因为软件保证了虚拟、物理页地址的最后几位是相同的。

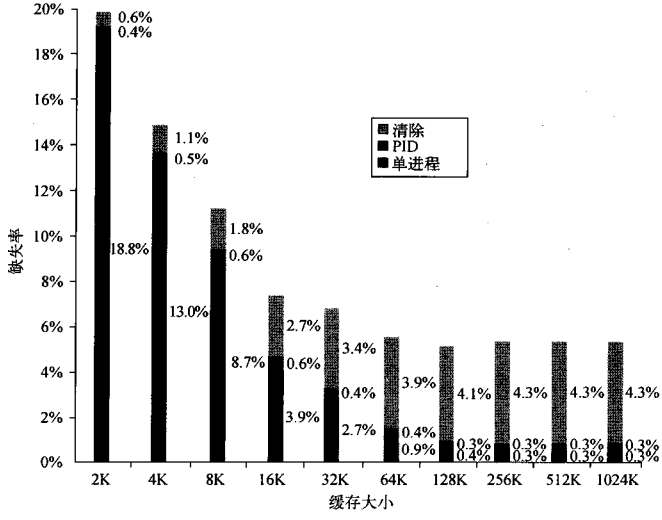


图 B-9 一个程序的缺失率随虚拟寻址缓存大小的变化，分三种情况测量：没有进程切换（单进程）、使用进程识别符标志（PID）进行进程切换，有进程切换但没有 PID，即清除（purge）模式。PID 使单进程绝对缺失率增加 0.3%～0.6%，比清除模式节省 0.6%～4.3%。Agarwal [1987]针对 VAX 上运行的 Ultrix 操作系统收集了这些统计数字，假定采用直接映射缓存，块大小为 16 字节。注意缓存大小从 128 K 到 256 K 时缺失率的增加。因为大小变化会改变存储器块到缓存块的映射，而这种变化又会改变冲突缺失率，所以在缓存中可能会发生这种与人们直观感觉不一致的行为

最后一部分与虚拟地址相关的领域是 I/O。I/O 通常使用物理地址，从而需要映射到虚拟地址，以与虚拟地址进行交换。（I/O 对缓存的影响将在附录 D 中另行深入讨论。）

一种使虚拟缓存与物理缓存均能实现最佳性能的备选方法是使用一部分页偏移量（也就是虚拟地址与物理地址保持一致的那一部分）来索引缓存。在使用索引读取缓存的同时，地址的虚拟部分被转换，标志匹配使用了物理地址。

这一备选方法允许缓存读取操作立即开始，而标志对比仍然使用物理地址。这种虚拟索引、物理标志备选方法的局限性是直接映射缓存不能大于页大小。例如，在图 B-3 中的数据缓存中，这个索引为 9 位，缓存块偏移量为 6 位。为了利用这一技巧，虚拟页大小至少为 $2^{(9+6)}$个字节，即 32 KB。如果不是这样，则必须将该索引的一部分由虚拟地址转换为物理地址。图 B-10 显示了在使用这一技术时的缓存、转换旁视缓冲区（TLB）和虚拟存储器的组织方式。

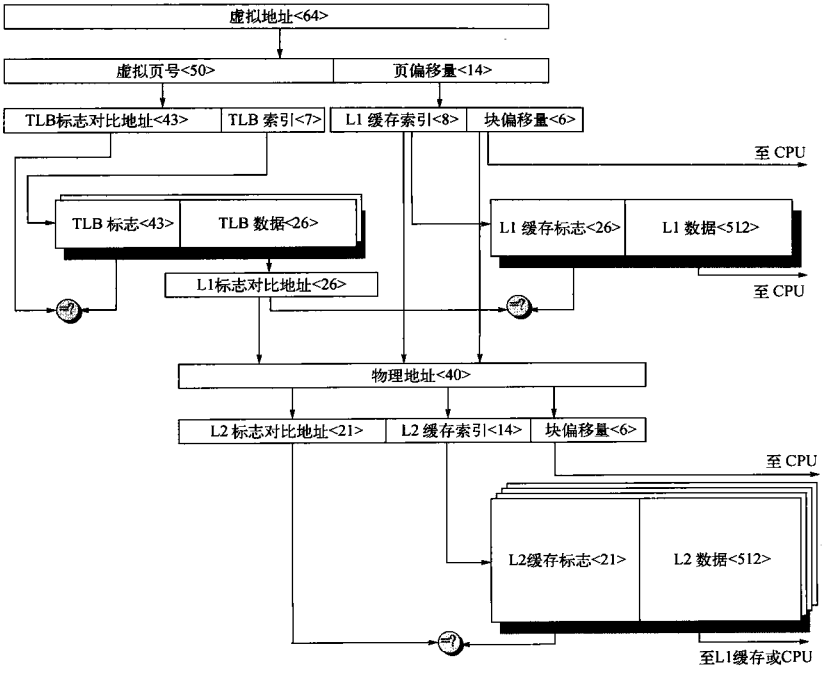


图 B-10　一种从虚拟地址到 L2 缓存访问的虚设存储器层次结构的整体图像。页大小为 16 KB。TLB 是拥有 256 项的两路组相联。L1 缓存是一个直接映射 16 KB，L2 缓存是一个总容量为 4 MB 的四路组相联。这两者的块大小都是 64 个字节。虚拟地址为 64 位，物理地址为 40 位

相联度可以将此索引保存在地址的物理部分，但仍支持大型缓存。回想一下，索引的大小受以下公式的控制：

$$2^{索引}=\frac{缓存大小}{块大小\times组相联}$$

B-39 例如，使相联度和缓存大小同时加倍并不会改变索引的大小。作为一个极端示例，IBM 3033 缓存是一个十六路组相联，尽管研究表明：在八路以上的组相联中，对缺失率没有什么好处。尽管 IBM 体系结构中存在页大小为 4 KB 这一障碍，这一高相联度允许使用物理索引对 64 KB 缓存进行寻址。

B.3.7　基本缓存优化方法小结

本节介绍了用于降低缺失率与缺失代价、缩短命中时间的技术，这些技术通常会影响到存储器平均访问公式的其他部分，还会影响到存储器层次结构的复杂性。表 B-8 总结了这些技术，并估计了对复杂性的影响，“+”表示该技术对该因素有改进，“–”表示该技术对该因素有伤害，

空白表示没有影响。本图中任何一项优化方法都不能对一个以上的类别提供帮助。

表B-8 关于本附录中介绍的技术，基本缓存优化对缓存性能和复杂度的影响汇总

技术	命中时间	缺失代价	缺失率	硬件复杂度	备注
较大的块大小		–	+	0	微小；Pentium 4 L2使用128字节
较大的缓存大小	–		+	1	广泛应用，特别是对L2缓存
较高的相联度	–		+	1	广泛应用
多级缓存		+		2	昂贵的硬件，如果L1块大小≠L2块大小，广泛应用
读取操作优先级高于写入操作		+		1	广泛应用
避免在缓存索引期间进行地址转换	+			1	广泛应用

* 从整体上来说，一项技术只能改善一个因素。+意味着该技术对该因素有改进，–意味着它对该因素有伤害，空白表示没有影响。复杂度是主观的，0表示最容易，3表示最富挑战性。

B.4 虚拟存储器

……已经设计了一种系统，使磁芯组合存储方式（core drum combination）在程序员看来就是单个存储层，必要的转移都是自动进行的。

——Kilburn 等人[1962]

计算机在任何时刻都在运行多个进程，每个进程有自己的地址空间。（进程将在下一节描述。）让每个进程专门使用存储器的完整地址空间，成本太高了，特别是许多进程连其自己的地址空间也只使用了很少一部分。因此，必须有一种方法，用于在许多进程之间共享较少量的物理空间。 B-40

其中一种做法——**虚拟存储器**，将物理存储器划分为块，并分配给不同的进程。这种方法必然要求采用一种保护机制来限制各个进程，使其仅能访问属于自己的块。虚拟存储器的许多形式还缩短了程序的启动时间，因为程序启动之前不再需要物理存储器中的所有代码和数据。

尽管由虚拟存储器提供的保护对于目前的计算机来说是必需的，但共享并不是发明虚拟内存的原因。如果一个程序对物理内存来说变得过于庞大，就需要由程序员负责将其装进去。程序员将程序划分为片段，然后确认这些互斥的片断，在执行时间根据用户程序控制来加载或卸载这些**覆盖段**（overlay）。程序员确保程序绝对不会尝试访问超出计算机现有的物理主存储器，并确保会在正确的时间加载正确的覆盖段。容易想到，这种责任降低了程序员的生产效率。

虚拟存储器的发明是为了减轻程序员的这一负担；它自动管理表示为主存储器和辅助存储的两级存储器层次结构。图 B-11 显示了程序从虚拟存储器到物理存储器的映射，共有 4 个页面。 B-41

除了共享受保护的存储器空间和自动管理存储器层次结构之外，虚拟存储器还简化了为执行程序而进行的加载过程。这种被称为**再定位**（relocation）的机制允许同一程序在物理存储器中的任意位置运行。图 B-11 中的程序可以定位在物理存储器中的任何位置，也可以放在磁盘上，只需要改变它们之间的映射即可。（在虚拟存储器普通之前，处理器中包含一个用于此目的的再定位寄存器。）硬件解决方案的一种替代方法是使用软件，在每次运行一个程序时，改变其所有地址。

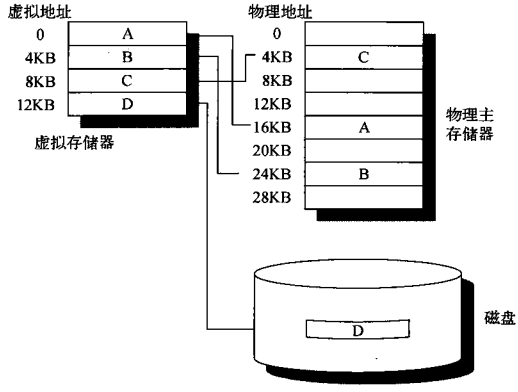


图 B-11　左侧给出位于相邻虚拟地址空间中的逻辑程序。它包括 A、B、C 和 D 4 个页。这些块中有 3 个的实际位置在物理主存储器中，另一个位于磁盘上

第 1 章中几个有关缓存的一般性存储器层次结构思想与虚拟存储器类似，当然，其中有许多术语不同。**页**或**段**表示块，**页错误**或**地址错误**用于缺失。有了虚拟存储器，处理器会给出**虚拟地址**，由软硬件组合方式转换为**物理地址**，再来访问主存储器。这一过程称为**存储器映射**或**地址转换**。今天，由虚拟地址控制的两级存储器层次结构为 DRAM 和磁盘。表 B-9 显示了虚拟存储器存储器层次结构参数的典型范围。

除了表 B-9 中提到的量化区别之外，缓存与虚拟存储器之间还有其他一些区别，如下所述。

- 发生缓存缺失时的替换主要由硬件控制，而虚拟存储器替换主要由操作系统控制。缺失代价越长，正确作出决定就显得越重要，所以操作系统可以参与其中，花费一些时间来决定要替换哪些块。
- 处理器地址的大小决定了虚拟存储器的大小，但缓存大小与处理器地址大小无关。
- 除了在层次结构中充当主存储器的低一级后援存储之外，辅助存储还用于文件系统。事实上，文件系统占用了大多数辅助存储。它通常不在地址空间中。

B-42

表B-9　缓存与虚拟存储器的典型参数范围

参　数	第一级缓存	虚拟存储器
块（页大小）	16~128字节	4096~65 536字节
命中时间	1~3个时钟周期	100~200个时钟周期
缺失代价	8~200个时钟周期	1 000 000~10 000 000个时钟周期
（访问时间）	（6~160个时钟周期）	（800 000~8 000 000个时钟周期）
（传输时间）	（2~40个时钟周期）	（200 000~2 000 000个时钟周期）
缺失率	0.1%~10%	0.000 01%~0.001%
地址映射	25~45位物理地址到14~20位缓存地址	32~64位虚拟地址到25~45位物理地址

* *虚拟存储器参数是缓存参数的 10 到 1 000 000 倍。通常，第一级缓存包含至少 1 MB 数据，而物理存储器包含 256 MB 到 1 TB 数据。*

虚拟存储器还包含几种相关技术。虚拟存储器系统可分为两类：**页**，采用大小固定的块；**段**，采用大小可变的块。页大小通常固定为 4 096 至 8 192 字节，而段大小是变化的。任意处理器所支持的最大段范围为 2^{16} 个字节至 2^{32} 个字节，最小段为 1 个字节。图 B-11 显示了这两种方法可以如何划分代码和数据。

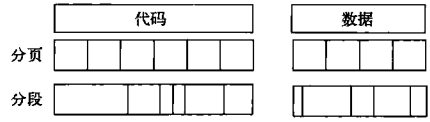


图 B-11 分页和分段方式对程序的划分示例

是使用页虚拟存储器还是段虚拟存储器，这一决定会影响处理器。页寻址方式的地址是单一固定大小，分为页编号和页内偏移量，与缓存寻址类似。单一地址对分段地址无效，可变大小的段需要 1 个字来表示段号，1 个字表示段内的偏移量，总共 2 个字。对编译器来说，不分段地址空间更简单一些。 B-43

这两种方法的优缺点已经在操作系统教科书中进行了很好的阐述，表 B-10 总结了这些观点。由于替换问题（表中第三行），今天很少再有计算机使用纯粹的分段方法。一些计算机使用一种名为**页式分段**的混合方式，在这种方式中，一个段由整数个页组成。由于存储器不需要是连续的，也不需要所有段都在主存储器中，从而简化了替换过程。最近的一种混合方式是由计算机提供多种页面大小，较大页面的大小为最小页面大小的整数倍，且为 2 的幂。例如，IBM 405CR 嵌入式处理器允许单个页面为 1 KB、4 KB ($2^2 \times 1$ KB)、16 KB ($2^4 \times 1$ KB)、64 KB ($2^6 \times 1$ KB)、256 KB ($2^8 \times 1$ KB)、1024 KB ($2^{10} \times 1$ KB)、4096 KB ($2^{12} \times 1$ KB)。

表B-10 分页与分段的对比

	页	段
每个地址的字	1	2（段和偏移）
程序员是否可见	应用程序程序员不可见	应用程序程序员可以看到
替换块	很容易（所有块的大小相同）	困难（必须查找主存储器中相邻的、可变大小的未使用部分）
存储器使用效率低下	内部分段（页的未使用部分）	外部分段（主存储器的未使用部分）
高效的磁盘通信	是的（调整页大小，以平衡访问时间和传输时间）	并非总是如此（小段可能仅传输几个字节）

* 这两者都可能浪费存储器，取决于块大小及各分段能否很好地容纳于主存储器中。采用不受限指针的编程语言需要传递段和地址。一种称为**页式分段**的混合方法可以发挥这两者的最佳状态：分段由页组成，所以替换一个块是很轻松的，而一个段仍被看作一个逻辑单位。

B.4.1 再谈存储器层次结构的 4 个问题

我们现在已经为回答虚拟存储器的四个存储器层次结构问题做好了准备。

问题 1：一个块可以放在主存储器的什么位置？

虚拟存储器的缺失代价涉及旋转磁存储设备的访问，因此非常高。如果在较低缺失率与较简单放置算法之间进行选择，操作系统设计人员通常选择较低缺失率，因为其缺失代价可能会高得离谱。因此，操作系统允许将块放在主存储器中的任意位置。根据图 B-1 中的术语，这一策略可以标记为全相联的。

问题 2：如果一个块在主存储器中，如何找到它？

分页和分段都依靠一种按页号或段号索引的数据结构。这种数据结构包含块的物理地址。对于分段方式，会将偏移量加到段的物理地址中，以获得最终物理地址。对于分页方式，该偏移量只是被串接到这一物理分布地址（见图 B-13）。

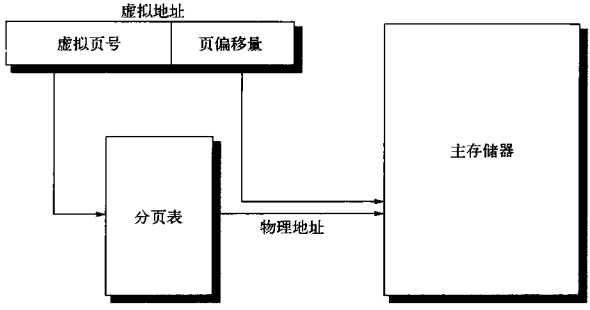


图 B-13　通过页表将虚拟地址映射到物理地址

这一包含物理页地址的数据结构通常采用一种分页表的形式。这种表通常根据虚拟页号进行索引，其大小就是虚拟地址空间中的页数。如果虚拟地址为 32 位、4 KB 页、每个页表项（PTE）大小为 4 字节，则页表的大小为（$2^{32}/2^{12}$）$\times 2^2=2^{22}$ 即 4 MB。

为了缩小这一数据结构，一些计算机向虚拟地址应用了一种散列功能。这种散列允许数据结构的长度等于主存储器中物理页的数目。这一数目可以远小于虚拟页的数目。这种结构被称为**反转分页表**。利用前面的例子，一个 512 MB 的物理存储器可能只需要 1 MB（8 × 512 MB/4 KB）的反转分页表；每个页表项另外需要 4 字节，用于表示虚拟地址。HP/Intel IA-64 同时支持传统页表和反转分页表，具体使用哪一种机制，交由操作系统程序员选择。

B-44

为了缩短地址转换时间，计算机使用一个专门进行这些地址变换的缓存，称为**变换旁视缓冲区**，或者简称为**变换缓冲区**，稍后将进行详细介绍。

问题 3：在虚拟存储器缺失时应当替换哪个块？

前面曾经提到，操作系统的最高指导原则是将页错误降至最低。几乎所有操作系统都与这一指导原则保持一致，尝试替换最近使用最少（LRU）的块，这是因为如果用过去的信息来预测未来，将来用到这种块的可能性最低。

为了帮助操作系统评估 LRU，许多处理器提供了一个**使用位**或**参考位**，从逻辑上来说，只要访问一个页，就应对其进行置位。（为了减少工作，通过仅在发生转换缓冲区缺失时对其进行置位，稍后将对此进行介绍。）操作系统定期对这些使用位清零，之后再记录它们，以判断在一个特定时间段时使用了哪些页。通过这种方式进行跟踪，操作系统可以选择最近引用最少的一个页。

问题 4：在写入时发生什么？

主存储器的下一级包含旋转磁盘，其访问会耗时数百万个时钟周期。由于访问时间的巨大差异，还没有人构建一种虚拟存储器操作系统，在处理器每次执行存储操作时将主存储器直写

到磁盘上。（不要因为看到这一叙述，就认为这是一个通过首次构建这种操作系统而成名的机会！）因此，这里总是采用写回策略。

由于对低一级的非必需访问会带来如此之高的成本，所以虚拟存储器系统通常会包含一个重写位。利用这一重写位，可以仅将上次读取磁盘之后经过修改的块写至磁盘。

B.4.2 快速地址变换技术

分页表通常很大，从而存储在主存储器中，有时它们本身就是分页的。分页意味着每次存储器访问在逻辑上至少要分两次进行，第一次存储器访问是为了获得物理地址，第二次访问是为了获得数据。第 2 章曾经提到，我们使用局域性来避免增加存储器访问次数。将地址变换局限在一个特殊缓存中，存储器访问就很少再需要第二次访问来转换数据。这一特殊地址变换缓存被称为**变换旁视缓冲区**（TLB），有时也称为**变换缓冲区**（TB）。

TLB 项就像是一个缓存项目，其中的标志保存了虚拟地址部分，数据部分保存了特殊页帧编号、保护字段、有效位，通常还有一个使用位和重写位。要改变页表中某一项的特殊页帧编号或保护字段，操作系统必须确保旧项不在 TLB 中；否则，系统就不能正常运行。注意，这个重写位意味着对应页曾被改写过，而不是指 TLB 中的地址变换或数据缓存中的特殊块经过改写。操作系统通过改变页表中的值，然后再使相应 TLB 项失效来重置这些位。在从分页表中重新加载该项时，TLB 会获得这些位的准确副本。

图 B-14 给出了 Opteron 数据 TLB 组织方式，并标出了每一个变换步骤。这个 TLB 使用全相联布置；因此，变换首先向所有标志发送虚拟地址（步骤 1 和步骤 2）。当然，这些标志必须标记为有效，以允许进行匹配。同时，根据 TLB 中的保护信息核对存储器访问的类型，以确认其是否有效（也在步骤 2 中完成）。

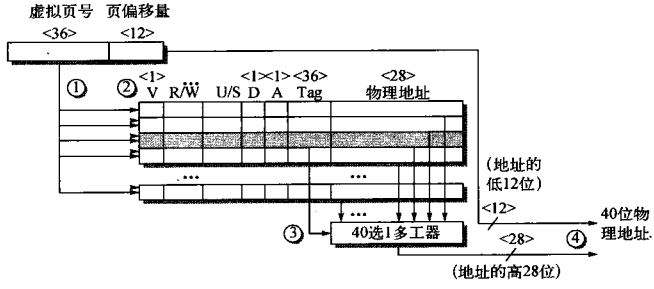


图 B-14 在地址变换期间 Opteron 数据 TLB 的操作。一次 TLB 命中的 4 个步骤用带圆圈的数字显示。这个 TLB 有 40 项。B.5 节描述了 Opteron 页表项的各种保护字段与访问字段

和缓存中的理由相似，TLB 中也不需要包含页偏移量的 12 个位。匹配标志通过一个 40 选 1 多工器，高效地发送相应的物理地址（步骤 3）。然后将页偏移量与物理页帧合并，生成一个完整的物理地址（步骤 4）。地址大小为 40 位。

关于处理器时钟周期的确定，地址变换很可能发挥至关重要的作用，所以 Opteron 使用虚拟寻址、物理标记的 L1 缓存。

B.4.3 选择页大小

最显而易见的体系结构参数是页大小。页大小的选择实际就是在偏向较大页与偏向较小页的力量之间进行平衡的问题。以下因素偏向较大尺寸。

B-46

- 页表的大小与页大小成反比，因此，增大页的大小可以节省存储器（或其他用于存储器映射的资源）。
- B.3 节曾经提到，分页较大时，可以允许缓存命中时间较短的较大缓存。
- 与传递较小页相比，从（向）辅助存储传递较大页（有可能通过网络）的效率更高一些。
- TLB 项目的数量受限，所以分页较大意味着可以高效地映射更多存储器，从而可以降低 TLB 缺失数量。

由于最后这个原因，近来的微处理器决定支持多种页大小；对于一些程序，TLB 缺失对 CPI 的重要性可能与缓存缺失相同。

采用较小分页的主要动机是节省存储。当虚拟内存的相邻区域不等于页大小的整数倍时，采用较小页可以减少存储的浪费空间。页面中这种未使用存储器的术语名称为**内部碎片**。假定每个进程有三个主要段（文本、堆和栈），每个进程的平均浪费存储量为页大小的 1.5 倍。对于有数百 M 存储器、页大小为 4 KB 至 8 KB 的计算机来说，这点数量是可以忽略的。当然，当页大小非常大（超过 32 KB）时，那就可能浪费存储（主存储器和辅助存储器）和 I/O 带宽了。

B-47

最后一项关注是进程启动时间；许多进程都很小，较大的页面可能会延长调用一个进程的时间。

B.4.4 虚拟存储器和缓存小结

由于虚拟存储器、TLB、第一级缓存、第二级缓存都映射到虚拟与物理地址空间的一部分，所以人们可能会混淆哪些位去了哪里。图 B-15 给出了一个从 64 位虚拟地址到 41 位物理地址的假设示例，它采用两级缓存。这一 L1 缓存的缓存大小和页大小都是 8 KB，所以它是虚拟寻址、物理标记的。L2 缓存为 4 MB。这两者的块大小都是 64 个字节。

B-48

第一，64 位虚拟地址在逻辑上被划分为虚拟页号和页偏移量。前者被发送到 TLB，以备变换为物理地址，后者的高位被发送到 L1 缓存，充当索引。如果 TLB 匹配命中，则将物理页号发送到 L1 缓存标志，检查是否匹配。如果匹配，则是 L1 缓存命中。块偏移随后为处理器选择该字。

如果 L1 缓存核对显示为缺失，则使用物理地址尝试 L2 缓存。物理地址的中间部分用作 4MB L2 缓存的索引。将所得到的 L2 缓存标志与物理地址的上半部分对比，以检查是否匹配。如果匹配，我们得到一次 L2 缓存命中，数据被送往处理器，它使用块偏移量来选择所需字。在 L2 缺失时，会使用物理地址从存储器获取该块。

尽管这是一个简单示例，但该图与真实缓存之间的主要区别只是重复问题。因为，只有一个 L1 缓存。如果有两个 L1 缓存，会重复该图的上半部分。注意，这会导致拥有两个 TLB，而这正是典型情况。因此，一个缓存和 TLB 用于指令，由 PC 驱动，一个缓存和 TLB 用于数据，由实际地址驱动。

第二种简化是所有缓存与 TLB 都是直接映射的。如果有任何一个是 n 路组相联的，则会将每一组标志存储器、比较器和数据存储器重复 n 次，并用一个 n 选 1 多工器将数据存储器连接在一起，以选择命中内容。当然，如果总缓存大小保持不变，则缓存索引也会收缩 $\log 2n$ 位，如图 B-4 中的公式所示。

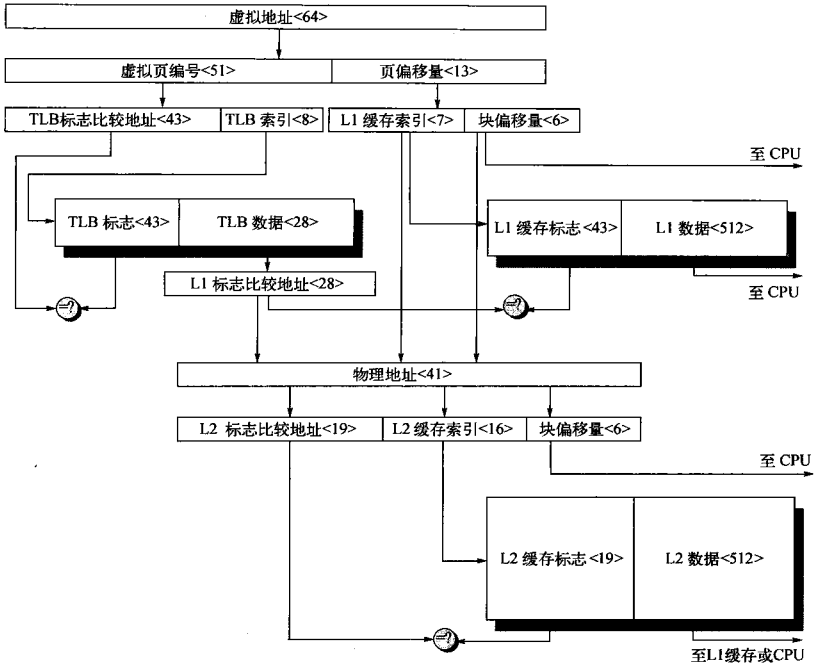


图 B-15 一个从虚拟地址到 L2 缓存访问的虚设存储器层次结构的整体图像。页大小为 8 KB。TLB 为直接映射，有 256 项。L1 缓存为直接映射 8 KB，L2 缓存为直接映射，大小为 4 MB。两者的块都是 64 字节。虚拟地址为 64 位，物理地址为 41 位。这一简单图像与实际缓存的主要差别在于这一图像多个组成部分的重复

B.5 虚拟存储器的保护与示例

在多重编程中，计算机由几个并行执行的程序共享，它的发明为程序之间的保护和共享提供了新的要求。这些要求与今天计算机中的虚拟存储器紧密捆绑在一起，所以我们在这里用两个虚拟存储器的示例来介绍这一主题。

多重编程导致了**进程**概念的出现。打个比方，进程就是程序呼吸的空气和生活的空间——即一个正在运行的程序加上持续运行它所需要的所有状态。时分共享是多重编程的一种变体，由几个同时进行交互的用户来共享处理器和存储器，给人的感觉是所有用户都拥有自己的计算机。因此，它在任何时刻都必须能够从一个进程切换到另一进程。这种交换被称为**进程切换**或**上下文切换**。

一个进程无论是从头到尾持续执行，还是被反复中断，与其他进程进行切换，其运行都必须正常进行。维护正确进程行为的责任由程序和操作系统的设计者共同分担。计算机设计师必 B-49

须确保进程状态的处理器部分能够保存和恢复。操作系统设计师必须确保这些进程不会相互干扰对方的计算。

保护一个进程的状态免受其他进程损害的最安全方法就是将当前信息复制到磁盘上。但是，一次进程切换可能需要几秒的时间——这对时分共享环境来说过长了。

这一问题的解决方法是由操作系统对主存储器进行划分，使几个不同进程能够在存储器同时拥有自己的状态。这种划分意味着操作系统设计师需要计算机设计师的帮助，以提供保护，使一个进程无法修改其他进程。除了保护之外，计算机还为进程之间共享代码和数据提供了支持，允许进程之间进行通信，或者通过减少相同信息的副本数目来节省存储器。

B.5.1 保护进程

使进程拥有自己的分页表，分别指向存储器的不同页面，这样可以为进程提供保护，避免相互损害。显然，必须防止用户修改它们的分页表，或者以欺骗方式绕过保护措施。

根据计算机设计者或购买者的理解，可以逐步升高保护级别。向处理器保护结构中加入环，可以将存储器访问保护扩展到远远超出最初的两级（用户级和内核级）。就像军用分类系统将信息划分为绝密、机密、秘密和非涉密相似，安全级别的同心环状结构可以让最受信任的人访问所有信息，第二受信任的人访问除最内层级别之外的所有信息，以此类推。“民用”程序是可信度最低的，因此对访问范围的限制也最多。关于存储器中可能包含代码的部分也有一些限制（执行保护），甚至对级别之间的入口点也要提供保护。本节后面介绍 Intel 80x86 保护结构，这一结构使用了环。在实践中，这些环相对于用户、内核模式的简单系统的改善并不是非常明显。

当设计者因为深入理解而加重安全恐惧时，这些简单的环可能就不够了。要限制程序在内层保密位置的自由度，需要采用一种新的分类系统。除了军用模型之外，还可以将这一系统比作钥匙和锁：没有钥匙的程序是不能对数据访问进行解锁的。为使这些钥匙（或者说能力）发挥作用，硬件和操作系统必须能够明确地在程序之间传送它们，而不允许程序自行伪造它们。为缩短钥匙核对时间，需要为这一核对机制提供大量硬件支持。

80x86 体系结构多年来已经尝试了这几种方法。由于保持后向兼容性是这种体系结构的指导原则之一，所以这一体系结构的最新版本包含了它在虚拟存储器中的所有试验探索。这里将回顾其中的两种：首先是较早的分段地址空间，然后是一种较新的平面 64 位地址空间。

B-50

B.5.2 分段虚拟存储器举例：Intel Pentium中的保护方式

一个人设计的第二个系统是他所有设计中最为危险的系统……第二个系统的功能设计过多是一种普遍倾向，设计师会将自己在第一个系统中谨慎排除的所有思想和装饰都塞到第二个系统中。

——F.P. Brooks, Jr.
The Mythical Man-Month (1975)

最早的 8086 使用分段寻址，但它没有提供任何虚拟存储器或保护。分段拥有基础寄存器，但没是界限寄存器，也没有访问核查，在能够载入分段寄存器之前，必须将相应段载入物理存储器中。Intel 对虚拟存储器和保护的专注在 8086 的后续产品中得到体现，扩展了一些字段来支持更大地址。这种保护机制非常精巧，精心设计了许多细节，以尝试避免安全漏洞。我们将其称为 IA-32。接下来的几页内容将重点介绍 Intel 的一些安全措施。如果你觉得阅读起来比较困难，那就想想实现它们的难度吧！

第一种增强是将传统的两级保护模型加倍：IA-32 有四级保护，最内层（0）对应于传统的内核模式，最外层（3）是权限最少的模型。IA-32 为每一级提供独立栈，以避免突破不同级别之间的安全措施。还有一些与传统分页表类似的数据结构，其中包含了段的物理地址，还有一个对变换地址进行的核对清单。

Intel 设计师并没有就此驻足不前。IA-32 划分了地址空间，让操作系统和用户都能访问整个空间。IA-32 用户能够在保持全面保护的情况下调用这一空间中的操作系统例程，甚至还可以向其传送参数。由于操作系统栈不同于用户栈，所以这一安全调用可不是一个简单操作。另外，IA-32 允许操作系统为那些传递给被调用例程的参数保持这些被调用例程的保护级别。禁止用户进程要求操作系统间接访问一些该进程自己不能访问的东西，就可以防止这一潜在保护漏洞。（此类安全漏洞称为特洛伊木马。）

Intel 设计师有一条指导原则：尽可能对操作系统持怀疑态度，并支持共享和保护。作为这种受保护共享的一个应用示例，假定有一个薪金支付系统，它要填写支票，并更新有关本年度截至当前为止的总薪金和津贴支付信息。因此，我们希望赋予该程序读取薪金、当前信息和修改当前信息的功能，但不能修改薪金。稍后将会看到支持这些功能的机制。在本小节的其他部分，将研究 IA-32 保护的整体描述，并研究其开发动机。

1. 增加界限检查和存储器映射

B-51

增强 Intel 处理器的第一个步骤是利用分段寻址来检查界限和提供基址。IA-32 中的分段寄存器中包含的不是基址，而是指向虚拟存储器数据结构的索引，这种结构称为**描述符表**。描述符表扮演着传统分页表的角色。IA-32 上与页表项等价的是**段描述符**。它包含可以在 PTE 中找到的字段，如下所述。

- **存在位**——等价于 PTE 有效位，用于表明这是一个有效变换。
- **基址字段**——等价于一个页帧地址，包含该段第一个字节的物理地址。
- **访问位**——类似于某些体系结构中的引用位或使用位，可以为替换算法提供帮助。
- **属性字段**——为使用这一段的操作指定有效操作和保护级别。

还有一个在分页系统中没有出现的**界限段**，它确定这一分段有效偏移量的上限。图 B-16 给出了 IA-32 段描述符的示例。

除了这种段式寻址之外，IA-32 提供了一种可选的分页系统。32 位地址的上面部分选择段描述符，中间部分是描述符所选页表中的索引。下面介绍不依赖分页的保护系统。

2. 增加共享和保护

为提供受保持的共享，地址空间的一半由所有进程共享，另一半由各进程独享，分别称为**全局地址空间**和**局部地址空间**。为每一半都提供一个拥有适当名字的描述符表。指向共享段的描述符被放在全局描述符表中，而指向专用段的描述符则被放在局部描述符表中。

程序向 IA-32 段寄存器中载入一个索引和一个位，索引指向描述符表，这一个位表明程序希望获得哪个表。根据描述符中的属性对操作进行检查，将来自处理器的偏移量加到描述符中的基址来构成物理地址，前提是这一偏移量要小于界限字段。每个段描述符都有一个独立的 2 位字段，提供这个段的合法访问级别。仅当程序尝试以段描述符中的较低保护级别使用段时，才会发生违反错误。

我们现在可以介绍如何调用上述薪金支付程序来更新当前信息，但不允许它更新薪金数据。可以向程序提供该信息的一个描述符，描述符的可写字段被清零，表明程序能够读取数据但不能写数据。然后可以提供一个受信任的程序，它只会写入当前最新信息。在向这一程序提供的

描述符中，其可写字段已被置位（见图 B-16）。薪金支付程序使用一个代码段描述符来调用受信任的代码，描述符的一致性字段已被置位。这种设置意味着被调用程序取得了被调用代码的权限级别，而不是调用者的权限级别。因此，薪金支付程序可以读取薪金信息，并调用受信任的程序来更新当前总值，但薪金支付程序不能修改这些薪金信息。如果系统中存在特洛伊木马，它必须位于受信任代码中才能生效，而这段代码的唯一任务就是更新截至目前的最新信息。这种保护类型的观点是通过限制易损范围来提高安全性。

B-52

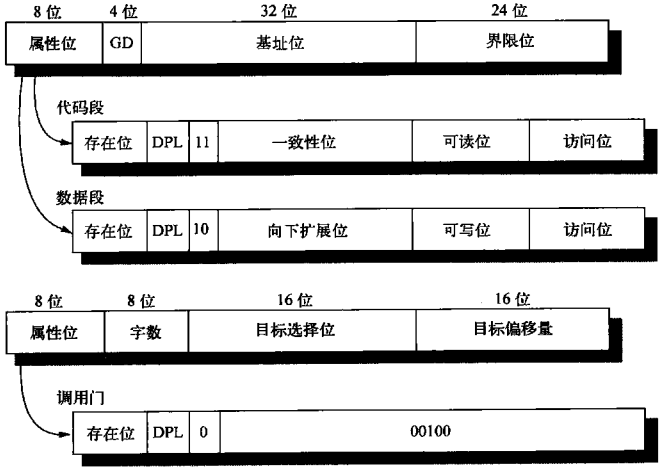


图 B-16　IA-32 段描述符由其属性字段中的位进行区分。基址位、界限位、存在位、可读位和可写位的用途都是不言自明的。D 是指令的默认寻址大小：16 位或 32 位。G 是段界限的粒度：0 表示采用字段，1 表示采用 4 KB 的页。在开启分页以设置页表大小时，将 G 设置为 1。DPL 表示**描述符权限级别**——根据代码权限级别核对 DPL，以查看是否允许访问。**一致性**是指代码采用被调用代码的权限级别，而不是调用者的权限级别；用于库例程。**向下扩展**段颠倒检查过程，以基址字段为高位标记，界限字段为低位标记。可以猜到，这种方式用于向下发展的栈段。**字数控制**从当前栈向调用门上新栈复制的字数。调用门描述符的其他两个字段——**目标选择位**和**目标偏移量**，分别选择该调用目标的描述符及其内部的偏移量。IA-32 保护模型中的段描述符远不止这三种

B-53

3. 增加从用户到操作系统门的安全调用，为参数继承保护级别

允许用户介入操作系统是非常大胆的一步。但是，硬件设计师如何能在不信任操作系统或其他代码的情况下增加安全系统的可能性呢？IA-32 方法是限制用户能够进入代码段的位置，将参数安全地放到正确的栈中，并确保用户参数不会取得被调用代码的保护级别。

为限制进入其他代码，IA-32 提供了一种被称为**调用门**（call gate）的特殊段描述符，用属性字段中的一位来识别。与其他描述符不同，调用门是一个对象在存储器中的完整物理地址；处理器提供的偏移量被忽略。如上所述，它们的目的是防止用户随机进入一段受保护或拥有更高权限的代码段中。在我们这个编程示例中，这意味着薪金支付程序唯一能够调用受信任代码的位置就是在准确的边界位置。一致性段的正常工作需要这一限制。

如果调用者和被调用者“相互怀疑”，都不相信对方，那会怎么样呢？在图 B-16 询问描述符的字数字段中可以找到解决方案。当一条调用指令调用一个调用门描述符时，描述符将局部栈中的一些字复制到这个段级别相对应的栈中，字的数量由描述符指定。这一复制过程允许用户首先将参数压入局部栈中，从而实现参数传递。随后由硬件将参数安全地传送给正确的栈。在从调用门返回时，会将参数从栈中弹出，并将返回值复制到正确的栈中。注意，这一模型与目前在寄存器中传递参数的实际做法不兼容。

这一机制仍然未能关闭潜在的安全漏洞：操作系统以操作系统的安全级别来使用作为参数传递的用户地址，而不是使用用户级别。IA-32 在每个处理器段寄存器中专门拿出 2 个位来指定**所请求的保护级别**，从而解决了上述问题。当这些地址参数载入段寄存器时，它们将所请求的保护级别设置为正确值。IA-32 硬件随后使用所请求的保护级别来防止出现欺骗：对于使用这些参数的系统例程，如果其权限保护级别高于被请求级别，则不允许该例程访问任何段。

B.5.3 分页虚拟存储器举例：64 位Opteron存储器管理

AMD 工程师发现人们很少使用上述这种精致、复杂的保护模型。常用的模型是在 80386 中引入的一种平面 32 位地址空间，它将段寄存器的所有基址值都设置为 0。因此，AMD 在 64 位模式中摒弃了多个段。它假定段基址为 0，忽略了界限字段。页大小为 4 KB、2 MB 和 4 MB。 B-54

AMD64 体系结构的 64 位虚拟地址被映射到 52 位物理地址，当然，具体实现可以采用较少的位数，以简化硬件。例如，Opteron 使用 48 位虚拟地址和 40 位物理地址。AMD64 需要虚拟地址的高 16 位就是低 48 位的符号扩展，这称为**规范格式**。

64 位地址空间页表的大小是惊人的。因此，AMD64 使用了一种多级层次结构分页表来映射地址空间，使其保持合理大小。级别数取决于虚拟地址空间的大小。图 B-17 显示了 Opteron 的 48 位虚拟地址的四级变换。

这些页表中各个表的偏移量来自 4 个 9 位字段。地址变换时，首先将第一个偏移量加到页映射第 4 级基址寄存器，然后从这个位置读取存储器，获取下一级页表的基址。然后再将下一级地址偏移量添加到这个新获取的地址，再次访问存储器，以确定第三个页表的基址。再次重复以上过程。最后一个地址字段被加到这一最终基址，使用两者之和读取存储器，（最终）获得所引用页面的物理地址。这个地址与 12 位页面偏移量串接在一起，获得完整的物理地址。注意，Opteron 体系结构中的页表可以放在单个 4 KB 页中。 B-55

Opteron 在每个页表中使用 64 位的项目。前 12 位为保留位，供以后使用，接下来的 52 位包含物理页帧编号，最后 12 位提供保护、使用信息。尽管不同页表级别之间的字段会有所变化，但基本上都有以下位。

- **存在位**——表明该页存在于存储器中。
- **读取/写入位**——表明一个页是只读的，还是读写的。
- **用户/管理员位**——表明用户是可以访问该页，还是仅限于上面三个权限级别。
- **脏位**——表明该页是否已经被修改。
- **访问位**——表明自该位上次清零以来，是否曾读取或写入过该页。
- **页面大小位**——表明最后一级是 4 KB 页面，还是 4 MB 页面；如果是后者，则 Opteron 只使用 3 个页面级别，而不是 4 个。
- **不执行位**——未出现在 80386 保护机制中，添加这个位是为了防止代码在某些页内执行。

- **页级缓存禁用**——表明是否可以缓存该页。
- **页级直写**——表明该页对数据缓存应用写回还是直写。

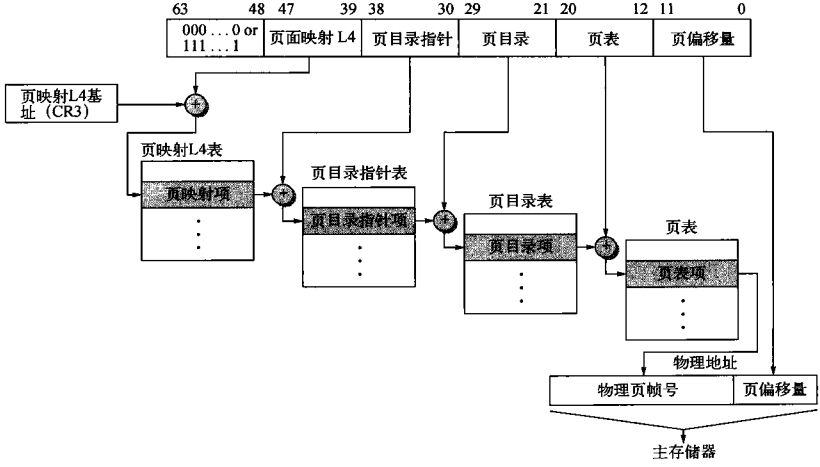


图 B-17　Opteron 虚拟地址的映射。拥有 4 个页表级别的 Opteron 虚拟存储器实现方式支持 40 位的有效物理地址大小。每个页表有 512 项，所以每一级字段的宽度为 9 位。AMD64 体系结构文档允许虚拟地址大小从当前的 48 位增长到 64 位，允许物理地址从当前的 40 位增长到 52 位

由于 Opteron 在 TLB 缺失时通常会经历四级页表，所以有 3 个可能位置来核对保护限制。Opteron 仅服从底级 PTE，检查其他各项只是为了确保设置了有效位。

由于该项目的长度为 8 个字节，每个页表有 512 项，而且 Opteron 拥有大小为 4 KB 的页，所以这些页表的长度恰好为一页。这些四级字段的每一个字段长 9 位，页偏移量为 12 位。这一推导留出 64 – (4 × 9 + 12)=16 位进行符号扩展，以确保地址的规范化。

尽管我们已经解释了合法地址的转换，那什么会防止用户创建非法地址转换并阻止故障发生呢？这些页表本身是受保护的，用户程序不能对其进行写入。因此，用户可以尝试任意虚拟地址，但操作系统通过控制页表项来控制访问哪个物理存储器。为了实现进程之间的存储器共享，在每个地址空间中都设置一个页表项，指向同一个物理存储器页。

Opteron 采用 4 个 TLB 来缩短地址转换时间，两个 TLB 用于指令访问，两个用于数据访问。与多级缓存类似，Opteron 通过两个较大的 L2 TLB 来减少 TLB 缺失：一个用于指令，一个用于数据。表 B-11 介绍了数据 TLB。

B-56

表B-11　Opteron L1和L2指令与数据TLB的存储器层次结构参数

参　数	说　明
块大小	1 PTE（8个字节）
L1命中时间	1个时间周期

（续）

参　数	说　明
L2命中时间	7个时钟周期
L1 TLB大小	指令TLB与数据TLB相同：每个TLB40个PTE，有32个4 KB的页和8个2 MB或4MB的页
L2 TLB大小	指令TLB与数据TLB相同：512个PTE，均为4 KB页
块选择	LRU
写入策略	（不适用）
L1块布置	全相联
L2块布置	四路组相联

B.5.4 小结：32位Intel Pentium与64位AMD Opteron的保护对比

Opteron中的存储器管理是当今大多数桌面或服务器计算机的典型代表，依靠页级地址变换和操作系统的正确操作，为共享计算机的多个进程提供安全性。尽管Intel也提出了一些替代方案，但它还是沿袭了AMD的领先作法，接纳了AMD64体系结构。因此，AMD和Intel都支持80x86的64位扩展；但出于兼容性原因，这两者都支持复杂的分段保护机制。

如果分段保护模型的构建看起来要难于AMD64，那是因为事实的确如此。由于很少有客户使用这种精心设计的保护机制，所以构建这种模型所付出的努力更让工程师体验到挫败感。此外，这种保护模型与UNIX等系统的简单分页保护机制也不一致，这一事实意味着只有专门为这一计算机编写操作系统的人们才会采用它，而这种情况还没有发生过。

B.6 谬论与易犯错误

即使是对存储器层次结构的回顾也存有谬论和易犯错误！

易犯错误 地址空间太小。

DEC和卡内基梅隆大学合作设计了新的PDP-11计算机系列，但仅仅五年之后就发现他们创造的作品中存在一个致命缺陷。IBM在PDP-11诞生六年前发布的一种体系结构在25年之后仍然生机勃勃，而且仅做了非常微小的修改。而曾被批评包含了多余功能的DEC VAX在PDP-11停产之后卖出了数百万件。为什么？ B-57

PDP-11的致使缺陷就在于它的地址大小（16位），可以与IBM 360和VAX的地址大小进行一下对比，IBM 360为24~31位，VAX为32位。由于程序的大小和程序所需要的数据量必须小于$2^{地址大小}$，所以地址大小限制了程序长度。地址大小之所以很难修改，其原因在于它确定了所有与地址相关的最小宽度：PC、寄存器、存储器字和实际地址运算。如果从开始就没有扩展地址的计划，那么成功改变地址大小的机会微乎其微，通常就意味着该计算机系列的终结。Bell和Strecker[1976]这样来讲述这一问题：

> 计算机设计中只有一种错误是很难挽回的，那就是没有足够的地址位用于存储器寻址和存储器管理。几乎所有著名计算机都未曾打破这一“魔咒”，PDP-11也不例外！[P2]

一些曾经取得成功的计算机最终因为地址位不足而“饥渴”致死，下面是此类计算机的一份不完整清单：PDP-8、PDP-10、PDP-11、Intel 8080、Intel 8086、Intel 80186、Intel 80286、

Motorola 6800、AMI 6502、Zilog Z80、CRAY-1、CRAY X-MP。

令人尊敬的 80x86 系列也已经进行了两次扩展，第一次是 1985 年的 Intel 80386 扩展到 32 位，最近一次是和 AMD Opteron 一起扩展到 64 位。

易犯错误　忽视操作系统对存储器层次结构性能的影响。

表 B-12 给出在执行三个大型工作负载时由于操作系统而产生的存储器停顿时间。大约 25% 的停顿时间与操作系统有关，或者消耗在操作系统的缺失中，或者是因为应用程序与操作系统互相干扰而导致的缺失。

表B-12　应用程序与操作系统中的缺失及消耗的时间百分比

工作负载	缺失		时间						
			由于应用程序缺失消耗的时间百分比		由于操作系统缺失直接消耗的时间百分比				
	应用程序中的百分比	操作系统中的百分比	固有的应用程序缺失	操作系统与应用程序冲突	操作系统指令缺失	迁移过程中的数据缺失	块操作中的数据缺失	其他操作系统缺失	操作缺失和应用程序冲突消耗的时间百分比
Pmake	47%	53%	14.1%	4.8%	10.9%	1.0%	6.2%	2.9%	25.8%
Multipgm	53%	47%	21.6%	3.4%	9.2%	4.2%	4.7%	3.4%	24.9%
Oracle	73%	27%	25.7%	10.2%	10.6%	2.6%	0.6%	2.8%	26.8%

* 操作系统使应用程序的执行时间增加了大约 25%。每个处理器有一个 64 KB 指令缓存和一个两级数据缓存，第一级为 64 KB，第二级为 256 KB；所有缓存都与 16 字节块直接映射。这些数据是在 Silicon Graphics POWER 工作站 4D/340 上收集，它是一个由 4 个 33 MHz R3000 处理器组成的多处理器，在 UNIX System V 上运行三个应用程序工作负载——Pamke，并行编译 56 个文件；Multipgm，并行数值程序 MP3D，与 Pmake 和五屏编辑会话同时运行；还有 Oracle，使用 Oracle 数据库运行 TP-1 基准测试的一个受限版本。（数据来自 Torrellas、Gupta 和 Hennessy [1992]。）

易犯错误　依靠操作系统来改变页面大小。

Alpha 体系结构有一个非常精细的计划：通过增大页面大小，甚至达到虚拟地址的大小，借以发展其体系结构。在后来的 Alpha 结构中增大页面大小时，操作系统设计人员拒绝了这项修改，最终通过修改虚拟存储器系统来增大地址空间，页面大小保持 8 KB 不变。

其他计算机的架构师注意到 TLB 缺失率极高，所以向 TLB 中增加了多种较大的页面大小。希望操作系统程序员会将一个对象分配到最大页中，从而保留 TLB 项目。在尝试了 10 来年之后，大多数操作系统仅在精心选择的功能中使用了这些“超级页面”，比如映射显示存储器或其他 I/O 设备，或者为数据库代码使用这种极大页面。

B-58

B.7　结语

要想制造出能够跟上处理器步伐的存储器系统，其难度极大，无论多么先进的计算机，其主存储器的制造原材料都与最廉价的计算机一致，这一事实为上述难度添加了新的注脚。这里能为我们提供帮助的是局域性原理——当前计算机中存储器层次结构的各个级别（从磁盘到 TLB）都证明了它的正确性。

但是，到存储器的相对延迟不断增加，2011 年达到数百个时钟周期，这就意味着，如果程序员和编译器编写人员希望自己的程序能够正常执行，就必须了解缓存和 TLB 的参数。

B.8 历史回顾与参考文献

附录 L.3 节中，我们研究了缓存、虚拟存储器和虚拟机的历史。（这一历史内容包含了本附录和第 3 章。）IBM 在所有这三种技术的历史上都扮演着重要角色。这一节还包含了供扩展阅读的参考文献。

B-59

练习（Amr Zaky设计）

B.1 [10/10/10/15] <B.1>你正在尝试理解局域性原理对于证明缓存存储器应用的正当性有多么重要，于是用一个拥有 L1 数据缓存和主存储器的计算机进行实验（专注于数据访问）。不同访问类型的延迟如下（用 CPU 周期表示）：缓存命中，1 个周期；缓存缺失，105 个周期；禁用缓存时的主存储器访问，100 个周期。

a. [10] <B.1>在运行一个总缺失率为 5%的程序时，存储器平均访问时间为多少（用 CPU 周期表示）？

b. [10] <B.1>运行一个专门设计用于生成完全随机数据访问的程序，这些访问中不存在局域性。为此，使用一个大小为 256 MB 的数组（整个数组都装在主存储器中）。持续访问这一数组的随机元素（使用均匀随机数生成器来生成元素索引）。如果数据缓存大小为 64 KB，存储器平均访问时间为多少？

c. [10] <B.1>如果将(b)部分得到的结果与禁用缓存时的主存储器访问时间对比，则可以说局域性原理在证明缓存存储器使用正当性方面扮演着什么样的角色？

d. [15] <B.1>你观察到一次缓存命中可以得到 99 个周期的收益（1 个周期对 100 个周期），但它会在缺失时造成 5 个周期的损失（105 个周期对 100 个周期）。在一般情况下，我们可以将这两个量表示为 G（收益）和 L（损失）。使用这两个量（G 和 L），给出缓存应用不会产生反作用的最高缺失率。

B.2 [15/15] <B.1>对于本练习，我们假定有 512 字节的缓存，块大小为 64 字节。我们还假定主存储器的大小为 2 KB。我们可以将存储器看作是一个由 64 字节块组成的数组：M0、M1、……、M31。表 B-13 列出了在缓存为全相联时，可以驻存于不同缓存块中的主存储器块。

a. [15] <B.1>如果缓存的组织方式采用直接映射，请给出表中内容。

b. [15] <B.1>如果缓存的组织方式采用四路组相联，重复(a)部分中的工作。

表B-13 可以驻存在缓存块中的存储器块

缓存块	组	路	可以驻存在缓存块中的存储器块
0	0	0	M0, M1, M2, …, M31
1	0	1	M0, M1, M2, …, M31
2	0	2	M0, M1, M2, …, M31
3	0	3	M0, M1, M2, …, M31
4	0	4	M0, M1, M2, …, M31
5	0	5	M0, M1, M2, …, M31
6	0	6	M0, M1, M2, …, M31
7	0	7	M0, M1, M2, …, M31

B.3 [10/10/10/10/15/10/15/20] <B.1>人们希望降低缓存的功耗，这一愿望经常会影响到缓存的组织方式。为此，我们假定缓存在物理上分布到一个数据数组（保存数据）、标志数据（保存标志）

B-60 和替换数组（保存替换策略所需要的信息）。此外，这些数组中的每一个数组都在物理上分布到多个可以各别访问的子数组中（每路一个子数组）；例如，四路组相联最近最少使用（LRU）缓存将拥有 4 个数据子数组、4 个标志子数组和 4 个替换子数组。我们假定在使用 LRU 替换策略时，在每次访问时都会访问一次替换子数组，如果使用先入先出（FIFO）替换策略，会在每次缺失时访问一次。在使用随机替换策略时不需要它。对于一个具体缓存，已经确定对不同数组的访问具有以下功耗权重：

数　组	功耗权重（每一被访问路）
数据数组	20个单位
标志数组	5个单位
其他数组	1个单位

估计以下配置的缓存功耗（以功率单位表示）。我们假定该缓存为四路组相联。这里不考虑主存储器访问功率（尽管它也非常重要）。给出 LRU、FIFO 和随机替换策略下的答案。

a. [10]<B.1>一次缓存读取命中。同时读取所有数组。

b. [10] <B.1>针对缓存读取缺失重复(a)部分。

c. [10] <B.1>假定缓存访问分跨在两个周期内，重复(a)部分。在第一个周期内，访问了所有标志子数组。在第二个周期内，仅访问那些标志匹配的子数组。

d. [10] <B.1>对缓存读取缺失重复(c)部分（第二个周期没有数据数组访问）。

e. [15] <B.1>假定添加了预测待访问缓存路的逻辑，重复(c)部分。在第一个周期时，仅访问预测路的标志子数组。一次路命中（在预测路内的地址匹配）意味着缓存命中。发生路缺失时则在第二个周期内查看所有标志子数组。在路命中时，在第二周期内仅访问一个数据子数组（标志匹配的那个子数组）。假定存在路命中。

B-61

f. [10] <B.1>假定路预测器缺失（选择的路是错误的），重复(e)部分。当其失败时，路预测器另外增加一个周期，在这个周期中访问所有标志子数组。假定一次缓存读取命中。

g. [15] <B.1>假定一次缓存读取缺失，重复(f)部分。

h. [20] <B.1>对于工作负载具有以下统计数字的一般情况，重复(c)、(f)和(g)部分：路预测器缺失率=5%，缓存缺失率=3%。（考虑不同替换策略）。

B.4 [10/10/15/15/15/20] <B.1>我们使用一个具体示例来对比直写缓存与写回缓存的写入带宽需求。我们假定有一个 64 KB 缓存，其行大小为 32 个字节。缓存会在写入缺失时分配一行。如果配置为写回缓存，它会在需要替换时写回整个脏行。我们还假定该缓存通过一个宽度为 64 位（8 个字节）的总线连接到层次结构的下一级。在这一总线上进行 B 字节写入访问的 CPU 周期数为：

$$10+5\left(\left\lceil\frac{B}{8}\right\rceil-1\right)$$

例如，一次 8 字节写入将需要 $10+5\left(\left\lceil\frac{B}{8}\right\rceil-1\right)$ 个周期，而使用同一公式时，12 字节的写入将需要 15 个周期。参考下面的 C 代码段，回答以下问题：

```
#define PORTION 1 … Base = 8*i; for (unsigned int j=base;
j < base+PORTION; j++) //假定 j 已被存储在寄存器中

data[j] = j;
```

a. [10] <B.1>对于一个直写缓存，j 循环的所有迭代中，在向存储器执行写入传输时，一共花费多少个 CPU 周期？

b. [10] <B.1>如果缓存配置为写回缓存，有多少个 CPU 周期花费在写回缓存行上？

c. [15] <B.1>将 PORTION 改为 8，重复(a)部分。

d. [15] <B.1>（在替换缓存行之前），对同一缓存行至少进行多少次数组更新时，才会使写回缓存占优？

e. [15] <B.1>给出这样一种情景：缓存行的所有字都将被写入（不一定使用上述代码），直写缓存需要的总 CPU 周期少于写回缓存。

B.5 [10/10/10/10/] <B.2>你正要采用一个具有以下特征的处理器构建系统：循序执行，运行频率为 1.1 GHz，排除存储器访问在外的 CPI 为 0.7。只有载入和存储指令能从存储器读写数据，载入指令占全部指令的 20%，存储指令占 5%。此计算机的存储器系统包括一个分离的 L1 缓存，它在命中时不会产生任何代价。I 缓存和 D 缓存都是直接映射，分别为 32 KB。I 缓存的缺失率 2%，块大小为 32 字节，D 缓存为直写缓存，缺失率为 5%，块大小为 16 字节。D 缓存上有一个写入缓冲区，消除了绝大多数写入操作的停顿，占总写入操作的 95%。512 KB 写回、统一 L2 缓存的块大小为 64 字节，访问时间为 15ns。它由 128 位数据总线连接到 L1 缓存，运行频率为 266 MHz，每条总线每个时间周期可以传送一个 128 位字。在发往此系统 L2 缓存的所有存储器引用中，其中 80%的引用无须进入主存储器就可以得到满足。另外，在被替换的所有块中，50%为脏块。主存储器的宽度为 128 位，访问延迟为 60ns，在此之后，可以在这个宽 128 位、频率为 133MHz 的主存储器总线上以每个周期传送一个字的速率来传送任意数目的总线字。 B-62

a. [10] <B.2>指令访问的存储器平均访问时间为多少？

b. [10] <B.2>数据读取的存储器平均访问时间为多少？

c. [10] <B.2>数据写入的存储器平均访问时间为多少？

d. [10] <B.2>包括存储器访问在内的整体 CPI 为多少？

B.6 [10/15/15] <B.2>在将缺失率（每次引用的缺失数）转换为每条指令的缺失数时，需要依靠两个因数：每条所提取指令的引用数，所提取指令中实际提交的比例。

a. [10] <B.2>B.1.1 节中每条指令缺失数的公式最初是用三个因数表示的：缺失率、存储器访问和指令数。这些因数中的每一个都代表实际事件。将每条指令的缺失数写为缺失率乘以因数**每条指令的存储器访问数**，会有什么不同？

b. [15] <B.2>推测处理器会提取一些最终不会提交的指令。B.1.1 节中每条指令缺失数的公式是指执行路径上每条指令的缺失数，也就是说，仅包括那些为运行程序而必须实际执行的指令。将 B.1.1 节中每条指令缺失数的公式转换为仅使用缺失率、所提取每条指令的引用数和所提交指令占所提取指令的比例。为什么要依靠这些因数而不是 B.1.1 节公式中的因数？

c. [15] <B.2>(b)部分的转换可能会得出一个错误值：每条所提取指令的引用值不等于任意特定指令的引用数。重写(b)部分的公式，以纠正这一不足。

B.7 [20] <B.1、B.3>如果系统采用直写 L1 缓存，再以写回 L2 缓存（而非主存储器）提供后备支援，则可以简化合并写入缓冲区。解释为什么可以这样做。拥有完整写入缓冲区（而不是你刚刚提议的简单版本）时能否有所帮助？

B.8 [20/20/15/25] <B.3>LRU 替换策略基于以下假定：如果最近访问地址 A1 的频率低于地址 A2，那么未来再次访问 A2 的时机要早于 A1。因此，为 A2 指定了高于 A1 的优先级。试讨论，当一个大于指令缓存的循环连续执行时，这一假定为什么不成立。例如，考虑一个全相联 128 字节指令缓存，其块大小为 4 个字节（每个块可以正好容纳一条指令）。此缓存使用 LRU 替换策略。 B-63

a. [20] <B.3>对于一个拥有大量迭代的 64 字节循环，渐近指令缺失率为多少？

b. [20] <B.3>对于大小为 192 字节和 320 字节的循环，重复(a)部分。

c. [15] <B.3>如果缓存替换策略改为最近使用最多（MRU）（替换最近访问最多的缓存行），以上三种情景（64、192、320 字节的循环）中的哪一种情景将因为这一策略而受益？

d. [25] <B.3>提出执行性能可能优于 LRU 的更多替换策略。

B.9 [20] <B.3>从统计的角度来看，增加缓存的相联度（所有其他常数保持恒定）可以降低缺失率。但也可能存在一些不正常情景：对于特定工作负载，增加缓存相联度反而会使缺失率增大。考

虑同等大小的直接映射与两路组相联缓存的对比。假定组相联缓存使用 LRU 替换策略。为进行简化，假定块大小为一个字。现在构造一组会在两路相联缓存中产生更多缺失的字访问。（提示：集中精力使构造的访问全部指向两路组相联缓存中的单个组，从而使同一种跟踪独占访问直接映射缓存中的两个块。）

B.10 [10/10/15] <B.3>考虑一个由 L1 和 L2 数据缓存组成的两级存储器层次结构。假定两个缓存在写入命中时都使用写回策略，两者的块大小相同。列出在以下事件时采取的操作。

a. [10] <B.3>当缓存组织方式为包含式层次结构时，发生 L1 缓存缺失。

b. [10] <B.3>当缓存组织方式为互斥式层次结构时，发生 L1 缓存缺失。

c. [15] <B.3>在(a)部分和(b)部分中，考虑被逐出行为脏行或洁净行的可能性（需要更新和不需要更新的可能性）。

B.11 [15/20] <B.2、B.3>禁止某些指令进入缓存可以降低冲突缺失。

a. [15] <B.3>画出一个程序层次结构，其中最好禁止其中一部分程序进入指令缓存。（提示：考虑一个程序，其代码块所在的循环嵌套要深于其他块所在的嵌套。）

b. [20] <B.2、B.3>给出一些软件或硬件技术，用于禁止特定块进入指令缓存。

B-64 B.12 [15] <B.4>一个程序运行于拥有四项全相联（微）变换旁视缓冲区（TLB）的计算机上。

虚拟页号	物理页号	有效项
5	30	1
7	1	0
10	10	1
15	25	1

下面是一组由程序访问的虚拟页号。指出每个访问是否会发生 TLB 命中/缺失，如果访问页表，它是发生页命中还是页错误。如果未被访问，则在页表列下放入一个 X。

虚拟页号	物理页号	是否存在
0	3	是
1	7	否
2	6	否
3	5	是
4	14	是
5	30	是
6	26	是
7	11	是
8	13	否
9	18	否
10	10	是
11	56	是
12	110	是
13	33	是
14	12	否
15	25	是

被访问的虚拟页	TLB（命中或缺失）	页表（命中或缺失）
1		
5		
9		
14		
10		
6		
15		
12		
7		
2		

B-65

B.13 [15/15/15/15/] <B.4>一些存储器系统以软件处理 TLB 缺失（将其作为异常），而另外一些则使用硬件来处理 TLB 缺失。

a. [15] <B.4>这两种用于处理 TLB 缺失的方法有哪些折中？

b. [15] <B.4>在软件中进行的 TLB 缺失处理是否总是慢于在硬件中进行的 TLB 缺失处理？请解释原因。

c. [15] <B.4>是否存在一些页表结构，在硬件中难以处理，但在软件中则有可能实现？是否有一些结构难以在软件中处理，但易于用硬件管理？

d. [15] <B.4>为什么浮点程序的 TLB 缺失率通常高于整数程序的缺失数？

B.14 [25/25/25/25/20] <B.4>TLB 应当有多大？TLB 缺失通常非常快（加上异常成本，少于 10 条指令），因此，仅仅为了使 TLB 缺失率降低一点点而使用庞大的 TLB 是不值得的。使用 SimpleScalar 模拟器（www.cs.wisc.edu/~mscalar/simplescalar.html）和一或多个 SPEC95 基准测试，计算以下 TLB 配置的 TLB 缺失率和 TLB 开销（以处理 TLB 缺失所浪费的时间百分比表示）。假定每个 TLB 缺失需要 20 条指令。

a. [25] <B.4>128 项，两路组相联，4KB 至 64KB 页面（均为 2 的幂）。

b. [25] <B.4>256 项，两路组相联，4KB 至 64KB 页面（均为 2 的幂）。

c. [25] <B.4>512 项，两路组相联，4KB 至 64KB 页面（均为 2 的幂）。

d. [25] <B.4>1024 项，两路组相联，4KB 至 64KB 页面（均为 2 的幂）。

e. [20] <B.4>多任务环境对 TLB 缺失率和开销的影响如何？上下文切换频率对开销有什么样的影响？

B.15 [15/20/20] <B.5>利用类似于 Hewlett-Packard Precision Architecture (HP/PA)中使用的保护机制，有可能提供一种比 Intel Pentium 体系结构更灵活的保护方式。在这种机制中，每个页表项包含一个“保护 ID”（键），还有对该页的访问权限。在每次引用中，CPU 将页表项中的保护 ID 与存储在 4 个保护 ID 寄存器中的保护 ID 逐一对比（对这些寄存器的访问要求 CPU 处于管理员模式）。如果寄存器内容与页表项中保护 ID 都不匹配，或者如果该访问不是受权访问（比如，写入只读页），则会生成异常。

a. [15] <B.5>进程如何在任意给定时刻都拥有 4 个以上的有效保护 ID？换句话说，假定进程希望同时拥有 10 个保护 ID。请给出一种可以实现这一愿望的机制（可能需要来自软件的帮助）。 B-66

b. [20] <B.5>请解释：如何利用这一模型，用一些不能相互改写的较小段代码（微内核）组合构造出操作系统。与整体操作系统（在这种操作系统中，操作系统的任意代码都可以写入任意存储器位置）相比，这种操作系统可能拥有哪些优势？

c. [20] <B.5>简单地改变这一系统的设计，就能使每个页表项有两个保护 ID，一个用于读取访问，一个用于写入或执行访问（如果可写位或可执行位都未被置位，则不使用这一字段）。为读取与写入功能使用不同保护 ID 会有什么好处？（提示：这样能否简化进程之间数据与代码的共享？） B-67

C

流水线：基础与中级概念

这个问题可能需要吸三袋烟的时间才能想明白。

——亚瑟·柯南·道尔，
《福尔摩斯探案集》

C.1 引言

本书的许多读者已经在其他教材（比如我们另一本更基础的教材 *Computer Organization and Design*）或其他课程中了解了流水线的基础知识。因为第 3 章主要以这一材料为基础，所以读者在学习该章之前，一定要熟悉本附录中讨论的概念。在阅读第 2 章时，快速回顾这一材料也会有所帮助。

我们在本附录中首先介绍流水线的基础知识，包括数据路径含义的讨论、冒险的介绍、流水线性能的研究。这一节介绍基本的五级 RISC，它是本附录其余部分的基础。C.2 节介绍冒险问题、它们为什么会导致性能问题、应当如何应对。C.3 节讨论如何实际实现这个简单的五级流水线，重点是控制和如何应对冒险。

C.4 节讨论流水线和指令集设计各个方面之间的相互关系，讨论了有关异常的重要主题以及它们与流水线的相互交互。如果读者不熟悉精确与非精确中断的概念以及发生异常后的恢复过程，会发现这一材料非常有用，因为它们是理解第 3 章中更高级方法的关键。

C.5 节讨论如何扩展五级流水线，以处理运行时间更长的浮点指令。C.6 节在一个案例研究中将这些概念结合在一起，这个案例研究的对象是一个深度流水线处理器——MIPS R4000/4400，它既包括八级整数流水线，又包括浮点流水线。

C.7 节介绍了动态调度的概念，以及记分板在动态调度实现中的用法。它是作为交叉问题介绍的，第 3 章的重点是动态调度方法，而这一部分内容可以充当第 3 章核心概念的引言。C.7 节还简单探讨了第 3 章中介绍的更复杂 Tomasulo 算法。尽管不用引入记分板也能介绍和理解 Tomasulo 算法，但记分卡方法更简单、更易于理解。

C.1.1 什么是流水线

流水线是一种将多条指令重叠执行的实现技术。一条指令的执行需要多个操作，流水线技术充分利用了这些操作之间的并行性。今天，流水线是用于加快 CPU 速度的关键实现技术。

流水线就像是一条装配线。在汽车装配线上存在许多步骤，每一步骤负责汽车生产中的某一项任务。每个步骤的执行与其他步骤是并行的，当然，装配的是不同汽车。在计算机流水线中，流水线中的每个步骤完成指令的一部分。就像装配线一样，不同步骤并行完成不同指令的不同部分。这些步骤中的每一步都称为**流水级**或**流水段**。流水级前后相连形成流水线——指令在一端进入，通过这些流水级，在另一段退出，就像汽车在装配线中经历的过程一样。 C-2

在汽车装配线中，**吞吐量**定义为每小时生产的汽车数，由完整汽车退出装配线的频率决定。与此类似，指令流水线的吞吐量由指令退出流水线的频率决定。由于流水线级是连在一起的，所以所有流水级都必须做好同时工作的准备，就像我们在装配线中的要求一样。将一条指令在流水线中下移一步所需的时间为**处理器周期**。由于所有各级同时进行，所以处理器周期的长度由最缓慢流水线级所需时间决定，就像汽车装配线上的最长步骤决定了汽车沿生产线前进的时间。在计算机中，这一处理器周期通常为 1 个时钟周期（有时为 2 个，但要少见得多）。

流水线设计者的目标是平衡每条流水线的长度，就像装配线的设计者尝试平衡装配过程中每个步骤的时间一样。如果各级达到完美平衡，那么每条指定在流水线处理器中的时间（假定为理想条件）等于：

$$\frac{\text{非流水线机器上每条指令的时间}}{\text{流水级的数目}}$$

在这些条件下，因为实现流水线而得到的加速比等于流水级的数目，就像一个 n 级装配线在理想情况下可以将汽车生产速度提高至 n 倍一样。但一般情况下，这些流水线之间不会达到完美平衡；此外，流水线还会产生一些开销。因此，在流水线处理器上，处理每条指令的时间不会等于其最低可能值，但可以非常接近。

流水线可以缩短每条指令的平均执行时间。根据我们认证的基准，这一缩短量可以计作每条指令时钟周期数（CPI）的下降、时钟周期时间的缩短，或者这两者的组合。如果在开始时，处理器需要多个时钟周期来处理一条指令，那我们通常将流水线的作用看作是降低了 CPI。这是我们采取的主视角。如果在开始时，处理器需要一个（长）时钟周期来处理一条指令，那就认为流水线缩短了时钟周期时间。

流水线技术开发了串行指令流中各指令之间的并行度。它与某些加速技术不同（见第 4 章），其真正的好处在于它对程序员是透明的。在这一附录中，我们将首先使用一个经典的五级流水线来介绍流水线的概念；其他章节研究了更复杂流水线技术在现代处理器的应用。在更深入地讨论流水线及其在处理器的应用之前，我们需要一个简单的指令集，下面将对此进行介绍。

C-3

C.1.2 RISC指令集基础知识

在本书中，我们一直使用 RISC（精简指令集计算机）体系结构或载入-存储体系结构来说明基本概念，当然，本书介绍的几乎所有思想都适用于其他处理器。本节介绍典型 RISC 体系结构的核心。在本附录及整本书中，我们的默认 RISC 体系结构为 MIPS。在许多地方，其概念非常类似，可以应用于任意 RISC。RISC 体系结构有几个关键属性，这些属性大大简化了其实现方式。

- 所有数据操作都是对寄存器中数据的操作，通常会改变整个寄存器（每个寄存器为 32 位或 64 位）。
- 只有载入和存储操作会影响到存储器，它们分别将数据从存储器移到寄存器或从寄存器移到存储器。通常存在一些可以载入或存储低于一个完整寄存器的内容（例如，一个字节、16 位或 32 位）。
- 指令格式的数量很少，所有指令通常都是同一种大小。

这些简单属性极大地简化了流水线的实现，这也是如此设计这些指令集的原因。

为与本书其他部分保持一致，我们使用 MIPS64，它是 MIPS 指令集的 64 位版本。这些扩展 64 位指令通常在助记符的开头或末尾加上字母 D 加以标识。例如，DADD 是加法指令的 64 位版本，而 LD 则是载入指令的 64 位版本。

与其他 RISC 体系结构类似，MIPS 指令集提供了 32 个寄存器，不过寄存器 0 的值总是 0。大多数 RISC 体系结构，比如 MIPS，通常有以下三类指令（更多详细信息请参见附录 A）。

(1) **ALU 指令**——这些指令取得两个寄存器或者一个寄存器与一个符号扩展立即数（称为 **ALU 立即数指令**，它们在 MIPS 中有一个 16 位偏移量），对它们进行操作，然后将结果存储到第三个寄存器中。典型操作包括加（DADD）、减（DSUB）和逻辑运算（例如 AND 或 OR），后者不区分 32 位和 64 位版本。这些指令的立即数版本使用相同助记符，但添加了后缀 I。在 MIPS 中，包括算术运算的有符号形式和无符号形式；无符号形式的末尾有一个字母 U（例如，DADDU、DSUBU、DADDIU），不会生成溢出异常（因此在 32 位和 64 位模式中是一样的）。

(2) **载入和存储指令**——这些指令获取一个寄存器源（称为**基址寄存器**）和一个立即数字段（在 MIPS 中为 16 位，称为**偏移量**）作为操作数。基址寄存器的内容与符号扩展偏移量之和（称为**有效地址**）用作存储器地址。对于载入指令，使用第二个寄存器操作数，用于存放从存储器载入的数据。对于存储指令，第二个寄存器操作数是要存入存储器的数据源。载入字（LD）和存储字（SD）等指令载入或存储整个 64 位寄存器内容。 C-4

(3) **分支与跳转**——分支是指控制的条件转移。在 RISC 体系结构中，通常有两种方式来指定分支条件：采用一组条件位（有时称为**条件码**），或者通过寄存器对之间、寄存器与零之间的有限对比来设定。MIPS 采用后者。对于本附录，我们仅考虑两个寄存器是否相等。在所有 RISC 体系结构中，分支目的地都是通过将符号扩展偏移量（MIPS 中为 16 位）加到当前程序计数器中获得的。在许多 RISC 体系结构中都提供了无条件跳转，但本附录中不会介绍跳转。

C.1.3 RISC指令集的简单实现

为了理解如何以流水线形式来实现 RISC 指令集，需要理解在没有流水线时它是如何实现的。这一节给出了一种简单实现，每一条指令最多需要 5 个时钟周期。我们将这一基本实现扩展到流水线版本，从而大幅降低 CPI。在所有不采用流水线的实现方式中，我们给出的方式并非最经济或性能最高的。它的设计只是可以很自然地引向流水线实现。实现此指令集需要引入几个不属于该体系结构的临时寄存器，引入它们是为了简化流水线。我们的实现将仅关注 RISC 体系结构中整数操作部分的流水线，这部分操作包括载入-存储字、分支和整数 ALU 操作。

这个 RISC 子集中的每条指令都可以在最多 5 个时钟周期内实现。这 5 个时钟周期如下所述。

(1) **指令提取周期**（IF）

将程序计数器（PC）发送到存储器，从存储器提取当前指令。向程序计数器加 4（因为每条指令的长度为 4 个字节），将程序计数器更新到下一个连续程序计数器。

(2) **指令译码/寄存器提取周期**（ID）

对指令进行译码，并从寄存器堆中读取与寄存器源说明符相对应的寄存器。在读取寄存器时对其进行相等测试，以确定是否为分支。必要时，对指令的偏移量字段进行符号扩展。符号扩展后的偏移量添加到所实现的程序计数器上，计算出可能的分支目标地址。在较为积极的实现方式中（我们稍后对此进行研究），如果这一条件判断的结果为真，则可以将分支目标地址存储到程序计数器中，以在这一级的末尾完成分支。

指令译码与寄存器的读取是并行执行的，这可能是因为在 RISC 体系结构中，寄存器说明符位于固定位置。这一技术称为**固定字段译码**。注意，我们可能会读取一个不会使用的寄存器，这样做没有什么好处，但也不会损及性能。（读取非必需寄存器的确会浪费能量，但看重功率因素的设计可以避免这一问题。）由于一个指令的立即数部分也位于同一位置，所以在需要符号扩展立即数时，也是在这一周期计算的。 C-5

(3) **执行/有效地址周期**（EX）

ALU 对上一周期准备的操作数进行操作，根据指令类型执行三条指令之一。

- 存储器引用——ALU 将基址寄存器和偏移量加到一起，形成有效地址。
- 寄存器-寄存器 ALU 指令——ALU 对读自寄存器堆的值执行由 ALU 操作码指定的操作。
- 寄存器-立即数 ALU 指令——ALU 对读自寄存器堆的第一个值和符号扩展立即数执行由 ALU 操作码指定的操作。

在载入-存储体系结构中，有效地址与执行周期可以合并到一个时钟周期中，这是因为没有指令需要计算数据地址并对数据执行操作。

(4) **存储器访问**（MEM）

如果该指令是一条载入指令，则使用上一周期计算的有效地址从存储器中读取数据。如果是一条存储指令，则使用有效地址将从寄存器堆的第二个寄存器读取的数据写入存储器。

(5) **写回周期**（WB）

- 寄存器-寄存器 ALU 指令或载入指令。

将结果写入寄存器堆，无论是来自寄存器系统（对于载入指令），还是来自 ALU（对于 ALU 指令）。

在这一实现中，分支指令需要 2 个周期，存储指令需要 4 个周期，所有其他指令需要 5 个周期。假定分支频率为 12%，存储频率为 10%，对于这一典型指令分布，总 CPI 为 4.54。但是，无论是在获取最佳性能方面，还是在给定性能级别的情况尽量减少使用硬件方面，这一实施方式都不是最优的；我们将对这一设计的改进留给读者作为练习，这里仅实现这一版本的流水线。

C.1.4　RISC处理器的经典五级流水线

C-6　我们几乎不需要进行什么改变就能实现上述执行过程的流水化，只要在每个时钟周期开始一条新的指令就行。（想想我们为什么选择这种设计？）上一节的每个时钟周期都变成一个**流水线级**——流水线中的一个周期。这样会得到表 C-1 所示的执行模式，这是绘制流水线结构的典型方式。尽管每条指令需要 5 个周期才能完成，但在每个时钟周期内，硬件都会启动一条新的命令，执行 5 个不同指令的某一部分。

表C-1　简单RISC流水线

	时钟编号								
指令编号	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

* 在每个时钟周期，提取另一条指令，并开始它的五周期执行过程。如果在每个时钟周期都启动一条指令，其性能最多可达到非流水化处理器的 5 倍。流水线中各个阶段的名称与非流水线实现方式中各个周期的名称相同：IF=指令提取、ID=指令译码、EX=执行、MEM=存储器访问、WB=写回。

读者可能觉得很难相信流水线就是这样简单，它的确也并非如此简单。在本节和后续各节中，我们将通过处理流水化带来的一些问题，使我们的 RISC 流水线显得更“真实”。

首先，我们必须确定在处理器的每个时钟周期都会发生什么，确保不会在同一时钟周期内对相同数据路径源执行两个不同操作。例如，不能要求同一个 ALU 同时计算有效地址和执行减法操作。因此，我们必须确保流水线中的指令重叠不会导致这种冲突。幸运的是，RISC 指令集比较简单，使资源评估变得相对容易。图 C-1 以流水线形式绘制了一个 RISC 数据路径的简化版本。可以看到，主要功能单元是在不同周期使用的，因此多条指令的执行重叠不会引入多少冲突。从以下三点可以看出这一事实。

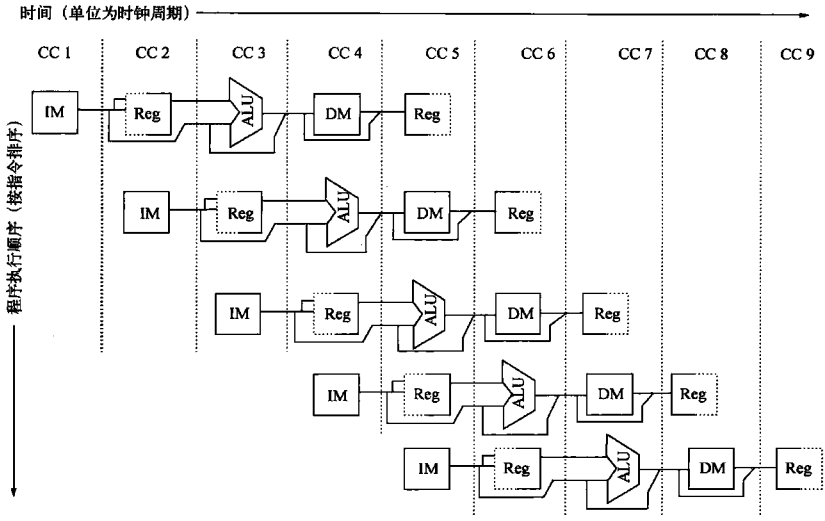


图 C-1 流水线可以看作一系列随时间移动的数据路径。本图给出了数据路径不同部分之间的重叠，时钟周期 5（CC 5）表示稳定状态。由于寄存器用作 ID 级中的一个源和 WB 级中的目的地，所以它出现两次。我们表示它在该级的一个部分进行读取，在另一部分进行写入，分别用左右两侧的实线或虚线来表示。缩写 IM 表示指令存储器、DM 表示数据存储器、CC 表示时钟周期

第一，我们使用分离的指令存储器和数据存储器，我们通常用分离的指令和数据缓存来实现它们（在第 2 章讨论）。在使用存储器时，在指令提取和数据存储器访问之间可能会发生冲突，而使用分离缓存则可以消除这种冲突。注意，如果我们的流水线处理器的时钟周期等于非流水线版本的时钟周期，则存储器系统必须提供 5 倍的带宽。这一需求的增加是提高性能的一种成本。

第二，在两个阶段都使用了寄存器堆：一个是在 ID 中进行读取，一个是在 WB 中进行写入。这些用法是不同的，所以我们干脆在两个地方画出了寄存器堆。因此，每个时钟周期需要 C-7
执行两次读取和一次写入。为了处理对相同寄存器的多次读取和一次写入（稍后将会明了，还有其他原因），我们在时钟周期的前半部分写寄存器，在后半部分读寄存器。

第三，图 C-1 没有涉及程序计数器。为了在每个时钟周期都启动一条新指令，我们必须在每个时钟周期使程序计数器递增并存储它，这必须在 IF 阶段完成，以便为下一条指令做好准备。此外，还必须拥有一个加法器，在 ID 期间计算潜在的分支目标。另外一个问题是分支在 ID 级改变程序计数器。这样会导致一个问题，现在暂时忽略这一问题，但稍后就会对之进行处理。

尽管确保流水线中的指令不会试图在相同时间使用硬件资源是至关重要的，我们还必须确保不同流水级中的指令不会相互干扰。这种分离是通过在连续流水级之间引入**流水线寄存器**来完成的，这样会在时钟周期的末尾，将一个给定流水级得出的所有结果都存储到寄存器中，在下一个时钟周期用作下一级的输入。图 C-2 给出了画有这些流水线寄存器的流水线。 C-8

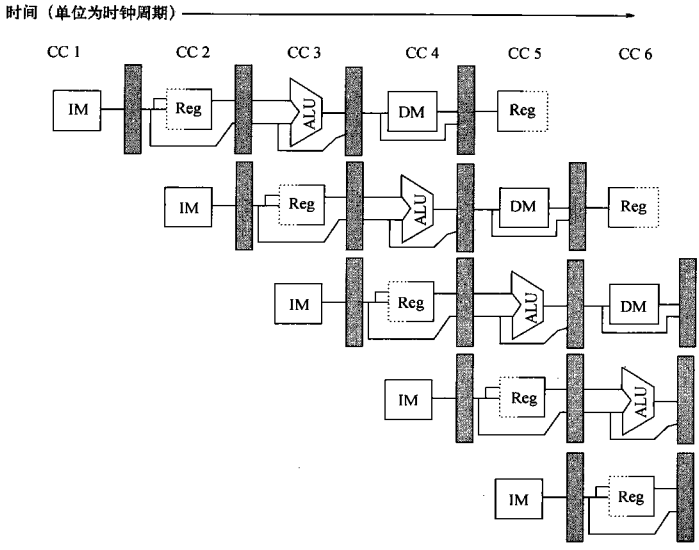


图 C-2 此流水线显示了连续流水级之间的流水线寄存器。注意，寄存器防止流水线相邻级中两条不同指令之间的干扰。在将一条给定指令的数据从一级带至另一级的过程中，寄存器也扮演着重要角色。寄存器的边沿触发特性（也就是说，取值在时钟沿即时改变）是非常关键的。否则，来自一条指令的数据可能会干扰另一指令的执行！

尽管许多图形都为了简便而省略了这些寄存器，但它们是流水线正常操作所必需的。当然，即使在一些没有采用流水化的多周期数据路径中也需要类似寄存器（因为只有寄存器中的值能够在跨过时钟边界之后仍然得以保存）。在流水化处理器中，如果要将中间结果从一级传送到另一级，而源位置与目标位置可能并非直接相邻，流水线寄存器也会在这种传送过程种扮演关键角色。例如，要在存储指令中存储的寄存器值是在 ID 期间读取的，但要等到 MEM 才会真正用到；它在 MEM 级中通过两个流水线寄存器传送给数据存储器。与此类似，ALU 指令的结果是在 EX 期间计算的，但要等到 WB 才会实际存储；它通过两个流水线寄存器才到达那里。有时对流水线寄存器进行命名是有用的，根据这些寄存器所连接的流水级对其进行命名，所以这些寄存器称为 IF/ID、ID/EX、EX/MEM 和 MEM/WB。

C-9

C.1.5 流水化的基本性能问题

流水化提高了 CPU 指令吞吐量（单位时间内完成的指令数），但不会缩短单条指令的执行时间。事实上，由于流水线控制会产生开销，它通常还会稍微延长每条指令的执行时间。尽管单条指令的运行速度并没有加快，指令吞吐量的增长意味着程序可以更快速地运行，总执行时间缩短。

由于单条指令的执行时间并没有缩短，所以在下一节将会看到，这一事实限制了流水线的实际深度。除了因为流水线延迟产生的局限之外，流水级之间的失衡和流化化开销也会造成限制。流水级之间的不平衡会降低性能，这是因为时钟的运行速度不可能快于最缓慢的流水级。

流水线开销包含流水线寄存器延迟和时钟偏差。流水线寄存器增加了建立时间，也就是在发出触发写操作的时钟信号之前，寄存器输入必须保持稳定的时间，而且时钟周期的传播也会产生延迟。时钟偏差是时钟到达任意两个寄存器时刻之间的最大延迟，时钟周期的下限也受此因素的影响。如果时钟周期小于时钟偏差与延迟开销之和，那时钟周期中就没有留给有用工作的时间了，所以再增加流水线也就没用了。有兴趣的读者可以阅读 Kunkel 和 Smith 等人的作品[1986]。我们在第 3 章曾经看到，这一开销影响了 Pentium 4 相对于 Pentium III 的性能改进。

例题 考虑上一节的非流水化处理器。假定其时钟周期为 1 ns，ALU 运算和分支需要 4 个周期，存储器操作需要 5 个周期。假定这些操作的相对频率分别为 40%、20%和 40%。假设由于时钟偏差和建立时间的原因，对处理器实现流水化使时钟增加了 0.2 ns 的开销。忽略所有延迟影响，通过流水线获得的指令执行速率加速比为多少？

解答 在非流水化处理器上，指令平均执行时间为：

$$\begin{aligned}\text{指令平均执行时间} &= \text{时钟周期} \times \text{平均 CPI}\\ &= 1\text{ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5]\\ &= 1\text{ns} \times 4.4\\ &= 4.4\text{ns}\end{aligned}$$

C-10

在流水线实现方式中，时钟的运行速度必须等于最慢流水级的速度加上开销时间，也就是 1+0.2=1.2 ns；这就是指令平均执行时间。因此，通过流水化获得的加速比为：

$$\begin{aligned}\text{流水化加速比} &= \frac{\text{非流水化指令平均执行时间}}{\text{流水化指令平均执行时间}}\\ &= \frac{4.4\text{ns}}{1.2\text{ns}} = 3.7\text{倍}\end{aligned}$$

0.2 ns 的开销基本上确定了流水化的效能限度。如果此开销不受时钟周期变化的影响，那么从 Amdahl 定律可知，这一开销限制了加速比。

如果流水线中每条指令独立于所有其他指令，那这种简单的 RISC 流水线对于整数指令可以正常运行。实际上，流水线中的指令可能是相互依赖的；这是下一节的主题。

C.2 流水化的主要阻碍——流水线冒险

有一些被称为冒险的情景，会阻止指令流中的下一条指令在其自己的指定时钟周期内执行。冒险降低了流水化所能获得的理想加速比。共有以下三类冒险。

(1) **结构冒险**，在重叠执行模式下，如果硬件无法同时支持指令的所有可能组合方式，就会出现资源冲突，从而导致结构冒险。

(2) **数据冒险**，根据流水线中的指令重叠，指令之间存在先后顺序，如果一条指令取决于先前指令的结果，就可能导致数据冒险。

(3) **控制冒险**，分支指令及其他改变程序计数器的指令实现流水化时可能导致控制冒险。

流水线中的冒险会使流水线停顿。为了避免冒险，经常要求在流水线中的一些指令延迟时，其他一些指令能够继续执行。对于本附录中讨论的流水线，当一条指令被停顿时，在停顿指令之后发射的所有指令也被停顿（这些指令在流水线中的位置不会远于停顿指令）。而在停顿指令

之前发射的指令必须继续执行（它们在流水线中的位置要更远一些），否则就永远不会清除冒险情况。结果，在停顿期间不会提取新的指令。在本节将会看到几个有关流水线如何停顿操作的示例——别担心，它们并没有听起来那么复杂！

C-11

C.2.1 带有停顿的流水线性能

停顿会导致流水线性能下降，低于理想性能。现在让我们看一个简单的公式，求解流水化的实际加速比，首先从上一节的公式开始：

$$\text{流水化加速比}=\frac{\text{非流水化指令平均执行时间}}{\text{流水化指令平均执行时间}}=\frac{\text{非流水化CPI}\times\text{非流水化时钟周期}}{\text{流水化CPI}\times\text{流水化时钟周期}}=\frac{\text{非流水化CPI}}{\text{流水化CPI}}\times\frac{\text{非流水化时钟周期}}{\text{流水化时钟周期}}$$

流水化可以看作 CPI 或时钟周期时间的降低。由于传统上使用 CPI 来比较流水线，所以让我们从这里开始。流水化处理器的理想 CPI 几乎总等于 1。因此，可以计算流水化 CPI 为：

$$\begin{aligned}\text{流水化 CPI}&=\text{理想 CPI}+\text{每条指令的流水线停顿时钟周期}\\&=1+\text{每条指令的流水线停顿时钟周期}\end{aligned}$$

如果忽略流水化的周期时间开销，并假定流水级之间达到完美平衡，则两个处理器的周期时间相等，得到：

$$\text{加速比}=\frac{\text{非流水化CPI}}{1+\text{每条指令的流水线停顿周期}}$$

一种简单而重要的情景是所有指令的周期数都相同，必然等于流水级数目（也称为**流水线深度**）。在这种情况下，非流水化 CPI 等于流水线的深度，得到：

$$\text{加速比}=\frac{\text{流水深度}}{1+\text{每条指令的流水线停顿周期}}$$

如果没有流水线停顿，由此公式可以得到一个很直观的结果：流水化可以使性能提高的倍数等于流水线深度。

或者，如果将流水化看作时钟周期时间的改善，那可以假定非流水化处理器的 CPI 以及流水化处理器的 CPI 为 1。于是得到：

$$\text{流水化加速比}=\frac{\text{非流水化CPI}}{\text{流水化CPI}}\times\frac{\text{非流水化时钟周期}}{\text{流水化时钟周期}}=\frac{1}{1+\text{每条指令的流水线停顿周期}}\times\frac{\text{非流水化时钟周期}}{\text{流水化时钟周期}}$$

C-12

当流水级达到完美平衡，而且没有开销时，流水化处理器的时钟周期小于非流水化处理器的时钟周期，缩小因子等于流水线深度：

$$\text{流水化时钟周期}=\frac{\text{非流水化时钟周期}}{\text{流水线深度}}$$

$$\text{流水线深度}=\frac{\text{非流水化时钟周期}}{\text{流水化时钟周期}}$$

从而得到以下公式：

$$流水化加速比=\frac{1}{1+每条指令的流水线停顿周期}\times\frac{非流水化时钟周期}{流水化时钟周期}$$

$$=\frac{1}{1+每条指令的流水线停顿周期}\times流水线深度$$

因此，如果没有停顿，则加速比等于流水级数，与我们对理想情况的直观感受一致。

C.2.2 结构冒险

当处理器以流水线方式工作时，指令的重叠执行需要实现功能单元的流水化和资源的复制，以允许在流水线中出现所有可能的指令组合。如果由于资源冲突而不能容许某些指令组合，就说该处理器存在**结构冒险**。

结构冒险最常见于某功能单元未能完全流水化的情况。这时，一系列使用该非流水化单元的指令不能以每时钟周期执行一条指令的速度前进。结构冒险另一常见出现方式是某一资源的复制不足以执行流水线中的所有指令组合。例如，处理器可能仅有一个寄存器堆写端口，但在特定情况下，流水线可能希望在一个时钟周期内执行两个写操作。这就会生成结构冒险。

当指令序列遇到这种冒险时，流水线将会使这些指令中的一个停顿，直到所需单元可用为止。这种停顿会增大 CPI 值，不再是其通常的理想值 1。

一些流水化处理器为数据和指令共享单存储器流水线。结果，当指令中包含数据存储器引用时，它会与后面指令的指令引用冲突，如图 C-3 所示。为避免这一冒险，在发生数据存储器访问时，我们使流水线停顿一个时钟周期。停顿通常被称为**流水线气泡**，或就称为**气泡**，因为它会飘浮穿过流水线，占据空间却不执行有用工作。在讨论数据冒险时，将会看到另一种停顿类型。

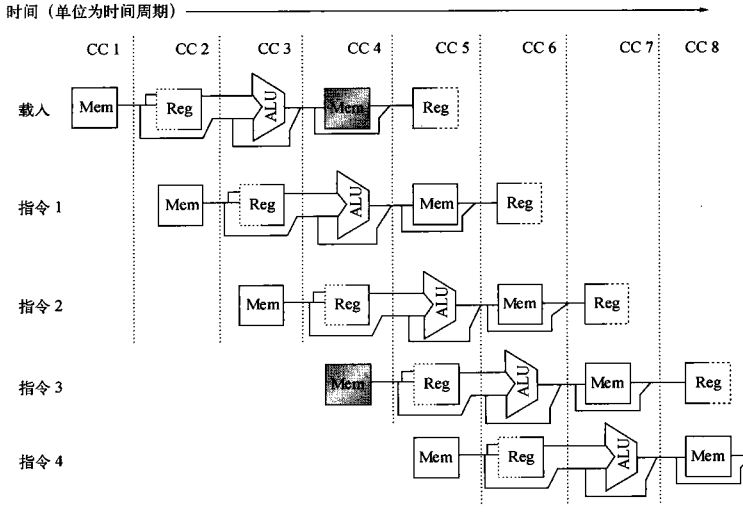


图 C-3 只要发生存储器引用，仅有一个存储器端口的处理器就会生成一次冲突。在这个示例中，载入指令因为数据访问而使用存储器的同时，指令 3 希望从存储器中提取一条指令

设计者经常使用仅含有流水级名称的简图来表示停顿行为，如表 C-2 所示。在表 C-2 中展示停顿时，指出那些没有执行操作的周期，只是将指令 3 向右移动（使其执行过程的开始与结束都推后 1 个周期）。当流水线气泡穿过流水线时，其效果就是占据了该指令时隙的资源。

C-13

表C-2　流水线因为结构冒险而停顿——用一个存储器端口进行载入

指令	时钟周期编号									
	1	2	3	4	5	6	7	8	9	10
载入指令	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				停顿	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

* 如表中所示，载入指令实际强占了指令提取周期，导致流水线停顿——在 4 号时钟周期没有启动指令（它通常会启动指令 i+3）。因为被提取的指令停顿，所以流水线中位于停顿指令之前的所有其他指令都可以正常进行。停顿周期将继续穿过流水线，所以在 8 号时钟周期中没有完成指令。有时在绘制这些流水线表时，让流水线占据整个水平行，指令 3 被移到下一行；无论采用哪种画法，效果都是一样的，因为指令 i+3 直到 5 号周期才开始执行。因为以上形式占据的空间较少，所以我们采用了这一形式。注意，本图假定指令 i+1 和 i+2 不是存储器引用。

例题　让我们看看载入结构冒险的成本有多高。假定数据引用占总体的 40%，流水化处理器的理想 CPI 为 1（忽略结构冒险）。假定与没有冒险的处理器相比，有结构冒险处理器的时钟频率为其 1.05 倍。不考虑所有其他性能损失，有结构冒险和无结构冒险相比，哪种流水线更快？快多少？

解答　有几种方法可以求解这一问题。最简单的一种可能就是计算两种处理器的平均指令时间：

$$\text{平均指令时间} = \text{CPI} \times \text{时钟周期时间}$$

C-14

由于没有停顿，所以理想处理器的平均指令时间就是时钟周期时间$_{\text{理想}}$。有结构冒险处理器的平均指令时间为：

$$\begin{aligned}\text{平均指令时间} &= \text{CPI} \times \text{时钟周期时间} \\ &= (1+0.4\times 1)\times \frac{\text{时钟周期时间}_{\text{理想}}}{1.05} \\ &= 1.3\times \text{时钟周期时间}_{\text{理想}}\end{aligned}$$

显然，没有结构冒险的处理器更快一些；根据平均指令时间的比值，我们可以得出结论，无冒险处理器的速度快 1.3 倍。

为避免出现这种结构冒险，设计师可以为指令提供独立的存储器访问，既可以将缓存分为独立的指令缓存和数据缓存，也可以使用一组缓冲区来保存指令，这种缓冲区通常称为指令缓冲区。第 5 章讨论了分离缓存和指令缓冲区的思想。

如果其他因素相同，无结构冒险处理器的 CPI 总是更低一些。那设计师为什么还会允许结构冒险呢？其主要目的是为了降低单元成本，因为要实现所有功能单元的流水化，或者复制它

们，成本都太高昂了。例如，（为了防止上面示例中的结构冒险，）那些在每个周期内支持指令与数据缓存访问的处理器需要有 2 倍的总存储器带宽，通常要求管脚处也可以承受较高的带宽。与此类似，要完全实现浮点（FP）乘法器需要耗用大量门电路。如果结构冒险很罕见，那就不值得花费这么大的代价来避免它。 C-15

C.2.3 数据冒险

流水化的主要效果是通过重叠指令的执行过程来改变它们的相对执行时间。这种重叠引入了数据与控制冒险。当流水线改变对操作数的读写访问顺序，使该顺序不同于在非流水化处理器上依次执行指令时的顺序，这时可能发生数据冒险。考虑以下指令的流水化执行：

```
DADD    R1,R2,R3
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
XOR     R10,R1,R11
```

DADD 之后的所有指令都用到了 DADD 指令的结果。如图 C-4 所示，DADD 指令在 WB 流水级写入 R1 的值，但 DSUB 指令在其 ID 级中读取这个值。这一问题称为**数据冒险**。除非提前防范这种问题，否则 DSUB 指令将会读取错误值并试图使用它。事实上，DSUB 指令使用的值甚至是不确定的：我们可能假定 DSUB 使用的 R1 值总是由 DADD 之前的指令赋值，认为这种假定是合乎逻辑的，但事实并非总是如此。如果在 DADD 和 DSUB 指令之间发生中断，DADD 的 WB 级将结束，而该点的 R1 值将是 DADD 的结果。这种不可预测的行为显然是不可接受的。

AND 指令也会受到这种冒险的影响。从图 C-4 中可以看出，在 5 号时钟周期之前，R1 的写入操作是不会完成的。因此，在 4 号时钟周期读取寄存器的 AND 指令会得到错误结果。

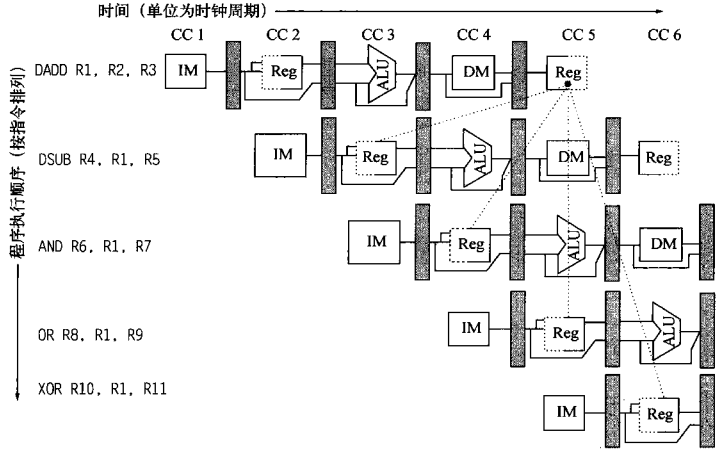


图 C-4 在后面三条指令中使用 DADD 指令的结果时，由于要等到这些指令读取寄存器之后才会向其中写入，所以会导致冒险

XOR 指令可以正确执行，因为它的寄存器读取是在 6 号时钟周期进行的，这时寄存器写入已经完成。OR 指令的执行也不会招致冒险，因为我们在该时钟周期的后半部分执行寄存器堆读取，而写入是在前半部分执行的。

下一小节将讨论一种技术，用于消除涉及 DSUB 和 AND 指令的冒险停顿。

1. 利用转发技术将数据冒险停顿减至最少

图 C-4 中提出的问题可以使用一种称为转发（forwarding）的简单硬件技术来解决（这一技术也称为旁路，有时也称为短路）。转发技术的关键是认识到 DSUB 要等到 DADD 实际生成结果之后才会真正用到它。DADD 将此结果放在流水线寄存器中，如果可以把它从这里转移到 DSUB 需要的地方，那就可以避免出现停顿。根据这一观察结果，转发的工作方式如下所述。

C-16

(1) 来自 EX/MEM 和 MEM/WB 流水线寄存器的 ALU 结果总是被反馈回 ALU 的输入端。

(2) 如果转发硬件检测到前一个 ALU 操作已经对当前 ALU 操作的源寄存器进行了写入操作，则控制逻辑选择转发结果作为 ALU 输入，而不是选择从寄存器堆中读取的值。

注意，采用转发技术后，如果 DSUB 停顿，则 DADD 将会完成，不会触发旁路。当两条指令之间发生中断时，这一关系同样成立。

如图 C-4 中的示例所示，我们需要转发的结果可能不只来自前一条指令，还可能来自提前两个周期启动的指令。图 C-5 显示了带有旁通路径的示例，它重点突出了寄存器读取与写入的时机。这一代码序列可以无停顿执行。

C-17

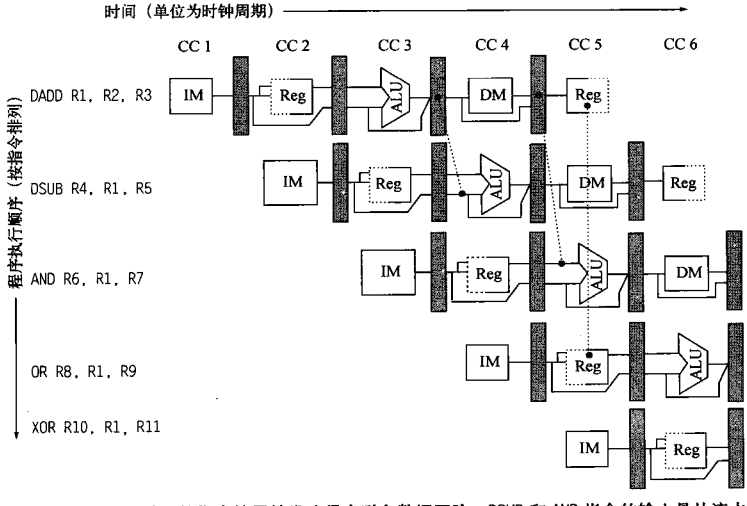


图 C-5　一组依赖 DADD 结果的指令使用转发路径来避免数据冒险。DSUB 和 AND 指令的输入是从流水线寄存器转发到第一个 ALU 输入。OR 接收的结果是通过寄存器堆转发而来的，这一点很容易实现，只需要在周期的后半部分读取寄存器、在前半部分写入寄存器就能轻松完成，如寄存器上的虚线所示。注意，转发结果可以到达任一 ALU 输入；事实上，ALU 输入既可以使用来自相同流水线寄存器的转发输入，也可以使用来自不同流水线寄存器的转发输入。例如，当 AND 指令为 AND R6, R1, R4 时就会发生这种情况

可以将转发技术加以推广，将结果直接传送给需要它的功能单元：可以将一个功能单元输出到寄存器中的结果直接转发到另一个功能单元的输入，而不仅限于同一单元的输出与输入之间。例如以下序列：

```
DADD    R1,R2,R3
LD      R4,0(R1)
SD      R4,12(R1)
```

C-18

为防止这一序列中出现停顿，我们需要将 ALU 输出值和存储器单元输出值从流水线寄存器转发到 ALU 和数据存储器输入。图 C-6 给出了这一示例的所有转发路径。

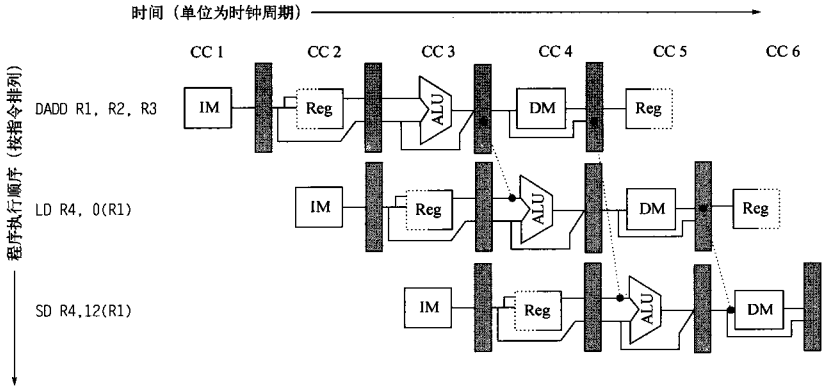


图 C-6 在 MEM 期间执行的存储操作需要转发操作数。载入结果由存储器输出转发到要存储的存储器输入端。此外，ALU 指令被转发到 ALU 输入，供载入和存储指令进行地址计算（这与转发到另一个 ALU 操作没有区别）。如果存储操作依赖与其直接相邻的前一个 ALU 操作（图中未示出），则需要转发其结果，以防止出现停顿

2. 需要停顿的数据冒险

遗憾的是，并非所有潜在数据冒险都可以通过旁路方式处理。考虑以下指令序列：

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

这一示例中旁通路径的流水化数据路径如图 C-7 所示。这种情况不同于背靠背 ALU 操作的情景。LD 指令在 4 号时钟周期（其 MEM 周期）结束之前不会得到数据，而 DSUB 指令需要在该时钟周期的开头就得到这一数据。因此，因为使用载入指令结果而产生的数据冒险无法使用简单的硬件消除。如图 C-7 所示，这种转发路径必须进行时间上的回退操作——计算机设计师还不具备这一能力！我们能够立即将该结果从流水线寄存器转发给 ALU，供 AND 操作使用，该操作是在载入操作之后两个时钟周期启动的。与此类似，OR 指令也没有问题，因为它是通过寄存器堆接收这个值的。对于 DSUB 指令，转发结果在时钟周期结束时才会抵达，这显然太晚了，因为这一指令需要在此时钟周期开始时启动。 C-19

载入指令有一种不能单由转发来消除的延迟。而是需要增加一种称为流水线互锁，以保持正确的执行模式。一般情况下，流水线互锁会检测冒险，并在该冒险被清除之前使流水线停顿。

在这种情况下，互锁使流水线停顿，让希望使用某一数据的指令等待，直到源指令生成该数据为止。这种流水线互锁引入一次停顿或气泡，就像应对结构冒险时所做的一样。停顿指令的 CPI 会增加，延长数目等于停顿的长度（在本例中为 1 个时钟周期）。

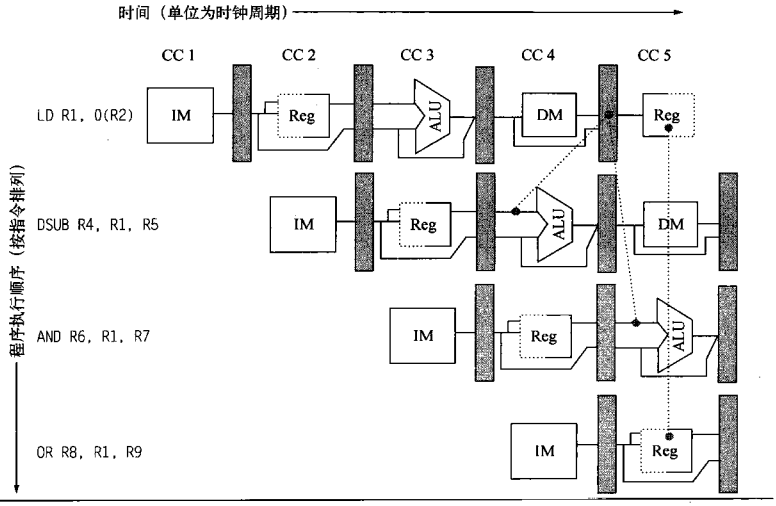


图 C-7 载入指令可以将其结果旁路至 AND 和 OR 指令，但不能旁路至 DSUB，因为这将意味着在“负向时间”上转发结果

表 C-3 使用流水级名称显示了停顿前后的流水线。因为停顿会导致从 DSUB 开始的指令在时间上向后移动 1 个周期，转发给 AND 指令的数据现在是通过寄存器堆到达的，而对于 OR 指令根本不需要转发。由于插入了气泡，需要增加一个周期才能完成这一序列。4 号时钟周期内没有启动指令（在 6 号周期没有指令完成）。

C-20

表C-3 在上半部分，我们可以看出为什么需要停顿：载入指令的MEM周期生成一个值，DSUB的EX周期会需要它，而它们是同时发生的

LD R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB 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	
LD R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	停顿	EX	MEM	WB		
AND R6,R1,R7			IF	停顿	ID	EX	MEM	WB	
OR R8,R1,R9				停顿	IF	ID	EX	MEM	WB

* 通过插入停顿可以解决这一部分，如下半部分所示。

C.2.4 分支冒险

对于我们的 MIPS 流水线，控制冒险造成的性能损失可能比数据冒险还要大。在执行分支时，修改后的程序计数器的值可能等于（也可能不等于）当前值加 4。回想一下，如果分支将程序计数器改为其目标地址，它就是选中分支；否则就是未选中分支。如果指令 i 为选中分支，通常会等到 ID 末尾，完成地址计算和对比之后才会改变程序计数器。

表 C-4 表明，处理分支的最简单方法是：一旦在 ID 期间（此时对指令进行译码）检测到分支，就对该分支之后的指令重新取值。第一个 IF 周期基本上是一次停顿，因为它从来不会执行有用工作。读者可能已经注意到，如果分支未被选中，由于事实上已经正确地提取了指令，所以 IF 级的重复是不必要的。我们稍后将开发几种机制，以充分利用这一事实。

表C-4 分支在五级流水线中导致一个周期的停顿

分支指令	IF	ID	EX	MEM	WB		
分支后续指令		IF	IF	ID	EX	MEM	WB
分支后续指令+1				IF	ID	EX	MEM
分支后续指令+2					IF	ID	EX

* 分支指令之后的指令已被提取，但随后被忽略，在已经知道分支目标之后，重新开始提取操作。如果分支未被选中，则分支后续指令的第二个 IF 就有些多余了，这一点可能比较明显。稍后将解决这一问题。

如果每个分支产生一个停顿周期，将会使性能损失 10%~30%，具体取决于分支频率，所以我们将研究一些用于应对这一损失的技术。 C-21

1. 降低流水线分支代价

有许多方法可以处理由分支延迟导致的流水线停顿，我们在这一小节讨论 4 种简单的编译时机制。在这 4 种机制中，分支的操作是静态的，也就是说，在整个执行过程中，它们对每条分支来说都是固定的。软件可以尝试利用硬件机制和分支行为方面的知识将分支代价降至最低。第 3 章研究了功能更强大的软硬件技术，可以处理静态与动态分支预测。

处理分支的最简单机制是**冻结**或**冲刷**流水线，保留或删除分支之后的所有指令，直到知道分支目标为止。这种解决方案的吸引力主要在于其软硬件都很简单。这也是表 C-4 所示流水线中较早使用的解决方案。在这种情况下，分支代价是固定的，不能通过软件来缩减。

一种性能更高但仅略微复杂一点的机制是将每个分支都看作未选中分支，允许硬件继续执行，就好像该分支未被执行一样。这时必须非常小心，在确切知道分支输入之前，不要改变处理器状态。这一机制的复杂性在于必须要知道处理器状态可能何时被指令改变，以及如何“撤销”这种改变。

在简单的五级流水线中，这种**预测未选中**机制的实现方式是继续提取指令，就好像分支指令是一条正常指令一样。流水线看起来好像没有什么异常发生。但是，如果分支被选中，就需要将已提取的指令转为空操作，重新开始在目标地址位提取指令。表 C-5 显示了这两种情况。 C-22

一种替代机制是将所有分支都看作选中分支。只要对分支指令进行了译码并计算了目标地址，我们就假定该分支将被选中，开始在目标位置提取和执行。因为在我们的五级流水线中，不可能在知道分支输出结果之前知道目标地址，所以这种方法对于这一流水线没有好处。在一些处理器中，特别是那些拥有隐性设定条件代码或者拥有更强大（因此也更慢一些）分支条件的处理器中，是可以在分支输出之前知道分支目标的，这时，预测选中机制可能就有意义了。无论是在预测选中还是预测未选中机制中，编译器总是可以通过代码组织方式，使最频繁的路

径与硬件选择相匹配，从而提高性能。我们的第四种机制为编译器提高性能提供了更多的机会。

表C-5　分支未选中（上）和被选中（下）时的预测未选中机制和流水线序列

未选中分支指令	IF	ID	EX	MEM	WB				
指令*i*+1		IF	ID	EX	MEM	WB			
指令*i*+2			IF	ID	EX	MEM	WB		
指令*i*+3				IF	ID	EX	MEM	WB	
指令*i*+4					IF	ID	EX	MEM	WB

选中指令分支	IF	ID	EX	MEM	WB				
指令*i*+1		IF	空闲	空闲	空闲	空闲			
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

* 当分支未被选中时（在 ID 期间确定），我们提取未选中指令，继续进行。当在 ID 期间确定选中该分支时，则在分支目标处重新开始提取。这将导致该分支后面的所有指令停顿 1 个时钟周期。

在某些处理器中使用的第四种机制称为**延迟分支**。这一技术在早期 RISC 处理器中的使用非常广泛，在五级流水线中的工作状态相当好。在延迟分支中，带有一个分支延迟的执行周期为：

分支指令
依序后续指令$_1$
选中时的分支目标

依序后续指令位于分支延迟时隙中。无论该分支是否被选中，这一指令都会执行。表 C-6 中给出具有分支延迟的五级流水线的流水线行为特性。尽管分支延迟可能长于一个 1 周期，但在实际中，几乎所有具有延迟分支的处理器都只有单个指令延迟；如果流水线的潜在分支代价更长，则使用其他技术。

表C-6　无论分支是否选中，延迟分支的行为特性都是相同的

未选择分支指令	IF	ID	EX	MEM	WB				
分支延迟指令(*i*+1)		IF	ID	EX	MEM	WB			
分支延迟指令*i*+2			IF	ID	EX	MEM	WB		
分支延迟指令*i*+3				IF	ID	EX	MEM	WB	
分支延迟指令*i*+4					IF	ID	EX	MEM	WB

选中分支指令	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

* 延迟时隙中的指令（对于 MIPS，只有一个延迟时隙）被执行。如果分支未被选中，则继续执行分支延迟指令之后的指令；如果分支被选中，则继续在分支目标处执行。当分支延迟时隙中的指令也是分支时，其含义就有些模糊：如果该分支未被选中，延迟分支时隙中的分支应当怎么办呢？由于这一混淆，采用延迟分支的体系结构经常禁止在延迟时隙中放入分支。

编译器的任务是让后续指令有效并可用。因此使用了多种优化方式。图 C-8 给出了三种可以调度分支延迟的方式。

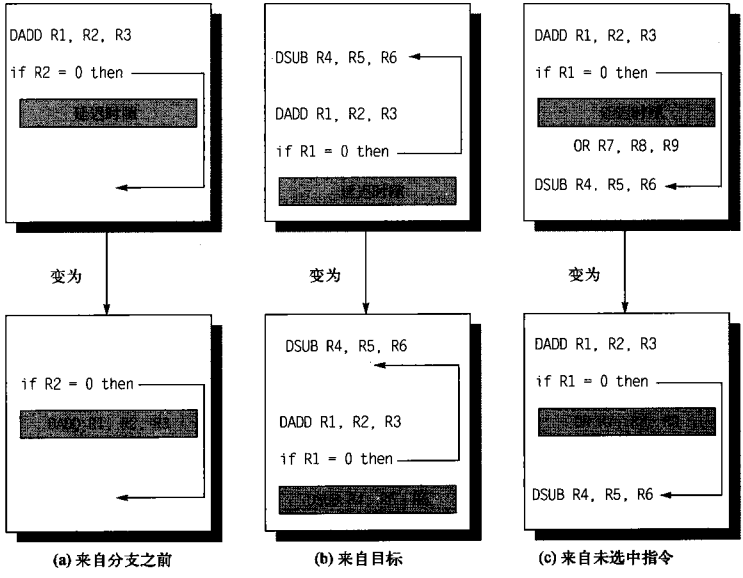


图 C-8 分支延迟时隙的调度。每一对中的上框显示调度之前的代码；下框表示调度后的代码。在(a)中，延迟时隙内载入的是分支之前的一条不相关指令。这是最佳选择。策略(b)与(c)在策略(a)不可能实现时使用。在(b)和(c)的序列中，由于分支条件中使用了 R1，所有不能将 DADD 指令移到分支之后（它的目的地是 R1）。在(b)中，分支延迟时隙中填充了分支的目标指令；这时一般需要复制目标指令，因为其他路径也可能会到达这一目标指令。当分支的选中机率很高时（比如循环分支）优选策略(b)。最后，如(c)中所示，可以用未被选中的指令填充延迟时隙。为使这一优化方法能够对(b)和(c)有效，当分支进入非预测方向时，必须可以执行经过移动的指令。这里所说的“可以”是指浪费了工作量，但程序仍能正确执行。例如，当分支进入非预测方向时，R7 成为一个未被用到的临时寄存器，那(c)策略就属于这一情景

延迟分支调度的局限性源于：(1) 对于可排在延迟时隙中的指令有限制；(2) 我们在编译时预测一个分支是否可能被选中的能力有限。为了提高编译器填充分支延迟时隙的能力，大多数具有条件分支的处理器都引入了一种取消或废除分支。在取消分支中，指令包含了预测分支的方向。当分支的行为与预期一致时，分支延迟时隙中的指令就像普通的延迟分支一样执行。当分支预测错误时，分支延迟时隙中的指令转为空操作。 C-24

2. 分支机制的性能

这些机制的各自实际性能怎么样呢？假定理想 CPI 为 1，考虑分支代价的实际流水线加速比为：

$$流水线加速比=\frac{流水线深度}{1+分支导致的流水线停顿周期}$$

由于：

$$分支导致的流水线停顿周期=分支频率\times分支代价$$

得到：

$$流水线加速比=\frac{流水线深度}{1+分支频率\times分支代价}$$

分支频率和分支代价可能都存在因为无条件分支和有条件分支导致的分量。但是，由于后者出现得更为频繁，所以它们起主导作用。

例题　对于一个更深的流水线，比如在 MIPS R4000 中，在知道分支目标地址之前至少需要三个流水级，在计算分支条件之前需要增加一个周期，这里假定条件比较时寄存器中没有停顿。三级延迟导致表 C-7 中所列三种最简单预测机制的分支代价。假定有如下频率，计算因分支使该流水线的 CPI 增加了多少。

无条件分支	4%
有条件分支、未选中	6%
条件分支、选中	10%

表C-7　三种最简单预测机制对于一个较深流水线的分支代价

分支机制	无条件分支代价	未选中分支代价	选中分支代价
冲刷流水线	2	3	3
预测选中	2	3	2
预测未选中	2	0	3

C-25

解答　将无条件、有条件未选中和有条件选中分支的相对频率乘以各自的代价，就可以求出 CPI。结果如表 C-8 所示。

表C-8　对于三种分支预测机制及较深流水线的CPI代价

分支机制	分支成本对CPI的增加量			
	无条件分支	未选中条件分支	选中条件分支	所有分支
事件频率	4%	6%	10%	20%
停顿流水线	0.08	0.18	0.30	0.56
预测选中	0.08	0.18	0.20	0.46
预测未选中	0.08	0.00	0.30	0.38

这些机制之间的差别大体随这一较长的延迟而增大。如果基础 CPI 为 1，分支是唯一的停顿源，则理想流水线的速度是使用停顿流水线机制的流水线的 1.56 倍。预测未选中机制在相同假定条件下优于停顿流水线机制，为其 1.13 倍。

C.2.5　通过预测降低分支成本

当流水线变得越来越深，而且分支的潜在代价增加时，仅使用延迟分支及类似机制就不够

了。这时需要寻求一种更积极的方式来预测分支。这些机制分为两类：依赖编译时可用信息的低成本静态机制；根据程序特性对分支进行动态预测的策略。下面将讨论这两种方法。

C.2.6 静态分支预测

改进编译时分支预测的一种重要方式是利用先前运行过程收集的一览数据。之所以值得这样做，是因为人们观测到分支的行为特性经常是双峰分布的；也就是说，各个分支经常是严重偏向于选中或未选中两种情景之一。图 C-9 显示了使用这一策略成功地进行了分支预测。使用相同输入数据来运行程序，以收集一览数据；其他研究表明，如果改变输入，使不同运行的一览数据发生变化，只会使基于一览数据的预测准确度有微小变化。

任意分支预测机制的有效性都同时取决于机制的精度和条件分支的频率，在 SPEC 中，其变化范围为 3%~24%。整数程序的错误预测率较高，此类程序的分支频率通常较高，这一事实是静态分支预测的主要限制。在下一节中，我们考虑动态分支预测器，最新的处理器都采用这种机制。 C-26

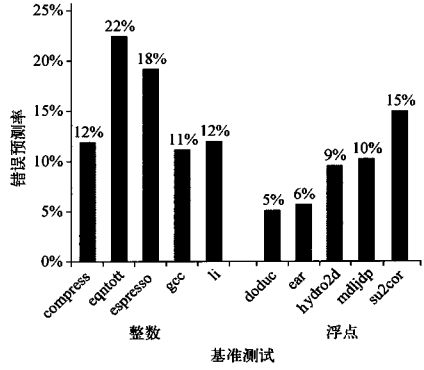


图 C-9 对于一种基于一览数据的预测器，SPEC92 的错误预测率变化幅度很大，但浮点程序通常优于整型程序，前者的平均错误预测率为 9%，标准偏差为 4%，后者的平均错误预测率为 15%，标准偏差为 5%。实际性能取决于预测精度和分支频率，其变化范围为 3%~24%

C.2.7 动态分支预测和分支预测缓冲区

最简单的动态分支预测机制是**分支预测缓冲区**或**分支历史表**。分支预测缓冲区是一个小型存储器，根据分支指令地址的低位部分进行索引。这个存储器中包含一个位（bit），表明该分支最近是否曾被选中。这一机制是最简单的缓冲区形式；它没有标志，仅当分支延迟过长，超过可能目标 PC 计算所需要的时间时，用于缩短分支延迟。

采用这样一种缓冲区时，我们事实上并不知道预测是否正确——它也可能是由另外一个具有相同低位地址的分支放入的。这个预测就是一种提示，我们假定它是正确的，并开始在预测方向上开始提取。如果这一提示最终是错误的，那将预测位反转后存回。

这个缓冲区实际上就是一个缓存，对其所有访问都会命中，而且在后面可以看到，缓冲区的性能取决于两点：对所关注分支的预测频繁程度，该预测在匹配时的准确度。在分析性能之

C-27 前，对分支预测机制的精度进行一点微小而重要的提升是很有用的。

这种简单的 1 位预测机制在性能上有一处短板：即使某个分支几乎总是被选中，在其未被选中时，我们也可能会得到两次错误预测，而不是一次，因为错误预测会导致该预测位反转。

为了弥补这一弱点，经常使用 2 位预测机制。在两位预测机制中，预测必须错过两次之后才会进行修改。图 C-10 给出了 2 位预测机制的有限状态处理器。

分支预测缓冲可以实现为一个小的特殊“缓存”，在 IF 流水线中使用指令地址进行访问，或者实现为一对比特，附加到指令缓存中的每个块，并随指令一起提取。如果指令的译码结果为一个分支，并且该分支被预测为选中，则在知道 PC 之后立即从目标位置开始提取。否则，继续进行顺序提取和执行。如图 C-10 所示，如果预测结果错误，将改变预测位。

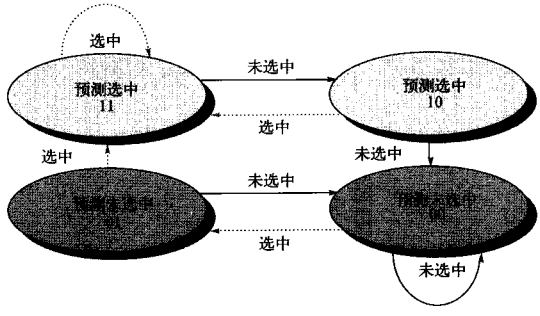


图 C-10　2 位预测机制中的状态。许多分支被选中和不被选中的概率并非均等，而是严重偏向其中一种状态，对于此类分支，2 位预测器的错误预测率经常低于 1 位预测器。在这种预测器中，使用 2 个数位对系统中的 4 种状态进行编码。这一 2 位机制实际上是一种更具一般性的机制的具体化，这种通用机制对于预测缓冲区中的每一项都有 n 位饱和计数器。对于一个 n 位计数器，其取值介于 2 至 2^n-1 之间：当计数器大于或等于其最大值（2^n-1）的一半时，分支被预测为选中；否则，预测其未选中。对 n 位预测器的研究已经证明，2 位预测器的效果几乎与 n 位预测器相同，所以大多数系统都采用 2 位分支预测器，而不是更具一般性的 n 位预测器

在实际应用程序中，如果使用每项两位的分支预测缓冲区，可以预测得到什么样的准确度？

C-28 图 C-11 显示，对于 SPEC89 基准测试，一个拥有 4096 项的分支预测缓冲区将得到超过 99%~82% 的预测准确度，或者说错误预测率为 1%~18%。根据 2005 年的标准，一个 4K 项的缓冲区（比如得出上述结果的缓冲区）算是很小了，较大的缓冲区可以得到更好一点的结果。

由于我们尝试开发更多的 ILP，所以分支预测的准确度变得非常关键。在图 C-11 中可以看出，整数程序的预测器精度低于循环密集的科学程序（前者的分支频率通常也更高一些）。我们可以采用两种方式来解决这一问题：增大缓冲区的大小，增加每种预测机制的准确度。但如图 C-12 所示，一个拥有 4000 项的缓冲区，其性能大体与无限缓冲区相当，至少对于 SPEC 这样的基准测试如此。图 C-12 中的数据清楚地表明缓冲区的命中率并非主要限制因素。前面曾经提到，

C-29 仅提高每个预测器的位数而不改变预测器结构，其影响也是微乎其微的。因此，我们需要研究一下如何提高每种预测器的精度。

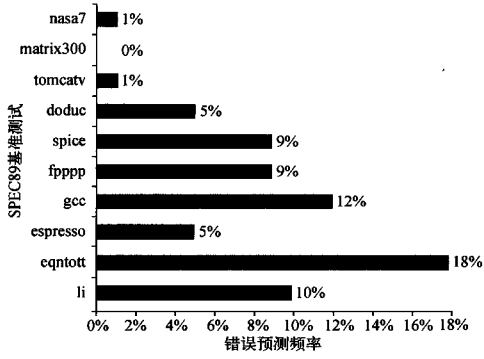


图 C-11 对于 SPEC89 基准测试，4096 项 2 位预测缓冲区的预测准确度。整数基准测试（gcc、espresso、eqntott 和 li）的预测错误率大体高于浮点程序，前者的均值为 11%，后者为 4%。忽略浮点内核（nasa7、matrix300 和 tomcatv）仍然会使浮点基准测试的准确度高于整数基准测试。收集这些数据及本节其余数据的分支预测研究采用 IBM Power 体系结构及针对该系统的优化代码执行。参见 Pan、So 和 Rameh 等人的文献[1992]。尽管这些数据来自一部分 SPEC 基准测试的较早版本，但新的基准测试结果值更大一些，行为特性稍差一些，整数基准测试尤其如此

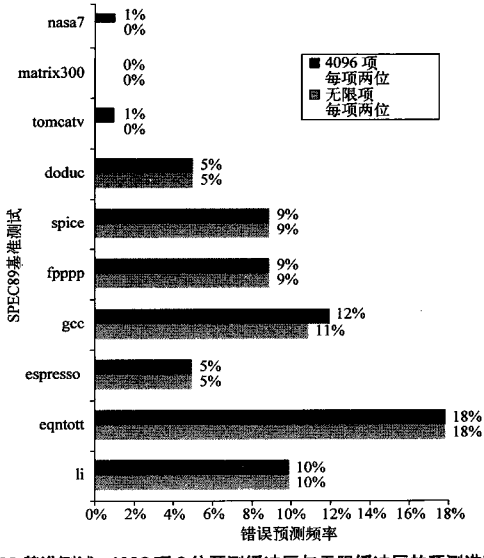


图 C-12 对于 SPEC89 基准测试，4096 项 2 位预测缓冲区与无限缓冲区的预测准确度对比。尽管这些数据是针对一部分 SPEC 基准测试的较早版本收集的，但较新版本的结果也大体相当，只是可能需要 8000 项来匹配一个无限 2 位预测器

C.3 如何实现流水化

C-30 在开始介绍基本流水化之前，需要回顾一下 MIPS 非流水版本的一种简单实现。

C.3.1 MIPS的简单实现

本节将继续 C.1 节的风格，首先给出一种简单的非流水化实现，然后是流水化实现。但我们这一次的例子是专门针对 MIPS 体系结构的。

在这一小节，我们主要关注 MIPS 中一部分整数运算的流水线，其中包括载入-存储字、等于零时分支和整数 ALU 运算。在本附录的后半部分将整合这些基础浮点操作。尽管我们仅讨论 MIPS 的一个子集，但这些基本原则可以扩展到对所有指令的处理。我们最初使用分支指令的一种不太积极的实现方式。在本节结束时，将展示如何实现一种更积极的版本。

每种 MIPS 指令都可以在最多 5 个时钟周期中实现。这 5 个时钟周期分述如下。

(1) **指令提取周期**（IF）。

```
IR ← Mem[PC];
NPC ← PC + 4;
```

操作——送出 PC，并将指令从存储器提取到指令寄存器中（IR）；将 PC 递增 4，以完成下一顺序指令的寻址。IR 用于保存将在后续时钟周期中需要的指令；与此类似，寄存器 NPC 用于保存下一顺序 PC。

(2) **指令译码/寄存器提取周期**（ID）。

```
A ← Regs[rs];
B ← Regs[rt];
Imm ← IR 的符号扩展立即数字段;
```

操作——对该指令进行译码，并访问寄存器堆，以读取寄存器（rs 和 rt 为寄存器识别符）。通用寄存器的输出被读入两个临时寄存器（A 和 B）中，供之后的时钟周期使用。IR 的低 16 位也进行了符号扩展，存储在临时寄存器 Imm 中，供下一个周期使用。

指令译码与寄存器读取是并行完成的，这一点之所以成为可能，是因为在 MIPS 格式中，这些字段放在固定位置。因为在所有 MIPS 格式中，指令的立即数部分都位于同一位置，所以在这一周期还会计算符号扩展立即数，以备在下一周期使用。

(3) **执行/实际地址周期**（EX）。

ALU 对前一周期准备的操作数进行操作，根据 MIPS 指令类型执行以下 4 种功能之一。

- 存储器引用：

  C-31
  ```
  ALUOutput ← A + Imm;
  ```

  操作——ALU 将操作数相加，得到实际地址，并将结果放在寄存器 ALUOutput 中。

- 寄存器-寄存器 ALU 指令：

  ```
  ALUOutput ← A func B;
  ```

  操作——ALU 对寄存器 A 和寄存器 B 中的取值执行由功能代码指定的操作。结果放在临时寄存器 ALUOutput 中。

- 寄存器-立即数 ALU 指令：

  ```
  ALUOutput ← A op Imm;
  ```

  操作——ALU 对寄存器 A 和寄存器 Imm 中的值执行由操作代码指定的操作。结果放在临时寄存器 ALUOutput 中。

- 分支：

```
ALUOutput ← NPC + (Imm << 2);
Cond ← (A == 0)
```

操作——ALU 将 NPC 加到 Imm 中的符号扩展立即数，将该立即数左移 2 位，得到一个字偏移量，以计算分支目标的地址。检查已经在上一周期读取的寄存器 A，以确保该分支是否被选中。由于我们仅考虑分支的一种形式（BEQZ），所以是与 0 进行对比。注意，BEQZ 实际上是一个伪指令，它会转换为一个以 R0 为操作数的 BEQ。为简单起见，我们这里仅考虑这一种分支形式。

MIPS 的载入-存储体系结构意味着实际地址与执行周期可以合并到一个时钟周期中，因为此时没有指令需要同时计算数据地址和指令目标地址，并对数据执行操作。各种形式的跳转指令未包含在上述整数指令中，它们与分支相类似。

(4) 存储器访问/分支完成计算（MEM）。

对所有指令更新 PC：PC ← NPC;

- 存储器引用：

```
LMD ← Mem[ALUOutput] 或
Mem[ALUOutput] ← B;
```

操作——在需要时访问存储器。如果指令为载入指令，则从存储器返回数据，将其放入 LMD（载入存储器数据）寄存器中；如果是存储指令，则将来自 B 寄存器的数据写入存储器。无论是哪种情况，所使用的地址都是在上一周期计算得出并放在寄存器 ALUOutput 中的地址。

- 分支：

```
if (cond) PC ← ALUOutput
```

操作——如果该指令为分支指令，则用寄存器 ALUOutput 中的分支目标地址代替 PC。

C-32

(5) 写回周期（WB）。

- 寄存器-寄存器 ALU 指令：

```
Regs[rd] ← ALUOutput;
```

- 寄存器-立即数 ALU 指令：

```
Regs[rt] ← ALUOutput;
```

- 载入指令：

```
Regs[rt] ← LMD;
```

操作——无论结果来自存储器系统（在 LMD 中），还是 ALU（在 ALUOutput 中），都将其写到寄存器堆中；寄存器目标字段也在两个位置之一（rd 或 rt）之一，具体取决于实际操作码。

图 C-13 显示了一条指令是如何流经数据路径的。在每个时钟周期结束时，在该时钟周期计算并会在后面时钟周期用到的所有值（无论是供本条指令使用，还是供下一条指令使用）都被写入存储设备中，可能是存储器、通用寄存器、PC 或临时寄存器（即，LMD、Imm、A、B、IR、NPC、ALUOutput 或 Cond）。临时寄存器在一个指令的时钟周期之间保存值，而其他存储元件则是状态的可视部分，在连续指令之间保存值。

尽管今天的所有处理器都是流水化的，但这种多周期实现方式合理地近似呈现了早些时候

是如何实现大多数处理器的。可以使用一种简单的有限状态机来实现一种遵循上述 5 周期结构的控制。对于更复杂的处理器，可以使用微代码控制。在任何一种情况下，类似上述内容的指令序列决定着控制结构。

在这种多周期实现中可以消除一些硬件冗余。例如，有两个 ALU：一个用于使 PC 递增，一个用于实际地址和 ALU 计算。由于不会在同一时钟周期用到它们，所以可以通过添加多工器和共享同一 ALU 来合并它们。同样，由于数据和指令访问发生在不同时钟周期内，所以指令和数据可以存储在相同存储器中。

我们没有优化这一简单实现方式，而是保持图 C-13 所示设计方式，为流水化实现提供了更好的基础。

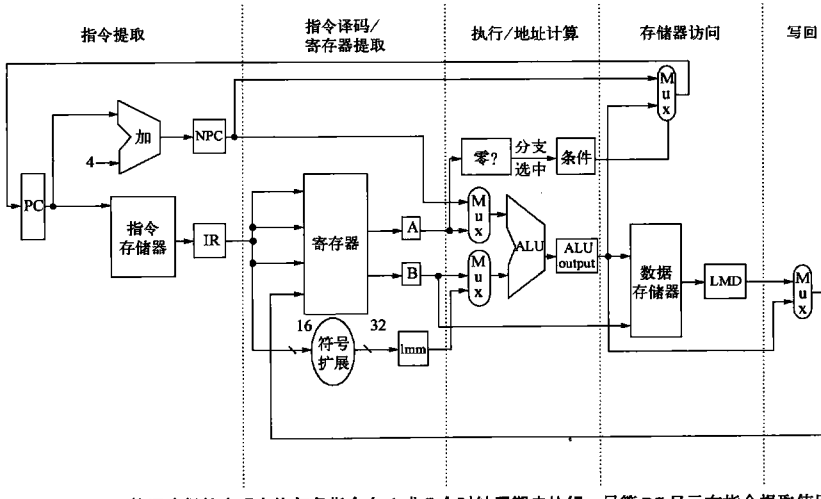


图 C-13　MIPS 数据路径的实现允许每条指令在 4 或 5 个时钟周期内执行。尽管 PC 显示在指令提取使用的数据路径部分、寄存器显示在指令译码/寄存器提取使用的数据路径部分，实际上这些功能单元都是由一条指令读取和写入的。尽管这些功能单元显示在对其进行读取的周期内，实际上，PC 是在存储器访问时钟周期内写入，寄存器是在写回时钟周期内写入的。在这两种情况下，在后续流水级中的写入是由多工器输出指示（在存储器访问或写回中），它将一个取值带回 PC 或寄存器。流水线的大多数复杂性是由这些反向流动信号引入的，因为它们表明存在冒险的可能

作为本节所讨论多周期设计的一种替代方式，我们也可以将 CPU 实现为每条指令占用 1 个长时钟周期。在这种情况下，由于一条指令之内的时钟周期之间不需要任何通信，所以将会删除临时寄存器。每条指令都在 1 个长时钟周期内执行，在该时钟周期结束时将结果写入数据存储器、寄存器或 PC。此类处理器的 CPI 为 1。不过，这时的时钟周期大约等于多周期处理器时钟周期的 5 倍，这是因为所有指令都需要遍历全部功能单元。设计人员从来不会使用这种单周期实现方式，其原因有两个。第一，单周期实现对于大多数 CPU 的效率极低，因为对于不同指令，它们的任务量会有合理的变动，从而时钟周期时间也会变动。第二，单周期实现需要重复功能单元，而在多周期实现中，这些功能单元是可以共享的。不过，这种单周期数据路径可以

C-33

让我们说明流水化是如何从时钟周期时间的角度（而非 CPI 的角度）来为处理器提供改进的。

C.3.2 MIPS基本流水线

和以前一样，只需要在每个时钟周期启动一个新指令，几乎不需要什么改变就可以对图 C-13 的数据路径实现流水线。因为每个流水级在每个时钟周期都是活动状态，所以流水级中的所有操作都必须在 1 个时钟周期内完成，任何操作组合都必须能够立即发生。此外，数据路径的流水化要求必须将流水级之间传递的数值放在寄存器中。图 C-14 显示的 MIPS 流水线中包含了每个流水级之间的适当寄存器，称为**流水线寄存器**或**流水线锁存器**。用这些寄存器相联的流水线名称对其进行标记。图 C-14 的绘制方式清楚地显示了各级之间通过流水线寄存器的连接。 C-34

用于在一条指令的时钟周期之间保存临时值的所有寄存器都包含在这些流水线寄存器中。指令寄存器（IR）是 IF/ID 寄存器的一部分，当它们用于提供寄存器名称时，对其字段进行标记。这些流水线寄存器用于从一个流水级向下一个流水级传送数据和控制。在后续流水级上需要的所有值都必须放在这样一个寄存器中，并从一个流水线寄存器中复制到下一个寄存器，直到不再需要这些值为止。如果我们尝试仅在早期未流水化数据路径中使用临时寄存器，就有可能在完成所有应用之前覆盖这些值。例如，寄存器操作数的字段，用于在载入或 ALU 操作中执行写入操作，这一字段是从 MEM/WB 流水线寄存器而非 IF/ID 寄存器中提供的。这是因为我们希望一个载入或 ALU 操作写入该操作指定的寄存器，而不是该指令当前从 IF 传送到 ID 的寄存器字段！这一目标寄存器字段就是从一个流水线寄存器复制到下一个寄存器，直到在 WB 级用到它为止。 C-35

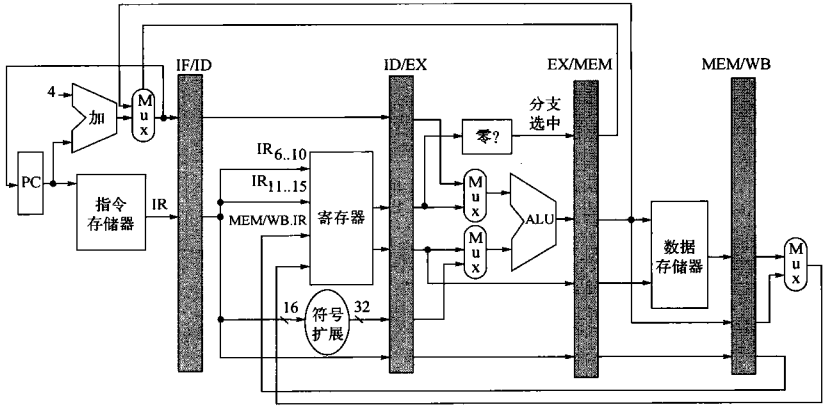


图 C-14 通过增加一组寄存器实现流水化的数据路径，每对流水线之间一个寄存器。寄存器用于从一个流水级向下一个流水级传送值和控制信息。我们也可以将 PC 看作一个流水线寄存器，它位于流水线的 IF 级之前，指向每个流水级的一个流水线寄存器。回想一下，PC 是一个边缘触发寄存器，在时钟周期结束时对其进行写入；因此，在写入 PC 时不存在争用条件。PC 的多工器已被移动，恰好在一个流水级（IF）中写入 PC。如果我们没有移动它，那在发生分支时可能会出现冲突，因为两条指令都会尝试将不同值写到 PC 中。大多数数据路径是由左向右流动，即从较早时间移向较晚时间。从右向左流动的数据（携带着分支的写回信息和 PC 信息）增加了流水线的复杂性

任何一条指令在某一时刻恰好在一个流水级中处于活动状态；任何以指令名义执行的操作都发生在一对流水线寄存器之间。因此，我们还可以通过以下方式来研究流水线的行为：查看在不同指令类型下，各流水级上必须执行什么操作。表 C-9 展示了这一观点。流水线寄存器的字段命名显示了数据在流水级之间的流动。注意，前两级的操作与当前指令类型无关。由于要等到 ID 级结束时才会对指令进行译码，所以前两级的操作必须与当前指令无关。IF 行为取决于 EX/MEM 中的指令是否为选中分支。如果是，则会在 IF 结束时将 EX/MEM 中分支指令的分支目标地址写入 PC 中；如果不是，则写回递增后的 PC。（前面曾经说过，分支指令的这一效果增加了流水线的复杂性，我们将在后面中进行讨论。）寄存器源操作数的固定位置编码对于在 ID 期间提取寄存器是至关重要的。

表C-9　MIPS流水线每个流水线级上的事件

流　水　级	任意指令		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);		
	ALU指令	**载入或存储指令**	**分支指令**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; 或 EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; 或 Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; 或 Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

* 让我们回顾一下流水级中专门用于流水线组织的操作。在 IF 中，除了提取指令和计算新 PC 之外，我们还将递增后的 PC 存储到 PC 和流水线寄存器（NPC）中，供以后计算分支目标地址时使用。这个结构与图 C-14 中的组织方式相同，在 IF 使用两个来源之一更新 PC。在 ID 中，我们提取寄存器，对 IR（立即数字段）的低 16 位进行符号扩展，并沿 IR 和 NPC 进行传送。在 EX 期间执行 ALU 运算或地址计算，沿 IR 和 B 寄存器进行传送（如果该指令为存储指令）。如果该指令为选中分支，我们还将 cond 的值设置为 1。在 MEM 阶段，我们循环使用存储器，必要时写 PC，并传送在最后流水级中使用的值。最后，在 WB 期间用 ALU 输出值或载入值来更新寄存器字段。为简单起见，我们总是将整个 IR 从一级传送到下一级，实际上，在一条指令沿流水线流动时，对 IR 的需要越来越少。

为了控制这一简单流水线，我们只需要决定如何设定图 C-14 数据路径中 4 个多工器的控制方式。ALU 级的 2 个多工器根据指令类型设定，由 ID/EX 寄存器的 IR 字段规定。上面的 ALU 输入多工器根据该指令是否为分支来进行设定，下面的多工器根据该指令是寄存器-寄存器 ALU 操作还是任意其他类型的操作来设定。IF 级中的多工器选择是使用递增 PC 的值，还是 EX/MEM.ALUOutput（分支目标）的值来写入 PC。这个多工器由 EX/MEM.cond 字段控制。第四个多工器由 WB 级的指令是载入指令还是 ALU 指令来控制。除了这 4 个多工器之外，还需要另外 1 个多工器，虽然未在图 C-14 中画出，但只要看一下 ALU 操作的 WB 级就可以清楚地看

出是存在该选择器的。根据指令类型（是寄存器-寄存器 ALU，还是 ALU 立即数或载入），目标寄存器字段为两个不同位置中的一个。因此，我们需要一个多工器来选择 MEM/WB 寄存器中 IR 的正确部分，以指定寄存器目标字段，假定该指令写入一个寄存器。

C.3.3 实现MIPS流水线的控制

将一条指令从指令译码级（ID）移入此流水线执行级（EX）的过程通常称为**指令发射**，已经执行这一步骤的指令称为**已发射**。对于 MIPS 整数流水线，所有数据冒险都可以在该流水线的 ID 阶段进行检查。如果存在数据冒险，该指令将在被发射之前停顿。与此类似，我们可以确定在 ID 期间需要哪种转发，并设定适当的控制。如果在流水线早期检查互锁，除非整个处理器停顿，否则硬件从来不需要挂起一条已经改变处理器状态的指令，从而降低了硬件的复杂性。或者，我们可以在使用操作数的一个时钟周期之始（对于此流水线来说，为 EX 和 MEM）检查冒险或转发。为了说明这两种方法之间的区别，我们将展示如何通过在 ID 中进行检查来消除因为载入指令所导致的写后读（RAW）冒险互锁（称为载入互锁），并说明如何实现指向 ALU 输入的转发路径。表 C-10 列出了我们必须处理的各种环境。

C-36
↓
C-37

表C-10 流水线冒险通过对比相邻指令的目标与源检测硬件的情景

情　景	示例代码序列	操　作
没有相关性	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	由于后面紧随的三条指令不存在对R1的依赖性，所以不可能出现冒险
相关性需要停顿	LD R1,45(R2) DADD R5,R1,R7 DSUB R8,R6,R7 OR R9,R6,R7	比较器在DADD开始EX之前检测DADD中的R1应用，并使DADD（及DSUB和OR）停顿
通过转发克服相关性	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R1,R7 OR R9,R6,R7	比较器检测DSUB中对R1的应用，并将载入结果及时转发到ALU中，供DSUB开始EX
循序访问的相关性	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R1,R7	由于OR中对R1的读取发生在ID阶段的后半部分，而载入数据的写入发生在前半部分，所以不需要操作

* 这个表显示，对于跟在写指令后面的两条指令，只需要对其目标和源进行对比。在发生停顿时，一旦继续执行后，流水线相关性与第三种情况类似。当然，涉及 R0 的冒险是可以忽略的，这是因为寄存器中总是包含零，而且上述测试经扩展后可以完成这一任务。

接下来实现载入互锁。如果存在一个因为载入指令导致的 RAW 冒险，当需要该载入数据的指令存位于 ID 级时，该载入数据将位于 EX 级。因此，我们可以用一个很小的表来描述所有可能存在的情景，可以直接将其转换为实现方式。表 C-11 显示了当使用载入结果的指令位于 ID 级时，检测到负载互锁的逻辑。

表C-11 用于检测一条指令的ID级中是否需要载入互锁的逻辑

ID/EX的操作码字段（ID/EX.IR$_{0..5}$）	IF/ID的操作码字段（IF/ID.IR$_{0..5}$）	匹配操作数字段
载入	寄存器-寄存器ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
载入	寄存器-寄存器ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
载入	载入、存储、ALU立即数或分支	ID/EX.IR[rt] == IF/ID.IR[rs]

* 表中第 1 行和第 2 行测试载入目标寄存器是否为 ID 中寄存器—寄存器操作的源寄存器之一。表中第 3 行判断载入目标寄存器是否为载入或存储有限地址、ALU 立即数或分支测试的源。请记住，IF/ID 寄存器保存着 ID 中指令的状态，它可能会用到载入结果，而 ID/EX 保存着 EX 中指令的状态，它是载入指令。

一旦检测到冒险，控制单元必须插入流水线停顿，并防止 IF 和 ID 级中的指令继续前进。前面曾经看到，所以控制信息都承载于流水线寄存器中。（仅承载指令就足够了，因为所有控制都是由其派生而来的。）因此，在检测冒险时，只需要将 ID/EX 流水线寄存器的控制部分改为全 0，它正好是一个空操作（一个不做任何事情的指令，比如 DADD R0，R0，R0）。此外，我们只需循环使用 IF/ID 寄存器中的内容，以保存被停顿的指令。在具有更复杂冒险的流水线中，这些思想同样适用：对比一组流水线寄存器，并转换为空操作，以防止错误执行。

C-38 转发逻辑的实现尽管需要考虑更多种情况，但大体类似。要实现转发逻辑，关键是要注意到流水线寄存器中既包含了要转发的数据，也包含了源、目标寄存器字段。所有转发在逻辑上都是从 ALU 或数据存储器的输出到 ALU 输入、数据存储器输入或零检测单元。因此，我们可以对比 EX/MEM 和 MEM/WB 级中所包含 IR 的目标寄存器与 ID/EX 和 EX/MEM 寄存器中所包含 IR 的源寄存器，以此来实现转发。表 C-12 显示了这些对比，以及当转发结果的目的地是 EX 中当前指令的 ALU 输入时，可能执行的转发操作。

表C-12　可以从ALU结果（EX/MEM中或MEM/WB中）或从MEM/WB中的载入结果向两个ALU输入转发数据（供EX中的指令使用）

包含源指令的流水线寄存器	源指令的操作码	包含目标指令的流水线寄存器	目标指令的操作码	转发结果的目的地	比较（若相等，则转发）
EX/MEM	寄存器-寄存器ALU	ID/EX	寄存器-寄存器ALU、ALU立即数、载入、存储、分支	顶部ALU输入	EX/MEM.IR[rd] ==ID/EX.IR[rs]
EX/MEM	寄存器-寄存器ALU	ID/EX	寄存器-寄存器ALU	底部ALU输入	EX/MEM.IR[rd] ==ID/EX.IR[rt]
MEM/WB	寄存器-寄存器ALU	ID/EX	寄存器-寄存器ALU、ALU立即数、载入、存储、分支	顶部ALU输入	MEM/WB.IR[rd] ==ID/EX.IR[rs]
MEM/WB	寄存器-寄存器ALU	ID/EX	寄存器-寄存器ALU	底部ALU输入	MEM/WB.IR[rd] ==ID/EX.IR[rt]
EX/MEM	ALU立即数	ID/EX	寄存器-寄存器ALU、ALU立即数、载入、存储、分支	顶部ALU输入	EX/MEM.IR[rt] ==ID/EX.IR[rs]
EX/MEM	ALU立即数	ID/EX	寄存器-寄存器ALU	底部ALU输入	EX/MEM.IR[rt] ==ID/EX.IR[rt]
MEM/WB	ALU立即数	ID/EX	寄存器-寄存器ALU、ALU立即数、载入、存储、分支	顶部ALU输入	MEM/WB.IR[rt] ==ID/EX.IR[rs]
MEM/WB	ALU立即数	ID/EX	寄存器-寄存器ALU	底部ALU输入	MEM/WB.IR[rt] ==ID/EX.IR[rt]
MEM/WB	载入	ID/EX	寄存器-寄存器ALU、ALU立即数、载入、存储，分支	顶部ALU输入	MEM/WB.IR[rt] ==ID/EX.IR[rs]
MEM/WB	载入	ID/EX	寄存器-寄存器ALU	底部ALU输入	MEM/WB.IR[rt] ==ID/EX.IR[rt]

* 为了判断是否应当发生转发操作，一共需要 10 次不同比较。顶部和底部的 ALU 输入分别指代与第一、第二 ALU 源操作数相对应的输入，如图 C-13 及图 C-15 中所示。请记住，EX 中目标指令的流水线锁存器是 ID/EX，而源值来自 EX/MEM 或 MEM/WB 的 ALUOutput 部分，或者 MEM/WB 的 LMD 部分。有一个复杂问题未能通过这一逻辑解决：处理多条向相同寄存器进行写入的指令。例如，在代码序列 DADD R1，R2，R3；DADDI R1，R1，#2；DSUB R4，R3，R1 期间，该逻辑必须确保 DSUB 指令使用的是 DADDI 指令的数据，而不是 DADD 指令的结果。为了处理这一情景，可以扩展上述逻辑：仅在未对相同输入转发来自 EX/MEM 的结果时，才检测来自 MEM/WB 的转发。由于 DADDI 结果将位于 EX/MEM 中，所以将转发该结果，而不是 MEM/WB 中的 DADD 结果。

除了在需要启用转发路径时必须确定的比较器和组合逻辑之外，还必须扩大 ALU 输入端的多工器，并添加一些连接，这些连接源于转发结果所用的流水线寄存器。图 C-15 给出了流水化数据路径的相关段，其中添加了所需要的多工器和连接。

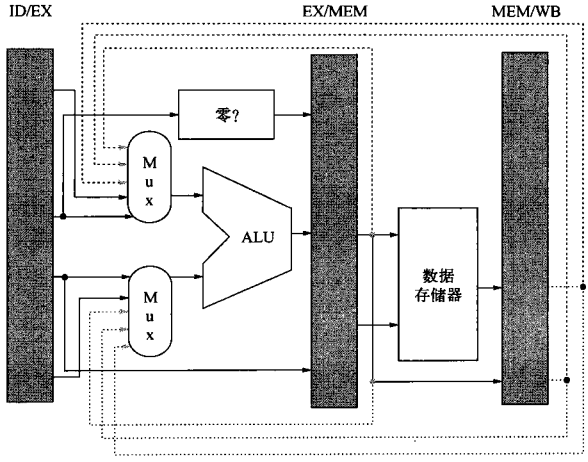


图 C-15 向 ALU 转发结果需要在每个 ALU 多工器上另外增加三个输入，增加三条指向这些新输入的路径。这些路径对应于以下三者的一个旁路：(1)EX 结束时的 ALU 输出；(2)MEM 级结束时的 ALU 输出；(3)MEM 级结束时中的存储器输出

对于 MIPS，冒险检测和转发硬件是相当简单的；我们将会看到，当为处理浮点数而对这一流水线进行扩展时，事情多少会变得复杂一些。在此之前，需要处理分支。

C.3.4 处理流水线中的分支

在 MIPS 中，分支（BEQ 和 BNE）需要检测一个寄存器，看其是否等于另一个寄存器（该寄存器可能为 R0）。如果仅考虑 BEQZ 和 BNEZ 的情景（它们需要零检测），那有可能通过将零检测移到周期内部，在 ID 周期结束时完成此判断。为了充分利用尽早判断出该分支是否命中的优势，都必须尽早计算 PC（无论命中还是未命中）。在 ID 期间计算分支目标地址需要增加一个加法器，因为前面一直用于实现此功能的主 ALU 在 EX 之前是不可用的。图 C-16 给出了修订后的流水化数据路径。通过增加独立的加法器，并在 ID 期间作出分支判断，分支仅需要停顿 1 个时钟周期。尽管这样将分支延迟缩减为 1 个周期，但这意味着如果一个 ALU 指令后面跟有一个依赖该指令结果的分支，那就会招致数据冒险停顿。表 C-13 显示了表 C-9 中修订后的流水线表的分支部分。

C-39 ~ C-40

在一些处理器中，分支冒险的代价（以时钟周期为单位）比我们这里的例子中更昂贵，因为评估分支条件和计算目的地的时间可能更长一些。例如，拥有独立译码和寄存器提取级的处理器可能存在**分支延迟**（控制冒险的长度），其长度至少为 1 个时钟周期。除非经过处理，否则分支延迟会转变为分支代价。许多实现更复杂指令集的较旧 CPU，其分支延迟为 4 个时钟周期，甚至更长一些，而大型深度流水化处理器的分支代价经常为 6 或 7 个时钟周期。一般来说，流

水线越深，以时钟周期度量的分支代价就越糟。当然，较长分支代价的相对性能性能影响取决于处理器的总 CPI。低 CPI 的处理器可以承受更昂贵的分支，这是因为分支导致处理器性能下降的百分比较低。

C-41

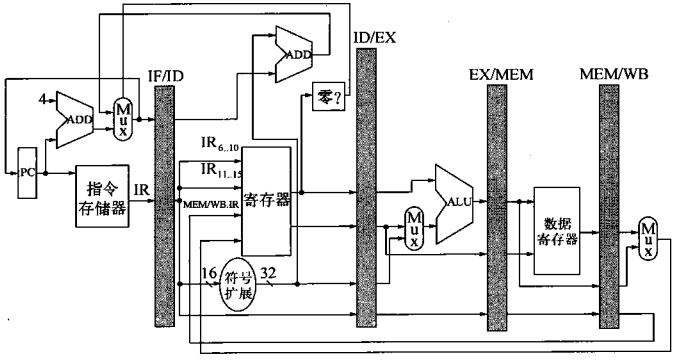


图 C-16　将零检测和分支目标计算移到流水线的 ID 级中，可以缩减因为分支冒险导致的停顿。注意，我们已经进行了两处重要改变，每个改变消除了分支指令的三周期停顿之一。第一处变化是将分支目标地址计算和分支条件判断移到 ID 周期。第二处变化是在 IF 阶段写入指令的 PC，或者使用在ID阶段计算的分支目标地址，或者使用在IF期间计算的递增PC。在对比中，图 C-14 从 EX/MEM 寄存器中获取分支目标地址，并在 MEM 时钟周期内写入结果。在图 C-14 中曾经提到，可以将 PC 看作流水线寄存器（例如，作为 ID/IF 的一部分），会在每个 IF 周期结束时向其中写入下一条指令的地址

表C-13　这一经过修订的流水线结构是以表C-9中的原流水线为基础的

流水级	分支指令
IF	IF/ID.IR ← Mem[PC]; **IF/ID.NPC,PC ← (if ((IF/ID.opcode == branch) & (Regs[IF/ID.IR$_{6..10}$] op 0)) {IF/ID.NPC + sign-extended (IF/ID.IR[immediate field] << 2) else {PC + 4});**
ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$]; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$
EX	
MEM	
WB	

* 它使用一个独立的加法器（和图 C-16 一样），在 ID 期间计算分支目标地址。新操作或发生变化的操作以黑体标出。由于分支目标地址加法是在 ID 期间发生的，所以针对所有指令都会经过这一步骤；分支条件（Regs[IF/ID.IR6..10] *op* 0）也对所有指令有效。是选择顺序 PC 还是分支目标 PC，仍然是在 IF 期间决定的，但它现在仅使用来自 ID 级的取值，这些值与上一条指令设定的值相对应。这一改变使分支代价降低两个周期，一个周期是因为提前计算分支目标和条件，一个周期是因为在同一时钟周期内控制 PC 选择，而没有推至下一周期。由于 cond 的值被设置为 0，所以除非 ID 中的指令为选中分支，则该处理器必须在 ID 结束之前对指令进行译码。因为该分支在 ID 结束时完成，所以分支指令没有用到 EX、MEM 和 WB 级。对于偏移量长于分支的跳转，还会进一步增加复杂性。我们可以增加一个加法器，将 IR 中低 26 位左移 2 位后的值与 PC 相加，以应对上述复杂性。

C-42

C.4 妨碍流水线实现的难题

我们已经理解了如何检测和解决冒险问题，现在就可以处理一些前面一直在回避的复杂问题。本节的第一部分考虑一些异常情景带来的挑战：指令的执行顺序发生意外变化。在本节的第二部分，我们将讨论一些由于不同指令集带来的挑战。

C.4.1 处理异常

异常情景在流水化 CPU 中更难处理：由于指令的重叠，更难以判断一条指令是否能安全地改变 CPU 的状态。在流水化 CPU 中，指令是一段一段地执行，在几个时钟周期内不会完成。不幸的是，流水线中的其他指令可能会引发一些异常，强制 CPU 在流水线中的指令尚未完成时中止执行。在详细讨论这些问题及其解决方案之前，需要了解可能出现哪些类型的情景，为它们提供支持有哪些体系结构方面的需求。

1. 异常的类型与需求

对于一些改变指令正常执行顺序的异常情景，在不同 CPU 中会采用不同的术语来进行描述。人们会使用**中断**、**错误**和**异常**等词，但具体用法并不一致。我们用**异常**一词来涵盖包括下列内容的所有此类机制：

- I/O 设备需求
- 从用户程序调用操作系统服务
- 跟踪指令执行
- 断点（程序员请求的中断）
- 整数算术溢出
- 浮点算术异常
- 页面错误（不在主存储器中）
- （在需要对齐时）存储器访问未对齐
- 违反存储器保护规则
- 使用未定义或未实现的指令
- 硬件故障
- 电源故障

当我们希望引用这些异常中的某一具体类别时，将使用一个较长的名称，比如 I/O 中断、浮点异常或页面错误。表 C-14 显示了上述常见异常事件的各种不同名称。 C-43

表C-14 4种不同体系结构对常见异常采用的名称不同

异常事件	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O设备请求	输入/输出中断	设备中断	异常（L0至L7自动向量）	向量化中断
从用户程序调用操作系统服务	管理员调用中断	异常（改变模式管理员陷阱）	异常（未实现的指令）——在Macintosh上	中断（INT中断）
跟踪指令异常	不适用	异常（跟踪错误）	异常（跟踪）	中断（单步陷阱）
断点	不适用	异常（断点错误）	异常（非法指令或断点）	中断（断点陷阱）
整型算术上溢或下溢、FP陷阱	程序中断（上溢或下溢异常）	异常（整型上溢陷阱或浮点下溢错误）	异常（浮点协处理器错误）	中断（溢出陷阱或数学单元异常）

（续）

异常事件	IBM 360	VAX	Motorola 680x0	Intel 80x86
页面错误（不在主存储器中）	不适用（仅370中）	异常（转移无效错误）	异常（存储器管理单元错误）	中断（页面错误）
未对齐存储器访问	程序中断（规范异常）	不适用	异常（地址错误）	不适用
违反存储器保护规则	程序中断（保护异常）	异常（违犯访问控制错误）	异常（总线错误）	中断（保护异常）
使用未定义指令	程序中断（操作异常）	异常（操作码权限/保留错误）	异常（非法指令或断点/未实现的指令）	中断（无效操作码）
硬件故障	机器检查中断	异常（机器检查中止）	异常（总线错误）	不适用
电源故障	机器检查中断	紧急中断	不适用	不可屏蔽中断

* IBM 360和80x86上的每个事件都称为中断，而680x0中的每个事件称为异常。VAX将事件划分为中断或异常。在VAX中断中使用了设备、软件和紧急等修饰词语，而VAX异常则细分为故障、陷阱和中止。

尽管我们使用异常一词来涵盖所有这些事件，但各种事件都有自己的重要特性，决定了硬件中需要采取什么操作。关于异常的需求，可以从五个半独立的方面进行描述。

C-44

(1) **同步与异步**——如果每次以相同数据和存储器分配执行程序时，事件都在同一位置发生，那事件就是**同步**的。由于硬件故障等异常，可能会由CPU和存储器外部的设备导致**异步**事件。异步事件通常可以在完成当前指令后处理，使其处理变得更容易一些。

(2) **用户请求与强制**——如果用户任务直接请求某一事件，那它就是**用户请求**事件。在某种意义上，用户请求的异常不是真正的异常，因为它们是可预测的。但是，由于对这些用户请求事件使用了用于存储和恢复状态的相同机制，所以也将它们看作异常。因为对于触发这一异常的指令来说，其唯一功能就是引发该异常，所以用户请求异常总是可以在完成该指令之后加以处理。强制异常是由某一不受用户程序控制的硬件事件导致的。强制异常不可预测，所以更难以实现。

(3) **用户可屏蔽与用户不可屏蔽**——如果一个事件可以借由用户任务来屏蔽或禁用，那它就是**用户可屏蔽**的。这一屏蔽只是控制硬件是否对异常进行回应。

(4) **指令内部与指令之间**——这种分类取决于妨碍指令完成的事件是发生在执行过程中间（无论多短），还是被看作是发生在指令之间。发生在指令内部的异常通常是同步的，因为就是这条指令触发了异常。在指令内部发生的异常实现起来要难于指令之间的异常，因为该指令必须被停止、重新启动。发生在指令内部的异步异常是因为灾难性情景（例如，硬件故障）出现的，所以总会导致程序终止。

(5) **恢复与终止**——如果程序的执行总是在中断之后停止，那它就是**终止**事件。如果程序的执行在中断之后继续，那它就是**恢复**事件。终止执行的异常实现起来更容易一些，因为CPU不需要在处理异常之后重启同一程序。

表C-15根据这5个类别对表C-14中的示例进行了划分。难点在于实现指令之间发生的中断，对于这种中断必须恢复指令的执行。此类异常的实现要求必须调用另一个程序，以保存所执行程序的状态、解决导致异常的原因，然后恢复程序的状态，之后才再次尝试导致该异常的指令。对正在执行的程序来说，这个过程事实上必须是不可见的。如果流水线使处理器能够在不影响程序执行的前提下处理异常、保存状态并重新启动，那就说流水线或处理器是**可重新启动的**。早期的超级计算机和微处理器通常缺少这一特性，但今天的几乎所有处理器都支持这一

特性，至少整数流水线是这样的，因为虚拟存储器的实现需要这一特性（见第 2 章）。 C-45

表C-15 用5个类别来确定表C-14所示不同异常类型所需要的操作

异常类型	同步与异步	用户请求与强制	用户可屏蔽与不可屏蔽	指令内部与指令之间	恢复与终止
I/O设备请求	异步	强制	不可屏蔽	之间	恢复
调用操作系统	同步	用户请求	不可屏蔽	之间	恢复
跟踪指令执行	同步	用户请求	用户可屏蔽	之间	恢复
断点	同步	用户请求	用户可屏蔽	之间	恢复
整型算术溢出	同步	强制	用户可屏蔽	内部	恢复
浮点算术上溢或下溢	同步	强制	用户可屏蔽	内部	恢复
页面错误	同步	强制	不可屏蔽	内部	恢复
非对齐存储器访问	同步	强制	用户可屏蔽	内部	恢复
违反存储器保护规则	同步	强制	不可屏蔽	内部	恢复
使用未定义指令	同步	强制	不可屏蔽	内部	终止
硬件故障	异步	强制	不可屏蔽	内部	终止
电源故障	异步	强制	不可屏蔽	内部	终止

* 必须允许恢复的异步被标记为恢复，尽管软件可能选择终止程序。发生在指令内部的可恢复、同步、强制异常是最难实现的。我们可能希望违反存储器保护访问的异常总是导致终止；但是，现代操作系统使用存储器保护来检测事件，比如首次尝试使用一个页面，或者首次尝试写入一个页面。因此，CPU 应当能够在此类异常之后恢复执行。

2. 停止和恢复执行

和在非流水化实现中一样，最困难的异步有两种特征：(1) 发生在指令内部（即，在指令执行过程期间发生，与 EX 或 MEM 流水级相对应）；(2) 必须可以重新启动。比如，在 MIPS 流水线中，由数据提取导致的虚拟存储器页面错误只可能发生在该指令 MEM 级的某一时间之后。在出现该错误时，会有其他几条指令正在执行。页面错误必须是可重新启动的，需要另一进程（比如操作系统）的干预。因此，必须能够安全关闭流水线并保存其状态，使指令能够以正确状态重新启动。重启过程的实现通常是通过保存待重启指令的 PC 来实现的。如果被重启的指令不是分支，则继续提取依次排在后面的指令，以正常方式开始执行。如果被重启的指令为分支指令，则重新计算分支条件，根据计算结果提取目标指令或直通指令。在发生异步时，流水线控制可以采取以下步骤安全地保存流水线状态。

(1) 在下一个 IF 向流水线中强制插入一个陷阱指令。

(2) 在选中该陷阱指令之前，禁止错误指令的所有写入操作，禁止流水线中后续所有指令的写入操作。可以通过以下方式来实现：从生成该异常的指令开始（不包括该指令之前的指令），将流水线中所有指令的流水线锁存置零。这样可以禁止在处理异常之前对未完成指令的状态进行任何更改。 C-46

(3) 在操作系统的异常处理例程接收控制权之后，它会立即保存故障指令的 PC。后面从异常返回时会用到这个值。

当我们使用延迟分支时，如上一节所述，由于流水线中的指令可能不是顺序相关的，所以仅用一个 PC 不再可能重建处理器的状态。因此，我们需要保存和恢复的 PC 数目等于分支延迟的长度加 1。这一步骤是在上述第(3)步完成的。

在处理异常之后，特殊指令通过重新加载 PC 并重启指令流（在 MIPS 中使用指令 RFE）从异常中返回处理器。如果流水线可以停止，使紧临错误指令之前的指令能够完成，使其之后的指令可以从头重新启动，那就说该流水线拥有**精确异常**。在理想情况下，错误指令可能还没有改变状态，要正确地处理一些异常，要求错误指令不产生任何影响。对于其他异常，比如浮点异常，某些处理器上的错误指令会首先写入其结果，然后才能处理异常。在此种情况下，即使目标位置与源操作数之一的目标位置相同，也必须准备硬件来提取源操作数。因为浮点操作可能持续许多个周期，所以其他某一指令非常可能已经写入了这些源操作数（在下一节将会看到，浮点操作经常是乱序完成的）。为了克服这一问题，最近的许多高性能 CPU 引入了两种操作模式。一种模式有精确异常，另一种（快速或性能模式）则没有精确异常。当然，精确异常模式要慢一些，因为它允许浮点指令之间的重叠较少。在一些高性能 CPU（包括 Alpha 21064、Power2 和 MIPS R8000）中，精确模式通常要慢得多（相差 10 倍以上），因此仅能用于代码调试。

在许多系统中，对精确异常的支持是必备功能，而在其他一些系统中，它“只是”存在一定的价值，因为它可以简化操作系统接口。至少，任何需要分页或 IEEE 算术陷阱处理程序的处理器都必须使其异常为精确异常，可以用硬件实现，也可以辅以一定的软件支持。对于整数流水线，创建精确异常的任务比较简单，支持虚拟内存是存储器引用支持精确异常的极大动力。在实践中，这些理由已经让设计师和架构师总是为整数流水线提供精确异常。本节将介绍如何为 MIPS 整数流水线实现精确异常。我们将在 C.5 节介绍一些技术，用来应对浮点流水线中出现的更复杂挑战。

C-47

3. MIPS 中的异常

表 C-16 显示了 MIPS 流水级，以及每一级中可能发生哪些问题异常。采用流水化时，由于有多条指令同时执行，所以在同一时钟周期中可能出现多个异常。例如，考虑如下指令序列：

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

这一对指令可能同时导致数据页面错误和算术异常，这是因为 ID 处于 MEM 级，而 DADD 位于 EX 级。要处理这一情景，可以先处理数据页面错误然后再重启执行过程。第二个异常将再次发生（如果软件正确，第一个异常将不再发生），这样，在发生第二个异常时，就可以单独对其进行处理。

表C-16　可能在MIPS流水线中发生的异常

流水级	所发生的问题异常
IF	指令提取时发生页面错误、非对齐存储器访问、违反存储器保护规则
ID	未定义或非法操作码
EX	算术异常
MEM	数据提取时发生页面错误、非对齐存储器访问、违反存储器保护规则
WB	无

* 由指令或数据存储器访问产生的异常可能占到 8 种情景的 6 种。

现实中的情景并不像这个简单例子中那样明了。异常可能乱序发生；也就是说，可能在一条指令先行产生异常之后，排在前面的指令才产生异常。再次考虑上述序列，LD 后面跟着 DADD。当 LD 处于 IF 时可能产生数据页面错误，而 DADD 指令位于 IF 时可能会产生指令页面错误。指令页面错误尽管由后一指令导致，但实际上它将会首先发生。

由于我们正在实现精确异常，所以流水线需要首先处理由 LD 指令导致的异常。为了解释如何实现这一过程，我们将位于 LD 指令位置的指令称为 i，将位于 DADD 指令位置的指令称为 $i+1$。流水线不能在发生异常时直接处理它，因为这样会导致这些异常的发生顺序不同于非流水化顺序。硬件会将一条给定指令产生的所有异常都记录在一个与该指令相关联的状态向量中。这个异常状态向量将一直随该指令向流水线下方移动。一旦在异常状态向量中设定了异常指示，则关闭任何可能导致数据值写入（包括寄存器写入和存储器写入）的控制信号。由于存储指令可能在 MEM 期间导致异常，所以硬件必须准备好在存储指令产生异常时阻止其完成。 C-48

当一条指令进入 WB 时（或者将要离开 MEM 时），将检查异步状态向量。如果发现存在任何异常，则按照它们在非流水化处理器中的发生顺序进行处理——首先处理与最早指令相对应的异常（通常位于该指令的最早流水级）。这样可以保证：指令 i 引发的所有异常将优先得到处理，早于指令 $i+1$ 引发的所有异常。当然，任何在较早流水级中以指令 i 名义采取的操作都是无效的，但由于对寄存器堆和存储器的写入操作都被禁用，所以还没有改变任何状态。在 C.5 节将会看到，要为浮点运算维护这一精确模型要困难得多。

一些处理器拥有功能更强大、运行时间更长的指令，下一小节将介绍在此类处理器的流水线中实现异常时存在的问题。

C.4.2 指令集的复杂性

所有 MIPS 指令的结果都不会超过 1 个，我们的 MIPS 流水线仅在指令执行结束时写入结果。在保证一条指令完成时，称为**已提交**。在 MIPS 整数流水线中，当所有指令到达 MEM 级的末尾（或者 WB 的开头），而且没有指令在该级之前更新状态，则说这些指令已提交。因此，精确异常非常简单。一些处理器的指令会在指令执行中间更改状态，更改状态时，该指令及其之前的指令可能还未完成。例如，IA-32 体系结构中的自动递增寻址模式可以在一条指令的执行过程中更新寄存器。在这种情况下，如果该指令由于异常而终止，则会使处理器状态发生变化。尽管我们知道哪些指令会导致异常，但由于该指令处于半完成状态，所以在未添加硬件支持的情况下，异常是不准确的。在这样一个非精确异常之后重启指令流是有难度的。我们也可以避免在指令提交之前更新状态，但这种做法的难度很大，或者成本很高，这是因为可能会用到经过更新的状态：考虑一条 VAX 指令，它多次递增同一寄存器。为了保持精确异常模型，大多数拥有此类指令的处理器能够在提交指令之间回退所做的状态更改。如果发生异常，处理器将使用这一功能将处理器状态还原为开始中断指令之前的值。在下一节，我们将会看到一种功能更强大的 MIPS 浮点流水线可能会引入类似问题，在 C.7 节介绍了一些用于完成复杂异常处理的技术。

一些在执行期间更新存储器状态的指令也会增加难度，比如 VAX 或 IBM 360 上的字符串复制操作（参见附录 K）。为中断和重启这些指令，规定这些指令使用通用寄存器作为工作寄存器。因此，部分完成指令的状态总是位于寄存器中，这些寄存器在发生异常时被保存，在异常之后恢复，使指令可以继续执行。在 VAX 中，有另外一个状态位记录指令何时开始更新存储器状态，从而在重启流水线时，CPU 知道是从头开始重启指令，还是从指令的中间开始。IA-32 字符串指令也使用寄存器作为工作存储器，这样，在保存和恢复寄存器时，也会保存和恢复这些指令的状态。 C-49

奇数个状态位可能会导致另外一组不同的难题：可能另外增加流水线冒险，可能需要额外的硬件来进行保存和恢复。条件代码就是这种情况的一个好例子。许多处理器隐式设定条件码，

将其作为指令的一部分。这种方法具有一定的优势，因为条件码将条件的判断与实际分支分离开来。但是，在调度条件代码设定与分支之间的流水线延迟时，由于大多数指令都会设定条件码，而且不能在条件判定与分支之间的延迟时隙中使用，所以隐式设定条件可能会增加调度难度。

另外，在具有条件码的处理器中，处理器必须判断何时确定分支条件。这就需要找出分支之前最后一次设置条件代码是在什么时候。在大多数隐式设定条件码的处理器中，其实现方式是推迟分支条件判断，直到先前的所有指令都有机会设定条件代码为止。

当然，显式设定条件码的体系结构允许在条件测量与待调度分支之间插入延迟；但是，流水线控制必须跟踪最后一条设定条件码的指令，以便知道何时确定分支条件。实际上，必须将条件码当作一个操作数对待，需要进行 RAW 冒险检测，就像 MIPS 必须对寄存器进行检测一样。

流水线中最后一个棘手领域是多周期操作。假定我们尝试实现下面这样一个 VAX 指令序列的流水化：

```
MOVL      R1,R2                  ;在寄存器之间移动
ADDL3     42(R1),56(R1)+,@(R1)   ;对存储器位置求和
SUBL2     R2,R3                  ;减去寄存器
MOVC3     @(R1)[R2],74(R2),R3    ;移动字符串
```

这些指令所需要的时钟周期数有很大差别，低至 1 个，高至数百个。它们所需要的数据存储器访问数也不一定，有的不需要访问数据存储器，有的可能需要数百次访问。数据冒险非常复杂，在指令之间和指令内部均会发生。一种简单的解决方案是让所有指令的执行周期数相同，但这种解决方案是不可接受的，因为它会引入数目庞大的冒险和旁通条件，形成一条极长的流水线。在指令级实现 VAX 的流水化是很困难的，但 VAX 8800 设计师找到了一种非常聪明的解决方案。他们实现了微指令执行的流水化。微指令就是一种简单指令，在序列中用于实现更复杂的指令集。由于微指令都很简单（它们看起来与 MIPS 非常相似），所以流水线控制要容易得多。从 1995 年开始，所有 Intel IA-32 微处理器都使用这一策略将 IA-32 指令转换为微操作，然后再实现微操作的流水化。

C-50

作为对比，载入-存储处理器拥有一些简单操作，完成的工作数量相似，更容易实现流水化。如果架构师认识到指令集设计与流水化之间的关系，他们就可以设计出能够高效流水化的设计体系结构。在下一节将会看到 MIPS 流水线如何处理长时间运行的指令，特别是浮点操作。

许多年以来，人们一直认为指令集与实现之间的互动非常少，在设计指令集时，实现问题不是主要关注点。20 世纪 80 年代，人们认识到指令集的复杂性会增加流水化的难度、降低流水化的效率。20 世纪 90 年代，所有公司都转向更简单的指令集，目标在于降低积极实现的复杂性。

C.5　扩展 MIPS 流水线，以处理多周期操作

我们现在希望研究如何扩展 MIPS 流水线，以处理浮点运算。这一节的重点是介绍基本方法和各种候选设计方案，最后是对一种 MIPS 浮点流水线的性能测量。

要求所有 MIPS 浮点运算都在 1 个时钟周期内完成是不太现实的，甚至在 2 个时钟周期内也有很大难度。这样做就意味着要么接受缓慢的时钟，要么在浮点单元中使用大量逻辑，或者同时接受两者。而实际情况是，浮点流水线将会允许更长的操作延迟。如果我们设想浮点指令拥有与整数指令相同的流水线，那就容易理解了，当然流水线中会有两处重要改变。第一，为了完成操作，EX 周期可能要根据需要重复多次，不同操作的重复次数可能不同。第二，可能存

在多个浮点功能单元。如果待发射指令会导致它所用功能单元的结构性冒险，或者导致数据冒险，将会出现停顿。

针对本节，我们假定 MIPS 实现中有以下 4 个独立的功能单元。

(1) 主整数单元，处理载入和存储、整型 ALU 操作，还有分支。

(2) 浮点与整数乘法器。

(3) 浮点加法器，处理浮点加、减和转换。

(4) 浮点和整型除法器。

如果我们还假定这些功能单元的执行级没有实现流水化，那么图 C-17 给出了最终的流水线结构。由于 EX 未被流水化，所以在前一指令离开 EX 之前，不会发射任何其他使用这一功能单元的指令。另外，如果一条指令不能进入 EX 级，该指令之后的整个流水线都会被停顿。 C-51

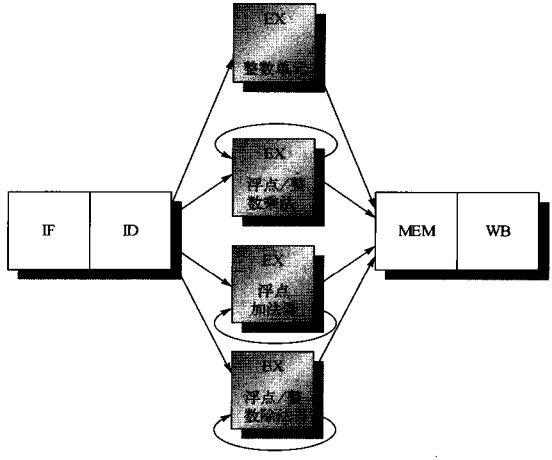


图 C-17 增加了三个未流水化浮点功能单元的 MIPS 流水线。因为每个时钟周期仅发射一条指令，所以所有指令都会经历整型运算的标准流水线。只是浮点运算在到达 EX 级时会循环。在它们完成 EX 级之后，则进入 MEM 和 WB 级，以完成执行

事实上，中间结果可能不会像图 C-17 所建议的那样围绕 EX 单元循环；而是在 EX 流水级拥有一些大于 1 的时钟延迟。我们可以推广如图 C-17 所示的浮点流水线结构，以允许实现某些级的流水化，并允许多个操作同时进行。为了描述这样一个流水线，我们必须定义功能单元的延迟以及**启动间隔**（或称**重复间隔**）。我们采用之前的相同方式来定义延迟：生成结果的指令与使用结果的指令之间的周期数。起始间隔或重复间隔是指在发出两个给定类型的操作之间必须间隔的周期数。例如，我们将使用如表 C-17 所示的延迟和启动间隔。

根据这一延迟定义，整型 ALU 运算的延迟为 0，因为其结果可以在下一时钟周期使用；载入指令的延迟为 1，因为这些结果可以相隔一个周期之后使用。由于大多数操作都会在 EX 的开头使用其操作数，所以延迟通常是指 EX 之后的级数（一条指令在 EX 生成结果），例如，ALU 运算之后有 0 个流水级，而载入指令则有 1 级。一个重要的例外是存储指令，它会在一个周期 C-52

之后使用被存储的值。因此，存储指令的延迟是针对被存储的值而言，而不是针对基址寄存器，所以少 1 个周期。流水线延迟基本上等于执行流水线深度减去 1 个时钟周期，而流水线深度等于从 EX 级到生成结果的流水级之间的级数。因此，对于上面的示例流水线，浮点加法中的级数为 4，而浮点乘法的级数为 7。为了获得更高的时钟频率，设计师需要降低每个流水级中的逻辑级数，而这样会增大更复杂操作所需要的流水级数。高时钟频率的代价是延长了操作的延迟。

表C-17　功能单元的延迟和启动间隔

功能单元	延　迟	启动间隔
整数ALU	0	1
数据存储器（整数和浮点载入）	1	1
浮点加	3	1
浮点乘（也是整数乘）	6	1
浮点除（也是整数除）	24	25

表 C-17 中的示例流水线结构允许多达 4 个同时执行的浮点加、7 个同时执行的浮点/整数乘，和一个浮点除。图 C-18 说明了如何通过扩展图 C-17 来绘制这个流水线。在图 C-18 中，重复间隔是通过增加额外的流水级来实现的，它们由增加的流水线寄存器隔开。由于这些单元是相互独立的，所以我们分别对各级进行命名。需要多个时钟周期的流水级，比如除法单元，将被进一步细分，以显示这些流水级的延迟。由于它们不是完整的流水级，所以只有一个操作是活动的。这一流水线结构还可以使用本附录前面的类似图表展示，表 C-18 显示了一组独立的浮点运算和浮点载入、存储指令。自然，浮点运算的较长延迟增加了 RAW 冒险和所导致停顿的频率，在本节后面将会看到这一点。

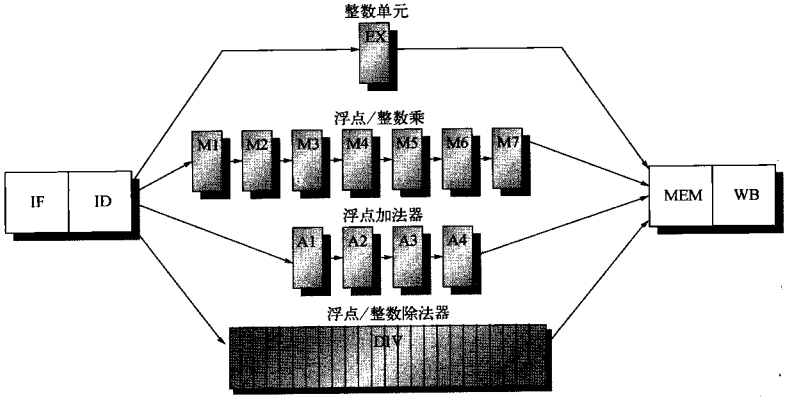


图 C-18　一条支持同时执行多个浮点操作的流水线。浮点乘法器和加法器被完全流水化，深度分别为 7 级和 4 级。浮点除法器未被流水化，而是需要 24 个时钟周期才能完成。在未招致 RAW 停顿的情况下，从发射浮点操作到使用操作结果之间的指令延迟由执行级中消耗的周期数来决定。例如，浮点加之后的第四条指令可以使用浮点加的结果。对于整数 ALU 操作，执行流水线的深度总是为 1，下一条指令就可以使用这些结果

表C-18 一组独立浮点运算的流水线时序

MUL.D	IF	ID	*M1*	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
L.D			IF	ID	*EX*	**MEM**	WB				
S.D				IF	ID	*EX*	*MEM*	WB			

* 用斜体表示的流水级显示了需要数据的位置，而用粗体表示的流水级显示了有结果可用的位置。指令标记符上的“.D”扩展表示双精度（64 位）浮点运算。浮点载入和存储使用 64 位路径连向存储器，所以流水线时序与整数载入或存储一样。

图 C-18 中的流水线结构需要另外引入流水线寄存器（例如，A1/A2、A2/A3、A3/A4），并修改连向这些寄存器的连接。ID/EX 寄存器必须进行扩展，以将 ID 连接到 EX、DIV、M1 和 A1；我们可以用标记 ID/EX、ID/DIV、ID/M1 或 ID/A1 来引用与之后流水线之一相关联的寄存器部分。ID 与所有其他流水级之间的流水线寄存器可以看作逻辑分离的寄存器，事实上也确实可以实现为分离寄存器。由于在一个流水级中只能同时有一个操作，所以控制信息可以与该流水级头部的寄存器关联在一起。 C-53

C.5.1 长延迟流水线中的冒险与转发

对于如图 C-18 所示的流水线，冒险检测与转发有许多不同方面。

(1) 因为除法单元未被完全流水化，所以可能发生结构冒险。需要对这些冒险进行检测，还将需要停顿指令发射。 C-54

(2) 因为指令的运行时间不同，所以一个周期内需要的寄存器写入次数可能会大于 1。

(3) 由于指令不会循序到达 WB，所以有可能存在写后写（WAW）冒险。注意，由于寄存器读总是在 ID 中发生，所以不可能存在读后写（WAR）冒险。

(4) 指令的完成顺序可能不同于其发射顺序，从而导致异常问题；我们将在下一小节解决这一问题。

(5) 由于操作的延迟较长，所以 RAW 冒险的停顿将会变得更为频繁。

由于操作延迟较长而导致停顿的增加，基本上与整数流水线一样。在描述这一浮点流水线中出现的新问题并探讨其解决方案之前，先让我们研究一下 RAW 冒险的可能影响。表 C-19 给出了一个典型浮点代码序列和最终的停顿。在这一节的末尾，将会研究这一浮点流水线中我们所选部分 SPEC 的性能。

表C-19 一个典型浮点代码序列展示了因为RAW冒险导致的停顿

指令		时钟周期数 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D	F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D	F0,F4,F6		IF	ID	停顿	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D	F2,F0,F8			IF	停顿	ID	停顿	停顿	停顿	停顿	停顿	停顿	A1	A2	A3	A4	MEM	WB
S.D	F2,0(R2)					IF	停顿	停顿	停顿	停顿	停顿	停顿	ID	EX	停顿	停顿	停顿	MEM

* 与较浅的整数流水线相比，较长的流水线大体会增大停顿频率。这个序列中的每条指令都会依赖于先前的指令，而且只要数据可用就可以继续进行，这里假定流水线具有完全旁通和转发。S.D 必须多停顿一个周期，使其 MEM 不会与 ADD.D 冲突。添加硬件可以很轻松地处理这种情况。

现在看看因为写入导致的问题，如前面列表中第(2)项和第(3)项所述。如果我们假定浮点寄存器堆有一个写端口，那么浮点操作序列（以及浮点载入指令与浮点运算的结合）可能导致寄存器写端口的冲突。考虑表 C-20 所示的流水线序列。在时钟周期 11 中，所有 3 条指令将到达

WB，希望写入寄存器堆。由于仅有一个寄存器堆写入端口，所以处理器必须依次完成各条指令。这个单一寄存器端口就代表着一种结构化冒险。我们可以增加端口的数目来解决这一问题，但由于所增加的写入端口可能很少用到，所以这种解决方案可能并没有什么吸引力。之所以很少用到这些端口，是因为写入端口的最大稳定状态数为 1。我们选择检测对写端口的访问，将其作为结构冒险加以实施。

表C-20　3条指令希望同时对浮点寄存器堆执行写回操作，如时钟周期11所示

	时钟周期编号										
指　令	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
…		IF	ID	EX	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
…						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

* 这不是最坏情况，因为浮点单元中的先前除法操作也可能在同一时钟周期完成。注意，尽管在时钟周期 10 中，MUL.D、ADD.D 和 L.D 都处于 MEM 级，但仅有 L.D 在实际使用该存储器，所以关于 MEM 不存在结构性冒险。

C-55 共有两种不同方法来实现这一互锁。第一种方法是跟踪 ID 级对写端口的使用，并在一条指令发射之前使其停顿，就像对于任何其他结构冒险一样。可以用一个移位寄存器来跟踪写端口的使用，这个移位寄存器可以指示已发射指令将会在何时使用这个寄存器堆。如果 ID 中的指令需要与已发射指令同时使用寄存器堆，那 ID 中的指令将会停顿一个周期。在每个时钟周期，保留寄存器将会移动 1 位。这种实现有一个好处——它能保持一个特性：所有互连检测与停顿插入都在 ID 流水级内进行。其成本是需要增加移位寄存器和写冲突逻辑。我们在本节始终假定采用这一机制。

一种替代方案是当一个冲突指令尝试进入 MEM 级或 WB 级时，使其停顿。如果我们等到冲突指令希望进入 MEM 或 WB 级时才使其停顿，可以选择停顿任一指令。一种简单的启动式方法（尽管有时是次优方法）是为那些延迟最长的单元赋予优先级，这是因为它是最可能导致另一指令因 RAW 冒险而停顿的指令。这种方案的好处在于在进入容易检测冲突的 MEM 或 WB 级之前不需要检测冲突。缺点是，由于停顿现在可能会出现在两个地方，所以使流水线控制变得复杂。注意，在进入 MEM 之前的停顿将会导致 EX、A4 或 M7 流水级被占用，可能会强制停顿返回流水线中。同样，WB 之前的停顿将会导致 MEM 倒退。

我们的另一个问题是可能出现 WAW 冒险。为了看到这些冒险的存在，考虑表 C-20 中的示例。如果 L.D 指令早一个周期发射，且其目的地为 F2，则会产生 WAW 冒险，因为它会早于 ADD.D 一个周期写入 F2。注意，只有当 ADD.D 的结果被改写，而且从来没有任何指令使用这一结果时，才会发生这一冒险！如果在 ADD.D 和 L.D 之间会用到 F2，那流水线将会因为 RAW 冒险而需要停顿，在 ADD.D 完成之前不会发射 L.D。对于这个流水线，我们可以宣称只有在发射无用指令时才会发生 WAW 冒险，但仍然必须检测这些冒险，并在完成工作时确保 L.D 的结果出现在 F2 中。（在 C.8 节将会看到，这些序列有时的确会出现在某些经过深思熟虑的代码中。）

C-56

有两种可能方式来处理这一 WAW 冒险。第一种方法是延迟载入指令的发射，直到 ADD.D 进入 MEM 为止。第二种方法是废除 ADD.D 的结果：检测冒险并改变控制，使 ADD.D 不会写入其结果。之后 L.D 就可以立即发射。由于这种冒险非常少见，所以两种方案都很有效——可以选择任何一种易于实现的方案。在任一情况下，都可以在发射 L.D 的 ID 期间检测冒险，使 L.D 停

顿或者使 ADD.D 成为空操作都很容易。难以处理的情景是检测 L.D 可能在 ADD.D 之前完成，因为这时需要知道流水线的长度和 ADD.D 的当前位置。幸运的是，这一代码序列（两个写操作之间没有插入读操作）很少出现，所以可以使用一种简单的解决方案：如果 ID 中的一条指令希望和一条已经发射的指令同时写同一寄存器，就不要向 EX 发射指令。在 C.7 节，我们将会看到如何增加硬件来消除因为这些冒险导致的停顿。接下来，我们将浮点流水线中的冒险和发射逻辑的实现细节结合在一起。

在检测可能出现的冒险时，必须考虑浮点指令之间的冒险，以及浮点指令与整型指令之间的冒险。除了浮点载入-存储和浮点-整数寄存器移动之外，浮点与整数寄存器是相互分离的。所有整数指令都是针对整数寄存器进行操作，而浮点操作仅对它们自己的寄存器进行操作。因此，在检测浮点与整数指令之间的冒险时，只需要考虑浮点载入-存储和浮点寄存器移动。流水线控制的这种简化是整数和浮点数据采用分离寄存器堆的另一项好处。（其主要好处是在各寄存器堆数目不变的情况下使寄存器数目加倍，还能在不增加各寄存器端口的情况下增加带宽。除了需要增加寄存器堆之外，其主要缺点是偶尔需要在两组寄存器之间进行移动所带来的微小成本。）假定流水线在 ID 中进行所有冒险检测，必须在执行 3 种检查之后才能发射一条指令。

(1) **检查结构冒险**——一直等到所需要功能单元不再繁忙为止（在这个流水线中，只有除法操作需要），并确保在需要寄存器写端口时可用。

(2) **检查 RAW 数据冒险**——一直等到源寄存器未被列为流水线寄存器中的目的地为止。（当这一指令需要结果时，这些寄存器不可用。）这里需要进行大量检查，具体取决于源指令和目标指令，前者决定结果何时可用，后者决定何时需要该取值。例如，如果 ID 中的指令是一个浮点运算，其源寄存器为 F2，那 F2 在 ID/A1、A1/A2 或 A2/A3 中不能被列为目的地，它们与一些浮点加法指令相对应，当 ID 中的指令需要结果时，这些指令还不能完成。（ID/A1 是 ID 输出寄存器中被发送给 A1 的部分。）如果我们希望重叠执行除法的最后几个周期，由于需要将除法接近完成时的情景作为特殊情况加以处理，所以除法运算需要的技巧性更强一些。实际上，设计师可能会忽略这一优化，以简化发射测试。 C-57

(3) **检查 WAW 数据冒险**——判断 A1，…，A4，D，M1，…，M7 中是否有任何指令的目标寄存器与这一指令相同。如果确实如此，则暂停发射 ID 中的指令。

尽管对于多周期浮点运算来说，冒险检测要更复杂一些，但其概念与 MIPS 整数流水线是一样的。对于转发逻辑也是如此。可通过以下方式来实现转发：检查 EX/MEM、A4/MEM、M7/MEM、D/MEM 或 MEM/WB 寄存器中的目标寄存器是否为浮点指令的源寄存器之一。如果确实如此，则必须启用适当的输入多工器，以选择转发数据。在练习中，读者将有机会为 RAW 和 WAW 冒险检测以及转发指定逻辑。

多周期浮点操作还为我们的的异常处理方案带来了问题，接下来将予以处理。

C.5.2 保持精确异常处理

用以下代码序列可以说明长时间运行指令所导致的另一个问题：

```
DIV.D     F0,F2,F4
ADD.D     F10,F10,F8
SUB.D     F12,F12,F14
```

这一代码序列看起来非常简单；其中没有相关性。但是，由于指令的完成顺序可能不同于其发射顺序，所以会出现一个问题。在这个示例中，我们可能预期 ADD.D 和 SUB.D 先于 DIV.D 之前完成。这称为**乱序完成**，在拥有长时操作的流水线中很常见（见 C.7 节）。由于冒险检测会禁止违

反指令之间的任何相关性，那乱序完成为什么会成为一个问题呢？假定在 ADD.D 已经完成而 DIV.D 还未完成时，SUB.D 导致了浮点算术异常，最终会出现我们应当尽力避免的不精确异常。看起来，我们似乎可以像对整数流水线那样，通过清空浮点流水线来解决这一问题。但是，异常的发生位置可能无法让我们清空流水线。例如，如果 DIV.D 决定在完成加法之后获取浮点算术异常，那可能无法获得硬件级别的精确异常。事实上，由于 ADD.D 破坏了它的一个操作数，所以即使在软件的帮助下，也无法恢复到 DIV.D 之前的状态。

这一问题的出现是因为指令的完成顺序与其发射顺序不同。共有四种可能方法来处理乱序完成情况。第一种是忽略问题，容忍非精确异常。20 世纪 60 年代和 70 年代早期采用这一方法。在某些超级计算机中仍在使用这种方法，在这种超级计算机中，某些特定类型的异常或者不允许出现，或者由硬件进行处理，不需要使流水线停止。在大多数现代处理器中很难使用这一方法，因为虚拟存储器等功能和 IEEE 浮点标准都需要通过软件和硬件的组合来实现精确异常。前面曾经提到，最近的一些处理器已经通过引入两种执行模式解决了这一问题，一种模式速度很快，但可能是非精确的，另一种模式较慢，但却是精确的。在实现较慢的精确模式时，或者使用模式切换，或者显式插入一些指令，用于测试浮点异常。无论采用哪种实现方式，浮点流水线中所允许的重叠和重新排序数目都受到严重限制，以实现在同一时间只有一条浮点指令是活动的。在 DEC Alpha 21064 和 21164、IBM Power1 和 Power2、MIPS R8000 中都采用这一解决方案。

C-58

第二种方法是缓冲一个操作的结果，直到先前发射的所有操作都完成为止。一些 CPU 实际上使用了这一方案，但是，当操作的运行时间差别很大时，由于要缓冲的结果数变得非常庞大，所以这种方法的成本就会非常高昂。此外，必须绕过来自队列的结果，以便在等待较长指令的同时继续发射指令。这就需要大量比较器和一个非常大的乘法器。

这一基本方法有两种可行的变化形式。第一种形式是 CYBER 180/990 中使用的**历史文件**。历史文件跟踪寄存器的原始值。在发生异常而且必须将状态回滚到某一乱序完成的指令之前时，可以从历史文件中恢复寄存器的原始值。在诸如 VAX 之类的处理器上，采用一种类似技术来实现自动递增和自动递减寻址。另一种方法是由 Smith 和 Pleszkum[1998]提出的**未来文件**，它跟踪寄存器的较新值；当所有先前指令均已完成时，则从未来文件中更新主寄存器堆。当发生异常时，主寄存器堆拥有中断状态的精确值。在第 3 章，我们看到在 PowerPC 620 和 MIPS R10000 等处理器中使用了这一思路的扩展，可以在保持精确异常的同时进行重叠和重新排序。

所使用的第三种方法是允许异常变得不十分精确，但保存足够的信息，以便陷阱处理例程可以生成精确的异常序列。这意味着要知道流水线中有哪些操作以及其 PC。因此，在处理异常之后，由软件完成那些最后完成指令之前的所有指令，然后该序列就能重新启动了。考虑以下最糟代码序列：

指令 $_1$——最终中断执行的长时运行指令。

指令 $_2$，…，指令 $_{n-1}$———系列未完成的指令。

指令 $_n$———条已完成指令。

C-59

给定流水线中所有指令的 PC 和异常返回 PC 之后，软件就可以得到指令 $_1$和指令 $_n$的状态。由于指令 $_n$已经完成，所以我们希望在指令 $_{n+1}$重新开始执行。在处理异常之后，软件必须模拟指令 $_1$，…，指令 $_{n-1}$的执行。然后，我们可以从异常中返回，并在指令 $_{n+1}$重新启动。由处理器正确执行这些指令的复杂性才是这一方案的主要挑战。

对于简单的类 MIPS 流水线，有一个重要简化：如果指令 $_2$，…，指令 $_n$都是整数指令，那我们就知道当指令 $_n$完成时，指令 $_2$，…，指令 $_{n-1}$也都已完成。因此，只有浮点操作需要处理。

为使这一情况易于处理，必须限制可以重叠执行的浮点指令数。例如，如果我们仅重叠两条指令，那只有中断指令需要由软件来完成。如果浮点流水线很深，或者如果存在大量浮点功能单元，那这一限制可能会限制吞吐量。SPARC 体系结构中使用了这一方法，以允许重叠浮点与整数操作。

最后一种方法是一种混合方案，它仅在确保所发射指令之前的所有指令都已完成，而且没有导致异常时，才允许继续指令发射。这样就能确保在发生异常时，中断指令之后的指令都不会完成，而中断指令之前的全部指令都可以完成。这意味着有时要停顿 CPU，以保持精确异常。为使这一方案有效，浮点功能单元必须在 EX 流水级的早期判断是否可能存在异常（这里所说的早期，在 MIPS 流水线中是指前三个周期），以阻止进一步完成其他指令。MIPS R2000/3000、R4000 和 Intel Pentium 中使用了这一方案。在附录 J 中将对其进行深入讨论。

C.5.3 MIPS浮点流水线的性能

图 C-18 中的 MIPS 浮点流水线既可以对除法单元生成结构性停顿，也可以对 RAW 冒险生成停顿（它还可能拥有 WAW 冒险，但在实际中很少发生）。图 C-19 以各实例为基础，列出了每种浮点操作的停顿周期数目（即，每个浮点基准测试的第一个长条表示每个浮点加、减或转换的浮点结果停顿数）。可以看到，每个操作的停顿周期与浮点运算的延迟相关，介于功能单元延迟的 46%~59%之间。

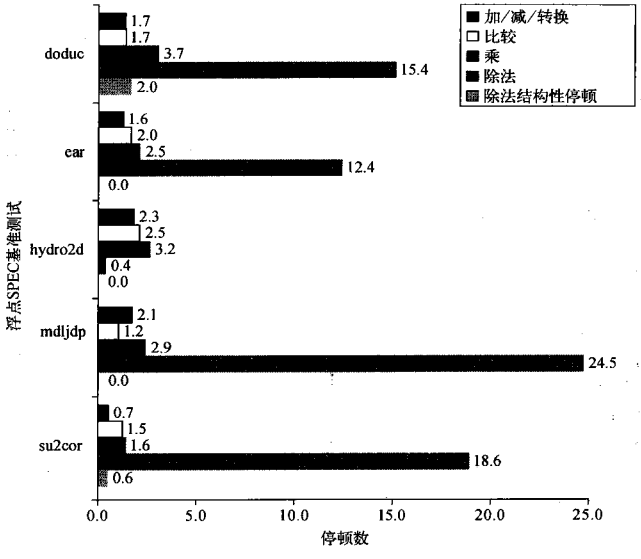


图 C-19 对于 SPEC89 浮点基准测试，每种主要浮点运算类型的停顿。除了除法结构性冒险之外，这些数据与运算的频率无关，仅取决于它的修改和使用结果之前的周期数。RAW 冒险造成的停顿数大体与浮点单元的延迟数保持一致。例如，每个浮点加、减或转换的平均停顿数为 1.7 个周期，也就是延迟（3 个周期）的 56%。与此类似，乘法与除法的平均停顿数分别为 2.8 和 14.2，也就是相应延迟的 46%和 59%。除法的结构性冒险很少见，这是因为除法频率很低的原因

图 C-20 给出了 5 种 SPECfp 基准测试整数与浮点停顿的完整分类。图中共给出 4 类停顿：浮点结果停顿、浮点比较停顿、载入与分支延迟、浮点结构延迟。编译器尝试在调度分支延迟之前调度载入与浮点延迟。每个指令的总停顿数介于 0.65~1.21 之间。 C-60

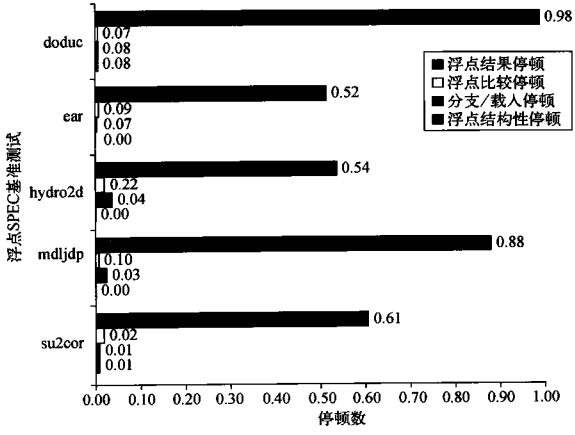


图 C-20 针对 5 种 SPEC89 浮点基准测试，MIPS 浮点流水线上发生的停顿。每条指令的总停顿数变化范围为 su2cor 的 0.65 到 doduc 的 1.21，平均值为 0.87。在所有情况下，浮点结果停顿都占绝大多数，每条指令平均为 0.71 次停顿，也就是停顿周期的 82%。比较操作生成的停顿数为每条指令平均 0.1 次，为第二大停顿源。除法结构性冒险的影响仅在 doduc 测试中比较明显一些

C.6 融会贯通：MIPS R4000 流水线

在本节，我们将研究 MIPS R4000 处理器系列的流水线结构和性能，该系列包括 4400。R4000 实现 MIPS64，但它为整数和浮点程序使用的流水线深度都超过我们使用的五级流水线设计。这一较深流水线可以将五级整数流水线分解为八级，以允许实现更高的时钟频率。由于缓存访问的时间要求很高，所以通过分解存储器访问可以获得更多的流水级。这种更深的流水线有时被称为**超流水线**。 C-61

图 C-21 显示了八级流水线结构，其中使用了数据路径的抽象版本。图 C-22 显示了流水线中连续指令的重叠。注意，尽管指令和数据存储器占用多个周期，但它们已经完全实现流水化，所以在每个时钟周期都可以开始一条新指令。事实上，流水线会在完成缓存命中检测之前使用数据；第 2 章更详细地讨论了如何完成这一过程。 C-62

每一流水级的过程如下所述。

- IF——指令提取的前半部分，PC 选择与指令缓存访问的初始化实际上发生在这里。
- IS——指令提取的后半部分，完成指令缓存访问。
- RF——指令译码与寄存器提取、冒险检查、指令缓存命中检测。
- EX——执行，包括实际地址计算、ALU 操作和分支目标计算与条件判断。
- DF——数据提取，数据缓存访问的前半部分。

- DS——数据提取的后半部分，完成数据缓存访问。
- TC——标记检查，判断数据缓存访问是否命中。
- WB——载入和寄存器-寄存器操作的写回过程。

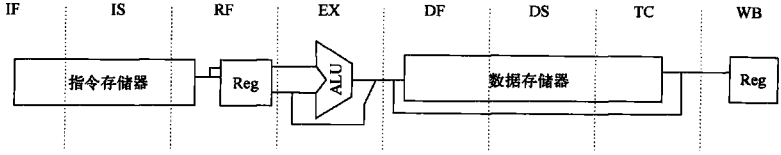


图 C-21 R4000 的八级流水线结构使用流水化指令与数据缓存。图中对流水级进行了标记，它们的详细功能用文字描述。垂直虚线表示流水级界限以及流水线锁的位置。指令实际上在 IS 结束时可供使用，但标记检查是在 RF 完成的，与此同时提取寄存器值。因此，我们将指令存储器标记为在整个 RF 中运行。由于在知道缓存访问是否命中之前，不能将数据写入寄存器，所以数据存储器访问需要 TC 级

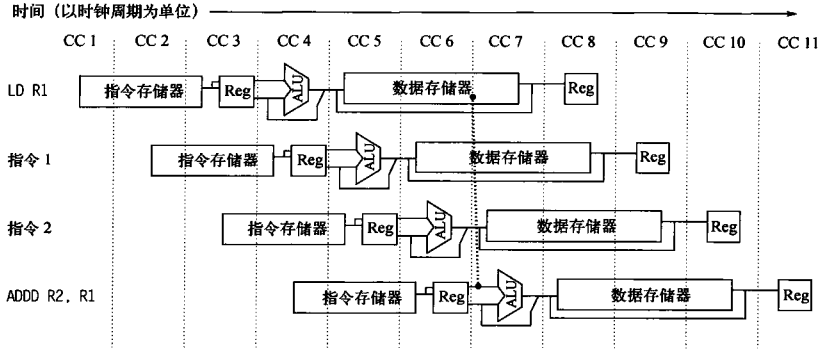


图 C-22 R4000 整数流水线的结构导致了 2 周期载入延迟。由于数据值在 DS 结束时可用，而且可能被旁路，所以有可能产生 2 周期延迟。如果 TC 中的标记检查显示这是一次缺失，则流水线将回退一个周期，在此时刻有正确数据可供使用

除了显著增加所需要的转发数量之外，这种较长延迟的流水线既会增加载入延迟，又会增加分支延迟。由于数据值在 DS 的末尾才可用，所以图 C-22 将载入延迟显示为 2 个周期。表 C-21 显示了在载入指令之后立即使用的简略流水线调度。它显示在 3 个或 4 个周期之后使用载入指令的结果时，就需要进行转发。 C-63

图 C-23 显示基本分支延迟为 3 个周期，这是因为分支条件是在 EX 期间计算的。MIPS 体系结构有一个延迟 1 周期的分支。R4000 为该分支延迟的其余 2 个周期使用预测未选中策略。如表 C-22 所示，未选中分支就是延迟 1 个周期的分支，而选中分支是在一个 1 周期延迟时隙之后跟有 2 个空闲周期。这一指令集提供了一种类似于分支的指令，前面已经对其进行了介绍，它可以帮助填充该延迟时隙。流水线互锁一方面要插入选中分支的 2 周期分支停顿代价，另一方面也造成因为使用载入结果而导致的数据冒险停顿。 C-64

表C-21　在载入指令之后立即使用结果时会产生2周期的停顿

指令编号	时钟编号 1	2	3	4	5	6	7	8	9
LD　R1,...	IF	IS	RF	EX	DF	DS	TC	WB	
DADD　R2,R1,...		IF	IS	RF	停顿	停顿	EX	DF	DS
DSUB　R3,R1,...			IF	IS	停顿	停顿	RF	EX	DF
OR　R4,R1,...				IF	停顿	停顿	IS	RF	EX

* 通常的转发路径可在 2 个周期之后使用，所以 DADD 和 DSUB 通过停顿之后的转发来获取其取值。OR 指令从寄存器堆中获取该值。由于载入之后的两条指令可能是独立的，因此不会停顿，所以旁路可能指向载入之后 3 或 4 个周期的指令。

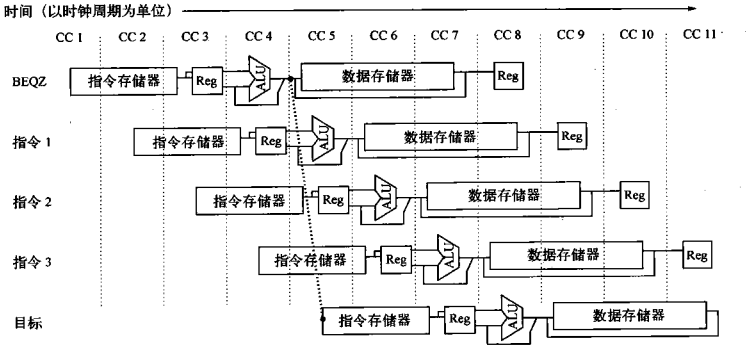


图 C-23　由于条件判断是在 EX 进行的，所以基本分支延迟为 3 个周期

深度流水线除了增加载入与分支的停顿之外，还会增加 ALU 运算的转发级别数。在 MIPS 五级流水线中，两个寄存器-寄存器 ALU 指令之间的转发可能发生于 ALU/MEM 或 MEM/WB 寄存器。在 R4000 流水线中，ALU 旁路可能有 4 种可能来源：EX/DF、DF/DS、DS/TC 和 TC/WB。

表C-22　如表中上半部分所示，选中分支有一个1周期延迟时隙，后面跟有一个2周期停顿，而如表中下半部分所示，未选中分支只有一个1周期延迟时隙

指令编号	时钟编号 1	2	3	4	5	6	7	8	9
分支指令	IF	IS	RF	EX	DF	DS	TC	WB	
延迟时隙		IF	IS	RF	EX	DF	DS	TC	WB
停顿			停顿	停顿	停顿	停顿	停顿	停顿	停顿
停顿				停顿	停顿	停顿	停顿	停顿	停顿
分支目标					IF	IS	RF	EX	DF

指令编号	时钟编号 1	2	3	4	5	6	7	8	9
分支指令	IF	IS	RF	EX	DF	DS	TC	WB	
延迟时隙		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

* 分支指令可以是一个普通延迟分支，也可能是类似于分支的指令，在分支未被选中时，取消延迟时隙中的指令效果。

C.6.1 浮点流水线

R4000 浮点单元由 3 个功能单元缓存：浮点除法器、浮点乘法器和浮点加法器。加法器逻辑在乘法或除法的最后一个步骤使用。双精度浮点运算可能占用 2 个周期（对于求相反数）到 112 个周期（对于求平方根）。此外，各种单元的起始速度不同。浮点功能单元可以看作拥有 8 个不同流水级，如表 C-23 中所列；以不同顺序组合这些流水级，即可执行各种浮点运算。

表C-23 R4000浮点流水线中使用的8个流水级

流水级	功能单元	描 述
A	浮点加法器	尾数加ADD流水级
D	浮点除法器	除法流水级
E	浮点乘法器	异常测试流水级
M	浮点乘法器	乘法器的第一流水级
N	浮点乘法器	乘法器的第二流水级
R	浮点加法器	舍入流水级
S	浮点加法器	操作数移位流水级
U		提取浮点数

这些流水级的每个流水级都有单个副本，各种指令对一个流水级可以使用 0 次或多次，使用顺序也可以不同。表 C-24 给出了最常见双精度浮点运算所使用的延迟、初始速率和流水级。 C-65

表C-24 浮点运算的延迟和初始间隔都取决于给定运算必须使用的浮点单元级

浮点指令	延 迟	初始间隔	流 水 级
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
平方根	112	111	U, E, $(A+R)^{108}$, A, R
求相反数	2	1	U, S
绝对值	2	1	U, S
浮点对比	3	2	U, A, R

* 延迟值假定目标指令是一个浮点运算。当目标指令为存储指令时，延迟会少 1 个周期。流水级的显示顺序就是各个运算使用它们的顺序。标记 S+A 表示在这个时钟周期内同时使用 S 和 A 流水级。标记 D^{28}表示 D 流水级在一行中使用 28 次。

根据表 C-24 中的信息，我们可以判断一个由不同独立浮点运算组成的序列是否可以无停顿发射。如果因为该序列的时序而对于共享流水级发生冲突，则需要停顿。表 C-25、表 C-26、表 C-27、表 C-28 给出了 4 种可能存在的常见两指令序列：乘法后面跟有加法、加法后面跟有乘法、除法后面跟有加法、加法后面跟有除法。这些表中显示了第二条指令所有感兴趣的起始位置，以及第二条指令在每个位置是发射还是停顿。当然，可能一共有三条指令是活动的，在这种情况下，发生停顿的可能性要高得多，列表也要更为复杂。 C-66

表C-25　在时钟周期0发射的浮点乘法在后面跟有一个在时钟周期1和时钟周期7之间发射的单个浮点加

		时钟周期												
运算	发射/停顿	0	1	2	3	4	5	6	7	8	9	10	11	12
乘	发射	U	E+M	M	M	M	N	N+A	R					
加	发射		U	S+A	A+R	R+S								
	发射			U	S+A	A+R	R+S							
	停顿				U	S+A	A+R	R+S						
	停顿					U	S+A	A+R	R+S					
	停顿						U	S+A	A+R	R+S				
	发射							U	S+A	A+R	R+S			
	发射								U	S+A	A+R	R+S		

* 第二列指出一个特定类型的指令在 n 个周期之后发射时是否会停顿，其中 n 为时钟周期编号，在此周期内发生第二指令的 U 级。导致停顿的流水级用黑体表示。注意，这个表仅给出了乘法指令与时钟周期 1、7 之间发射的一个加法指令之间的交互。在这种情况下，如果加法指令在乘法之后 4 或 5 个周期发射，则该加法指令会停顿；否则，它会无停顿发射。注意，如果加法指令在周期 4 发射，由于它在下一个时钟周期仍然会与乘法指令相冲突，所以加法指令会停顿 2 个周期；但是，如果加法指令在周期 5 发射，由于这样会消除冲突，所以它仍然仅停顿 1 个时钟周期。

表C-26　在加法之后发射的乘法总是可以无停顿继续进行，这是因为较长指令到达共享流水级之前，较短指令会清空它们

		时钟周期												
运算	发射/停顿	0	1	2	3	4	5	6	7	8	9	10	11	12
加	发射	U	S+A	A+R	R+S									
乘	发射		U	E+M	M	M	M	N	N+A	R				
	发射			U	M	M	M	M	N	N+A	R			

表C-27　如果加法指令在浮点除法指令结束时启动，该浮点除法可能导致该加法指令的停顿

		时钟周期											
运算	发射/停顿	25	26	27	28	29	30	31	32	33	34	35	36
除	在周期0发射	D	D	D	D	D	D+A	D+R	D+A	D+R	A	R	
加	发射		U	S+A	A+R	R+S							
	发射			U	S+A	A+R	R+S						
	停顿				U	S+A	A+R	R+S					
	停顿					U	S+A	A+R	R+S				
	停顿						U	S+A	A+R	R+S			
	停顿							U	S+A	A+R	R+S		
	停顿								U	S+A	A+R	R+S	
	停顿									U	S+A	A+R	R+S
	发射										U	S+A	A+R
	发射											U	S+A
	发射												U

* 除法在周期 0 处开始，在周期 35 处完成；表中给出除法的最后 10 个周期。由于除法指令大量使用了加法指令所需要的舍入硬件，因此，只要加法指令是在周期 28 至周期 33 中的任一周期中启动，该除法指令都会使其停顿。注意，在周期 28 处启动的加法指令将一直停顿到周期 36。如果加法指令在除法指令之后立即启动，由于加法指令可能在除法指令用到共享流水级之前完成，所以不会导致冲突，如同我们在表 C-26 中的乘加一样。和前面的表一样，这个示例假定在时钟周期 26 和 35 之间只有一个加法指令到达 U 级。

表C-28 双精度加之后跟有双精度除

运算	发射/停顿	时钟周期 0	1	2	3	4	5	6	7	8	9	10	11	12
加	发射	U	S+A	A+R	R+S									
除	停顿		U	A	R	D	D	D	D	D	D	D	D	D
	发射			U	A	R	D	D	D	D	D	D	D	D
	发射				U	A	R	D	D	D	D	D	D	D

* 如果除法指令晚于加法 1 个周期启动，则除法指令会停顿，但在此之后不存在冲突。

C.6.2 R4000 流水线的性能

本节，我们将研究在 R4000 流水线结构上运行 SPEC92 基准测试时所发生的停顿。流水线停顿或损失的原因共有四大类。

(1) **载入停顿**——在载入之后 1 或 2 个周期再使用载入结果时导致的延迟。

(2) **分支停顿**——每个选中分支上发生的两周期停顿再加上未填充或已取消分支延迟时隙。

(3) **浮点结果停顿**——因为浮点操作数的 RAW 冒险所导致的停顿。 C-67

(4) **浮点结构停顿**——因为浮点流水线中功能单元的冲突产生发射限制，进而导致的延迟。

图 C-24 给出了对于 10 个 SPEC92 基准测试，R4000 流水线的流水线 CPI 分类。表 C-29 给出了相同的数据。

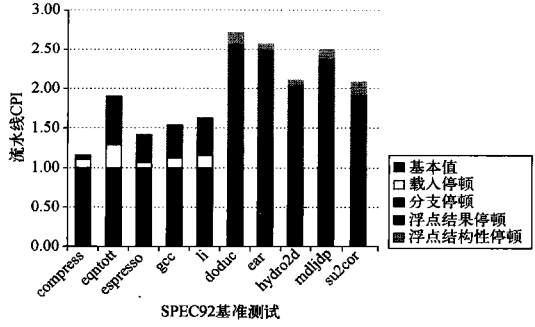


图 C-24 10 个 SPEC92 基准测试的流水线 CPI，假定采用完美缓存。流水线 CPI 的变化范围为 1.2~2.8。最左边的 5 个程序为整数程序，分支延迟是 CPI 的主要组成因素。最右边 5 个程序为浮点程序，浮点结果停顿是其主要因素。表 C-29 给出了绘制这一图形的数值

表C-29 显示了总流水线CPI的4个主要停顿源

基准测试	流水线CPI	载入停顿	分支停顿	浮点结果停顿	浮点结构性停顿
Compress	1.20	0.14	0.06	0.00	0.00
Eqntott	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
整数平均值	1.54	0.16	0.38	0.00	0.00

（续）

基准测试	流水线CPI	载入停顿	分支停顿	浮点结果停顿	浮点结构性停顿
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
浮点平均值	2.48	0.01	0.33	0.95	0.18
整体平均值	2.00	0.10	0.36	0.46	0.09

* 主要因素为浮点结果停顿（对于分支和浮点输入均是如此）和分支停顿，载入停顿和浮点结构性停顿的影响很小。

C-68 ~ C-69 根据图 C-24 和表 C-29 中的数据，可以看出深度流水线的代价。与经典五级流水线相比，R4000 流水线的分支延迟要长得多。较长的分支延迟会显著增加在分支上花费的周期数，特别是对于分支频率较高的整数程序。浮点程序一个值得注意的影响是：与结构性冒险相比，浮点功能单元的延迟会导致更多的停顿，主要源于初始间隔限制和不同浮点指令对功能单元的冲突。因此，降低浮点运算的延迟应当是第一目标，而不是实现功能单元的更多流水线或重复。当然，降低延迟可能会增加结构性停顿，这是因为许多潜在的结构性停顿被隐藏在数据冒险之后。

C.7　交叉问题

C.7.1　RISC指令集及流水线效率

我们已经讨论了指令集简化对于构建流水线的好处。简单指令集还有另外一个好处：这样可以更轻松地调度代码，以提高流水线的执行效率。为了解这一点，考虑一个简单示例：假定我们需要对存储器中的两个值相加，并将结果存回存储器。在一些高级指令集中，这一任务只需要一条指令；而在其他一些指令集中则需要两条或三条指令。一个典型的 RISC 体系结构需要四条指令（两条载入指令、一条加法指令和一条存储指令）。在大多数流水线中，不可能在没有插入停顿的情况下顺序调度这些指令。

对于 RISC 指令集，各个操作都是单独的指令，可以使用编译器进行各别调度（使用我们前面讨论的技术以及第 3 章讨论的功能更强大的技术）或动态硬件调度技术进行各别调度（将在下面讨论，第 3 章进行了更详细的讨论）。这些效率优势如此明显，再加上其实现非常容易，所以复杂指令集的几乎所有近期流水线实现实际上都将其复杂指令转换为类似于 RISC 的简单操作，然后再对这些操作进行调度和流水化。第 3 章中显示 Pentium III 和 Pentium 4 都使用了这一方法。

C.7.2　动态调度流水线

简单流水线提取一条指令并发射它，除非流水线中的已有指令和被提取的指令之间存在数据相关性，而且不能通过旁路或转发来隐藏。转发逻辑降低了实际流水线延迟，使特定的相关性不会导致冒险。如果存在不可避免的冒险，则冒险检测硬件会使流水线停顿（从使用该结构的冒险开始）。在清除这种相关性之前，不会提取或发射新指令。为了弥补这些性能损失，编译器可以尝试调度指令来避免冒险；这种方法称为**编译器调度**或**静态调度**。 C-70

几种早期处理器使用了另外一种名为动态调度的方法，硬件借此方法重新安排指令的执行过程，以减少停顿。本节通过解释 CDC6600 的记分卡技术，简要介绍了动态调度问题。一些读者会发现，在阅读这一材料之后再去深入研究第 3 章讨论的较为复杂的 Tomasulo 方案，会更容易一些。

到目前为止，本附录讨论的所有技术都使用循序指令发射，这意味着如果一条指令在流水线中停顿，将不能处理后续指令。在采用循序发射时，如果两条指令之间存在冒险，即使后面存在一些不相关的、不会停顿的指令，流水线也会停顿。

在前面开发的 MIPS 流水线中，结构性冒险和数据冒险都是在指令译码（ID）期间进行检查的：当一条指令可以正确执行时，该指令是从 ID 发射出去的。为使一条指令在其操作数可用时立即开始执行，不受其先前停顿指令的影响，我们必须将发射过程分为两部分：检查结构性冒险，等待数据冒险的消失。循序对指令进行译码和发射；但是，我们希望指令在其数据操作数可用时立即开始执行。因此，流水线是**乱序执行**的，也就暗示是**乱序完成**的。为了实现乱序执行，我们必须将 ID 流水级分为两级。

(1) **发射**——指令译码，检查结构性冒险。

(2) **读取操作数**——等到没有数据冒险，随后读取操作数。

IF 级进入发射级，EX 级跟在读取操作数级之后，这一点与 MIPS 流水线中一样。同 MIPS 浮点流水线一样，执行可能占用多个周期，具体取决于所执行的操作。因此，我们可能需要区分一条指令何时**开始执行**，何时**完成执行**；在这两个时刻之间，指令处于**执行过程**中。这样就允许多条指令同时处于执行过程中。除了对流水线结构的修改之外，我们还将改变功能单元设计：改变单元数、操作延迟和功能单元流水化，以更好地探索这些更高级的流水线技术。

采用记分卡的动态调度

在动态调度流水线中，所有指令都循序通过发射级（循序发射）；但是，它们可能在第二级（读取操作数级）停顿，或绕过其他指令，然后进行乱序执行状态。**记分卡**技术在有足够资源、没有数据依赖性时，允许指令乱序执行；这一功能是在 CDC 6600 记分卡中开发的，并因此而得名。

C-71

在我们了解如何在 MIPS 流水线中使用记分卡之前，非常重要的一点是要观察到当指令乱序执行时可能会出现 WAR 冒险，这种冒险在 MIPS 浮点或整数流水线中是不存在的。例如，考虑以下代码序列：

```
DIV.D    F0,F2,F4
ADD.D    F10,F0,F8
SUB.D    F8,F8,F14
```

ADD.D 和 SUB.D 之间存在一种反相关性：如果流水线在 ADD.D 之前执行 SUB.D，它将违犯反相关性，产生错误的执行结果。与此类似，为避免违反输出相关性，也必须检查 WAW 冒险（例如，当 SUB.D 的目标寄存器为 F10 时将会发生此种冒险）。后面将会看到，记分卡通过停顿反相关中涉及的后续指令，避免了这两种冒险。

记分卡的目标是：通过尽早执行指令，保持每时钟周期 1 条指令的执行速率（在没有结构性冒险时）。因此，当下一条要执行的指令停顿时，如果其他指令不依赖于任何活动指令或停顿指令，则发射和执行这些指令。记分卡全面负责指令发射与执行，包括所有冒险检测任务。要充分利用乱序执行，需要在其 EX 级中同时有多条指令。这一点可以通过多个功能单元、流水

化功能单元或同时利用两者来实现。由于这两种功能（流水化功能单元和多个功能单元）对于流水线控制来说基本上是等价的，所以我们将假定处理器拥有多个功能单元。

CDC 6600 拥有 16 个独立的功能单元，包括 4 个浮点单元、5 个存储器引用单元和 7 个整数运算单元。在采用 MIPS 体系结构的处理器上，记分卡主要在浮点单元上发挥作用，因为其他功能单元的延迟非常小。让我们假定一共有两个乘法器、一个加法器、一个除法单元和一个完成所有存储器引用、分支和整数运算的整数单元。尽管这个例子要比 CDC 6600 简单，但它的功能足以演示这些原理，不需要大量细节，也不需要非常长的示例。因为 MIPS 和 CDC 6600 都是载入-存储体系结构，所以这些技术对于这两种技术来说几乎是相同的。图 C-25 给出了该处理器的基本结构。

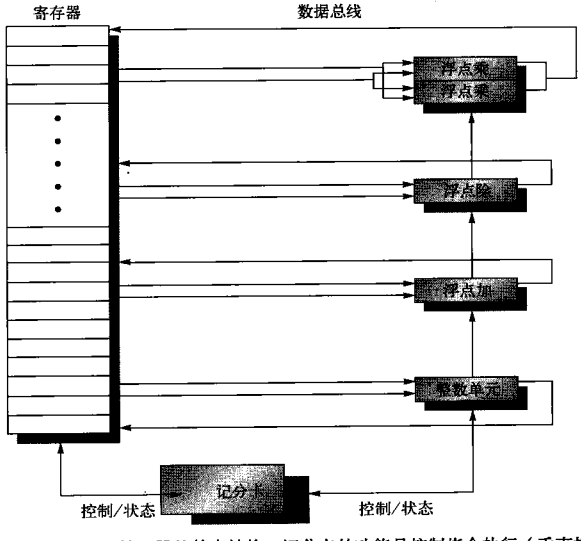


图 C-25　带有记分卡的 MIPS 处理器的基本结构。记分卡的功能是控制指令执行（垂直控制线）。所有数据在寄存器堆和总线上的功能单元之间流动（水平线，在 CDC6600 中称为干线）。共有两个浮点乘法器、一个浮点除法器、一个浮点加法器和一个整数单元。一组总线（两个输入和一个输出）充当一组功能单元。记分卡的细节在表 C-30 至表 C-33 中给出

每条指令都进入记分卡，在这里构建一条数据相关性记录；这一步与指令发射相对应，并替换 MIPS 流水线中的 ID 步骤。记分卡随后判断指令什么时候能够读取它的操作数并开始执行。如果记分卡判断该指令不能立即执行，它监控硬件中的所有变化，以判断该指令何时能够执行。记分卡还控制一条指令什么时候能将其结果写到目标寄存器中。因此，所有冒险检测与解决都集中在记分卡。我们后面将会看到记分卡的一张表格（表 C-30），但首先需要理解流水线发射与执行部分的步骤。

C-72

每条指令需要经历 4 个执行步骤。（由于我们现在主要考虑浮点运算，所以不考虑存储器访问步骤。）我们先粗略地查看一下这些步骤，然后再详细研究记分卡如何记录一些必要信息，用

于判断执行过程何时由一个步骤进行到下一个步骤。这四个步骤代替了标准 MIPS 流水线中的 ID、EX 和 WB 步骤，如下所示。

(1) **发射**——如果指令的一个功能单元空闲，没有其他活动指令以同一寄存器为目标寄存器，则记分卡向功能单元发射指令，并更新其内部数据结构。这一步代替了 MIPS 流水线中 ID 步骤的一部分。只要确保没有其他活动功能单元希望将自己的结果写入目标寄存器，就能保证不会出现 WAW 冒险。如果存在结构性冒险或 WAW 冒险，则指令发射停顿，在清除这些冒险之前，不会再发射其他指令。当发射级停顿时，会导致指令提取与发射之间的缓冲区填满；如果缓冲区只是一项，则指令提取立即停顿。如果缓冲区是拥有多条指令的队列，则在队列填满后停顿。 C-73

(2) **读取操作数**——记分卡监视源操作数的可用性。如果先前发射的活动指令都不再写入源操作数，而该源操作数可用。当源操作数可用时，记分卡告诉功能单元继续从寄存器读取操作数，并开始执行。记分卡在这一步动态解决 RAW 冒险，可以发送指令以进行乱序执行。这一步和发射步骤一起，完成了简单 MIPS 流水线中 ID 步骤的功能。

(3) **执行**——功能单元接收到操作数后开始执行。结果准备就绪后，它通知记分卡已经完成执行。这一步代替了 MIPS 流水线中的 EX 步骤，在 MIPS 浮点流水线中耗用多个周期。

(4) **写结果**——一旦记分卡知道功能单元已经完成执行，则检查 WAR 冒险，并在必要时停顿正在完成的指令。

如果有一个与我们先前示例相类似的代码序列，其中 ADD.D 和 SUB.D 都使用 F8，则存在 WAR 冒险。在这个示例中，有如下代码：

```
DIV.D     F0,F2,F4
ADD.D     F10,F0,F8
SUB.D     F8,F8,F14
```

ADD.D 有一个源操作数为 F8，就是 SUB.D 的目标寄存器。但 ADD.D 实际上取决于前面的一条指令。记分卡仍将 SUB.D 停顿于它的写结果阶段，直到 ADD.D 读取它的操作数为止。一般来说，在以下情况下，不能允许一条正在执行的指令写入其结果：

- 在正在执行的指令前面（即按发射顺序）有一条指令还没有读取其操作数；
- 这些操作数之一与正执行指令的结果是同一寄存器。

如果不存在这一 WAR 冒险，或者已经清除，则记分卡会告诉功能单元将其结果存储到目标寄存器中。这一步骤代替了简单 MIPS 流水线中的 WB 步骤。

乍看起来，记分卡在区分 RAW 和 WAR 冒险时似乎会有困难。 C-74

因为只有当寄存器堆中拥有一条指令的两个操作数时，才会读取这些操作数，所以记分卡未能利用转发。只有当寄存器都可用时才会进行读取。这一代价并没有读者最初想象得那么严重。这里与我们前面的简单流水线不同，指令会在完成执行之后立即将结果写入寄存器堆（假定没有 WAR 冒险），而不是等待可能间隔几个周期的静态指定写入时隙。由于结果的写入和操作数的读取不能重叠，所以仍然会增加一个周期的延迟。我们需要增加缓冲，以消除这一开销。

记分卡根据自己的数据结构，通过与功能单元的沟通来控制指令从一个步骤到下一个步骤的进展。但这种做法有一点点复杂。指向寄存器堆的源操作数总线和结果总线数目是有限的，所以可能会存在结构性冒险。记分卡必须确保允许进入第(2)、(4)步的功能单元数不会超过可用总线数。这里不会进行深入讨论，仅提及 CDC 6600 在解决这一问题时，将 16 个功能单元分为四组，并为每一组提供一组总线，称为**数据干线**。在一个时钟周期内，一个组中只有一个单元可以读取其操作数或写入其结果。

现在让我们看一个拥有五个功能单元的 MIPS 记分卡所保持的详尽数据结构。表 C-30 显示了在如下这一简单指令序列执行时，记分卡中的信息。

```
L.D      F6,34(R2)
L.D      F2,45(R3)
MUL.D    F0,F2,F4
SUB.D    F8,F6,F2
DIV.D    F10,F0,F6
ADD.D    F6,F8,F2
```

表C-30　记分卡的组成。每条已经发射或正在发射的指令在指令状态表中都有一个对应项

指　令		指令状态			
		发射	读取操作数	执行完成	写结果
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	
MUL.D	F0,F2,F4	√			
SUB.D	F8,F6,F2	√			
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2				

	功能单元状态								
名称	忙	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
整数	是	载入	F2	R3				否	
Mult1	是	乘	F0	F2	F4	整数		否	是
Mult2	否								
加	是	减	F8	F6	F2		整数	是	否
除	是	除	F10	F0	F6	乘		否	是

	寄存器结果状态								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	整数			加	除			

* 每个功能单元在功能单元状态表中有一个对应项。一旦发射一条指令后，就在功能单元状态表中保留其操作数记录。最后，寄存器状态表指示哪个单元将生成每个未给出的结果；项数与寄存器数相等。指令状态表表明：(1) 第一个 L.D 已经完成并写入其结果，(2) 第二个 L.D 已经完成执行，但还没有写入其结果。MUL.D、SUB.D 和 DIV.D 都已经发射，但正在停顿，等待其操作数。功能单元状态表明第一个乘法单元正在等待整数单元，加法单元正在等待整数单元，除法单元正在等待第一个乘法单元。ADD.D 指令因为结构性冒险而停顿，当 SUB.D 完成时将会清除这一冒险。如果这些记分卡中某个中的项目没有用到，则保持为空。例如，Rk 字段在载入时没有用到，Mult2 单元没有用到，因此它们的字段没有意义。另外，一旦读取一个操作数之后，Rj 和 Rk 字段将被设置为“否”。表 C-33 表明了最后一步为什么至关重要。

记分卡共有三个部分，如下所述。

(1) **指令状态**——指出该指令处于四个步骤中的哪一步。

(2) **功能单元状态**——指出功能单元（FU）的状态。共有 9 个字段用来表示每个功能单元的状态。

- 忙——指示该单元是否繁忙。
- Op——在此单元中执行的运算（例如，加或减）。

- Fi——目标寄存器。
- Fj, Fk——源寄存器编号。
- Qj, Qk——生成源寄存器 Fj、Fk 的功能单元。
- Rj, Rk——指示 Fj、Fk 已准备就绪但尚未读取的标记。在读取操作数后将其设置为“否”。 C-75

(3) **寄存器结果状态**——如果一条活动指令以该寄存器为目标寄存器，则指出哪个功能单元将写入每个寄存器。只要没有向该寄存器写入的未完成指令，则将此字段设置为空。

现在让我们看一下在表 C-30 中开始的代码序列如何继续执行。之后，我们就能更详细地研究记分卡用于控制执行的条件了。 C-76

例题 假定浮点功能单元的 EX 周期延迟如下（选择这些延迟是为了说明行为特性，并非代表性数值）：加法为 2 个时钟周期，乘法为 10 个时钟周期，除法为 40 个时钟周期。利用表 C-30 前面的代码段，并从表 C-30 中指令状态指示的时刻开始，说明当 MUL.D 和 DIV.D 分别准备好写入结果状态时，状态表中是什么样的。

解答 从第二个 L.D 到 MUL.D、ADD.D 和 SUB.D，从 MUL.D 到 DIV.D 和从 SUB.D 到 ADD.D，存在 RAW 数据冒险。在 DIV.D 和 ADD.D 以及 SUB.D 之间存在 WAR 数据冒险。最后，加法功能单元对于 ADD.D 和 SUB.D 中存在结构性冒险。当 MUL.D 和 DIV.D 准备好写入其结果时，这些表分别如表 C-31 和表 C-32 所示。

表C-31 在MUL.D写结果之前的记分卡表

指令		指令状态			
		发射	读取操作数	执行完成	写结果
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2	√	√	√	

名称	功能单元状态								
	忙	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
整数	否								
Mult1	是	乘	F0	F2	F4			否	否
Mult2	否								
加	是	加	F6	F8	F2			否	否
除	是	除	F10	F0	F6	Mult1		否	是

	寄存器结果状态								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1			加		除			

* DIV.D 还没有读取它的任何一个操作数，因为它依赖于乘法结果。ADD.D 已经读取其操作数，并在执行过程中，当然它必须要等到 SUB.D 完成之后才能得到功能单元。由于存在对 F6 的 WAR 冒险，所以 ADD.D 不能继续写结果，F6 由 DIV.D 使用。只有当功能单元正在等待另一个单元时，Q 字段才有关联。

表C-32　在DIV.D写结果之前的记分卡表

指令		指令状态			
		发射	读取操作数	执行完成	写结果
L.D	F6,34(R2)d	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	√
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√	√	√	
ADD.D	F6,F8,F2	√	√	√	√

名称	功能单元状态								
	忙	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
整数	否								
Mult1	是	乘	F0	F2	F4			否	否
Mult2	否								
加	是	减	F6	F8	F2			否	否
除	是	除	F10	F0	F6			否	是

	寄存器结果状态								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1			加		除			

*在 DIV.D 读取操作数并获得 F6 的副本之后，ADD.D 立即就能完成。只有 DIV.D 指令仍然有待完成。

现在，我们可以研究一下为使每条指令能够继续，记分卡必须做些什么，以此来详细了解记分卡是如何工作的。表 C-33 说明，为使每条指令能够继续执行，记分卡需要些什么，并记录在指令继续执行时需要哪些必要的操作。记分卡记录操作数标志符信息，比如寄存器编号。例如，在发射指令时，必须记录源寄存器。因为我们将寄存器的内容称为 Regs[D]，其中 D 为寄存器名称，所以不存在模糊性。例如，Fj[FU]←S1 会导致寄存器名称 S1 被放在 Fj[FU]中，而不是寄存器 S1 的内容。

表C-33　指令执行过程中每个步骤需要的检查和记录操作

指令状态	在达到以下条件之前一直等待	记　录
发射	不忙[FU]，没有结果[D]	Busy[FU]←yes; Op[FU]←op; Fi[FU]←D; Fj[FU]←S1; Fk[FU]←S2; Qj←Result[S1]; Qk← Result[S2]; Rj← not Qj; Rk← not Qk; Result[D]←FU;
读取操作数	Rj和Rk	Rj← No; Rk← No; Qj←0; Qk←0
执行完成	功能单元完成	
写结果	∀f((Fj[f] ≠ Fi[FU] or Rj[f] = No) & (Fk[f] ≠ Fi[FU] or Rk[f] = No))	∀f(if Qj[f]=FU then Rj[f]←Yes); ∀f(if Qk[f]=FU then Rk[f]←Yes); Result[Fi[FU]]← 0; Busy[FU]← No

* FU 表示该指令使用的功能单元，D 为目标寄存器名称，S1 和 S2 为源寄存器名称，op 是要完成的操作。要访问记分卡中名称为 Fj、针对功能单元 FU 的项目，使用标记 Fj[FU]。Result[D]是将写入寄存器 D 的功能单元名称。当存在 WAR 冒险时，对写入结果情况的测试将禁止写入，如果另一条指令以这一指令的目标寄存器(Fi[FU])为源寄存器(Fj[f]或 Fk[f])，或者如果某一其他指令已经写入了寄存器(Rj=Yes 或 Rk=Yes)，则存在 WAR 冒险。对所有功能单元均使用变量 f。

记分卡的成本和收益也是人们的关注点。CDC 6600 设计师测量到 FORTRAN 程序的性能改进因数为 1.7，对于人工编码的汇编语言改进因数为 2.5。但是，这些数据是在软件流水线调度、半导体主存储器和缓存（缩短了存储器访问时间）之前的那一时期测得的。CDC 6600 上记分卡所拥有的逻辑数与一个功能单元相当，这是相当低的。主要成本在于存在大量总线——其数量大约是 CPU 循序执行（或者每个执行周期仅启动一条执行）时所需数量的四倍。人们近来对动态调度的关注有所增加，其目的就是希望在每个时钟周期内发射更多条指令（反正都要支付增加总线带来的成本），一些很自然地以动态调度为基础的思想（比如 4.7 节研究的推测）也是提升此关注度的推动因素。

记分卡利用可用 ILP，在最大程度上降低因为程序真数据相关所导致的停顿数目。在消除停顿方面，记分卡受以下几个因素的影响。

(1) **指令间可用并行数**——这一因素决定了能否找到要执行的独立指令。如果每条指令都依赖于它前面的指令，那就找不到减少停顿的动态调度方案。如果必须从同一基本模块中选择同时存在于流水线中的指令（在 6600 中就是如此），那这一限制是十分严重的。 C-77

(2) **记分卡的项数**——这一因素决定了流水线为了查找不相关指令可以向前查找多少条指令。这组作为潜在执行对象的指令被称为窗口。记分卡的大小决定了窗口的大小。在这一节，我们假定窗口不会超过一个分支，所以窗口（及记分卡）总是包含来自单个基本模块的直行代码。第 3 章说明如何将窗口扩展到超出一个分支之外。

(3) **功能单元的数目和类型**——这一因素决定了结构性冒险的重要性，它可能会在使用动态调度时增加。 C-78

(4) **存在反相关和输出相关**——它们会导致 WAR 和 WAW 停顿。

第 3 章重点研究了一些方法，用于解决在开发指令级并行（ILP）及更好地利用可用 ILP 时出现的问题。第二个、第三个因素可以通过增加记分卡的大小及功能单元的数目来解决；但是，这些改变可能会增加成本，还可能影响周期时间。WAW 和 WAR 冒险在动态调度处理器中变得更为重要，这是因为流水线暴露了更多的名称相关。如果我们使用采用分支预测方案的动态调度，允许重叠执行一个循环的多个迭代，那么 WAW 冒险也会变得更为重要。 C-79

C.8 谬论与易犯错误

易犯错误 预料外的执行序列可能导致预料外的冒险。

乍看起来，WAW 冒险似乎永远不可能在一个代码序列中出现，因为没有哪个编译器会生成对同一寄存器的两次写入操作，却在中间没有读取操作，但当序列出乎意料之外时，却可能发生 WAW 冒险。例如，第一次写入操作可能在一个选中分支的延迟时隙中，而调度器认为该分支未被选中。下面是可能导致这一情景的代码序列：

```
        BNEZ   R1,foo
        DIV.D  F0,F2,F4; 从未被选中移入延迟时隙
        .....
        .....
foo:    L.D    F0,qrs
```

如果该分支被选中，则在 DIV.D 可以完成之前，L.D 将到达 WB，导致 WAW 冒险。硬件必须检测这一冒险，并暂停发射 L.D。另外一种可能发生这种情景的方式是第二次读取操作存在于陷

陷例程中。一条要写入结果的指令导致陷阱中断，当陷阱处理器中的一条指令完成对同一寄存器的写入之后，原指令继续完成，这时就会发生上述情景。硬件也必须检测并阻止这一情景。

C-80 **易犯错误**　*全面流水化可能影响到设计的其他方面，从而严重降低整体性价比。*

这一现象的最佳示例来自 VAX 的两种实现——8600 和 8700。在 8600 最初交付时，其时钟周期为 80 ns。后来发布了一个名为 8650 的再设计版本，其时钟周期为 55 ns。8700 的流水线要简单得多，工作在微指令级别，从而得到一个较小的 CPU，其时钟更快，周期时间为 45 ns。最后的结果是：8650 在 CPI 方面具有大约 20%的优势，8700 的时钟速率大约快 20%。因此，8700 以少得多的硬件实现了相同性能。

易犯错误　*根据未经优化的代码来评估动态或静态调度。*

与“严格”优化的代码相比，未经优化的代码（包括可以由优化器消除的冗余载入、存储和其他操作）调度起来要容易得多。在调度控制延迟（带有延迟分支）和因为 RAW 冒险所导致的延迟时也是如此。R3000 拥有一个几乎与 C.1 节相同的流水线，在 R3000 上运行的 gcc 中，从未优化的调度后代码到经过优化的调度后代码，空闲时钟周期的频率提高 18%。当然，优化后的程序要快得多，因为其指令数较少。为了公平地评估编译时调度器或运行时动态调度，必须使用优化代码，因为在实际系统中，除了调度之外，还会通过其他优化方法来提高性能。

C.9　结语

在 20 世纪 80 年代早期，流水线技术主要用于超级计算机和价值数百万美元的大型机。到了 20 世纪 80 年代中期，第一批流水线微处理器出现，推动了计算领域的转变，使微处理器在性能上超过小型计算机，最终赶上和超过了大型机。到 20 世纪 90 年代早期，高端嵌入式微处理器也采用了流水线，台式计算机首先开始使用第 3 章讨论的高级动态调度多发射方法。本附录中的材料在 20 世纪 90 年代首先出现时，被认为对研究生来说也是相当高级的内容，现在被看作是非常基础的本科生知识，可以在低于 2 美元处理器中找到相关应用。

C.10　历史回顾与参考文献

附录 L.5 节讨论了流水线与指令级并行的开发，涵盖了本附录及第 3 章中的内容。我们提供了大量参考文献，以便深入阅读和进一步研究这些主题。

C-81

练习（由Diana Franklin更新）

C.1　[15/15/15/15/25/10/15] <A.2>使用以下代码段：

```
Loop:   LD      R1,0(R2)      ;从地址 0+R2 载入 R1
        DADDI   R1,R1,#1      ;R1=R1+1
        SD      R1,0,(R2)     ;将 R1 存储在 0+R2
        DADDI   R2,R2,#4      ;R2=R2+4
        DSUB    R4,R3,R2      ;R4=R3-R2
        BNEZ    R4,Loop       ;如果 R4!=0 则分支到循环
```

假定 R3 的初始值为 R2+396。

a. [15] <C.2>数据冒险是由代码中的数据相关性导致的。相关性是否会导致冒险，取决于机器

实现（比如流水级的数目）。列出上述代码中的所有数据相关。记录寄存器、源指令和目标指令；例如，从 LD 到 DADDI，存在对于寄存器 R1 的数据相关性。

b. [15] <C.2>给出这一指令序列对于 5 级 RISC 流水线的时序，该流水线没有任何转发或旁路硬件，但假定在同一时钟周期中的寄存器读取与写入通过寄存器堆进行“转发”，如图 C-4 所示。请使用如表 C-2 中所示的流水线时序表。假定该分支是通过冲刷流水线来处理的。如果所有存储器引用耗时 1 个周期，这一循环的执行需要多少个周期?

c. [15] <C.2>给出这一指令序列对于拥有完整转发、旁路硬件的 5 级 RISC 流水线的时序。请使用如表 C-2 中所示的流水线时序表。假定在处理分支时，预测它未被选中。如果所有存储器引用耗时 1 个周期，这一循环的执行需要多少个周期?

d. [15] <C.2>给出这一指令序列对于拥有完整转发、旁路硬件的 5 级 RISC 流水线的时序。请使用如表 C-2 中所示的流水线时序表。假定在处理分支时，预测它被选中。如果所有存储器引用耗时 1 个周期，这一循环的执行需要多少个周期?

e. [25] <C.2>高性能处理器拥有很深的流水线——超过 15 级。设想我们拥有一个 10 级流水线，其中 5 级流水线的每一级被分为 2 级。唯一的难题是：对于数据转发操作，数据是由每一对流水级的末尾转发到需要这些数据的两个流水线的开头。例如，数据从第二执行级的输出转发到第一执行级的输入，仍然导致 1 个周期的延迟。对于一个拥有完整转发、旁路硬件的 10 级 RISC 流水线，给出这一指令序列的时序。请使用如表 C-2 所示的流水线时序表。假定在处理分支时，预测它被选中。如果所有存储器引用耗时 1 个周期，这一循环的执行需要多少个周期? C-82

f. [10] <C.2>假定在一个 5 级流水线中，最长的流水级需要 0.8 ns，流水线寄存器延迟为 0.1 ns。这个 5 级流水线的时钟周期时间为多少？如果 10 级流水线将所有流水级都分为两半，那么 10 级机器的周期时间为多少呢?

g. [15] <C.2>利用第(d)、(e)部分的答案，判断该循环在 5 级流水线和 10 级流水线上的每指令周期数（CPI）。确保仅计算从第一条指令到达写回级再到最后的周期数。不要计算第一条指令的开始时间。利用第(f)部分计算的时钟周期数，计算每种机器的平均指令执行时间。

C.2 [15/15] <C.2>假定分支频率如下所示（以占全部指令的百分比表示）：

条件分支 15%

跳转与调用 1%

选中条件分支 60%被选中

a. [15] <C.2>我们正在研究一个深度为 4 的流水线，其中，无条件分支在第二周期结束时执行，而条件分支则在第三个周期结束时执行。假定仅第一个流水级总会完成，与是否执行该分支无关，忽略其他流水线停顿，在没有分支冒险的情况下，该机器的速度快多少?

b. [15] <C.2>现在假定有一个高性能处理器，其中有一个深度为 15 的流水线，无条件分支在第五周期结束时执行，条件分支在第十周期结束时执行。假定仅第一个流水级总会完成，与是否执行该分支无关，忽略其他流水线停顿，在没有分支冒险的情况下，该机器的速度快多少?

C.3 [5/15/10/10] <C.2>我们首先考虑一个采用单周期实现的计算机。在按功能分割流水级时，这些流水级需要的数目不一定相同。原机器的时钟周期时间为 7 ns。在流水线被分割之后，测得的时间数据为：IF，1 ns；ID，1.5 ns；EX，1 ns；MEM，2 ns；WB，1.5 ns。流水线寄存器延迟为 0.1 ns。

a. [5] <C.2>5 级流水化机器的时钟周期时间为多少?

b. [15] <C.2>如果每 4 条指令有一次停顿，新机器的 CPI 为多少?

c. [10] <C.2>流水化机器相对于单周期机器的加速比为多少?

d. [10] <C.2>如果流水化机器有无限个流水级，那它相对于单周期机器的加速比为多少? C-83

C.4 [15] <C.1、C.2>典型 5 级 RISC 流水线的精简硬件实现可能使用 EX 级硬件来执行分支指令对比，

分支指令会在某一时钟周期而到达 MEM 级，在此时钟周期之前，不会将分支目标 PC 实际提交给 IF 级。通过求解 ID 中的分支指令可以缩减控制冒险停顿，但在某一方面的性能提升可能会降低其他情况下的性能。写一小段代码，在此代码中计算 ID 级的分支时会导致数据冒险，甚至在拥有数据转发时也是如此。

C.5 [12/13/20/20/15/15] <C.2、C.3>对于这些问题，我们将研究一种寄存器-存储器体系结构的流水线。这一体系结构有两种指令格式：寄存器-寄存器格式和寄存器-存储器格式。存在一种单存储器寻址方式（偏移量+基地寄存器）。还有一组采用以下格式的 ALU 运算：

```
ALUop Rdest, Rsrc1, Rsrc2
```

或

```
ALUop Rdest, Rsrc1, MEM
```

其中 ALUop 是以下指令之一：加、减、AND、OR、载入（忽略 Rsrc 1）或存储。Rsrc 或 Rdest 为寄存器。MEM 是基址寄存器和偏移量对。分支对两个寄存器进行全面对比，采用 PC 相对寻址。假定此机器实现了流水化，从而在每个时钟周期都会启动一条新指令。此流水线结构类似于 VAX 8700 微流水线中使用的结构[Clark 1987]，为：

IF	RF	ALU1	MEM	WB					
	IF	RF	ALU1	MEM	ALU2	WB			
		IF	RF	ALU1	MEM	ALU2	WB		
			IF	RF	ALU1	MEM	ALU2	WB	
				IF	RF	ALU1	MEM	ALU2	WB

第一个 ALU 流水级用于为存储器引用和分支的计算实际地址。第二个 ALU 周期用于运算和分支比较。RF 既是解码周期也是寄存器提取周期。假定当在同一时钟周期对同一寄存器进行读取和写入时，将转发所写数据。

a. [12] <C.2>求出所需加法器的个数，包括所有加法器和递增器；给出一组指令和流水级，证明这一答案的合理性。只需要给出一种使加法器数目最大的组合方式。

b. [13] <C.2>求出所需要的寄存器读、写端口数目以及存储器读、写端口。给出指令和流水级的一种组合方式，以证明你的答案是正确的，要指出指令以及该指令所需要的读取端口和写入端口。

C-84

c. [20] <C.3>判断任何 ALU 所需要的数据转发。假定 ALU1 和 ALU2 流水级拥有独立的 ALU。在 ALU 之间放入为避免或减免停顿所需要的全部转发。以表 C-12 的形式，给出转发所涉及的两条指令之间的关系，忽略表中最后两列。要仔细考虑跨越中间指令的转发，比如，

```
ADD          R1, ...
任意指令
ADD          ..., R1, ...
```

d. [20] <C.3>当源单元或目标单元不是 ALU 时，给出为避免和减少停顿所需要的全部数据转发。使用表 C-12 中的相同格式，再次忽略最后两列。别忘了向（自）存储器引用的转发。

e. [15] <C.3>给出所有其他符合以下条件的冒险：至少有一个源单元或目标单元不是 ALU。使用表 C-11 所示的表格，但用冒险的长度来代替最后一列。

f. [15] <C.2>以示例方式给出所有控制冒险，并列出停顿的长度。使用表 C-4 所示的格式，标记出每个示例。

C.6 [12/13/13/15/15] <C.1、C.2、C.3>我们现在将向经典 5 级 RISC 流水线中添加对寄存器-存储器 ALU 运算的支持。为了抵消复杂性的增长，所有存储器寻址都限于寄存器间接寻址（例如，所有地址都只是保存在寄存器中的值；没有向寄存器值添加偏移量或位移）。例如，寄存器-存储器指令 ADD R4, R5 (R1)表示将寄存器 R5 的内容添加到某一存储器位置（其地址等于寄存器 R1

中的值）的值，并将和值放到寄存器 R4 中。寄存器-寄存器 ALU 操作不变。以下各项适用于整数 RISC 流水线。

a. [12] <C.1>列出 RISC 流水线五个传统流水级重新排列后的顺序，该流水线将支持由寄存器间接寻址独占实现的寄存器-存储器操作。

b. [13] <C.2、C.3>给出重排序流水线需要哪些新转发路径，列出每条新路径的来源、目的地及其上面传送的信息。

c. [13] <C.2、C.3>对于 RISC 流水线重排序后的流水级，这种寻址模式生成了哪些新数据冒险？给出一段指令序列，阐明每种新冒险。

d. [15] <C.3>对于一个给定程序，与原 RISC 流水线相比，拥有寄存器-存储器 ALU 运算的 RISC 流水线的指令数可能不同，列出所有这些差别。给出一对特定的指令序列，一个用于原流水线，一个用于重新排序后的流水线，以说明每种差别方式。

e. [15] <C.3>假定所有指令在每个流水线上花费 1 个时钟周期。与原 RISC 流水线相比，寄存器-存储器 RISC 对一个给定程序的 CPI 可能会有所不同，列出所有这些差别。 C-85

C.7 [10/10] <C.3>在这个问题中，我们将研究流水线的加深如何以两种不同方式来影响性能：加快时钟周期，因为数据与控制冒险而延长停顿。假定原机器是一个 5 级流水线，其时钟周期为 1 ns。第二种机器为 12 级流水线，时钟周期为 0.6 ns。由于数据冒险，5 级流水线每 5 条指令经历 1 次停顿，而 12 级流水线每 8 条指令经历 3 次停顿。此外，分支占全部指令的 20%，两台机器的错误预测率都是 5%。

a. [10] <C.3>仅考虑数据冒险，12 级流水线相对于 5 级流水线的加速比为多少？

b. [10] <C.3>如果第一台机器的分支错误预测代价为 2 个周期，而第二台机器为 5 个周期，则每种机器的 CPI 为多少？由于分支错误预测而导致的停顿考虑在内。

C.8 [15] <C.5>绘制一个表格，采用表 C-12 所示的格式，说明 R4000 整数流水线的转发逻辑。仅包含我们在表 C-12 中考虑的 MIPS 指令。

C.9 [15] <C.5>绘制一个表格，采用表 C-11 所示的格式，说明 R4000 整数冒险检测。仅包含我们在表 C-12 中考虑的 MIPS 指令。

C.10 [25] <C.5>假定 MIPS 仅有一个寄存器组。使用表 C-12 的格式，构造浮点和整数指令的转发表。忽略浮点除法与整数除法。

C.11 [15] <C.5>构造一个类似于如表 C-11 所示的表格，检查图 C-18 中 MIPS 浮点流水线中的 WAW 停顿。不考虑浮点除法。

C.12 [20/22/22] <C.4、C.6>在这个练习中，我们将研究一个常见向量循环如何在 MIPS 流水线的静态、动态调度版本上运行。这个循环就是所谓的 DAXPY 循环（在附录 G 中进行了全面讨论），它是高斯消去法中的核心运算。该循环对于一个长度为 100 的向量实现了向量运算 $Y=a*X+Y$。下面是该循环的 MIPS 代码：

```
foo:    L.D     F2, 0(R1)        ; 载入 X(i)
        MUL.D   F4, F2, F0       ; 求乘积 a*X(i)
        L.D     F6, 0($2)        ; 载入 Y(i)
        ADD.D   F6, F4, F6       ; 求和 a*X(i) + Y(i)
        S.D     0(R2), F6        ; 存储 Y(i)
        DADDIU  R1, R1, #8       ; 递增 X 索引
        DADDIU  R2, R2, #8       ; 递增 Y 索引
        SGTIU   R3, R1, done     ; 测试是否完成
        BEQZ    R3, foo          ; 如果没有完成则继续循环
```

对于第(a)部分至第(c)部分，假定整数运算在一个时钟周期内发射和完成（包括载入），它们的结果被完全旁路。忽略分支延迟。（仅）使用表 C-17 所示的浮点延迟，但假定浮点单元被完全流水化。对于以下记分卡，假定一个等待另一功能单元结果的指令可以在写入该结果的同时读取操作数。另外假定一个指令正在执行 WR，它允许在自己执行 WR 的同一时钟周期内发射另 C-86

外一条指令，而所发射的这一指令可以与正在执行 WR 的指令等待同一功能单元。

a. [20] <C.5>对于这个问题，使用 C.5 节的 MIPS 流水线，其流水线延迟如表 C-17 所示，但浮点单元实现了完全流水化，所以启动间隔为 1。画出与表 C-19 类似的时序图，显示每条指令的执行时序。每个循环迭代耗用多少个时钟周期？从第一条指令进入 WB 级开始计算，直到最后一条指令进入 WB 级为止。

b. [22] <C.6>使用上面 DAXPY 的 MIPS 代码，给出当 SGTIU 指令到达写结果阶段时记分卡表的状态（如表 C-31 所示）。假定发射和读取操作数各需要 1 个周期。假定有一个整数功能单元仅需要一个执行周期（延迟为 0 个周期，包括载入和存储）。假定采用图 C-25 的浮点单元配置，其浮点延迟如表 C-17 所示。该分支不应包含在记分卡中。

c. [22] <C.6>使用上面 DAXPY 的 MIPS 代码，假定一个记分卡具有图 C-25 所述的浮点功能单元，再加上一个整数功能单元（也用于载入-存储）。假定延迟如表 C-34 所示。给出第二次分支发射时的记分卡状态（如表 C-31 所示）。假定分支被正确预测为选中，耗时 1 个周期。每次循环迭代需要多少个时钟周期？可以忽略所有寄存器端口/总线冲突。

表C-34 流水线延迟（其中延迟用时钟周期数表示）

生成结果的指令	使用结果的指令	延迟（单位：时钟周期）
浮点乘	浮点ALU操作	6
浮点加	浮点ALU操作	4
浮点乘	浮点存储	5
浮点加	浮点存储	3
整数运算（包括载入）	任意	0

C.13 [25] <C.8>由于 WAR 冒险需要暂停正在执行写入的指令，直到读取指令的指令开始执行为止，而 RAW 冒险需要延迟正在进行读取的指令，直到正在进行写入的指令完成为止，正好与 WAR 冒险相反，因此，记分卡能够区分 RAW 和 WAR 冒险是非常重要的。例如，考虑以下序列：

C-87

```
MUL.D     F0,F6,F4
DSUB.D    F8,F0,F2
ADD.D     F2,F10,F2
```

DSUB.D 依赖于 MUL.D（一次 RAW 冒险），因此，必须允许 MUL.D 在 DSUB.D 之前完成。如果由于不能区别 RAW 和 WAR 冒险，MUL.D 因为 DSUB.D 而停顿，处理器将会死锁。这一序列包含 ADD.D 与 DSUB.D 之间的 WAR 冒险，在 DSUB.D 开始执行之前，不允许 ADD.D 完成。难度在于区分 MUL.D 和 DSUB.D 之间的 RAW 冒险和 DSUB.D 和 ADD.D 之间的 WAR 冒险。为了明白为什么三指令情景非常重要，逐级跟踪每条指令的处理：发射、读取操作数、执行和写结果。假定 MUL.D 指令需要 3 个时钟周期来执行，DSUB.D 和 ADD.D 指令各需要 1 个周期来执行。最后，假定处理器有两个乘法功能单元和两个加法功能单元。给出跟踪过程如下。

(1) 制作一张表，各列的标题分别为：指令、发射、读取操作数、执行、写结果、备注。在第一列中，按程序顺序列出这些指令（指令之间留出足够的空间，表格的单元格越大，越容易保存分析结果）。首先在 MUL.D 指令列的“发射”列中写下 1，表明 MUL.D 在时钟周期 1 中完成发射级。现在，填充表中各个流水级列，直到记分卡首次暂停一条指令的周期为止。

(2) 对于停顿指令，在适当的表列中写下如下文字“在时钟周期 X 等待”，其中 X 是当前时钟周期的编号，以表明该记分卡正在通过停顿该流水级来解决 RAW 或 WAR 冒险。在“备注”列中，写出是哪种类型的冒险和哪个相关指令导致等待。

(3) 向“正在等待”的表项添加文字“在时钟周期 Y 完成”，填充表中其余各项，直到所有指令完成为止。对于一条被暂停的指令，在“备注”列中添加一项备注，说明该等待为什么结

束以及如何避免死锁。（提示：考虑一下如何避免 WAW 冒险，以及它对活动指令序列意味着什么。）注意，三条指令的完成顺序与其程序顺序的对比。

C.14 [10/10/10] <C.5>在这个问题中，需要编写一系列短小代码，说明在使用具有不同延迟的功能单元时所导致的发射。对于每个代码段，画出与表 C-20 类似的时序表，说明每个概念，明确指出问题所在。

a. [10] <C.5>展示当硬件中仅有一个 MEM 和 WB 级时的结构性冒险，所使用的代码应当不同于表 C-20 中使用的代码。

b. [10] <C.5>展示需要停顿的 WAW 冒险。 C-88

参考文献

Adve, S. V., and K. Gharachorloo [1996]. "Shared memory consistency models: A tutorial,"*IEEE Computer* 29:12 (December), 66–76.

Adve, S. V., and M. D. Hill [1990]. "Weak ordering—a new definition," *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash.,2–14.

Agarwal, A. [1987]. "Analysis of Cache Performance for Operating Systems and Multiprogramming," Ph.D. thesis, Tech. Rep. No. CSL-TR-87-332, Stanford University, Palo Alto, Calif.

Agarwal, A. [1991]. "Limits on interconnection network performance," *IEEE Trans. on Parallel and Distributed Systems* 2:4 (April), 398–412.

Agarwal, A., and S. D. Pudar [1993]. "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," *20th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 16–19, 1993, San Diego, Calif. Also appears in *Computer Architecture News* 21:2 (May), 179–190, 1993.

Agarwal, A., R. Bianchini, D. Chaiken, K. Johnson, and D. Kranz [1995]. "The MIT Alewife machine: Architecture and performance," *Int'l. Symposium on Computer Architecture* (Denver, Colo.), June, 2–13.

Agarwal, A., J. L. Hennessy, R. Simoni, and M. A. Horowitz [1988]. "An evaluation of directory schemes for cache coherence," *Proc. 15th Int'l. Symposium on Computer Architecture* (June), 280–289.

Agarwal, A., J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin [1993]. "Sparcle: An evolutionary processor design for large-scale multiprocessors," *IEEE Micro* 13 (June), 48–61.

Agerwala, T., and J. Cocke [1987]. High Performance Reduced Instruction Set Processors, IBM Tech. Rep. RC12434, IBM, Armonk, N.Y.

Akeley, K. and T. Jermoluk [1988]. "High-Performance Polygon Rendering," *Proc. 15th Annual Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH 1988)*, August 1–5, 1988, Atlanta, Ga., 239–246.

Alexander, W. G., and D. B. Wortman [1975]. "Static and dynamic characteristics of XPL programs," *IEEE Computer* 8:11 (November), 41–46.

Alles, A. [1995]. "ATM Internetworking," White Paper (May), Cisco Systems, Inc., San Jose, Calif. (*www.cisco.com/warp/public/614/12.html*).

Alliant. [1987]. *Alliant FX/Series: Product Summary*, Alliant Computer Systems Corp., Acton, Mass.

Almasi, G. S., and A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, Calif.

Alverson, G., R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith [1992]. "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," *Proc. ACM/IEEE Conf. on Supercomputing*, November 16–20, 1992, Minneapolis, Minn.,188–197.

Amdahl, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS*

Spring Joint Computer Conf., April 18–20, 1967, Atlantic City, N.J., 483–485.

Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr. [1964]. "Architecture of the IBM System 360," *IBM J. Research and Development* 8:2 (April), 87–101.

Amza, C., A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel [1996]. "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer* 29:2 (February), 18–28.

Anderson, D. [2003]. "You don't know jack about disks," *Queue*, 1:4 (June), 20–30.

Anderson, D., J. Dykes, and E. Riedel [2003]. "SCSI vs. ATA—More than an interface," *Proc. 2nd USENIX Conf. on File and Storage Technology (FAST '03)*, March 31–April 2, 2003, San Francisco.

Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo [1967]. "The IBM 360 Model 91: Processor philosophy and instruction handling," *IBM J. Research and Development* 11:1 (January), 8–24.

Anderson, M. H. [1990]. "Strength (and safety) in numbers (RAID, disk storage technology)," *Byte* 15:13 (December), 337–339.

Anderson, T. E., D. E. Culler, and D. Patterson [1995]. "A case for NOW (networks of workstations)," *IEEE Micro* 15:1 (February), 54–64.

Ang, B., D. Chiou, D. Rosenband, M. Ehrlich, L. Rudolph, and Arvind [1998]. "StarTVoyager: A flexible platform for exploring scalable SMP issues," *Proc. ACM/IEEE Conf. on Supercomputing*, November 7–13, 1998, Orlando, FL.

Anjan, K. V., and T. M. Pinkston [1995]. "An efficient, fully-adaptive deadlock recovery scheme: Disha," *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy.

Anon. et al. [1985]. *A Measure of Transaction Processing Power*, Tandem Tech. Rep. TR85.2. Also appears in *Datamation* 31:7 (April), 112–118, 1985.

Apache Hadoop. [2011]. *http://hadoop.apache.org*.

Archibald, J., and J.-L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* 4:4 (November), 273–298.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia [2009]. *Above the Clouds: A Berkeley View of Cloud Computing*, Tech. Rep. UCB/EECS-2009-28, University of California, Berkeley (*http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html*).

Arpaci, R. H., D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick [1995]. "Empirical evaluation of the CRAY-T3D: A compiler perspective," *22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy.

Asanovic, K. [1998]. "Vector Microprocessors," Ph.D. thesis, Computer Science Division, University of California, Berkeley.

Associated Press. [2005]. "Gap Inc. shuts down two Internet stores for major overhaul," *USATODAY.com*, August 8, 2005.

Atanasoff, J. V. [1940]. *Computing Machine for the Solution of Large Systems of Linear Equations*, Internal Report, Iowa State University, Ames.

Atkins, M. [1991]. Performance and the i860 Microprocessor, *IEEE Micro*, 11:5 (September), 24–27, 72–78.

Austin, T. M., and G. Sohi [1992]. "Dynamic dependency analysis of ordinary programs," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 342–351.

Babbay, F., and A. Mendelson [1998]. "Using value prediction to increase the power of speculative execution hardware," *ACM Trans. on Computer Systems* 16:3 (August), 234–270.

Baer, J.-L., and W.-H. Wang [1988]. "On the inclusion property for multi-level cache hierarchies," *Proc. 15th Annual Int'l. Symposium on Computer Architecture*, May 30–June 2, 1988, Honolulu, Hawaii, 73–80.

Bailey, D. H., E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga [1991]. "The NAS parallel benchmarks," *Int'l. J. Supercomputing Applications* 5, 63–73.

Bakoglu, H. B., G. F. Grohoski, L. E. Thatcher, J. A. Kaeli, C. R. Moore, D. P. Tattle, W. E. Male, W. R. Hardell, D. A. Hicks, M. Nguyen Phu, R. K. Montoye, W. T. Glover, and S. Dhawan [1989]. "IBM second-generation RISC processor organization," *Proc. IEEE Int'l. Conf. on Computer Design*, September 30–October 4, 1989, Rye, N.Y., 138–142.

Balakrishnan, H., V. N. Padmanabhan, S. Seshan, and R. H. Katz [1997]. "A comparison of mechanisms for improving TCP performance over wireless links," *IEEE/ACM Trans. on Networking* 5:6 (December), 756–769.

Ball, T., and J. Larus [1993]. "Branch prediction for free," *Proc. ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, June 23–25, 1993, Albuquerque, N.M., 300–313.

Banerjee, U. [1979]. "Speedup of Ordinary Programs," Ph.D. thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, and R. Neugebauer [2003]. "Xen and the art of virtualization," *Proc. of the 19th ACM Symposium on Operating Systems Principles*, October 19–22, 2003, Bolton Landing, N.Y.

Barroso, L. A. [2010]. "Warehouse Scale Computing [keynote address]," *Proc. ACM SIGMOD*, June 8–10, 2010, Indianapolis, Ind.

Barroso, L. A., and U. Hölzle [2007], "The case for energy-proportional computing," *IEEE Computer*, 40:12 (December), 33–37.

Barroso, L. A., and U. Hölzle [2009]. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan & Claypool, San Rafael, Calif.

Barroso, L. A., K. Gharachorloo, and E. Bugnion [1998]. "Memory system characterization of commercial workloads," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 3–14.

Barton, R. S. [1961]. "A new approach to the functional design of a computer," *Proc Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., 393–396.

Bashe, C. J., W. Buchholz, G. V. Hawkins, J. L. Ingram, and N. Rochester [1981]. "The architecture of IBM's early computers," *IBM J. Research and Development* 25:5 (September), 363–375.

Bashe, C. J., L. R. Johnson, J. H. Palmer, and E. W. Pugh [1986]. *IBM's Early Computers*, MIT Press, Cambridge, Mass.

Baskett, F., and T. W. Keller [1977]. "An evaluation of the Cray-1 processor," in *High Speed Computer and Algorithm Organization*, D. J. Kuck, D. H. Lawrie, and A. H. Sameh, eds., Academic Press, San Diego, 71–84.

Baskett, F., T. Jermoluk, and D. Solomon [1988]. "The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 10,000 lighted polygons per second," *Proc. IEEE COMPCON*, February 29–March 4, 1988, San Francisco, 468–471.

BBN Laboratories. [1986]. *Butterfly Parallel Processor Overview*, Tech. Rep. 6148, BBN Laboratories, Cambridge, Mass.

Bell, C. G. [1984]. "The mini and micro industries," *IEEE Computer* 17:10 (October), 14–30.

Bell, C. G. [1985]. "Multis: A new class of multiprocessor computers," *Science* 228 (April 26), 462–467.

Bell, C. G. [1989]. "The future of high performance computers in science and engineering," *Communications of the ACM* 32:9 (September), 1091–1101.

Bell, G., and J. Gray [2001]. *Crays, Clusters and Centers*, Tech. Rep. MSR-TR-2001-76, Microsoft Research, Redmond, Wash.

Bell, C. G., and J. Gray [2002]. "What's next in high performance computing?" *CACM* 45:2 (February), 91–95.

Bell, C. G., and A. Newell [1971]. *Computer Structures: Readings and Examples*, McGraw-Hill, New York.

Bell, C. G., and W. D. Strecker [1976]. "Computer structures: What have we learned from the PDP-11?," *Third Annual Int'l. Symposium on Computer Architecture (ISCA)*, January 19–21, 1976, Tampa, Fla., 1–14.

Bell, C. G., and W. D. Strecker [1998]. "Computer structures: What have we learned from the PDP-11?" *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ACM, New York, 138–151.

Bell, C. G., J. C. Mudge, and J. E. McNamara [1978]. *A DEC View of Computer Engineering*, Digital Press, Bedford, Mass.

Bell, C. G., R. Cady, H. McFarland, B. DeLagi, J. O'Laughlin, R. Noonan, and W. Wulf [1970]. "A new architecture for mini-computers: The DEC PDP-11," *Proc. AFIPS Spring Joint Computer Conf.*, May 5–May 7, 1970, Atlantic City, N.J., 657–675.

Benes, V. E. [1962]. "Rearrangeable three stage connecting networks," *Bell System Technical Journal* 41, 1481–1492.

Bertozzi, D., A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli [2005]. "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Trans. on Parallel and Distributed Systems* 16:2 (February), 113–130.

Bhandarkar, D. P. [1995]. *Alpha Architecture and Implementations*, Digital Press, Newton, Mass.

Bhandarkar, D. P., and D. W. Clark [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Palo Alto, Calif., 310–319.

Bhandarkar, D. P., and J. Ding [1997]. "Performance characterization of the Pentium Pro processor," *Proc. Third Int'l. Symposium on High-Performance Computer Architecture*, February 1–February 5, 1997, San Antonio, Tex., 288–297.

Bhuyan, L. N., and D. P. Agrawal [1984]. "Generalized hypercube and hyperbus structures for a computer network," *IEEE Trans. on Computers* 32:4 (April), 322–333.

Bienia, C., S. Kumar, P. S. Jaswinder, and K. Li [2008]. *The Parsec Benchmark Suite: Characterization and Architectural Implications*, Tech. Rep. TR-811-08, Princeton University, Princeton, N.J.Bier, J. [1997]. "The Evolution of DSP

Processors," presentation at Univesity of California, Berkeley, November 14.

Bird, S., A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru [2007]. "Characterization of performance of SPEC CPU benchmarks on Intel's Core Microarchitecture based processor," *Proc. 2007 SPEC Benchmark Workshop*, January 21, 2007, Austin, Tex.

Birman, M., A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barnes [1990]."Developing the WRL3170/3171 SPARC floating-point coprocessors," *IEEE Micro* 10:1, 55–64.

Blackburn, M., R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann [2006]. "The DaCapo benchmarks: Java benchmarking development and analysis," *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 22–26, 2006, 169–190.

Blaum, M., J. Bruck, and A. Vardy [1996]. "MDS array codes with independent parity symbols," *IEEE Trans. on Information Theory*, IT-42 (March), 529–42.

Blaum, M., J. Brady, J. Bruck, and J. Menon [1994]. "EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago, 245–254.

Blaum, M., J. Brady, J. Bruck, and J. Menon [1995]. "EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures," *IEEE Trans. on Computers* 44:2 (February), 192–202.

Blaum, M., J. Brady, J., Bruck, J. Menon, and A. Vardy [2001]. "The EVENODD code and its generalization," in H. Jin, T. Cortes, and R. Buyya, eds., *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, Wiley–IEEE, New York, 187–208.

Bloch, E. [1959]. "The engineering design of the Stretch computer," *1959 Proceedings of the Eastern Joint Computer Conf.*, December 1–3, 1959, Boston, Mass., 48–59.

Boddie, J. R. [2000]. "History of DSPs," *www.lucent.com/micro/dsp/dsphist.html.*

Bolt, K. M. [2005]. "Amazon sees sales rise, profit fall," *Seattle Post-Intelligencer*, October 25 (http://seattlepi.nwsource.com/business/245943_techearns26.html).

Bordawekar, R., U. Bondhugula, R. Rao [2010]. "Believe It or Not!: Multi-core CPUs can Match GPU Performance for a FLOP-Intensive Application!" *19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010)*. Vienna,Austria, September 11–15, 2010, 537–538.

Borg, A., R. E. Kessler, and D. W. Wall [1990]. "Generation and analysis of very long address traces," *19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 270–279.

Bouknight, W. J., S. A. Deneberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick [1972]. "The Illiac IV system," *Proc. IEEE* 60:4, 369–379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 306–316.

Brady, J. T. [1986]. "A theory of productivity in the creative process," *IEEE CG&A* (May), 25–34.

Brain, M. [2000]. "Inside a Digital Cell Phone," *www.howstuffworks.com/insidecellphone.htm.*

Brandt, M., J. Brooks, M. Cahir, T. Hewitt, E. Lopez-Pineda, and D. Sandness [2000]. *The Benchmarker's Guide for Cray SV1 Systems*. Cray Inc., Seattle, Wash.

Brent, R. P., and H. T. Kung [1982]. "A regular layout for parallel adders," *IEEE Trans. on Computers* C-31, 260–264.

Brewer, E. A., and B. C. Kuszmaul [1994]. "How to get good performance from the CM-5 data network," *Proc. Eighth Int'l. Parallel Processing Symposium*, April 26–27, 1994,Cancun, Mexico.

Brin, S., and L. Page [1998]. "The anatomy of a large-scale hypertextual Web search engine," *Proc. 7th Int'l. World Wide Web Conf.*, April 14–18, 1998, Brisbane, Queensland, Australia, 107–117.

Brown, A., and D. A. Patterson [2000]. "Towards maintainability, availability, and growth benchmarks: A case study of software RAID systems." *Proc. 2000 USENIX Annual Technical Conf.*, June 18–23, 2000, San Diego, Calif.

Bucher, I. V., and A. H. Hayes [1980]. "I/O performance measurement on Cray-1 and CDC 7000 computers," *Proc. Computer Performance Evaluation Users Group, 16th Meeting*, NBS 500-65, 245–254.

Bucher, I. Y. [1983]. "The computational speed of supercomputers," *Proc. Int'l. Conf. on Measuring and Modeling of Computer Systems (SIGMETRICS 1983)*, August 29–31,1983, Minneapolis, Minn., 151–165.

Bucholtz, W. [1962]. *Planning a Computer System: Project Stretch*, McGraw-Hill, New York.

Burgess, N., and T. Williams [1995]. "Choices of operand truncation in the SRT division algorithm," *IEEE Trans. on Computers* 44:7, 933–938.

Burkhardt III, H., S. Frank, B. Knobe, and J. Rothnie [1992]. *Overview of the KSR1 Computer System*, Tech. Rep. KSR-TR-9202001, Kendall Square Research, Boston, Mass.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, Mass., and Tomash Publishers, Los Angeles, Calif., 1987, 97–146.

Calder, B., G. Reinman, and D. M. Tullsen [1999]. "Selective value prediction," *Proc. 26th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 2–4, 1999, Atlanta, Ga.

Calder, B., D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn [1997]. "Evidence-based static branch prediction using machine learning," *ACM Trans. Program. Lang. Syst.* 19:1, 188–222.

Callahan, D., J. Dongarra, and D. Levine [1988]. "Vectorizing compilers: A test suite and results," *Proc. ACM/IEEE Conf. on Supercomputing*, November 12–17, 1988, Orland, Fla., 98–105.

Cantin, J. F., and M. D. Hill [2001]. "Cache Performance for Selected SPEC CPU2000 Benchmarks," *www.jfred.org/cache-data.html* (June).

Cantin, J. F., and M. D. Hill [2003]. "Cache Performance for SPEC CPU2000 Benchmarks, Version 3.0," *www.cs.wisc.edu/multifacet/misc/spec2000cache-data/index.html.*

Carles, S. [2005]. "Amazon reports record Xmas season, top game picks," *Gamasutra*, December 27 (*http://www.gamasutra.com/php-bin/news_index.php?story=7630.*)

Carter, J., and K. Rajamani [2010]. "Designing energy-efficient servers and data centers," *IEEE Computer* 43:7 (July), 76–78.

Case, R. P., and A. Padegs [1978]. "The architecture of the IBM System/370," *Communications of the ACM* 21:1, 73–96. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 830–855.

Censier, L., and P. Feautrier [1978]. "A new solution to coherence problems in multicache systems," *IEEE Trans. on Computers* C-27:12 (December), 1112–1118.

Chandra, R., S. Devine, B. Verghese, A. Gupta, and M. Rosenblum [1994]. "Scheduling and page migration for multiprocessor compute servers," *Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 4–7, 1994, San Jose, Calif., 12–24.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber [2006]. "Bigtable: A distributed storage system for structured data," *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, November 6–8, 2006, Seattle, Wash.

Chang, J., J. Meza, P. Ranganathan, C. Bash, and A. Shah [2010]. "Green server design: Beyond operational energy to sustainability," *Proc. Workshop on Power Aware Computing and Systems (HotPower '10)*, October 3, 2010, Vancouver, British Columbia.

Chang, P. P., S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu [1991]. "IMPACT: An architectural framework for multiple-instruction-issue processors," *18th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 27–30, 1991, Toronto, Canada, 266–275.

Charlesworth, A. E. [1981]. "An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family," *Computer* 14:9 (September), 18–27.

Charlesworth, A. [1998]. "Starfire: Extending the SMP envelope," *IEEE Micro* 18:1 (January/February), 39–49.

Chen, P. M., and E. K. Lee [1995]. "Striping in a RAID level 5 disk array," *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 15–19, 1995, Ottawa, Canada, 136–145.

Chen, P. M., G. A. Gibson, R. H. Katz, and D. A. Patterson [1990]. "An evaluation of redundant arrays of inexpensive disks using an Amdahl 5890," *Proc.ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 22–25, 1990, Boulder, Colo.

Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson [1994]. "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys* 26:2 (June), 145–188.

Chen, S. [1983]. "Large-scale and high-speed multiprocessor system for scientific applications," *Proc. NATO Advanced Research Workshop on High-Speed Computing*, June 20–22, 1983, Jülich, West Germany. Also appears in K. Hwang, ed.,

"Superprocessors: Design and applications," *IEEE* (August), 602–609, 1984.

Chen, T. C. [1980]. "Overlap and parallel processing," in H. Stone, ed., *Introduction to Computer Architecture*, Science Research Associates, Chicago, 427–486.

Chow, F. C. [1983]. "A Portable Machine-Independent Global Optimizer—Design and Measurements," Ph.D. thesis, Stanford University, Palo Alto, Calif.

Chrysos, G. Z., and J. S. Emer [1998]. "Memory dependence prediction using store sets," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 142–153.

Clark, B., T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Neefe Matthews [2004]. "Xen and the art of repeated research," *Proc. USENIX Annual Technical Conf.*, June 27–July 2, 2004, 135–144.

Clark, D. W. [1983]. "Cache performance of the VAX-11/780," *ACM Trans. on Computer Systems* 1:1, 24–37.

Clark, D. W. [1987]. "Pipelining and performance in the VAX 8800 processor," *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 173–177.

Clark, D. W., and J. S. Emer [1985]. "Performance of the VAX-11/780 translation buffer: Simulation and measurement," *ACM Trans. on Computer Systems* 3:1 (February), 31–62.

Clark, D., and H. Levy [1982]. "Measurement and analysis of instruction set use in the VAX-11/780," *Proc. Ninth Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 26–29, 1982, Austin, Tex., 9–17.

Clark, D., and W. D. Strecker [1980]. "Comments on 'the case for the reduced instruction set computer,'" *Computer Architecture News* 8:6 (October), 34–38.

Clark, W. A. [1957]. "The Lincoln TX-2 computer development," *Proc. Western Joint Computer Conference*, February 26–28, 1957, Los Angeles, 143–145.

Clidaras, J., C. Johnson, and B. Felderman [2010]. Private communication.

Climate Savers Computing Initiative. [2007]. "Efficiency Specs," *http://www. climatesaverscomputing.org/.*

Clos, C. [1953]. "A study of non-blocking switching networks," *Bell Systems Technical Journal* 32 (March), 406–424.

Cody, W. J., J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson [1984]. "A proposed radix- and wordlengthindependent standard for floating-point arithmetic," *IEEE Micro* 4:4, 86–100.

Colwell, R. P., and R. Steck [1995]. "A 0.6 μm BiCMOS processor with dynamic execution." *Proc. of IEEE Int'l. Symposium on Solid State Circuits (ISSCC)*, February 15–17, 1995, San Francisco, 176–177.

Colwell, R. P., R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman [1987]. "A VLIW architecture for a trace scheduling compiler," *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 180–192.

Comer, D. [1993]. *Internetworking with TCP/IP*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

Compaq Computer Corporation. [1999]. *Compiler Writer's Guide for the Alpha 21264*, Order Number EC-RJ66A-TE, June, *www1.support.compaq.com/alpha-tools/ documentation/current/21264_EV67/ec-rj66a_te_comp_writ_gde_for_alpha21264.pdf.*

Conti, C., D. H. Gibson, and S. H. Pitkowsky [1968]. "Structural aspects of the System/ 360 Model 85. Part I. General organization," *IBM Systems J.* 7:1, 2–14.

Coonen, J. [1984]. "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic," Ph.D. thesis, University of California, Berkeley.

Corbett, P., B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar [2004]. "Row-diagonal parity for double disk failure correction," *Proc. 3rd USENIX Conf. on File and Storage Technology (FAST '04)*, March 31–April 2, 2004, San Francisco.

Crawford, J., and P. Gelsinger [1988]. *Programming the 80386*, Sybex Books, Alameda, Calif.

Culler, D. E., J. P. Singh, and A. Gupta [1999]. *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco.

Curnow, H. J., and B. A. Wichmann [1976]. "A synthetic benchmark," *The Computer J.* 19:1, 43–49.

Cvetanovic, Z., and R. E. Kessler [2000]. "Performance analysis of the Alpha 21264- based Compaq ES40 system," *Proc. 27th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 10–14, 2000, Vancouver, Canada, 192–202.

Dally, W. J. [1990]. "Performance analysis of *k*-ary *n*-cube interconnection networks," *IEEE Trans. on Computers* 39:6 (June), 775–785.

Dally, W. J. [1992]. "Virtual channel flow control," *IEEE Trans. on Parallel and Distributed Systems* 3:2 (March), 194–205.

Dally, W. J. [1999]. "Interconnect limited VLSI architecture," *Proc. of the International Interconnect Technology Conference*, May 24–26, 1999, San Francisco.

Dally, W. J., and C. I. Seitz [1986]. "The torus routing chip," *Distributed Computing* 1:4, 187–196.

Dally, W. J., and B. Towles [2001]. "Route packets, not wires: On-chip interconnection networks," *Proc. 38th Design Automation Conference*, June 18–22, 2001, Las Vegas.

Dally, W. J., and B. Towles [2003]. *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, San Francisco.

Darcy, J. D., and D. Gay [1996]. "FLECKmarks: Measuring floating point performance using a full IEEE compliant arithmetic benchmark," CS 252 class project, University of California, Berkeley (see *HTTP.CS.Berkeley.EDU/~darcy/Projects/cs252/*).

Darley, H. M. et al. [1989]. "Floating Point/Integer Processor with Divide and Square Root Functions," U.S. Patent 4,878,190, October 31.

Davidson, E. S. [1971]. "The design and control of pipelined function generators," *Proc. IEEE Conf. on Systems, Networks, and Computers*, January 19–21, 1971, Oaxtepec, Mexico, 19–21.

Davidson, E. S., A. T. Thomas, L. E. Shar, and J. H. Patel [1975]. "Effective control for pipelined processors," *Proc. IEEE COMPCON*, February 25–27, 1975, San Francisco, 181–184. Colwell, R. P., and R. Steck [1995]. "A 0.6 μm BiCMOS processor with dynamic execution." *Proc. of IEEE Int'l. Symposium on Solid State Circuits (ISSCC)*, February 15–17, 1995, San Francisco, 176–177.

Colwell, R. P., R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman [1987]. "A VLIW architecture for a trace scheduling compiler," *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 180–192.

Comer, D. [1993]. *Internetworking with TCP/IP*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

Compaq Computer Corporation. [1999]. *Compiler Writer's Guide for the Alpha 21264*, Order Number EC-RJ66A-TE, June, *www1.support.compaq.com/alpha-tools/ documentation/current/21264_EV67/ec-rj66a_te_comp_writ_gde_for_alpha21264.pdf.*

Conti, C., D. H. Gibson, and S. H. Pitkowsky [1968]. "Structural aspects of the System/ 360 Model 85. Part I. General organization," *IBM Systems J.* 7:1, 2–14.

Coonen, J. [1984]. "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic," Ph.D. thesis, University of California, Berkeley.

Corbett, P., B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar [2004]. "Row-diagonal parity for double disk failure correction," *Proc. 3rd USENIX Conf. on File and Storage Technology (FAST '04)*, March 31–April 2, 2004, San Francisco.

Crawford, J., and P. Gelsinger [1988]. *Programming the 80386*, Sybex Books, Alameda,Calif.

Culler, D. E., J. P. Singh, and A. Gupta [1999]. *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco.

Curnow, H. J., and B. A. Wichmann [1976]. "A synthetic benchmark," *The Computer J.* 19:1, 43–49.

Cvetanovic, Z., and R. E. Kessler [2000]. "Performance analysis of the Alpha 21264- based Compaq ES40 system," *Proc. 27th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 10–14, 2000, Vancouver, Canada, 192–202.

Dally, W. J. [1990]. "Performance analysis of *k*-ary *n*-cube interconnection networks," *IEEE Trans. on Computers* 39:6 (June), 775–785.

Dally, W. J. [1992]. "Virtual channel flow control," *IEEE Trans. on Parallel and Distributed Systems* 3:2 (March), 194–205.

Dally, W. J. [1999]. "Interconnect limited VLSI architecture," *Proc. of the International Interconnect Technology Conference*, May 24–26, 1999, San Francisco.

Dally, W. J., and C. I. Seitz [1986]. "The torus routing chip," *Distributed Computing* 1:4, 187–196.

Dally, W. J., and B. Towles [2001]. "Route packets, not wires: On-chip interconnection networks," *Proc. 38th Design Automation Conference*, June 18–22, 2001, Las Vegas.

Dally, W. J., and B. Towles [2003]. *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, San Francisco.

Darcy, J. D., and D. Gay [1996]. "FLECKmarks: Measuring floating point performance using a full IEEE compliant arithmetic benchmark," CS 252 class project, University of California, Berkeley (see *HTTP.CS.Berkeley.EDU/~darcy/Projects/cs252/*).

Darley, H. M. et al. [1989]. "Floating Point/Integer Processor with Divide and Square Root Functions," U.S. Patent 4,878,190, October 31.

Davidson, E. S. [1971]. "The design and control of pipelined function generators," *Proc. IEEE Conf. on Systems, Networks, and Computers*, January 19–21, 1971, Oaxtepec, Mexico, 19–21.

Davidson, E. S., A. T. Thomas, L. E. Shar, and J. H. Patel [1975]. "Effective control for pipelined processors," *Proc. IEEE COMPCON*, February 25–27, 1975, San Francisco, 181–184.

Duato, J., and T. M. Pinkston [2001]. "A general theory for deadlock-free adaptive routing using a mixed set of resources," *IEEE Trans. on Parallel and Distributed Systems* 12:12 (December), 1219–1235.

Duato, J., S. Yalamanchili, and L. Ni [2003]. *Interconnection Networks: An Engineering Approach*, 2nd printing, Morgan Kaufmann, San Francisco.

Duato, J., I. Johnson, J. Flich, F. Naven, P. Garcia, and T. Nachiondo [2005a]. "A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks," *Proc. 11th Int'l. Symposium on High-Performance Computer Architecture*, February 12–16, 2005, San Francisco.

Duato, J., O. Lysne, R. Pang, and T. M. Pinkston [2005b]. "Part I: A theory for deadlockfree dynamic reconfiguration of interconnection networks," *IEEE Trans. on Parallel and Distributed Systems* 16:5 (May), 412–427.

Dubois, M., C. Scheurich, and F. Briggs [1988]. "Synchronization, coherence, and event ordering," *IEEE Computer* 21:2 (February), 9–21.

Dunigan, W., K. Vetter, K. White, and P. Worley [2005]. "Performance evaluation of the Cray X1 distributed shared memory architecture," *IEEE Micro* January/February, 30–40.

Eden, A., and T. Mudge [1998]. "The YAGS branch prediction scheme," *Proc. of the 31st Annual ACM/IEEE Int'l. Symposium on Microarchitecture*, November 30–December 2, 1998, Dallas, Tex., 69–80.

Edmondson, J. H., P. I. Rubinfield, R. Preston, and V. Rajagopalan [1995]. "Superscalar instruction execution in the 21164 Alpha microprocessor," *IEEE Micro* 15:2, 33–43.

Eggers, S. [1989]. "Simulation Analysis of Data Sharing in Shared Memory Multiprocessors," Ph.D. thesis, University of California, Berkeley.

Elder, J., A. Gottlieb, C. K. Kruskal, K. P. McAuliffe, L. Randolph, M. Snir, P. Teller, and J. Wilson [1985]. "Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach," *Proc. 12th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 17–19, 1985, Boston, Mass., 126–135.

Ellis, J. R. [1986]. *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass.

Emer, J. S., and D. W. Clark [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1984, Ann Arbor, Mich., 301–310.

Enriquez, P. [2001]. "What happened to my dial tone? A study of FCC service disruption reports," poster, *Richard Tapia Symposium on the Celebration of Diversity in Computing*, October 18–20, Houston, Tex.

Erlichson, A., N. Nuckolls, G. Chesson, and J. L. Hennessy [1996]. "SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory," *Proc. Seventh Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1–5, 1996, Cambridge, Mass., 210–220.

Esmaeilzadeh, H., T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley [2011]. "Looking Back on the Language and Hardware Revolution: Measured Power, Performance, and Scaling," *Proc. 16th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 5–11, 2011, Newport Beach, Calif.

Evers, M., S. J. Patel, R. S. Chappell, and Y. N. Patt [1998]. "An analysis of correlation and predictability: What makes two-level branch predictors work," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 52–61.

Fabry, R. S. [1974]. "Capability based addressing," *Communications of the ACM* 17:7 (July), 403–412.

Falsafi, B., and D. A. Wood [1997]. "Reactive NUMA: A design for unifying S-COMA and CC-NUMA," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo., 229–240.

Fan, X., W. Weber, and L. A. Barroso [2007]. "Power provisioning for a warehouse-sized computer," *Proc. 34th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 9–13, 2007, San Diego, Calif.

Farkas, K. I., and N. P. Jouppi [1994]. "Complexity/performance trade-offs with nonblocking loads," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago.

Farkas, K. I., N. P. Jouppi, and P. Chow [1995]. "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors?," *Proc. First IEEE Symposium on High-Performance Computer Architecture*, January 22–25, 1995, Raleigh, N.C., 78–89.

Farkas, K. I., P. Chow, N. P. Jouppi, and Z. Vranesic [1997]. "Memory-system design considerations for dynamically-scheduled processors," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo., 133–143.

Fazio, D. [1987]. "It's really much more fun building a supercomputer than it is simply inventing one," *Proc. IEEE COMPCON*,

February 23–27, 1987, San Francisco, 102–105.

Fisher, J. A. [1981]. "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. on Computers* 30:7 (July), 478–490.

Fisher, J. A. [1983]. "Very long instruction word architectures and ELI-512," *10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 140–150.

Fisher, J. A., and S. M. Freudenberger [1992]. "Predicting conditional branches from previous runs of a program," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston, Mass., 85–95.

Fisher, J. A., and B. R. Rau [1993]. *Journal of Supercomputing*, January (special issue).

Fisher, J. A., J. R. Ellis, J. C. Ruttenberg, and A. Nicolau [1984]. "Parallel processing: A smart compiler and a dumb processor," *Proc. SIGPLAN Conf. on Compiler Construction*, June 17–22, 1984, Montreal, Canada, 11–16.

Flemming, P. J., and J. J. Wallace [1986]. "How not to lie with statistics: The correct way to summarize benchmarks results," *Communications of the ACM* 29:3 (March), 218–221.

Flynn, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December), 1901–1909.

Forgie, J. W. [1957]. "The Lincoln TX-2 input-output system," *Proc. Western Joint Computer Conference* (February), Institute of Radio Engineers, Los Angeles, 156–160.

Foster, C. C., and E. M. Riseman [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411– 1415.

Frank, S. J. [1984]. "Tightly coupled multiprocessor systems speed memory access time," *Electronics* 57:1 (January), 164–169.

Freiman, C. V. [1961]. "Statistical analysis of certain binary division algorithms," *Proc. IRE* 49:1, 91–103.

Friesenborg, S. E., and R. J. Wicks [1985]. *DASD Expectations: The 3380, 3380-23, and MVS/XA*, Tech. Bulletin GG22-9363-02, IBM Washington Systems Center, Gaithersburg, Md.

Fuller, S. H., and W. E. Burr [1977]. "Measurement and evaluation of alternative computer architectures," *Computer* 10:10 (October), 24–35.

Furber, S. B. [1996]. *ARM System Architecture*, Addison-Wesley, Harlow, England (see *www.cs.man.ac.uk/amulet/publications/books/ARMsysArch*).

Gagliardi, U. O. [1973]. "Report of workshop 4—software-related advances in computer hardware," *Proc. Symposium on the High Cost of Software*, September 17–19, 1973, Monterey, Calif., 99–120.

Gajski, D., D. Kuck, D. Lawrie, and A. Sameh [1983]. "CEDAR—a large scale multiprocessor," *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, August, Columbus, Ohio, 524–529.

Gallagher, D. M., W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu [1994]. "Dynamic memory disambiguation using the memory conflict buffer," *Proc. Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 4–7, Santa Jose, Calif., 183–193.

Galles, M. [1996]. "Scalable pipelined interconnect for distributed endpoint routing: The SGI SPIDER chip," *Proc. IEEE HOT Interconnects '96*, August 15–17, 1996, Stanford University, Palo Alto, Calif.

Game, M., and A. Booker [1999]. "CodePack code compression for PowerPC processors," *MicroNews*, 5:1, *www.chips.ibm.com/micronews/vol5_no1/codepack.html.*

Gao, Q. S. [1993]. "The Chinese remainder theorem and the prime memory system," *20th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 16–19, 1993, San Diego, Calif. (*Computer Architecture News* 21:2 (May), 337–340).

Gap. [2005]. "Gap Inc. Reports Third Quarter Earnings," *http://gapinc.com/public/ documents/PR_Q405EarningsFeb2306.pdf.*

Gap. [2006]. "Gap Inc. Reports Fourth Quarter and Full Year Earnings," *http:// gapinc.com/public/documents/Q32005PressRelease_Final22.pdff.*

Garner, R., A. Agarwal, F. Briggs, E. Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnick, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck [1988]. "Scalable processor architecture (SPARC)," *Proc. IEEE COMPCON*, February 29–March 4, 1988, San Francisco, 278–283.

Gebis, J., and D. Patterson [2007]. "Embracing and extending 20th-century instruction set architectures," *IEEE Computer* 40:4 (April), 68–75.

Gee, J. D., M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith [1993]. "Cache performance of the SPEC92 benchmark suite," *IEEE Micro* 13:4 (August), 17–27.

Gehringer, E. F., D. P. Siewiorek, and Z. Segall [1987]. *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, Mass.

Gharachorloo, K., A. Gupta, and J. L. Hennessy [1992]. "Hiding memory latency using dynamic scheduling in shared-memory multiprocessors," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia.

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy [1990]. "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash., 15–26.

Ghemawat, S., H. Gobioff, and S.-T. Leung [2003]. "The Google file system," *Proc. 19th ACM Symposium on Operating Systems Principles*, October 19–22, 2003, Bolton Landing, N.Y.

Gibson, D. H. [1967]. "Considerations in block-oriented systems design," *AFIPS Conf. Proc.* 30, 75–80.

Gibson, G. A. [1992]. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*, ACM Distinguished Dissertation Series, MIT Press, Cambridge, Mass.

Gibson, J. C. [1970]. "The Gibson mix," Rep. TR. 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y. (research done in 1959).

Gibson, J., R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich [2000].

"FLASH vs. (simulated) FLASH: Closing the simulation loop," *Proc. Ninth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 12–15, Cambridge, Mass., 49–58.

Glass, C. J., and L. M. Ni [1992]. "The Turn Model for adaptive routing," *19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic," *Computing Surveys* 23:1, 5–48.

Goldberg, I. B. [1967]. "27 bits are not enough for 8-digit accuracy," *Communications of the ACM* 10:2, 105–106.

Goldstein, S. [1987]. *Storage Performance—An Eight Year Outlook*, Tech. Rep. TR 03.308-1, Santa Teresa Laboratory, IBM Santa Teresa Laboratory, San Jose, Calif.

Goldstine, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, N.J.

González, J., and A. González [1998]. "Limits of instruction level parallelism with data speculation," *Proc. Vector and Parallel Processing (VECPAR) Conf.*, June 21–23, 1998, Porto, Portugal, 585–598.

Goodman, J. R. [1983]. "Using cache memory to reduce processor memory traffic," *Proc. 10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 124–131.

Goralski, W. [1997]. *SONET: A Guide to Synchronous Optical Network*, McGraw-Hill, New York.

Gosling, J. B. [1980]. *Design of Arithmetic Units for Digital Computers*, Springer-Verlag, New York.

Gray, J. [1990]. "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. on Reliability*, 39:4 (October), 409–418.

Gray, J. (ed.) [1993]. *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd ed., Morgan Kaufmann, San Francisco.

Gray, J. [2006]. Sort benchmark home page, *http://sortbenchmark.org/*.

Gray, J., and A. Reuter [1993]. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco.

Gray, J., and D. P. Siewiorek [1991]. "High-availability computer systems," *Computer* 24:9 (September), 39–48.

Gray, J., and C. van Ingen [2005]. *Empirical Measurements of Disk Failure Rates and Error Rates*, MSR-TR-2005-166, Microsoft Research, Redmond, Wash.

Greenberg, A., N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta [2009]. "VL2: A Scalable and Flexible Data Center Network," in *Proc. ACM SIGCOMM*, August 17–21, 2009, Barcelona, Spain.

Grice, C., and M. Kanellos [2000]. "Cell phone industry at crossroads: Go high or low?," *CNET News*, August 31, *technews.netscape.com/news/0-1004-201-2518386-0.html?tag=st.ne.1002.tgif.sf*.

Groe, J. B., and L. E. Larson [2000]. *CDMA Mobile Radio Design*, Artech House, Boston.

Gunther, K. D. [1981]. "Prevention of deadlocks in packet-switched data transport systems," *IEEE Trans. on Communications* COM–29:4 (April), 512–524.

Hagersten, E., and M. Koster [1998]. "WildFire: A scalable path for SMPs," *Proc. Fifth Int'l. Symposium on High-Performance Computer Architecture*, January 9–12, 1999, Orlando, Fla.

Hagersten, E., A. Landin, and S. Haridi [1992]. "DDM—a cache-only memory architecture," *IEEE Computer* 25:9 (September), 44–54.

Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky [1984]. *Computer Organization*, 2nd ed., McGraw-Hill, New York.

Hamilton, J. [2009]. "Data center networks are in my way," paper presented at the Stanford Clean Slate CTO Summit, October 23, 2009 (*http://mvdirona.com/jrh/ TalksAndPapers/JamesHamilton_CleanSlateCTO2009. pdf*).

Hamilton, J. [2010]. "Cloud computing economies of scale," paper presented at the *AWS Workshop on Genomics and Cloud Computing*, June 8, 2010, Seattle, Wash. (*http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_GenomicsCloud20100608.pdf*).

Handy, J. [1993]. *The Cache Memory Book*, Academic Press, Boston.

Hauck, E. A., and B. A. Dent [1968]. "Burroughs' B6500/B7500 stack mechanism," *Proc. AFIPS Spring Joint Computer Conf.*, April 30–May 2, 1968, Atlantic City, N.J., 245–251.

Heald, R., K. Aingaran, C. Amir, M. Ang, M. Boland, A. Das, P. Dixit, G. Gouldsberry, J. Hart, T. Horel, W.-J. Hsu, J. Kaku, C. Kim, S. Kim, F. Klass, H. Kwan, R. Lo, H. McIntyre, A. Mehta, D. Murata, S. Nguyen, Y.-P. Pai, S. Patel, K. Shin, K. Tam, S. Vishwanthaiah, J. Wu, G. Yee, and H. You [2000]. "Implementation of thirdgeneration SPARC V9 64-b microprocessor," *ISSCC Digest of Technical Papers*, 412–413 and slide supplement.

Heinrich, J. [1993]. *MIPS R4000 User's Manual*, Prentice Hall, Englewood Cliffs, N.J.

Henly, M., and B. McNutt [1989]. *DASD I/O Characteristics: A Comparison of MVS to VM*," Tech. Rep. TR 02.1550 (May), IBM General Products Division, San Jose, Calif.

Hennessy, J. [1984]. "VLSI processor architecture," *IEEE Trans. on Computers* C-33:11 (December), 1221–1246.

Hennessy, J. [1985]. "VLSI RISC processors," *VLSI Systems Design* 6:10 (October), 22–32.

Hennessy, J., N. Jouppi, F. Baskett, and J. Gill [1981]. "MIPS: A VLSI processor architecture," in *CMU Conference on VLSI Systems and Computations*, Computer Science Press, Rockville, Md.

Hewlett-Packard. [1994]. *PA-RISC 2.0 Architecture Reference Manual*, 3rd ed., Hewlett- Packard, Palo Alto, Calif.

Hewlett-Packard. [1998]. "HP's '5NINES:5MINUTES' Vision Extends Leadership and Redefines High Availability in Mission-Critical Environments," February 10,*www.future.enterprisecomputing.hp.com/ia64/news/5nines_vision_pr.html.*

Hill, M. D. [1987]. "Aspects of Cache Memory and Instruction Buffer Performance," Ph.D. thesis, Tech. Rep. UCB/CSD 87/381, Computer Science Division, University of California, Berkeley.

Hill, M. D. [1988]. "A case for direct mapped caches," *Computer* 21:12 (December), 25–40.

Hill, M. D. [1998]. "Multiprocessors should support simple memory consistency models," *IEEE Computer* 31:8 (August), 28–34.

Hillis, W. D. [1985]. *The Connection Multiprocessor*, MIT Press, Cambridge, Mass.

Hillis, W. D. and G. L. Steele [1986]. "Data parallel algorithms," *Communications of the ACM* 29:12 (December), 1170–1183. (*http://doi.acm.org/10.1145/7902.7903*).

Hinton, G., D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel [2001]. "The microarchitecture of the Pentium 4 processor," *Intel Technology Journal*, February.

Hintz, R. G., and D. P. Tate [1972]. "Control data STAR-100 processor design," *Proc. IEEE COMPCON*, September 12–14, 1972, San Francisco, 1–4.

Hirata, H., K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa [1992]. "An elementary processor architecture with simultaneous instruction issuing from multiple threads," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 136–145.

Hitachi. [1997]. *SuperH RISC Engine SH7700 Series Programming Manual*, Hitachi, Santa Clara, Calif. (see *www.halsp.hitachi. com/tech_prod/* and search for title).

Ho, R., K. W. Mai, and M. A. Horowitz [2001]. "The future of wires," *Proc. of the IEEE* 89:4 (April), 490–504.

Hoagland, A. S. [1963]. *Digital Magnetic Recording*, Wiley, New York.

Hockney, R. W., and C. R. Jesshope [1988]. *Parallel Computers 2: Architectures, Programming and Algorithms*, Adam Hilger, Ltd., Bristol, England.

Holland, J. H. [1959]. "A universal computer capable of executing an arbitrary number of subprograms simultaneously," *Proc. East Joint Computer Conf.* 16, 108–113.

Holt, R. C. [1972]. "Some deadlock properties of computer systems," *ACM Computer Surveys* 4:3 (September), 179–196.

Hopkins, M. [2000]. "A critical look at IA-64: Massive resources, massive ILP, but can it deliver?" *Microprocessor Report*, February.

Hord, R. M. [1982]. *The Illiac-IV, The First Supercomputer*, Computer Science Press, Rockville, Md.

Horel, T., and G. Lauterbach [1999]. "UltraSPARC-III: Designing third-generation 64-bit performance," *IEEE Micro* 19:3

(May–June), 73–85.

Hospodor, A. D., and A. S. Hoagland [1993]. "The changing nature of disk controllers." *Proc. IEEE* 81:4 (April), 586–594.

Hölzle, U. [2010]. "Brawny cores still beat wimpy cores, most of the time," *IEEE Micro* 30:4 (July/August).

Hristea, C., D. Lenoski, and J. Keen [1997]. "Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks," *Proc. ACM/IEEE Conf. on Supercomputing*, November 16–21, 1997, San Jose, Calif.

Hsu, P. [1994]. "Designing the TFP microprocessor," *IEEE Micro* 18:2 (April), 2333.

Huck, J. et al. [2000]. "Introducing the IA-64 Architecture" *IEEE Micro*, 20:5 (September–October), 12–23.

Hughes, C. J., P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan [2001]. "Variability in the execution of multimedia applications and implications for architecture," *Proc. 28th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 30–July 4, 2001, Goteborg, Sweden, 254–265.

Hwang, K. [1979]. *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, New York.

Hwang, K. [1993]. *Advanced Computer Architecture and Parallel Programming*, McGraw-Hill, New York.

Hwu, W.-M., and Y. Patt [1986]. "HPSm, a high performance restricted data flow architecture having minimum functionality," *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1986, Tokyo, 297–307.

Hwu, W. W., S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery [1993]. "The superblock: An effective technique for VLIW and superscalar compilation," *J. Supercomputing* 7:1, 2 (March), 229–248.

IBM. [1982]. *The Economic Value of Rapid Response Time*, GE20-0752-0, IBM, White Plains, N.Y., 11–82.

IBM. [1990]. "The IBM RISC System/6000 processor" (collection of papers), *IBM J. Research and Development* 34:1 (January).

IBM. [1994]. *The PowerPC Architecture*, Morgan Kaufmann, San Francisco.

IBM. [2005]. "Blue Gene," *IBM J. Research and Development*, 49:2/3 (special issue).

IEEE. [1985]. "IEEE standard for binary floating-point arithmetic," *SIGPLAN Notices* 22:2, 9–25.

IEEE. [2005]. "Intel virtualization technology, computer," *IEEE Computer Society* 38:5 (May), 48–56.

IEEE. 754-2008 Working Group. [2006]. "DRAFT Standard for Floating-Point Arithmetic 754-2008," *http:// dx.doi.org/10.1109/ IEEESTD.2008.4610935*.

Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB, Document No. 64402302, Imprimis, Dallas, Tex.

InfiniBand Trade Association. [2001]. *InfiniBand Architecture Specifications Release 1.0.a, www.infinibandta.org.*

Intel. [2001]. "Using MMX Instructions to Convert RGB to YUV Color Conversion," *cedar.intel.com/cgi-bin/ids.dll/content/content.jsp? cntKey=Legacy::irtm_AP548_9996& cntType=IDS_ EDITORIAL.*

Internet Retailer. [2005]. "The Gap launches a new site—after two weeks of downtime," *Internet® Retailer*, September 28, *http://www.internetretailer.com/2005/09/28/thegap- launches-a-new-site-after-two-weeks-of-downtime.*

Jain, R. [1991]. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, New York.

Jantsch, A., and H. Tenhunen (eds.) [2003]. *Networks on Chips*, Kluwer Academic Publishers, The Netherlands.

Jimenez, D. A., and C. Lin [2002]. "Neural methods for dynamic branch prediction," *ACM Trans. on Computer Systems* 20:4 (November), 369–397.

Johnson, M. [1990]. *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J.

Jordan, H. F. [1983]. "Performance measurements on HEP—a pipelined MIMD computer," *Proc. 10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 207–212.

Jordan, K. E. [1987]. "Performance comparison of large-scale scientific processors: Scalar mainframes, mainframes with vector facilities, and supercomputers," *Computer* 20:3 (March), 10–23.

Jouppi, N. P. [1990]. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash., 364–373.

Jouppi, N. P. [1998]. "Retrospective: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ACM, New York, 71–73.

Jouppi, N. P., and D. W. Wall [1989]. "Available instruction-level parallelism for superscalar and superpipelined processors," *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, 272–282.

Jouppi, N. P., and S. J. E. Wilton [1994]. "Trade-offs in two-level on-chip caching," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago, 34–45.

Kaeli, D. R., and P. G. Emma [1991]. "Branch history table prediction of moving target branches due to subroutine returns," *Proc. 18th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 27–30, 1991, Toronto, Canada, 34–42.

Kahan, J. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University of California, Berkeley.

Kahan, W. [1968]. "7094-II system support for numerical analysis," *SHARE Secretarial Distribution* SSD-159, Department of Computer Science, University of Toronto.

Kahaner, D. K. [1988]. "Benchmarks for 'real' programs," *SIAM News*, November.

Kahn, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November), 1397–1407.

Kane, G. [1986]. *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

Kane, G. [1996]. *PA-RISC 2.0 Architecture*, Prentice Hall, Upper Saddle River, N.J.

Kane, G., and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

Katz, R. H., D. A. Patterson, and G. A. Gibson [1989]. "Disk system architectures for high performance computing," *Proc. IEEE* 77:12 (December), 1842–1858.

Keckler, S. W., and W. J. Dally [1992]. "Processor coupling: Integrating compile time and runtime scheduling for parallelism," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 202–213.

Keller, R. M. [1975]. "Look-ahead processors," *ACM Computing Surveys* 7:4 (December), 177–195.

Keltcher, C. N., K. J. McGrath, A. Ahmed, and P. Conway [2003]. "The AMD Opteron processor for multiprocessor servers," *IEEE Micro* 23:2 (March–April), 66–76 (dx.doi.org/10.1109.MM.2003.119116).

Kembel, R. [2000]. "Fibre Channel: A comprehensive introduction," *Internet Week*, April.

Kermani, P., and L. Kleinrock [1979]. "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks* 3 (January), 267–286.

Kessler, R. [1999]. "The Alpha 21264 microprocessor," *IEEE Micro* 19:2 (March/April) 24–36.

Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner [1962]. "One-level storage system," *IRE Trans. on Electronic Computers* EC-11 (April) 223–235. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 135–148.

Killian, E. [1991]. "MIPS R4000 technical overview–64 bits/100 MHz or bust," *Hot Chips III Symposium Record*, August 26–27, 1991, Stanford University, Palo Alto, Calif., 1.6–1.19.

Kim, M. Y. [1986]. "Synchronized disk interleaving," *IEEE Trans. on Computers* C-35:11 (November), 978–988.

Kissell, K. D. [1997]. "MIPS16: High-density for the embedded market," *Proc. Real Time Systems '97*, June 15, 1997, Las Vegas, Nev. (see *www.sgi.com/MIPS/arch/MIPS16/ MIPS16.whitepaper.pdf*).

Kitagawa, K., S. Tagaya, Y. Hagihara, and Y. Kanoh [2003]. "A hardware overview of SX- 6 and SX-7 supercomputer," *NEC Research & Development J.* 44:1 (January), 2–7.

Knuth, D. [1981]. *The Art of Computer Programming*, Vol. II, 2nd ed., Addison-Wesley, Reading, Mass.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

Kohn, L., and S.-W. Fu [1989]. "A 1,000,000 transistor microprocessor," *Proc. of IEEE Int'l. Symposium on Solid State Circuits (ISSCC)*, February 15–17, 1989, New York, 54–55.

Kohn, L., and N. Margulis [1989]. "Introducing the Intel i860 64-Bit Microprocessor," *IEEE Micro*, 9:4 (July), 15–30.

Kontothanassis, L., G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott [1997]. "VM-based shared memory on lowlatency, remote-memory-access networks," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo.

Koren, I. [1989]. *Computer Arithmetic Algorithms*, Prentice Hall, Englewood Cliffs, N.J.

Kozyrakis, C. [2000]. "Vector IRAM: A media-oriented vector processor with embedded

DRAM," paper presented at Hot Chips 12, August 13–15, 2000, Palo Alto, Calif, 13–15.

Kozyrakis, C., and D. Patterson, [2002]. "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," *Proc. 35th Annual Int'l. Symposium on Microarchitecture (MICRO-35)*, November 18–22, 2002, Istanbul, Turkey.

Kroft, D. [1981]. "Lockup-free instruction fetch/prefetch cache organization," *Proc. Eighth Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 12–14, 1981, Minneapolis, Minn., 81–87.

Kroft, D. [1998]. "Retrospective: Lockup-free instruction fetch/prefetch cache organization," *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ACM, New York, 20–21.

Kuck, D., P. P. Budnik, S.-C. Chen, D. H. Lawrie, R. A. Towle, R. E. Strebendt, E. W. Davis, Jr., J. Han, P. W. Kraska, and Y. Muraoka [1974]. "Measurements of parallelism in ordinary FORTRAN programs," *Computer* 7:1 (January), 37–46.

Kuhn, D. R. [1997]. "Sources of failure in the public switched telephone network," *IEEE Computer* 30:4 (April), 31–36.

Kumar, A. [1997]. "The HP PA-8000 RISC CPU," *IEEE Micro* 17:2 (March/April), 27–32.

Kunimatsu, A., N. Ide, T. Sato, Y. Endo, H. Murakami, T. Kamei, M. Hirano, F. Ishihara, H. Tago, M. Oka, A. Ohba, T. Yutaka, T. Okada, and M. Suzuoki [2000]. "Vector unit architecture for emotion synthesis," *IEEE Micro* 20:2 (March–April), 40–47.

Kunkel, S. R., and J. E. Smith [1986]. "Optimal pipelining in supercomputers," *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1986, Tokyo, 404–414.

Kurose, J. F., and K. W. Ross [2001]. *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison-Wesley, Boston.

Kuskin, J., D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy [1994]. "The Stanford FLASH multiprocessor," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago.

Lam, M. [1988]. "Software pipelining: An effective scheduling technique for VLIW processors," *SIGPLAN Conf. on Programming Language Design and Implementation*, June 22–24, 1988, Atlanta, Ga., 318–328.

Lam, M. S., and R. P. Wilson [1992]. "Limits of control flow on parallelism," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 46–57.

Lam, M. S., E. E. Rothberg, and M. E. Wolf [1991]. "The cache performance and optimizations of blocked algorithms," *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Santa Clara, Calif. (*SIGPLAN Notices* 26:4 (April), 63–74).

Lambright, D. [2000]. "Experiences in measuring the reliability of a cache-based storage system," *Proc. of First Workshop on Industrial Experiences with Systems Software (WIESS 2000), Co-Located with the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, October 22, 2000, San Diego, Calif.

Lamport, L. [1979]. "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Computers* C-28:9 (September), 241–248.

Lang, W., J. M. Patel, and S. Shankar [2010]. "Wimpy node clusters: What about nonwimpy workloads?" *Proc. Sixth International Workshop on Data Management on New Hardware (DaMoN)*, June 7, Indianapolis, Ind.

Laprie, J.-C. [1985]. "Dependable computing and fault tolerance: Concepts and terminology," *Proc. 15th Annual Int'l. Symposium on Fault-Tolerant Computing*, June 19–21, 1985, Ann Arbor, Mich., 2–11.

Larson, E. R. [1973]. "Findings of fact, conclusions of law, and order for judgment," File No. 4-67, Civ. 138, *Honeywell v. Sperry-Rand and Illinois Scientific Development*, U.S. District Court for the State of Minnesota, Fourth Division (October 19).

Laudon, J., and D. Lenoski [1997]. "The SGI Origin: A ccNUMA highly scalable server," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo., 241–251.

Laudon, J., A. Gupta, and M. Horowitz [1994]. "Interleaving: A multithreading technique targeting multiprocessors and workstations," *Proc. Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 4–7, San Jose, Calif., 308–318.

Lauterbach, G., and T. Horel [1999]. "UltraSPARC-III: Designing third generation 64-bit performance," *IEEE Micro* 19:3 (May/June).

Lazowska, E. D., J. Zahorjan, G. S. Graham, and K. C. Sevcik [1984]. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice Hall, Englewood Cliffs, N.J. (Although out of print, it is available online at *www.cs.washington.edu/homes/lazowska/qsp/*.)

Lebeck, A. R., and D. A. Wood [1994]. "Cache profiling and the SPEC benchmarks: A case study," *Computer* 27:10 (October), 15–26.

Lee, R. [1989]. "Precision architecture," *Computer* 22:1 (January), 78–91.

Lee, W. V. et al. [2010]. "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 19–23, 2010, Saint-Malo, France.

Leighton, F. T. [1992]. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Francisco.

Leiner, A. L. [1954]. "System specifications for the DYSEAC," *J. ACM* 1:2 (April), 57–81.

Leiner, A. L., and S. N. Alexander [1954]. "System organization of the DYSEAC," *IRE Trans. of Electronic Computers* EC-3:1 (March), 1–10.

Leiserson, C. E. [1985]. "Fat trees: Universal networks for hardware-efficient supercomputing," *IEEE Trans. on Computers* C-34:10 (October), 892–901.

Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy [1990]. "The Stanford DASH multiprocessor," *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash., 148–159.

Lenoski, D., J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. A.

Horowitz, and M. Lam [1992]. "The Stanford DASH multiprocessor," *IEEE Computer* 25:3 (March), 63–79.

Levy, H., and R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.

Li, K. [1988]. "IVY: A shared virtual memory system for parallel computing," *Proc. 1988 Int'l. Conf. on Parallel Processing*, Pennsylvania State University Press, University Park, Penn.

Li, S., K. Chen, J. B. Brockman, and N. Jouppi [2011]. "Performance Impacts of Nonblocking Caches in Out-of-order Processors," HP Labs Tech Report HPL-2011-65 (full text available at http://Library.hp.com/techpubs/2011/Hpl-2011-65.html).

Lim, K., P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt [2008]. "Understanding and designing new system architectures for emerging warehouse-computing environments," *Proc. 35th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 21–25, 2008, Beijing, China.

Lincoln, N. R. [1982]. "Technology and design trade offs in the creation of a modern supercomputer," *IEEE Trans. on Computers* C-31:5 (May), 363–376.

Lindholm, T., and F. Yellin [1999]. *The Java Virtual Machine Specification*, 2nd ed., Addison-Wesley, Reading, Mass. (also available online at *java.sun.com/docs/ books/ vmspec/*).

Lipasti, M. H., and J. P. Shen [1996]. "Exceeding the dataflow limit via value prediction," *Proc. 29th Int'l. Symposium on Microarchitecture*, December 2–4, 1996, Paris, France.

Lipasti, M. H., C. B. Wilkerson, and J. P. Shen [1996]. "Value locality and load value prediction," *Proc. Seventh Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1–5, 1996, Cambridge, Mass., 138–147.

Liptay, J. S. [1968]. "Structural aspects of the System/360 Model 85, Part II: The cache," *IBM Systems J.* 7:1, 15–21.

Lo, J., L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh [1998]. "An analysis of database workload performance on simultaneous multithreaded processors," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 39–50.

Lo, J., S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen [1997]. "Converting threadlevel parallelism into instruction-level parallelism via simultaneous multithreading," *ACM Trans. on Computer Systems* 15:2 (August), 322–354.

Lovett, T., and S. Thakkar [1988]. "The Symmetry multiprocessor system," *Proc. 1988 Int'l. Conf. of Parallel Processing*, University Park, Penn., 303–310.

Lubeck, O., J. Moore, and R. Mendez [1985]. "A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2," *Computer* 18:12 (December), 10–24.

Luk, C.-K., and T. C Mowry [1999]. "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE Trans. on Computers* 48:2 (February), 134–141.

Lunde, A. [1977]. "Empirical evaluation of some features of instruction set processor architecture," *Communications of the ACM* 20:3 (March), 143–152.

Luszczek, P., J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi [2005]. "Introduction to the HPC challenge benchmark suite," Lawrence Berkeley National Laboratory, Paper LBNL-57493 (April 25), *repositories.cdlib.org/lbnl/LBNL-57493*.

Maberly, N. C. [1966]. *Mastering Speed Reading*, New American Library, New York.

Magenheimer, D. J., L. Peters, K. W. Pettis, and D. Zuras [1988]. "Integer multiplication and division on the HP precision architecture," *IEEE Trans. on Computers* 37:8, 980–990.

Mahlke, S. A., W. Y. Chen, W.-M. Hwu, B. R. Rau, and M. S. Schlansker [1992]. "Sentinel scheduling for VLIW and superscalar processors," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston, 238–247.

Mahlke, S. A., R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu [1995]. "A comparison of full and partial predicated

execution support for ILP processors," *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy, 138–149.

Major, J. B. [1989]. "Are queuing models within the grasp of the unwashed?," *Proc. Int'l. Conf. on Management and Performance Evaluation of Computer Systems*, December 11–15, 1989, Reno, Nev., 831–839.

Markstein, P. W. [1990]. "Computation of elementary functions on the IBM RISC System/ 6000 processor," *IBM J. Research and Development* 34:1, 111–119.

Mathis, H. M., A. E. Mercias, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel [2005]. "Characterization of the multithreading (SMT) efficiency in Power5," *IBM J. Research and Development*, 49:4/5 (July/September), 555–564.

McCalpin, J. [2005]. "STREAM: Sustainable Memory Bandwidth in High Performance Computers," *www.cs. virginia.edu/stream/*.

McCalpin, J., D. Bailey, and D. Takahashi [2005]. *Introduction to the HPC Challenge Benchmark Suite*, Paper LBNL-57493 Lawrence Berkeley National Laboratory, University of California, Berkeley, *repositories. cdlib.org/lbnl/LBNL-57493*.

McCormick, J., and A. Knies [2002]. "A brief analysis of the SPEC CPU2000 benchmarks on the Intel Itanium 2 processor," paper presented at Hot Chips 14, August 18–20, 2002, Stanford University, Palo Alto, Calif.

McFarling, S. [1989]. "Program optimization for instruction caches," *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, 183–191.

McFarling, S. [1993]. *Combining Branch Predictors*, WRL Technical Note TN-36, Digital Western Research Laboratory, Palo Alto, Calif.

McFarling, S., and J. Hennessy [1986]. "Reducing the cost of branches," *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1986, Tokyo, 396–403.

McGhan, H., and M. O'Connor [1998]. "PicoJava: A direct execution engine for Java bytecode," *Computer* 31:10 (October), 22–30.

McKeeman, W. M. [1967]. "Language directed computer design," *Proc. AFIPS Fall Joint Computer Conf.*, November 14–16, 1967, Washington, D.C., 413–417.

McMahon, F. M. [1986]. "*The Livermore FORTRAN Kernels: A Computer Test of Numerical Performance Range*," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, University of California, Livermore.

McNairy, C., and D. Soltis [2003]. "Itanium 2 processor microarchitecture," *IEEE Micro* 23:2 (March–April), 44–55.

Mead, C., and L. Conway [1980]. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass.

Mellor-Crummey, J. M., and M. L. Scott [1991]. "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems* 9:1 (February), 21–65.

Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," *Bibliothèque Universelle de Genève*, 82 (October).

Menon, A., J. Renato Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel [2005]. "Diagnosing performance overheads in the xen virtual machine environment," *Proc. First ACM/USENIX Int'l. Conf. on Virtual Execution Environments*, June 11–12, 2005, Chicago, 13–23.

Merlin, P. M., and P. J. Schweitzer [1980]. "Deadlock avoidance in store-and-forward networks. Part I. Store-and-forward deadlock," *IEEE Trans. on Communications* COM-28:3 (March), 345–354.

Metcalfe, R. M. [1993]. "Computer/network interface design: Lessons from Arpanet and Ethernet," *IEEE J. on Selected Areas in Communications* 11:2 (February), 173–180.

Metcalfe, R. M., and D. R. Boggs [1976]. "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM* 19:7 (July), 395–404.

Metropolis, N., J. Howlett, and G. C. Rota (eds.) [1980]. *A History of Computing in the Twentieth Century*, Academic Press, New York.

Meyer, R. A., and L. H. Seawright [1970]. A virtual machine time sharing system, *IBM Systems J.* 9:3, 199–218.

Meyers, G. J. [1978]. "The evaluation of expressions in a storage-to-storage architecture," *Computer Architecture News* 7:3 (October), 20–23.

Meyers, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley, New York.

Micron. [2004]. "Calculating Memory System Power for DDR2," *http://download. micron.com/pdf/pubs/ designline/dl1Q04.pdf*.

Micron. [2006]. "The Micron® System-Power Calculator," *http://www.micron.com/ systemcalc*.

MIPS. [1997]. "MIPS16 Application Specific Extension Product Description," *www.sgi.com/MIPS/arch/ MIPS16/mips16.pdf*.

Miranker, G. S.; J. Rubenstein, and J. Sanguinetti [1988]. "Squeezing a Cray-class supercomputer into a single-user package," *Proc. IEEE COMPCON*, February 29–March 4, 1988, San Francisco, 452–456.

Mitchell, D. [1989]. "The Transputer: The time is now," *Computer Design* (RISC suppl.), 40–41.

Mitsubishi. [1996]. *Mitsubishi 32-Bit Single Chip Microcomputer M32R Family Software Manual*, Mitsubishi, Cypress, Calif.

Miura, K., and K. Uchida [1983]. "FACOM vector processing system: VP100/200," *Proc. NATO Advanced Research Workshop on High-Speed Computing*, June 20–22, 1983, Jülich, West Germany. Also appears in K. Hwang, ed., "Superprocessors: Design and applications," *IEEE* (August 1984), 59–73.

Miya, E. N. [1985]. "Multiprocessor/distributed processing bibliography," *Computer Architecture News* 13:1, 27–29.

Montoye, R. K., E. Hokenek, and S. L. Runyon [1990]. "Design of the IBM RISC System/ 6000 floating-point execution," *IBM J. Research and Development* 34:1, 59–70.

Moore, B., A. Padegs, R. Smith, and W. Bucholz [1987]. "Concepts of the System/370 vector architecture," *14th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1987, Pittsburgh, Penn., 282–292.

Moore, G. E. [1965]. "Cramming more components onto integrated circuits," *Electronics*, 38:8 (April 19), 114–117.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel microprocessors—8080 to 8086," *Computer* 13:10 (October).

Moshovos, A., and G. S. Sohi [1997]. "Streamlining inter-operation memory communication via data dependence prediction," *Proc. 30th Annual Int'l. Symposium on Microarchitecture*, December 1–3, Research Triangle Park, N.C., 235–245.

Moshovos, A., S. Breach, T. N. Vijaykumar, and G. S. Sohi [1997]. "Dynamic speculation and synchronization of data dependences," *24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo. Moussouris, J., L. Crudele, D. Freitas, C. Hansen, E. Hudson, S. Przybylski, T. Riordan, and C. Rowen [1986]. "A CMOS RISC processor with integrated system functions," *Proc. IEEE COMPCON*, March 3–6, 1986, San Francisco, 191.

Mowry, T. C., S. Lam, and A. Gupta [1992]. "Design and evaluation of a compiler algorithm for prefetching," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston (*SIGPLAN Notices* 27:9 (September), 62–73).

MSN Money. [2005]. "Amazon Shares Tumble after Rally Fizzles," *http://moneycentral .msn.com/content/CNBCTV/Articles/ Dispatches/P133695.asp.*

Muchnick, S. S. [1988]. "Optimizing compilers for SPARC," *Sun Technology* 1:3 (Summer), 64–77.

Mueller, M., L. C. Alves, W. Fischer, M. L. Fair, and I. Modi [1999]. "RAS strategy for IBM S/390 G5 and G6," *IBM J. Research and Development* 43:5–6 (September–November), 875–888.

Mukherjee, S. S., C. Weaver, J. S. Emer, S. K. Reinhardt, and T. M. Austin [2003]. "Measuring architectural vulnerability factors," *IEEE Micro* 23:6, 70–75.

Murphy, B., and T. Gent [1995]. "Measuring system and software reliability using an automated data collection process," *Quality and Reliability Engineering International* 11:5 (September–October), 341–353.

Myer, T. H., and I. E. Sutherland [1968]. "On the design of display processors," *Communications of the ACM* 11:6 (June), 410–414.

Narayanan, D., E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron [2009]. "Migrating server storage to SSDs: Analysis of trade-offs," *Proc. 4th ACM European Conf. on Computer Systems*, April 1–3, 2009, Nuremberg, Germany.

National Research Council. [1997]. *The Evolution of Untethered Communications*, Computer Science and Telecommunications Board, National Academy Press, Washington, D.C.

National Storage Industry Consortium. [1998]. "Tape Roadmap," *www.nsic.org.*

Nelson, V. P. [1990]. "Fault-tolerant computing: Fundamental concepts," *Computer* 23:7 (July), 19–25.

Ngai, T.-F., and M. J. Irwin [1985]. "Regular, area-time efficient carry-lookahead adders," *Proc. Seventh IEEE Symposium on Computer Arithmetic*, June 4–6, 1985, University of Illinois, Urbana, 9–15.

Nicolau, A., and J. A. Fisher [1984]. "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. on Computers* C-33:11 (November), 968–976.

Nikhil, R. S., G. M. Papadopoulos, and Arvind [1992]. "*T: A multithreaded massively parallel architecture," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 156–167.

Noordergraaf, L., and R. van der Pas [1999]. "Performance experiences on Sun's WildFire prototype," *Proc. ACM/IEEE Conf. on Supercomputing*, November 13–19, 1999, Portland, Ore.

Nyberg, C. R., T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet [1994]. "AlphaSort: A RISC machine sort," *Proc. ACM SIGMOD*, May 24–27, 1994, Minneapolis, Minn.

Oka, M., and M. Suzuoki [1999]. "Designing and programming the emotion engine," *IEEE Micro* 19:6 (November–December),

20–28.

Okada, S., S. Okada, Y. Matsuda, T. Yamada, and A. Kobayashi [1999]. "System on a chip for digital still camera," *IEEE Trans. on Consumer Electronics* 45:3 (August), 584–590.

Oliker, L., A. Canning, J. Carter, J. Shalf, and S. Ethier [2004]. "Scientific computations on modern parallel vector systems," *Proc. ACM/IEEE Conf. on Supercomputing*, November 6–12, 2004, Pittsburgh, Penn., 10.

Pabst, T. [2000]. "Performance Showdown at 133 MHz FSB—The Best Platform for Coppermine," *www6.tomshardware.com/mainboard/00q1/000302/*.

Padua, D., and M. Wolfe [1986]. "Advanced compiler optimizations for supercomputers," *Communications of the ACM* 29:12 (December), 1184–1201.

Palacharla, S., and R. E. Kessler [1994]. "Evaluating stream buffers as a secondary cache replacement," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago, 24–33.

Palmer, J., and S. Morse [1984]. *The 8087 Primer*, John Wiley & Sons, New York, 93.

Pan, S.-T., K. So, and J. T. Rameh [1992]. "Improving the accuracy of dynamic branch prediction using branch correlation," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston, 76–84.

Partridge, C. [1994]. *Gigabit Networking*, Addison-Wesley, Reading, Mass.

Patterson, D. [1985]. "Reduced instruction set computers," *Communications of the ACM* 28:1 (January), 8–21.

Patterson, D. [2004]. "Latency lags bandwidth," *Communications of the ACM* 47:10 (October), 71–75.

Patterson, D. A., and D. R. Ditzel [1980]. "The case for the reduced instruction set computer," *Computer Architecture News* 8:6 (October), 25–33.

Patterson, D. A., and J. L. Hennessy [2004]. *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed., Morgan Kaufmann, San Francisco.

Patterson, D. A., G. A. Gibson, and R. H. Katz [1987]. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Tech. Rep. UCB/CSD 87/391, University of California,Berkeley. Also appeared in *Proc. ACM SIGMOD*, June 1–3, 1988, Chicago, 109–116.

Patterson, D. A., P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, and K. Van Dyke [1983]. "Architecture of a VLSI instruction cache for a RISC," *10th Annual Int'l. Conf. on Computer Architecture Conf. Proc.*, June 13–16, 1983, Stockholm, Sweden, 108–116.

Pavan, P., R. Bez, P. Olivo, and E. Zanoni [1997]. "Flash memory cells—an overview." *Proc. IEEE* 85:8 (August), 1248–1271.

Peh, L. S., and W. J. Dally [2001]. "A delay model and speculative architecture for pipelined routers," *Proc. 7th Int'l. Symposium on High-Performance Computer Architecture*, January 22–24, 2001, Monterrey, Mexico.

Peng, V., S. Samudrala, and M. Gavrielov [1987]. "On the implementation of shifters, multipliers, and dividers in VLSI floating point units," *Proc. 8th IEEE Symposium on Computer Arithmetic*, May 19–21, 1987, Como, Italy, 95–102.

Pfister, G. F. [1998]. *In Search of Clusters*, 2nd ed., Prentice Hall, Upper Saddle River, N.J.

Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfekder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss [1985]. "The IBM research parallel processor prototype (RP3): Introduction and architecture," *Proc. 12th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 17–19, 1985, Boston, Mass., 764–771.

Pinheiro, E., W. D. Weber, and L. A. Barroso [2007]. "Failure trends in a large disk drive population," *Proc. 5th USENIX Conference on File and Storage Technologies (FAST '07)*, February 13–16, 2007, San Jose, Calif.

Pinkston, T. M. [2004]. "Deadlock characterization and resolution in interconnection networks," in M. C. Zhu and M. P. Fanti, eds., *Deadlock Resolution in Computer- Integrated Systems*, CRC Press, Boca Raton, FL, 445–492.

Pinkston, T. M., and J. Shin [2005]. "Trends toward on-chip networked microsystems," *Int'l. J. of High Performance Computing and Networking* 3:1, 3–18.

Pinkston, T. M., and S. Warnakulasuriya [1997]. "On deadlocks in interconnection networks," *24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo.

Pinkston, T. M., A. Benner, M. Krause, I. Robinson, and T. Sterling [2003]. "InfiniBand: The 'de facto' future standard for system and local area networks or just a scalable replacement for PCI buses?" *Cluster Computing* (special issue on communication architecture for clusters) 6:2 (April), 95–104.

Postiff, M. A., D. A. Greene, G. S. Tyson, and T. N. Mudge [1999]. "The limits of instruction level parallelism in SPEC95

applications," *Computer Architecture News* 27:1 (March), 31–40.

Przybylski, S. A. [1990]. *Cache Design: A Performance-Directed Approach*, Morgan Kaufmann, San Francisco.

Przybylski, S. A., M. Horowitz, and J. L. Hennessy [1988]. "Performance trade-offs in cache design," *15th Annual Int'l. Symposium on Computer Architecture*, May 30–June 2, 1988, Honolulu, Hawaii, 290–298.

Puente, V., R. Beivide, J. A. Gregorio, J. M. Prellezo, J. Duato, and C. Izu [1999]. "Adaptive bubble router: A design to improve performance in torus networks," *Proc. 28th Int'l. Conference on Parallel Processing*, September 21–24, 1999, Aizu-Wakamatsu, Fukushima, Japan.

Radin, G. [1982]. "The 801 minicomputer," *Proc. Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 1–3, 1982, Palo Alto, Calif., 39–47.

Rajesh Bordawekar, Uday Bondhugula, Ravi Rao: Believe it or not!: mult-core CPUs can match GPU performance for a FLOP-intensive application! 19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010: 537-538.

Ramamoorthy, C. V., and H. F. Li [1977]. "Pipeline architecture," *ACM Computing Surveys* 9:1 (March), 61–102.

Ranganathan, P., P. Leech, D. Irwin, and J. Chase [2006]. "Ensemble-Level Power Management for Dense Blade Servers," *Proc. 33rd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 17–21, 2006, Boston, Mass., 66–77.

Rau, B. R. [1994]. "Iterative modulo scheduling: An algorithm for software pipelining loops," *Proc. 27th Annual Int'l. Symposium on Microarchitecture*, November 30–December 2, 1994, San Jose, Calif., 63–74.

Rau, B. R., C. D. Glaeser, and R. L. Picard [1982]. "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," *Proc. Ninth Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 26–29, 1982, Austin, Tex., 131–139.

Rau, B. R., D. W. L. Yen, W. Yen, and R. A. Towle [1989]. "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs," *IEEE Computers* 22:1 (January), 12–34.

Reddi, V. J. , B. C. Lee, T. Chilimbi, and K. Vaid [2010]. "Web search using mobile cores: Quantifying and mitigating the price of efficiency," *Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 19–23, 2010, Saint-Malo, France.

Redmond, K. C., and T. M. Smith [1980]. *Project Whirlwind—The History of a Pioneer Computer*, Digital Press, Boston.

Reinhardt, S. K., J. R. Larus, and D. A. Wood [1994]. "Tempest and Typhoon: User-level shared memory," *21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago, 325–336.

Reinman, G., and N. P. Jouppi. [1999]. "Extensions to CACTI," *research.compaq.com/ wrl/people/jouppi/CACTI. html.*

Rettberg, R. D., W. R. Crowther, P. P. Carvey, and R. S. Towlinson [1990]. "The Monarch parallel processor hardware design," *IEEE Computer* 23:4 (April), 18–30.

Riemens, A., K. A. Vissers, R. J. Schutten, F. W. Sijstermans, G. J. Hekstra, and G. D. La Hei [1999]."Trimedia CPU64 application domain and benchmark suite," *Proc. IEEE Int'l. Conf. on Computer Design: VLSI in Computers and Processors (ICCD'99)*, October 10–13, 1999, Austin, Tex., 580–585.

Riseman, E. M., and C. C. Foster [1972]. "Percolation of code to enhance paralled dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

Robin, J., and C. Irvine [2000]. "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor." *Proc. USENIX Security Symposium*, August 14–17, 2000, Denver, Colo.

Robinson, B., and L. Blount [1986]. *The VM/HPO 3880-23 Performance Results*, IBM Tech. Bulletin GG66-0247-00, IBM Washington Systems Center, Gaithersburg, Md.

Ropers, A., H. W. Lollman, and J. Wellhausen [1999]. *DSPstone: Texas Instruments TMS320C54x*, Tech. Rep. IB 315 1999/9-ISS-Version 0.9, Aachen University of Technology, Aaachen, Germany (*www.ert.rwth-aachen.de/Projekte/Tools/coal/dspstone_c54x/index.html*).

Rosenblum, M., S. A. Herrod, E. Witchel, and A. Gupta [1995]. "Complete computer simulation: The SimOS approach," in *IEEE Parallel and Distributed Technology* (now called *Concurrency*) 4:3, 34–43.

Rowen, C., M. Johnson, and P. Ries [1988]. "The MIPS R3010 floating-point coprocessor," *IEEE Micro* 8:3 (June), 53–62.

Russell, R. M. [1978]. "The Cray-1 processor system," *Communications of the ACM* 21:1 (January), 63–72.

Rymarczyk, J. [1982]. "Coding guidelines for pipelined processors," *Proc. Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 1–3, 1982, Palo Alto, Calif., 12–19.

Saavedra-Barrera, R. H. [1992]. "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum

Benchmarking," Ph.D. dissertation, University of California, Berkeley.

Salem, K., and H. Garcia-Molina [1986]. "Disk striping," *Proc. 2nd Int'l. IEEE Conf. on Data Engineering*, February 5–7, 1986, Washington, D.C., 249–259.

Saltzer, J. H., D. P. Reed, and D. D. Clark [1984]. "End-to-end arguments in system design," *ACM Trans. on Computer Systems* 2:4 (November), 277–288.

Samples, A. D., and P. N. Hilfinger [1988]. *Code Reorganization for Instruction Caches*, Tech. Rep. UCB/CSD 88/447, University of California, Berkeley.

Santoro, M. R., G. Bewick, and M. A. Horowitz [1989]. "Rounding algorithms for IEEE multipliers," *Proc. Ninth IEEE Symposium on Computer Arithmetic*, September 6–8, Santa Monica, Calif., 176–183.

Satran, J., D. Smith, K. Meth, C. Sapuntzakis, M. Wakeley, P. Von Stamwitz, R. Haagens, E. Zeidner, L. Dalle Ore, and Y. Klein [2001]. "iSCSI," IPS Working Group of IETF, Internet draft *www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-07.txt*.

Saulsbury, A., T. Wilkinson, J. Carter, and A. Landin [1995]. "An argument for Simple COMA," *Proc. First IEEE Symposium on High-Performance Computer Architectures*, January 22–25, 1995, Raleigh, N.C., 276–285.

Schneck, P. B. [1987]. *Superprocessor Architecture*, Kluwer Academic Publishers, Norwell, Mass.

Schroeder, B., and G. A. Gibson [2007]. "Understanding failures in petascale computers," *J. of Physics Conf. Series* 78(1), 188–198.

Schroeder, B., E. Pinheiro, and W.-D. Weber [2009]. "DRAM errors in the wild: a largescale field study," *Proc. Eleventh Int'l. Joint Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 15–19, 2009, Seattle, Wash.

Schurman, E., and J. Brutlag [2009]. "The user and business impact of server delays," *Proc. Velocity: Web Performance and Operations Conf.*, June 22–24, 2009, San Jose, Calif.

Schwartz, J. T. [1980]. "Ultracomputers," *ACM Trans. on Programming Languages and Systems* 4:2, 484–521.

Scott, N. R. [1985]. *Computer Number Systems and Arithmetic*, Prentice Hall, Englewood Cliffs, N.J.

Scott, S. L. [1996]. "Synchronization and communication in the T3E multiprocessor," *Seventh Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1–5, 1996, Cambridge, Mass.

Scott, S. L., and J. Goodman [1994]. "The impact of pipelined channels on *k*-ary *n*-cube networks," *IEEE Trans. on Parallel and Distributed Systems* 5:1 (January), 1–16.

Scott, S. L., and G. M. Thorson [1996]. "The Cray T3E network: Adaptive routing in a high performance 3D torus," *Proc. IEEE HOT Interconnects '96*, August 15–17, 1996, Stanford University, Palo Alto, Calif., 14–156.

Scranton, R. A., D. A. Thompson, and D. W. Hunter [1983]. *The Access Time Myth*," Tech. Rep. RC 10197 (45223), IBM, Yorktown Heights, N.Y.

Seagate. [2000]. *Seagate Cheetah 73 Family: ST173404LW/LWV/LC/LCV Product Manual*, Vol. 1, Seagate, Scotts Valley, Calif. (*www.seagate.com/support/disc/manuals/ scsi/29478b.pdf*).

Seitz, C. L. [1985]. "The Cosmic Cube (concurrent computing)," *Communications of the ACM* 28:1 (January), 22–33.

Senior, J. M. [1993]. *Optical Fiber Commmunications: Principles and Practice*, 2nd ed., Prentice Hall, Hertfordshire, U.K.

Sharangpani, H., and K. Arora [2000]. "Itanium Processor Microarchitecture," *IEEE Micro* 20:5 (September–October), 24–43.

Shurkin, J. [1984]. *Engines of the Mind: A History of the Computer*, W. W. Norton, New York.

Shustek, L. J. [1978]. "Analysis and Performance of Computer Instruction Sets," Ph.D. dissertation, Stanford University, Palo Alto, Calif.

Silicon Graphics. [1996]. *MIPS V Instruction Set* (see http://www.sgi.com/MIPS/arch/ ISA5/#MIPSV_indx).

Singh, J. P., J. L. Hennessy, and A. Gupta [1993]. "Scaling parallel programs for multiprocessors: Methodology and examples," *Computer* 26:7 (July), 22–33.

Sinharoy, B., R. N. Koala, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner [2005]. "POWER5 system microarchitecture," *IBM J. Research and Development*, 49:4–5, 505–521.

Sites, R. [1979]. *Instruction Ordering for the CRAY-1 Computer*, Tech. Rep. 78-CS-023, Dept. of Computer Science, University of California, San Diego.

Sites, R. L. (ed.) [1992]. *Alpha Architecture Reference Manual*, Digital Press, Burlington, Mass.

Sites, R. L., and R. Witek, (eds.) [1995]. *Alpha Architecture Reference Manual*, 2nd ed., Digital Press, Newton, Mass.

Skadron, K., and D. W. Clark [1997]. "Design issues and tradeoffs for write buffers," *Proc. Third Int'l. Symposium on High-Performance Computer Architecture*, February 1–5, 1997, San Antonio, Tex., 144–155.

Skadron, K., P. S. Ahuja, M. Martonosi, and D. W. Clark [1999]. "Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques," *IEEE Trans. on Computers* 48:11 (November).

Slater, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, Mass.

Slotnick, D. L., W. C. Borck, and R. C. McReynolds [1962]. "The Solomon computer," *Proc. AFIPS Fall Joint Computer Conf.*, December 4–6, 1962, Philadelphia, Penn., 97–107.

Smith, A. J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September), 473–530.

Smith, A., and J. Lee [1984]. "Branch prediction strategies and branch-target buffer design," *Computer* 17:1 (January), 6–22.

Smith, B. J. [1978]. "A pipelined, shared resource MIMD computer," *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, August, Bellaire, Mich., 6–8.

Smith, B. J. [1981]. "Architecture and applications of the HEP multiprocessor system," *Real-Time Signal Processing IV* 298 (August), 241–248.

Smith, J. E. [1981]. "A study of branch prediction strategies," *Proc. Eighth Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 12–14, 1981, Minneapolis, Minn., 135–148.

Smith, J. E. [1984]. "Decoupled access/execute computer architectures," *ACM Trans. on Computer Systems* 2:4 (November), 289–308.

Smith, J. E. [1988]. "Characterizing computer performance with a single number," *Communications of the ACM* 31:10 (October), 1202–1206.

Smith, J. E. [1989]. "Dynamic instruction scheduling and the Astronautics ZS-1," *Computer* 22:7 (July), 21–35.

Smith, J. E., and J. R. Goodman [1983]. "A study of instruction cache organizations and replacement policies," *Proc. 10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 132–137.

Smith, J. E., and A. R. Pleszkun [1988]. "Implementing precise interrupts in pipelined processors," *IEEE Trans. on Computers* 37:5 (May), 562–573. (This paper is based on an earlier paper that appeared in *Proc. 12th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 17–19, 1985, Boston, Mass.)

Smith, J. E., G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon [1987]. "The ZS-1 central processor," *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 199–204.

Smith, M. D., M. Horowitz, and M. S. Lam [1992]. "Efficient superscalar performance through boosting," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston, 248–259.

Smith, M. D., M. Johnson, and M. A. Horowitz [1989]. "Limits on multiple instruction issue," *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, 290–302.

Smotherman, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September), 5–15. Reprinted in *Computer Architecture Readings*, M. D. Hill, N. P. Jouppi, and G. S. Sohi, eds., Morgan Kaufmann, San Francisco, 1999, 451–461.

Sodani, A., and G. Sohi [1997]. "Dynamic instruction reuse," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo.

Sohi, G. S. [1990]. "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Trans. on Computers* 39:3 (March), 349–359.

Sohi, G. S., and S. Vajapeyam [1989]. "Tradeoffs in instruction format design for horizontal architectures," *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, 15–25.

Soundararajan, V., M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. L. Hennessy [1998]. "Flexible use of memory for replication/migration in cachecoherent DSM multiprocessors," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 342–355.

SPEC. [1989]. *SPEC Benchmark Suite Release 1.0* (October 2).

SPEC. [1994]. *SPEC Newsletter* (June).

Sporer, M., F. H. Moss, and C. J. Mathais [1988]. "An introduction to the architecture of the Stellar Graphics supercomputer," *Proc. IEEE COMPCON*, February 29–March 4, 1988, San Francisco, 464.

Spurgeon, C. [2001]. "Charles Spurgeon's Ethernet Web Site," *wwwhost.ots.utexas.edu/ ethernet/ethernet-home.html*.

Spurgeon, C. [2006]. "Charles Spurgeon's Ethernet Web SITE," *www.ethermanage.com/ ethernet/ethernet.html*.

Stenström, P., T. Joe, and A. Gupta [1992]. "Comparative performance evaluation of cache-coherent NUMA and COMA architectures," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 80–91.

Sterling, T. [2001]. *Beowulf PC Cluster Computing with Windows and Beowulf PC Cluster Computing with Linux*, MIT Press, Cambridge, Mass.

Stern, N. [1980]. "Who invented the first electronic digital computer?" *Annals of the History of Computing* 2:4 (October), 375–376.

Stevens, W. R. [1994–1996]. *TCP/IP Illustrated* (three volumes), Addison-Wesley, Reading, Mass.

Stokes, J. [2000]. "Sound and Vision: A Technical Overview of the Emotion Engine," arstechnica.com/reviews/1q00/playstation2/ee-1.html.

Stone, H. [1991]. *High Performance Computers*, Addison-Wesley, New York.

Strauss, W. [1998]. "DSP Strategies 2002," www.usadata.com/ market_research/spr_05/ spr_r127-005.htm.

Strecker, W. D. [1976]. "Cache memories for the PDP-11?," *Proc. Third Annual Int'l. Symposium on Computer Architecture (ISCA)*, January 19–21, 1976, Tampa, Fla., 155–158.

Strecker, W. D. [1978]. "VAX-11/780: A virtual address extension of the PDP-11 family," *Proc. AFIPS National Computer Conf.*, June 5–8, 1978, Anaheim, Calif., 47, 967–980.

Sugumar, R. A., and S. G. Abraham [1993]. "Efficient simulation of caches under optimal replacement with applications to miss characterization," *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 17–21, 1993, Santa Clara, Calif., 24–35.

Sun Microsystems. [1989]. *The SPARC Architectural Manual*, Version 8, Part No. 8001399-09, Sun Microsystems, Santa Clara, Calif.

Sussenguth, E. [1999]. "IBM's ACS-1 Machine," *IEEE Computer* 22:11 (November).

Swan, R. J., S. H. Fuller, and D. P. Siewiorek [1977]. "Cm*—a modular, multimicroprocessor," *Proc. AFIPS National Computing Conf.*, June 13–16, 1977, Dallas, Tex., 637–644.

Swan, R. J., A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout [1977]. "The implementation of the Cm* multi-microprocessor," *Proc. AFIPS National Computing Conf.*, June 13–16, 1977, Dallas, Tex., 645–654.

Swartzlander, E. (ed.) [1990]. *Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, Calif.

Takagi, N., H. Yasuura, and S. Yajima [1985]."High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Trans. on Computers* C-34:9, 789–796.

Talagala, N. [2000]. "Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks," Ph.D. dissertation, Computer Science Division, University of California, Berkeley.

Talagala, N., and D. Patterson [1999]. *An Analysis of Error Behavior in a Large Storage System*, Tech. Report UCB//CSD-99-1042, Computer Science Division, University of California, Berkeley.

Talagala, N., R. Arpaci-Dusseau, and D. Patterson [2000]. *Micro-Benchmark Based Extraction of Local and Global Disk Characteristics*, CSD-99-1063, Computer Science Division, University of California, Berkeley.

Talagala, N., S. Asami, D. Patterson, R. Futernick, and D. Hart [2000]. "The art of massive storage: A case study of a Web image archive," *Computer* (November).

Tamir, Y., and G. Frazier [1992]. "Dynamically-allocated multi-queue buffers for VLSI communication switches," *IEEE Trans. on Computers* 41:6 (June), 725–734.

Tanenbaum, A. S. [1978]. "Implications of structured programming for machine architecture," *Communications of the ACM* 21:3 (March), 237–246.

Tanenbaum, A. S. [1988]. *Computer Networks*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

Tang, C. K. [1976]. "Cache design in the tightly coupled multiprocessor system," *Proc. AFIPS National Computer Conf.*, June 7–10, 1976, New York, 749–753.

Tanqueray, D. [2002]. "The Cray X1 and supercomputer road map," *Proc. 13th Daresbury Machine Evaluation Workshop*, December 11–12, 2002, Daresbury Laboratories, Daresbury, Cheshire, U.K.

Tarjan, D., S. Thoziyoor, and N. Jouppi [2005]. "HPL Technical Report on CACTI 4.0," www.hpl.hp.com/techeports/2006/HPL=2006+86.html.

Taylor, G. S. [1981]. "Compatible hardware for division and square root," *Proc. 5th IEEE Symposium on Computer Arithmetic*, May 18–19, 1981, University of Michigan, Ann Arbor, Mich., 127–134.

Taylor, G. S. [1985]. "Radix 16 SRT dividers with overlapped quotient selection stages," *Proc. Seventh IEEE Symposium on Computer Arithmetic*, June 4–6, 1985, University of Illinois, Urbana, Ill., 64–71.

Taylor, G., P. Hilfinger, J. Larus, D. Patterson, and B. Zorn [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1986, Tokyo.

Taylor, M. B., W. Lee, S. P. Amarasinghe, and A. Agarwal [2005]. "Scalar operand networks," *IEEE Trans. on Parallel and Distributed Systems* 16:2 (February), 145–162.

Tendler, J. M., J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy [2002]. "Power4 system microarchitecture," *IBM J. Research and Development* 46:1, 5–26.

Texas Instruments. [2000]. "History of Innovation: 1980s," www.ti.com/corp/docs/ company/ history/1980s.shtml.

Tezzaron Semiconductor. [2004]. *Soft Errors in Electronic Memory*, White Paper, Tezzaron Semiconductor, Naperville, Ill. (*http://www.tezzaron.com/about/papers/ soft_errors_1_1_secure.pdf*).

Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs [1982].

"Alto: A personal computer," in D. P. Siewiorek, C. G. Bell, and A. Newell, eds., *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 549–572.

Thadhani, A. J. [1981]. "Interactive user productivity," *IBM Systems J.* 20:4, 407–423.

Thekkath, R., A. P. Singh, J. P. Singh, S. John, and J. L. Hennessy [1997]. "An evaluation of a commercial CC-NUMA architecture—the CONVEX Exemplar SPP1200," *Proc. 11th Int'l. Parallel Processing Symposium (IPPS)*, April 1–7, 1997, Geneva, Switzerland.

Thorlin, J. F. [1967]. "Code generation for PIE (parallel instruction execution) computers," *Proc. Spring Joint Computer Conf.*, April 18–20, 1967, Atlantic City, N.J., 27.

Thornton, J. E. [1964]. "Parallel operation in the Control Data 6600," *Proc. AFIPS Fall Joint Computer Conf., Part II*, October 27–29, 1964, San Francisco, 26, 33–40.

Thornton, J. E. [1970]. *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview, Ill.

Tjaden, G. S., and M. J. Flynn [1970]. "Detection and parallel execution of independent instructions," *IEEE Trans. on Computers* C-19:10 (October), 889–895.

Tomasulo, R. M. [1967]. "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Research and Development* 11:1 (January), 25–33.

Torrellas, J., A. Gupta, and J. Hennessy [1992]. "Characterizing the caching and synchronization performance of a multiprocessor operating system," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston (*SIGPLAN Notices* 27:9 (September), 162–174).

Touma, W. R. [1993]. *The Dynamics of the Computer Industry: Modeling the Supply of Workstations and Their Components*, Kluwer Academic, Boston.

Tuck, N., and D. Tullsen [2003]. "Initial observations of the simultaneous multithreading Pentium 4 processor," *Proc. 12th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*, September 27–October 1, 2003, New Orleans, La., 26–34.

Tullsen, D. M., S. J. Eggers, and H. M. Levy [1995]. "Simultaneous multithreading: Maximizing on-chip parallelism," *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy, 392–403.

Tullsen, D. M., S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm [1996]. "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," *Proc. 23rd Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 22–24, 1996, Philadelphia, Penn., 191–202.

Ungar, D., R. Blau, P. Foley, D. Samples, and D. Patterson [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1984, Ann Arbor, Mich., 188–197.

Unger, S. H. [1958]. "A computer oriented towards spatial problems," *Proc. Institute of Radio Engineers* 46:10 (October), 1744–1750.

Vahdat, A., M. Al-Fares, N. Farrington, R. Niranjan Mysore, G. Porter, and S. Radhakrishnan [2010]. "Scale-Out Networking in the Data Center," *IEEE Micro* 30:4 (July/August), 29–41.

Vaidya, A. S., A Sivasubramaniam, and C. R. Das [1997]. "Performance benefits of virtual channels and adaptive routing: An application-driven study," *Proc. ACM/IEEE Conf. on Supercomputing*, November 16–21, 1997, San Jose, Calif.

Vajapeyam, S. [1991]. "Instruction-Level Characterization of the Cray Y-MP Processor," Ph.D. thesis, Computer Sciences

Department, University of Wisconsin-Madison. van Eijndhoven, J. T. J., F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. I. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken [1999]. "Trimedia CPU64 architecture," *Proc. IEEE Int'l. Conf. on Computer Design: VLSI in Computers and Processors (ICCD'99)*, October 10–13, 1999, Austin, Tex., 586–592.

Van Vleck, T. [2005]. "The IBM 360/67 and CP/CMS," http://www.multicians.org/thvv/ 360-67.html.

von Eicken, T., D. E. Culler, S. C. Goldstein, and K. E. Schauser [1992]. "Active Messages: A mechanism for integrated communication and computation," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia.

Waingold, E., M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P.

Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal [1997]. "Baring it all to software: Raw Machines," *IEEE Computer* 30 (September), 86–93.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.

Wall, D. W. [1991]. "Limits of instruction-level parallelism," *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Palo Alto, Calif., 248–259.

Wall, D. W. [1993]. *Limits of Instruction-Level Parallelism*, Research Rep. 93/6, Western Research Laboratory, Digital Equipment Corp., Palo Alto, Calif.

Walrand, J. [1991]. *Communication Networks: A First Course*, Aksen Associates/Irwin, Homewood, Ill.

Wang, W.-H., J.-L. Baer, and H. M. Levy [1989]. "Organization and performance of a two-level virtual-real cache hierarchy," *Proc. 16th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–June 1, 1989, Jerusalem, 140–148.

Watanabe, T. [1987]. "Architecture and performance of the NEC supercomputer SX system," *Parallel Computing* 5, 247–255.

Waters, F. (ed.) [1986]. *IBM RT Personal Computer Technology*, SA 23-1057, IBM, Austin, Tex.

Watson, W. J. [1972]. "The TI ASC—a highly modular and flexible super processor architecture," *Proc. AFIPS Fall Joint Computer Conf.*, December 5–7, 1972, Anaheim, Calif., 221–228.

Weaver, D. L., and T. Germond [1994]. *The SPARC Architectural Manual*, Version 9, Prentice Hall, Englewood Cliffs, N.J.

Weicker, R. P. [1984]. "Dhrystone: A synthetic systems programming benchmark," *Communications of the ACM* 27:10 (October), 1013–1030.

Weiss, S., and J. E. Smith [1984]. "Instruction issue logic for pipelined supercomputers," *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1984, Ann Arbor, Mich., 110–118.

Weiss, S., and J. E. Smith [1987]. "A study of scalar compilation techniques for pipelined supercomputers," *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 105–109.

Weiss, S., and J. E. Smith [1994]. *Power and PowerPC*, Morgan Kaufmann, San Francisco.

Wendel, D., R. Kalla, J. Friedrich, J. Kahle, J. Leenstra, C. Lichtenau, B. Sinharoy, W. Starke, and V. Zyuban [2010]. "The Power7 processor SoC," *Proc. Int'l. Conf. on IC Design and Technology*, June 2–4, 2010, Grenoble, France, 71–73.

Weste, N., and K. Eshraghian [1993]. *Principles of CMOS VLSI Design: A Systems Perspective*, 2nd ed., Addison-Wesley, Reading, Mass.

Wiecek, C. [1982]. "A case study of the VAX 11 instruction set usage for compiler execution," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 1–3, 1982, Palo Alto, Calif., 177–184.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* EC-14:2 (April), 270–271.

Wilkes, M. V. [1982]. "Hardware support for memory protection: Capability implementations," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 1–3, 1982, Palo Alto, Calif., 107–116.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, Mass.

Wilkes, M. V. [1995]. *Computing Perspectives*, Morgan Kaufmann, San Francisco.

Wilkes, M. V., D. J. Wheeler, and S. Gill [1951]. *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, Cambridge, Mass.

Williams, S., A. Waterman, and D. Patterson [2009]. "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, 52:4 (April), 65–76.

Williams, T. E., M. Horowitz, R. L. Alverson, and T. S. Yang [1987]. "A self-timed chip for division," in P. Losleben, ed., *1987 Stanford Conference on Advanced Research in VLSI*, MIT Press, Cambridge, Mass.

Wilson, A. W., Jr. [1987]. "Hierarchical cache/bus architecture for shared-memory multiprocessors," *Proc. 14th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1987, Pittsburgh, Penn., 244–252.

Wilson, R. P., and M. S. Lam [1995]. "Efficient context-sensitive pointer analysis for C programs," *Proc. ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation*, June 18–21, 1995, La Jolla, Calif., 1–12.

Wolfe, A., and J. P. Shen [1991]. "A variable instruction stream extension to the VLIW architecture," *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Palo Alto, Calif., 2–14.

Wood, D. A., and M. D. Hill [1995]. "Cost-effective parallel computing," *IEEE Computer* 28:2 (February), 69–72.

Wulf, W. [1981]. "Compilers and computer architecture," *Computer* 14:7 (July), 41–47.

Wulf, W., and C. G. Bell [1972]. "C.mmp—A multi-mini-processor," *Proc. AFIPS Fall Joint Computer Conf.*, December 5–7, 1972, Anaheim, Calif., 765–777.

Wulf, W., and S. P. Harbison [1978]. "Reflections in a pool of processors—an experience report on C.mmp/Hydra," *Proc. AFIPS National Computing Conf.* June 5–8, 1978, Anaheim, Calif., 939–951.

Wulf, W. A., and S. A. McKee [1995]. "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, 23:1 (March), 20–24.

Wulf, W. A., R. Levin, and S. P. Harbison [1981]. *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York.

Yamamoto, W., M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirosky [1994]. "Performance estimation of multistreamed, superscalar processors," *Proc. 27th Annual Hawaii Int'l. Conf. on System Sciences*, January 4–7, 1994, Maui, 195–204.

Yang, Y., and G. Mason [1991]. "Nonblocking broadcast switching networks," *IEEE Trans. on Computers* 40:9 (September), 1005–1015.

Yeager, K. [1996]. "The MIPS R10000 superscalar microprocessor," *IEEE Micro* 16:2 (April), 28–40.

Yeh, T., and Y. N. Patt [1993a]. "Alternative implementations of two-level adaptive branch prediction," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 124–134.

Yeh, T., and Y. N. Patt [1993b]. "A comparison of dynamic branch predictors that use two levels of branch history," *Proc. 20th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 16–19, 1993, San Diego, Calif., 257–266.

索　引

索引中的页码为英文原书页码，与本书中页边标注的页码一致。读者可到图灵社区（ituring.com.cn/book/888）下载本索引完整版。

数字

2:1 cache rule of thumb, definition（缓存经验规则，定义），B-29
80x86，见 Intel 80x86 处理器

A

ABI，见 Application binary interface
Access bit，IA-32 descriptor table（访问位，IA-32 描述符表），B-52
Access time, 另见 Average Memory Access Time
ACID, 见 Atomicity-consistency-isolation-durability
Active low power modes, WSCs（活跃低功率模式，WSC），472
Address aliasing prediction（地址别名预测），213
Address Coalescing Unit（地址接合单元），310
Address fault, virtual memory（地址错误，虚拟存储器），B-42
Addressing modes（寻址方式），A-11
Address offset, virtual memory（地址偏移量，虚拟存储器），B-56
Address space（地址空间）
　Fermi GPU architecture（Fermi GPU 体系结构），306~307
　memory hierarchy（存储器层次结构），B-48~B-49，B-57~B-58
　Multimedia SIMD vs. GPUs,（多媒体 SIMD 对 GPU），312
　SMP/DSM shared memory（SMP/DSM 共享存储器），348
　virtual memory（虚拟存储器），B-40~B-41
Address specifier（地址标识符）
　instruction set encoding（指令集编码），A-21
Address trace, cache performance（地址跟踪，缓存性能），B-4
Address translation（地址变换）
AMD64 paged virtual memory（AMD64 分页虚拟存储器），B-55~B-56
Administrative costs, WSC *vs.* datacenters（管理成本，WSC 对数据中心），455
Advanced directory protocol（高级目标协议）
　basic function（基本功能），283
　case studies（案例研究），420~426
Advanced RISC Machine, 见 ARM（Advanced RISC Machine）
Advanced Technology Attachment disks, 见 ATA（Advanced Technology Attachment）disks
Advanced Vector Extensions（AVX）（高级向量扩展），284
Affine, loop-level parallelism dependences（仿射，循环级并行相关性），318~320
Airflow（气流），466
Airside econimization, WSC cooling systems（经济化供风，WSC 制冷系统），449

Akamai, as Content Delivery Network（Akamai，作为内容交付网络），460
Aliased variables, and compiler technology（别名变量和编译器技术），A-27~A-28
Aliases, address translation（别名，地址变换），B-38
Alignment, memory address interpretation(对齐，存储器地址解读），A-7~A-8，**A-8**
shared-memory workloads(共享存储器工作负载），369，**370**
ALUs, 见 Arithmetic-logical units
AMAT, 见 Average Memory Access Time
Amazon，452~455
Amazon Elastic Computer Cloud（EC2）（Amazon 弹性计算机云），456~457
Amazon Simple Storage Service（S3）（Amazon 简单存储服务），456~457
Amazon Web Services（AWS）（Amazon Web 服务），456~459
Amdahl's law（Amdahl 定律）
computer design principles（计算机设计原则），46~48
computer system power consumption case study（计算机系统功耗案例研究），63~64
DRAM，99
parallel computers（并行计算机），406~407
parallel processing calculations（并行处理计算），349~350
pitfalls（易犯错误），55~56
processor performance equation（处理器性能公式），51
scalar performance（标量性能），331
WSC processor cost-performance（WSC 处理器成本性能），472~473
AMD Barcelona microprocessor（AMD Barcelona 微处理器），467
AMD Opteron
address translation（地址变换），B-38
Amazon Web Services（Amazon Web 服务），457
architecture（体系结构），15
cache coherence（缓存一致性），361
data cache example(数据缓存示例），B-12~B-15，**B-13**
Google WSC servers（Google WSC 服务器），468~469
inclusion（包含性），398
manufacturing cost（制造成本），**62**
misses per instruction（每条指令的缺失数），**B-15**
MOESI protocol（MOESI 协议），362
multicore processor performance（多核处理器性能），400-401
multilevel exclusion（多级互斥），B-35
paged virtual memory example(分页虚拟存储器示例），B-54~B-57
Pentium protection（Pentium 保护），B-57
real-world server considerations（现实服务器考虑事项），52~55
server energy savings（服务器节能），**25**
snooping limitations（监听局限性），363~364
SPEC benchmarks（SPEC 基准测试），**43**
TLB during address translation(地址变换期间的 TLB），**B-47**
AMD processors（AMD 处理器）
architecture flaws vs. success（体系结构缺陷与成功），A-45
shared-memory multiprogramming workload（共享存储器多重编程工作负载），378
terminology（术语），313~315
tournament predictors（竞赛预测器），164
Virtual Machines（虚拟机），110
VMMs，129
Android OS，**324**
Antialiasing, address translation（别名消去，地址变换），B-38
Antidependences（反相关），152
Apogee Software（Apogee 软件），**A-44**
Apple iPad
ARM Cortex-A8，114
memory hierarchy basics（存储器层次结构基础），78
Application binary interface（ABI）（应用程序二进制接口）A-20
Architect-compiler writer relationship（架构师—编译器编写人员关系），A-29~A-30
Architecturally visible registers（从体系结构角度可见的寄存器）
register renaming *vs.* ROB（寄存器重命名与 ROB），208~209
Architecture, 另见 Computer architecture; CUDA（Compute Unified Device Architecture）; Instruction set architecture（ISA）; Vector architectures
Arithmetic intensity（算术密度）
Roofline model（屋顶轮廓线模型），**326**，326~327
Arithmetic-logical units（ALUs）（算术逻辑单元）
ARM Cortex-A8，**234**，236
basic MIPS pipeline（基本 MIPS 流水线），C-36
branch condition evaluation（分支条件判断），**A-19**
data forwarding（数据转发），**C-40~C-41**
data hazards requiring stalls（需要停顿的数据冒险），C-19~C-20
data hazard stall minimization（数据冒险停顿最小化），C-17~C-19
effective address cycle（实际地址周期），C-6

hardware-based execution（基于硬件的扩展），185
hardware-based speculation（基于硬件的推测），200~201，**201**
immediate operands（立即操作数），**A-12**
Intel Core i7，238
ISA operands（ISA 操作数），A-4~A-5
ISA performance and efficiency prediction（ISA 性能与效率预测），241
load interlocks（载入互锁），**C-39**
microarchitectural techniques case study（微体系结构技术案例研究），253
MIPS operations（MIPS 操作），A-35，**A-37**
MIPS pipeline control（MIPS 流水线控制），C-38~C-39
MIPS pipeline FP operations（MIPS 流水线浮点运算），C-52~C-53
MIPS R4000，C-65
operand forwarding（操作数转发），**C-19**
operands per instruction example（每个指令示例的操作数），**A-6**
parallelism（并行度），45
pipeline branch issues（流水线分支发射），C-39~C-41
pipeline execution rate（流水线执行速率），C-10~C-11
power/DLP issues（功率/DLP 发射），322
RISC classic pipeline（RISC 经典流水线），C-7
RISC instruction set（RISC 指令集），C-4
simple MIPS implementation（简单 MIPS 实现），C-31~C-33
ARM（Advanced RISC Machine）（高级 RISC 机）
control flow instructions（控制流指令），14
ISA class（ISA 分类），11
memory addressing（存储器寻址），11
operands（操作数），12
Arrays（数组）
access age（访问阶段），**91**
blocking（分块），89~90
cluster server outage/anomaly statistics（集群服务器输出/异常编译数字），**435**
examples（举例），**90**
FFT kernel（FFT 内核），I-7
Google WSC servers（Google WSC 服务器），469
Layer 3 network linkage（L3 网络链路），**445**
loop interchange（循环交换），88~89
loop-level parallelism dependences（循环级并行相关），318~319
WSC memory hierarchy（WSC 存储器层次结构），445
WSCs，443
Array switch, WSCs,（阵列交换机，WSC），443~444
ASCII character format（ASCII 字符格式），12，A-14
Assembly language（汇编语言），2
Associativity，另见 Set associativity
ATA（Advanced Technology Attachment）disks（(高级技术附件）磁盘）
server energy savings（服务器节能），**25**
ATM systems（ATM 系统）
server benchmarks（服务器基准测试），41
Atomic exchange（原子交换）
lock implementation（锁实现），389-390
synchronization（同步），387-388
Atomic instructions（原子指令）
Core i7，329
Fermi GPU，308
T1 multithreading unicore performance（T1 多线程单核性能），**229**
Atomicity-consistency-isolation-durab ility(ACID), *vs.* WSC storage
(原子性—一致性–隔离性–持久性与 WSC 存储），439
Atomic operations（原子操作）
cache coherence（缓存一致性），360~361
snooping cache coherence implementation（监听缓存一致性实现），365
Availability（可用性）
computer architecture（计算机体系结构），11，15
data on Internet（互联网上的数据），344
fault detection（错误检测），57~58
loop-level parallelism（循环级并行），217~218
mainstream computing classes（主流计算类别），5
modules（模块），34
open-source software（开源软件），457
RAID systems（RAID 系统），60
as server characteristic（作为服务器特性），7
servers（服务器），16
source operands（源操作数），C-74
WSCs，8，433~435，438~439
Average Memory Access Time（AMAT）（存储器平均访问时间），B-30~B-31
AVX, 见 Advanced Vector Extensions（AVX）
AWS, 见 Amazon Web Services（AWS）

B

Backside bus, centralized shared-memory multiprocessors(背部总线，集中式共享存储器多处理器），351
Bandwidth，另见 Throughput
Banked memory，另见 Memory banks
Banks Fermi GPUs（分组，Fermi GPU），2[illegible]
Barriers（屏障）
commercial workloads（商业工作负载），378
hardware primitives（硬件基元），387

synchronization（同步），298，**313**，329
BARRNet，见 Bay Area Research Network
Base field, IA-32 descriptor table（基址字段，IA-32 描述符表），B-52~B-53
Basic block, ILP（基本块，ILP），149
Batch processing workloads（批处理工作负载）
WSC goals/requirements（WSC 目标/需求），433
WSC MapReduce and Hadoop（WSC MapReduce 和 Hadoop），437~438
Between instruction exceptions（指令间异常），C-45
Big Endian（大端）
memory address interpretation（存储器地址解释），A-7
MIPS data transfers（MIPS 数据传输），A-34
Bigtable（Google），438，441
Binary code compatibility（二进制代码兼容性）
VLIW processors（VLIW 处理器），196
Binary-coded decimal, definition（二进制编码十进制数，定义），A-14
Bing search（必应搜索）
delays and user behavior（延迟与用户行为），**451**
latency effects（延迟效果），450~452
WSC processor cost-performance（WSC 处理器成本性能），473
Bisection bandwidth, WSC array switch（二分带宽，WSC 阵列交换机），443
Bit selection, block placement（位选择，块放置），B-7
Block addressing（块寻址）
block identification（块识别），B-7~B-8
interleaved cache banks（交错缓存组），**86**
memory hierarchy basics（存储器层次结构基础），74
Block identification（块识别）
memory hierarchy considerations（存储器层次结构考虑事项），B-7~B-9
virtual memory（虚拟存储器），B-44~B-45
Blocking（分块）
benchmark fallacies（基础测试谬论），56
Blocking calls, shared-memory multiprocessor workload（分块调用，共享存储器多处理器工作负载），369
Blocking factor, definition（分块因数，定义），90
Block offset（块偏移），B-7~B-8
Block replacement（块替换）
memory hierarchy considerations（存储器层次结构考虑事项），B-9~B-10
virtual memory（虚拟存储器），B-45
Blocks, 另见 Cache block; Thread Block
Block size（块大小）
access time（访问时间），**B-28**
memory hierarchy basics（存储器层次结构基础），76
miss rate（缺失率），**B-27**
Body of Vectorized Loop（向量循环体），292，313
Bose-Einstein formula, definition（玻色-爱因斯坦公式，定义），30
Bounds checking, segmented virtual memory（边界检查，分段虚拟存储器），B-52
Branch delay slot（分支延迟时隙）
characteristics（特性），C-23~C-25
control hazards（控制冒险），C-41
MIPS R4000，C-64
scheduling（调度），**C-24**
Branches（分支）
canceling（取消），C-24~C-25
conditional branches（条件分支），300~303，**A-17**，**A-19~A-20**，A-21
control flow instructions（控制流指令），A-16，A-18
delayed（延迟），**C-23**
delay slot（延迟时隙），**C-65**
MIPS control flow instructions（MIPS 控制流指令），A-38
MIPS operations（MIPS 操作），A-35
nullifying（废除），C-24~C-25
RISC instruction set（RISC 指令集），C-5
Branch folding, definition（分支折合，定义），206
Branch hazards（分支冒险）
basic considerations（基本考虑事项），C-21
penalty reduction（降低代价），C-22~C-25
pipeline issues（流水线发射），C-39，C-42
scheme performance（方案性能），C-25~C-26
stall reduction（减少停顿），**C-42**
Branch history table, basic scheme（分支历史表，基本方案），C-27~C-30
Branch offsets, control flow instructions（分支偏移量，控制流指令），A-18
Branch penalty（分支代价）
examples（示例），**205**
instruction fetch bandwidth（指令提取带宽），203~206
reduction（降低），C-22~C-25
simple scheme examples（简单方案举例），**C-25**
Branch prediction（分支预测）
accuracy（精度），**C-30**
branch cost reduction（降低分支成本），162~167
correlation（相关性），162~164
cost reduction（降低成本），C-26
dynamic（动态），C-27~C-30
ideal processor（理想处理器），214
ILP exploitation（ILP 开发），**201**
instruction fetch bandwidth（指令提取带宽），205
integrated instruction fetch units（集成指令提取单元），207

Intel Core i7，166~167，239~241
misprediction rates on SPEC89（SPEC89 测得的预测错误率），**166**
static（静态），C-26~C-27
two-bit predictor comparison（两位预测比较），**165**
Branch-prediction buffers, basic considerations（分支预测缓冲区，基本考虑事项），C-27~C-30，**C-29**
Branch stalls, MIPS R4000 pipeline（分支停顿，MIPS R4000 流水线），**C-67**
Branch-target address（分支目标地址）
branch hazards（分支冒险），**C-42**
MIPS control flow instructions（MIPS 控制流指令），A-38
MIPS pipeline（MIPS 流水线），C-36，**C-37**
MIPS R4000，C-25
pipeline branches（流水线分支），C-39
RISC instruction set（RISC 指令集），C-5
Branch-target buffers（分支目标缓冲区）
ARM Cortex-A8，233
branch hazard stalls（分支冒险停顿），**C-42**
example（示例），**203**
instruction fetch bandwidth（分支提取带宽），203~206
instruction handling（指令处理），**204**
MIPS control flow instructions（MIPS 控制流指令），A-38
Branch-target cache,见 Branch-target buffers
Bubbles（冒泡）
stall（停顿），C-13
Buffers（缓冲区）
branch-prediction（分支预测），C-27~C-30，**C-29**
branch-target（分支目标），203~206，**204**，233，A-38，**C-42**
memory（存储器），208
MIPS scoreboarding（MIPS 记分卡），C-74
ROB，184~192，**188~189**，**199**，208~210，238
TLB，见 Translation lookaside buffer
translation buffer（转换缓冲区），B-45~B-46
write buffer（写缓冲区），B-11，B-14，B-32，B-35~B-36
Buses（总线），351
Bypassing，另见 Forwarding
Byte offset（字节偏移量）
misaligned addresses（非对齐地址），A-8
PTX instructions（PTX 指令），300
Bytes（字节）
aligned/misaligned addresses（对齐/非对齐地址），**A-8**
arithmetic intensity example（算术密度示例），**286**
memory address interpretation（存储器地址解释），A-7~A-8
MIPS data transfers（MIPS 数据传输），A-34
MIPS data types（MIPS 数据类型），A-34
operand types/sizes（操作数类型/大小），A-14
per reference, vs. block size（每次引用对块大小），**378**

C

CAC,见 Computer aided design（CAD）tools
Cache bandwidth（缓存带宽）
caches（缓存），78
multibanked caches（多组缓存），85~86
nonblocking caches（非阻塞缓存），83~85
pipelined cache access（流水线缓存访问），82
Cache block（缓存块），B-2
Cache coherence（缓存一致性）
advanced directory protocol case study（高级目录协议案例研究），420~426
basic considerations（基本考虑事项），112~113
directory-based,见 Directory-based cache coherence
enforcement（实施），354~355
extensions（扩展），362~363
hardware primitives（硬件原语），388
latency hiding with speculation（以推测隐藏延迟），396
lock implementation（锁实现），389~391
mechanism（方案），**358**
memory hierarchy basics（存储器层次结构基础），75
multiprocessor-optimized software（多处理器优化软件），409
multiprocessors（多处理器），352~353
protocol definitions（协议定义），354~355
single-chip multicore processor case study（单芯片多核处理器案例研究），412~418
single memory location example（单存储器位置示例），**352**
snooping,见 Snooping cache coherence
state diagram（状态图），**361**
steps and bus traffic examples（步长与总线缓冲区示例），**391**
write-back cache（写回缓存），**360**
Cache definition（缓存定义），B-2
Cache hit（缓存命中），B-2
Cache latency, nonblocking cache（缓存延迟，非阻塞缓存），83~84
Cache miss（缓存缺失），B-2
Cache optimizations（缓存优化）
basic categories（基本类别），B-22
basic optimizations（基本优化），**B-40**
case studies（案例研究），131~133
compiler-controlled prefetching（编译器控制的预取）92~95

compiler optimizations（编译器优化），87~90
critical word first（关键字优化），86~87
energy consumption（能耗），**81**
hardware instruction prefetching（硬件指令预取），91~92，**92**
hit time reduction（缩短命中时间），B-36~B-40
miss categories（缺失类别），B-23~B-26
miss penalty reduction（降低缺失代价），B-30~B-36
miss rate reduction（降低缺失率），B-26~B-30
multibanked caches（多组缓存），85~86，**86**
nonblocking caches（非阻塞缓存），83~85，**84**
overview（概述），78~79
pipelined cache access（流水线缓存访问），82
simple first-level caches（简单第一级缓存），79~80
techniques overview（技术概述），**96**
way prediction（路预测），81~82
write buffer merging（写缓冲区合并），87，**88**
Cache organization（缓存组织方式）
blocks（块），B-7，**B-8**
Opteron data cache（Opteron数据缓存），B-12~B-13，**B-13**
optimization（优化），B-19
performance impact（性能影响），B-19
Cache performance（缓存性能）
average memory access time（存储器平均访问时间），B-16~B-20
basic considerations（基本考虑事项），B-3~B-6，B-16
basic equations（基本公式），**B-22**
basic optimizations（基本优化），**B-40**
cache optimization（缓存优化），**96**
case study（案例研究），131~133
example calculation（示例计算），B-16~B-17
out-of-order processors（乱序处理器），B-20~B-22
prediction（预测），125~126
Cache prefetch, cache optimization（缓存预测，缓存优化），92
Caches，另见 Memory hierarchy
Cache size（缓存大小）
access time（缓存时间），**77**
AMD Opteron example（AMD Opteron示例），B-13~B-14
energy consumption（能耗），**81**
highly parallel memory systems（高度并行存储器系统），133
memory hierarchy basics（存储器层次结构基础），76
misses per instruction（每条指令的缺失数），**126**，371
miss rate（缺失率），**B-24~B-25**
*vs*miss rate（缺失率对比），**B-27**
miss rate reduction（降低缺失率），B-28
multilevel caches（多级缓存），**B-33**
relative execution time（相对执行时间），B-34
CACTI
cache optimization（缓存优化），79~80，**81**
memory access times（存储器访问题意），**77**
Caller saving, control flow instructions（调用方保存，控制流指令），A-19~A-20
Call gate（调用门）
IA-32 segment descriptors（IA-32段描述符），B-53
segmented virtual memory（分段虚拟存储器），B-54
Calls（调用）
compiler structure（编译器结构），A-25~A-26
control flow instructions（控制流指令），**A-17**，A-19~A-21
CUDA Thread（CUDA线程），297
dependence analysis（相关性分析），321
high-level instruction set（高级指令集），A-42~A-43
invocation options（调用选项），A-19
MIPS control flow instructions（MIPS控制流指令），A-38
MIPS registers（MIPS寄存器），12
multiprogrammed workload（多重编程工作负载），378
NVIDIA GPU Memory structures（NVIDIA GPU存储器结构），304~305
return address predictors（返回地址预测器），206
shared-memory multiprocessor workload（共享存储器多处理器工作负载），369
user-to-OS gates（“用户到OS”门），B-54
Canceling branch, branch delay slots（取消分支，分支延迟时隙），C-24~C-25
Canonical form, AMD64 paged virtual memory（规范形式，AMD64分页虚拟存储器），B-55
Capacity misses（容量缺失），B-23
Capital expenditures（CAPEX）（资本性支出）
WSC costs（WSC成本），452~455，**453**
WSC Flash memory（WSC闪存），475
WSC TCO case study（WSC TCO案例研究），476~478
Case statements（Case语句）
control flow instruction addressing modes（控制流指令寻址方式），A-18
return address predictors（返回地址预测器），206
Case studies（案例研究）
advanced directory protocol（高级目录协议），420~426
cache optimization（缓存优化），131~133
chip fabrication cost（芯片制造成本），61~62
computer system power consumption（计算机系统功耗），63~64
directory-based coherence（目录式一致性），418~420

highly parallel memory systems（高度并行的存储器系统），133~136
instruction set principles（指令集原理），A-47~A-54
memory hierarchy（存储器层次结构），B-60~B-67
microarchitectural techniques（微体系结构技术），247~254
pipelining example（流水线示例），C-82~C-88
single-chip multicore processor(单芯片多核心处理器)，412~418
vector kernel on vector processor and GPU（向量处理器和 GPU 上的向量内核），334~336
WSC resource allocation（WSC 资源分配），478~479
WSC TCO（WSC TCO），476~478
C/C++ language（C/C++语言）
hardware impact on software development（硬件对软件开发的影响），4
loop-level parallelism dependences（循环级并行相关性），318，320~321
NVIDIA GPU programming（NVIDIA GPU 编程），289
return address predictors（返回地址预测器），206
CDC, 见 Control Data Corporation
CDF, datacenter（CDF，数据中心），**487**
Centralized shared-memory multiprocessors（集中式共享存储器多处理器）
basic considerations（基本考虑事项），351~352
basic structure（基本结构），346~347，**347**
cache coherence（缓存一致性），352~353
cache coherence enforcement（缓存一致性实施），354~355
cache coherence example（缓存一致性示例），357~362
cache coherence extensions(缓存一致性扩展)，362~363
invalidate protocol implementation（失效协议实现），356~357
SMP and snooping limitations（SMP 和监听局限性），363~364
snooping coherence implementation(监听一致性实现)，365~366
snooping coherence protocols（监听一致性协议），355~356
Central processing unit（CPU）（中央处理器）
Amdahl's law（Amdahl 定律），48
average memory access time（存储器平均访问时间），B-17
cache performance（缓存性能），B-4
coarse-grained multithreading（粗粒度多线程），224
exception stopping/restarting（异常停顿/重新启动），C-47
extensive pipelining（全面流水线），C-81
Google server usage（Google 服务器应用），**440**
vs GPUs（GPU 对比），288
instruction set complications（指令集复杂性），C-50
MIPS implementation（MIPS 实现），C-33~C-34
MIPS precise exceptions(MIPS 精确异常)，C-59~C-60
MIPS scoreboarding（MIPS 记分卡），C-77
pipeline branch issues（流水线分支发射），C-41
pipelining exceptions（流水线异常），C-43~C-46
pipelining performance（流水线性能），C-10
SPEC server benchmarks（SPEC 服务器基准测试），40
Central processing unit（CPU）time（中央处理器单元时间）
execution time（执行时间），36
modeling（建模），B-18
processor performance calculations(处理器性能计算)，B-19~B-21
processor performance equation（处理器性能公式），49~51
processor performance time（处理器性能时间），49
Chaining（链接）
VMIPS，268~269
Character（字符）
floating-point performance（浮点性能），A-2
operand type（操作数类型），A-13~A-14
operand types/sizes（操作数类型/大小），12
Chillers（冷凝机）
Google WSC，466，468
WSC containers（WSC 集装箱），464
WSC cooling systems（WSC 制冷系统），448~449
Chime（钟鸣），309
Chipkill
memory dependability（存储器可靠性），104~105
WSCs，473
Circulating water system（CWS）（循环水系统(CWS)）
cooling system design（制冷系统设计），**448**
WSCs，448
Clean block, definition（清洁块，定义），B-11
Climate Savers Computing Initiative, power supply efficiencies (计算产业拯救气候计划，电源效率），**462**
Clock cycles（时钟周期）
basic MIPS pipeline（基本 MIPS 流水线），C-34~C-35
and branch penalties（与分支代价），**205**
cache performance（缓存性能），B-4
FP pipeline（浮点流水线），**C-66**
full associativity（全相联），B-23
GPU conditional branching（GPU 条件分支），303
ILP exploitation（ILP 开发），197，200
ILP exposure（揭示 ILP），157
instruction fetch bandwidth（指令提取带宽），202~203
instruction steps（指令步骤），173~175

Intel Core i7 branch predictor（Intel Core i7 分支预测器），166
MIPS exceptions（MIPS 异常），C-48
MIPS pipeline（MIPS 流水线），**C-52**
MIPS pipeline FP operations（MIPS 流水线浮点运算），C-52~C-53
MIPS scoreboarding（MIPS 记分卡），C-77
miss rate calculations（缺失率计算），B-31~B-32
multithreading approaches（多线程方法），225~226
pipelining performance（流水线性能），C-10
processor performance equation（处理器性能公式），49
RISC classic pipeline（RISC 经典流水线），C-7
Sun T1 multithreading（Sun T1 多线程），226-227
vector execution time（向量执行时间），269
vector multiple lanes（向量多车道），271~273
VLIW processors（VLIW 处理器），**195**
Clock cycles per instruction（CPI）（每条指令的时钟周期数）
addressing modes（寻址方式），A-10
branch schemes（分支方案），C-25~C-26，**C-26**
cache behavior impact（缓存行为影响），B-18~B-19
cache hit calculation（缓存命中计算），B-5
data hazards requiring stalls（需要停顿的数据冒险），C-20
extensive pipelining（全面流水线），C-81
floating-point calculations（浮点计算），50~52
ILP concepts（ILP 概念），148~149，**149**
ILP exploitation（ILP 开发），192
MIPS R4000 performance（MIPS R4000 性能），**C-69**
miss penalty reduction（降低缺失代价），B-32
multiprocessing/multithreading-based performance（基于多重处理/多线程的性能），398~400
multiprocessor communication calculations（多处理器通信计算），350
pipeline branch issues（流水线分支发射），C-41
pipeline with stalls（具有停顿的流水线），C-12~C-13
pipeline structural hazards（流水线结构性冒险），C-15~C-16
pipelining concept（流水线概念），C-3
processor performance calculations（处理器性能计算），218~219
processor performance time（处理器性能时间），49~51
processor speed（处理器速度），244
shared-memory workloads（共享存储器工作负载），369~370
simple MIPS implementation（简单 MIPS 实现），C-33~C-34
structural hazards（结构性冒险），C-13
Sun T1 multithreading unicore performance（Sun T1 多线程单核心性能），**229**
Sun T1 processor（Sun T1 处理器），**399**
Tomasulo's algorithm（Tomasulo 算法），181
Clock cycle time（时钟周期时间）
associativity（相联度），B-29
average memory access time（存储器平均访问时间），B-21~B-22
cache optimization（缓存优化），B-19~B-20，B-30
cache performance（缓存性能），B-4
CPU time equation（CPU 时间公式），49~50，B-18
MIPS implementation（MIPS 实现），**C-34**
miss penalties（缺失代价），219
pipeline performance（流水线性能），C-12，C-14~C-15
pipelining（流水化），C-3
Clock periods, processor performance equation（时钟周期，处理器性能公式），48~49
Clock rate（时钟速率）
DDR DRAMS and DIMMS（DDR DRAMS 和 DIMMS），101
ILP for realizable processors（可实现处理器的 ILP），218
microprocessors（微处理器），**24**
MIPS pipeline FP operations（MIPS 流水线浮点），C-53
multicore processor performance（多核处理器性能），400
processor speed（处理器速度），**244**
Clocks, processor performance equation（时钟，处理器性能公式），48~49
Clock skew, pipelining performance（时钟偏移，流水线性能），C-10
Clock ticks（时钟嘀嗒）
cache coherence（缓存一致性），**391**
processor performance equation（处理器性能公式），48~49
Cloud computing（云计算）
basic considerations（基本考虑事项），455~461
clusters（集群），345
provider issues（提供商问题），471~472
Clusters（集群）
cloud computing（云计算），345
computer class（计算机类别），5
Google WSC servers（Google WSC 服务器），469
outage/anomaly statistics（储运/异常编译数字），**435**
WSC forerunners（WSC 先驱），435~436
WSC storage（WSC 存储），442~443
CMOS
DRAM，**99**
Coarse-grained multithreading, definition（粗粒度多线程，定义），224~226
Code generation（代码生成）

compiler structure（编译器结构），A-25~A-26，A-30
dependences（相联性），220
general-purpose register computers（通用寄存器计算机），**A-6**
ILP limitation studies（ILP局限性研究），220
loop unrolling/scheduling（循环展开/调度），162
Code size（代码大小）
architect-compiler considerations(体系结构编译器考虑事项），A-30
benchmark information（基准测试信息），A-2
comparisons（比较），**A-44**
flawless architecture design（无瑕疵体系结构设计），A-45
instruciton set encoding（指令集编码），A-22~A-23
ISA and compiler technology（ISA和编译器技术），A-43~A-44
loop unrolling（循环展开），160~161
multiprogramming（多重编程），375~376
PMDs，6
RISCs，A-23~A-24
VAX design（VAX设计），A-45
VLIW model（VLIW模型），195~196
Coerced exceptions（强制异常），C-45
Coherence, 见 Cache coherence
Coherence misses（一致性缺失），366
Cold-start misses, definition（冷启动缺失，定义），B-23
Collision misses, definition（冲突缺失，定义），B-23
Column access strobe（CAS），DRAM（列访问选通，DRAM），98~99
Column major order（列主序）
blocking（分块），89
stride（步幅），278
COMA, 见 Cache-only memory architecture
Commercial workloads（商业工作负载）
execution time distribution（执行时间分布），**369**
symmetric shared-memory multiprocessors(对称共享存储器多处理器），367~374
Commit stage, ROB instruction(提交阶段，ROB指令），186~187，**188**
Commodities（大众化商品）
Amazon Web Services（Amazon Web服务），456~457
array switch（阵列交换机），443
cloud computing（云计算），455
cost vs. price（成本对价格），32~33
cost trends（成本趋势），27~28，32
Ethernet rack switch（以太网机架交换机），442
HPC hardware（HPC硬件），436
shared-memory multiprocessor(共享存储器多处理器），441
Common data bus（CDB）（通用数据总线(CDB））
dynamic scheduling with Tomasulo's algorithm（用Tomasulo算法实现动态调度），172，175
FP unit with Tomasulo's algorithm（采用Tomasulo算法的浮点单元），**185**
reservation stations/register tags（保留站点/寄存器标签），**177**
Tomasulo's algorithm（Tomasulo算法），**180**，182
Communication mechanism（通信机制）
multiprocessor communication calculations（多处理器通信计算），350
SMP limitations（SMP限制条件），363
Communication subnets, 见 Interconnection networks
Communication subsystems, 见 Interconnection networks
Compiler-controlled prefetching, miss penalty/rate reduction (编译器控制的预取，降低缺失代价/缺失率），92~95
Compiler optimizations（编译器优化）
blocking（分块），89~90
cache optimization（缓存优化），131~133
compiler assumptions（编译器假设），A-25~A-26
consistency model（一致性模型），396
loop interchange（循环交换），88~89
miss rate reduction（降低缺失率），87~90
passes（扫描遍数），**A-25**
performance impact（性能影响），A-27
types and classes（类型与分类），**A-28**
Compiler scheduling（编译器调度），C-71
Compiler technology（编译器技术）
architecture decisions（体系结构决策），A-27~A-29
ISA and code size（ISA和代码大小），A-43~A-44
multimedia instruction support（多媒体指令支持），A-31~A-32
register allocation（寄存器分配），A-26~A-27
structure（结构），A-24~A-26，**A-25**
Compiler writer-architect relationship（编译器编写人员-架构师关系），A-29~A-30
Compulsory misses（强制缺失），B-23
Computer aided design（CAD）tools, cache optimization(计算机辅助设计工具，缓存优化），79~80
Computer architecture,另见 Architecture
Computer chipfabrication（计算机芯片制造）
Cost case study（成本案例研究），61~62
Computer classes（计算机分类），5
Computer design principles（计算机设计原理）
Amdahl's law（Amdahl定律），46~48
common case（常见情景），45~46
parallelism（并行度），44~45
principle of locality（局域性原理），45

processor performance equation（处理器性能公式），48~52
Computer history, technology and architecture（计算机历史，技术与体系结构），2~5
Computer room air-conditioning（CRAC），WSC infrastructure（计算机室空调，WSC基础设施），448~449
Compute Unified Device Architecture, 见 CUDA（Compute Unified Device Architecture）
Conditional branches（条件分支）
branch folding（分支折合），206
compare frequencies（比较频率），A-20
compiler performance（编译器性能），C-24~C-25
control flow instructions（控制流指令），14，A-16，**A-17**，A-19，A-21
evaluation（求值），**A-19**
GPUs，300~303
ideal processor（理想处理器），214
ISAs，A-46
MIPS control flow instructions（MIPS控制流指令），A-38，**A-40**
predictor misprediction rates（预测器错误预测率），166
PTX instruction set（PTX指令集），298~299
static branch prediction（静态分支预测），C-26
types（类型），**A-20**
vector-GPU comparison（向量-GPU比较），311
Condition codes（条件码），C-5
Conflict misses（冲突缺失），B-23
Connection Multiprocessor（连接多处理器），2
Consistency, 见 Memory consistency
Containers（集装箱）
airflow（气流），**466**
Google WSCs，464~465，**465**
Context Switching（上下文切换），106，B-49
definition（定义），106，B-49
Control Data Corporation（CDC）6600（控制数据公司6600）
dynamically scheduling with scoreboard（用记分卡动态调度），C-71~C-72
MIPS scoreboarding（MIPS记分卡），C-75~C-77
Control Data Corporation（CDC）STAR-100（控制数据公司STAR-100）
peak performance *vs.* start-up overhead（峰值性能对启动开销），331
Control dependences（控制相关性）
data dependence（数据相关性），150
hardware-based speculation（基于硬件的推测），163
ILP，154~156
ILP hardware model（ILP硬件模型），214
Tomasulo's algorithm（Tomasulo算法），170
vector mask registers（向量遮罩寄存器），275~276
Control flow instructions（控制流指令）
addressing modes（寻址方式），A-17~A-18
basic considerations（基本考虑事项），A-16~A-17，A-20~A-21
classes（分类），**A-17**
conditional branch options（条件分支选项），A-19
hardware vs. software speculation（硬件对软件推测），221
procedure invocation options（过程调用选项），A-19~A-20
Control hazards（控制冒险），C-11
Control instructions（控制指令）
VAX，**B-73**
Controller transitions（控制器转换）
directory-based（基于目录），**422**
snooping cache（监听缓存），**421**
Control Processor（控制处理器），309
Conventional datacenters, *vs.* WSCs（传统数据中心对WSC），436
Cooling systems（制冷系统）
Google WSC，465~468
mechanical design（机械设计），**448**
WSC infrastructure（WSC基础设施），448~449
Core definition（核心定义），15
Correlating branch predictors, branch costs（相关分支预测器，分支成本），162~163
Cost（成本）
Amazon EC2，**458**
Amazon Web Services（Amazon Web服务），457
branch predictors（分支预测器），162-167，C-26
chip fabrication case study（芯片制造案例研究），61~62
cloud computing providers（云计算提供商），471~472
MapReduce calculations（MapReduce计算），458~459，**459**
memory hierarchy design（存储器层次结构设计），72
multiprocessor cost relationship（多处理器成本关系），409
multiprocessor linear speedup（多处理器线程加速），407
server calculations（服务器计算），**454**，454~455
server usage（服务器使用），7
speculation（推测），210
tournament predictors（竞赛预测器），164~166
WSC array switch（WSC阵列交换机），443
WSC vs. datacenters（WSC对数据中心），455~456
WSC efficiency（WSC效率），450~452
WSC facilities（WSC设施），472

WSC network bottleneck（WSC 网络瓶颈），461
WSCs *vs.* servers（WSC 对服务器），434
WSC TCO case study（WSC TCO 案例研究），476~478
Cost associativity, cloud computing（成本相关性，云计算），460~461
Cost-performance（成本-性能）
computer trends（计算机趋势），3
extensive pipelining（全面流水化），C-80~C-81
IBM eServer p5 processor（IBM eSever p5 处理器），**409**
WSC Flash memory（WSC 闪存），474~475
WSC goals/requirements（WSC 目标/需求），433
WSC hardware inactivity（WSC 硬件不活跃），474
WSC processors（WSC 处理器），472~473
Cost trends（成本趋势）
integrated circuits（集成电路），28~32
manufacturing vs. operation（制造对运行），33
overview（概述），27
price（价格），32~33
time, volume, commoditization（时间、体积、大众商品化），27~28
CPI, 见 Clock cycles per instruction
CPU, 见 Central processing unit
CRAC, 见 Computer room air-conditioning
Cray-1
peak performance *vs.* start-up overhead（峰值效率对启动开销），331
vector performance（向量性能），332
as VMIPS basis（作为 VMIPS 基础），264，270~271，276~277
Cray，C-90
Cray T90, memory bank calculations（Cray T90，存储器组计算），276
Cray Y-MP
vector architecture programming（向量体系结构编程），**281**，281~282
Critical word first（关键字优先），86~87
Carry-skip adder（CSA）（进位跳跃加法器）
C# language, hardware impact on software development(C# 语言，硬件对软件开发的影响），4
CUDA（Compute Unified Device Architecture）（计算统一设备体系结构）
GPU conditional branching（GPU 条件分支），303
GPUs *vs.* vector architectures（GPU 对向量体系结构），310
NVIDIA GPU programming（NVIDIA GPU 编程），289
PTX，298，300
sample program（示例程序），289~290
SIMD instructions（SIMD 指令），297
terminology（术语），313~315
CUDA Thread（CUDA 线程），292，313
CYBER 180/990, precise exceptions（CYBER 180/990，精确异常），C-59
Cycles, processor performance equation（周期，处理器性能计算），49

D

DaCapo benchmarks（DaCapo 基准测试）
ISA，242
SMT，230~231，**231**
DAMQs, 见 Dynamically allocatable multi-queues
Database program speculation, via multiple branches（数据库程序推测，通过多分支），211
Data cache（数据缓存）
ARM Cortex-A8，**236**
cache optimization（缓存优化），B-33，B-38
cache performance（缓存性能），B-16
GPU Memory（GPU 存储器），306
ISA，241
locality principle（局域性原理），B-60
MIPS R4000 pipeline（MIPS R4000 流水线），C-62~C-63
multiprogramming（多重编程），374
page level write-through（页级直写），B-56
RISC processor（RISC 处理器），C-7
structural hazards（结构性冒险），C-15
TLB，B-46
Data cache miss（数据缓存缺失）
applications vs. OS（应用程序对操作系统），**B-59**
cache optimization（缓存优化），B-25
Intel Core i7，240
Opteron，B-12~B-15
sizes and associativities（大小与相联性），B-10
writes（写入），B-10
Data cache size, multiprogramming（数据缓存大小，多重编程），376~377
Datacenters（数据中心）
CDF，**487**
cooling systems（制冷系统），449
layer 3 network example（L3 网络示例），445
PUE statistics（PUE 统计数字），**451**
tier classifications（层分类），**491**
WSC costs（WSC 成本），455~456
WSC efficiency measurement（WSC 效率测量），450~452
Data dependences（数据相关性）
data hazards（数据冒险），167~168

dynamically scheduling with scoreboard（以记分卡进行动态调度），C-71
hazards（冒险），153~154
ILP，150~152
ILP hardware model（ILP硬件模型），214~215
ILP limitation studies（ILP局限性研究），220
vector execution time（向量执行时间），269
Data fetching（数据提取）
ARM Cortex-A8，**234**
directory-based cache coherence protocol example(目录式缓存一致性协议示例），382~383
dynamically scheduled pipelines（动态调度流水线），C-70~C-71
ILP, instruction bandwidth（ILP，指令带宽），202~207
MIPS R4000，C-63
snooping coherence protocols（监听一致性协议），355~356
Data flow（数据流）
control dependence（控制相关性），154~156
dynamic scheduling（动态调度），168
ILP limitation studies（ILP局限性研究），220
Data flow execution, hardware-based speculation（数据流执行，基于硬件的推测），184
Data hazards（数据冒险）
stall minimization by forwarding（通过转发将停顿降至最低），C-16~C-19，**C-18**
stall requirements（停顿需求），C-19~C-21
VMIPS，264~267
Data-level parallelism（DLP）（数据级并行），9
Data-race-free, synchronized programs（无数据竞赛，同步程序），394
Data races, synchronized programs（数据竞赛，同步程序），394
Data transfers（数据传送）
cache miss rate calculations（缓存缺失率计算），B-16
computer architecture（计算机体系结构），15
gather-scatter（集中/分散），281，291
instruction operators（指令运算符），**A-15**
MIPS, addressing modes（MIPS，寻址模式），A-34
MIPS64 instruction subset（MIPS64指令子集），**A-40**
MIPS64 ISA formats（MIPS64 ISA格式），14
MIPS operations（MIPS操作），A-36~A-37
MMX，283
multimedia instruction compiler support（多媒体指令编译器支持），A-31
operands（操作数），**A-12**
PTX，305
SIMD extensions（SIMD扩展），284
"typical" programs（"典型"程序），A-43
VAX，**B-73**
vector vs. GPU（向量对CPU），300
Data trunks, MIPS scoreboarding(数据干线，MIPS记分卡），C-75
Data types（数据类型）
architect-compiler writer relationship（架构师-编译器编写人员关系），A-30
desktop computing（桌面计算），A-2
MIPS，A-34，A-36
MIPS64 architecture（MIPS64体系结构），A-34
multimedia compiler support（多媒体编译器支持），A-31
operand types/sizes（操作数类型/大小），A-14~A-15
SIMD Multimedia Extensions（SIMD多媒体扩展），282~283
DAXPY loop（DAXPY循环）
memory bandwidth（存储器带宽），332
MIPS/VMIPS calculations（MIPS/VMIPS计算），267~268
peak performance vs. start-up overhead（峰值性能对启动开销），331
VLRs，274~275
D-caches（D缓存）
case study examples（案例研究示例），B-63
way prediction（路预测），81~82
DDR, 见 Double data rate
Deadlock（死锁）
cache coherence（缓存一致性），361
directory protocols（目录协议），386
synchronization（同步），388
Decimal operands, formats（十进制操作数，格式），A-14
Decision support system（DSS），shared-memory workloads(决策支持系统，共享工作负载），368~369，**369**，369~370
DEC PDP-11, address space（DEC PDP-11，地址空间），B-57~B-58
Defect tolerance, chip fabrication cost case study（缺陷容忍，芯片制造成本案例研究），61~62
Delayed branch（延迟分支）
basic scheme（基本方案），**C-23**
stalls（停顿），**C-65**
Dell Poweredge servers, prices（Dell Poweredge服务器，价格），**53**
Dell servers（Dell服务器）
economies of scale（规模经常），456
real-world considerations（现实考虑事项），52~55
WSC services（WSC服务），441
Dependences（相关），152~153，315~316
Descriptor privilege level（DPL），segmented virtual memory(描

述符权限级别，分段虚拟存储器），B-53
Descriptor table, IA-32（描述符表，IA-32），B-52
Desktop computers（台式计算机）
characteristics（特性），6
compiler structure（编译器结构），A-24
as computer class（作为计算机类别），**5**
memory hierarchy basics（存储器层次结构基础），78
multiprocessor importance（多处理器重要性），344
performance benchmarks（性能基准测试），38-40
processor comparison（处理器比较），**242**
RAID history（RAID历史），C-80
Destination offset, IA-32 segment（目标偏移量，IA-32段），**B-53**
Dies（晶片）
integrated circuits（集成电路），28~30，29
Nehalem floorplan（Nehalem平面图），**30**
wafer example（晶圆示例），**31**，31~32
Die yield（晶片正品率），basic equation（基本公式），30~31
Digital Alpha
branches（分支），**A-18**
Digital Alpha 21264
cache hierarchy（缓存层次结构），**368**
floorplan（平面图），**143**
Digital Alpha processors（Digital Alpha处理器）
control flow instruction branches（控制流指令分支），A-18
displacement addressing mode（位移量寻址方式），**A-12**
exception stopping/restarting（异常停止/重新启动），C-47
immediate value distribution（立即值分布），**A-13**
MIPS precise exceptions（MIPS精确异常），C-59
shared-memory workload（共享存储器工作负载），367~369
DIMMs, 见 Dual inline memory modules
Direct-mapped cache（直接映射缓存）
address parts（地址部分），**B-9**
address translation（地址转换），B-38
block placement（块布置），B-7
memory hierarchy basics（存储器层次结构基础），74
memory hierarchy（存储器层次结构），B-48
optimization（优化），79~80
Directory-based cache coherence（目录式缓存一致性），354
Dirty bit（脏位（重写位）），B-11
Dirty block（脏块），B-11
Disk technology（磁盘技术）
failure rate calculation（故障率计算），48
Google WSC servers（Google WSC服务器），469
performance trends（性能趋势），19~20，20
WSC Flash memory（WSC闪存），474~475
Dispatch stage（分派级）
instruction steps（指令步骤），174
microarchitectural techniques case study（微体系结构技术案例研究），247~254
Displacement addressing mode（位移量寻址方式）
basic considerations（基本考虑事项），A-10
MIPS，12
MIPS data transfers（MIPS数据传输），A-34
MIPS instruction format（MIPS指令格式），A-35
value distributions（值分布），**A-12**
Distributed shared memory（DSM）（分布式共享存储器）
basic considerations（基本考虑事项），378~380
basic structure（基本结构），347~348，**348**
directory-based cache coherence（目录式缓存一致性），354，**380**，418~420
multichip multicore multiprocessor（多芯片多核多处理器），**419**
snooping coherence protocols（监听一致性协议），355
DLP, 见 Data-level parallelism
Double data rate（DDR）（双数据率）
ARM Cortex-A8，117
DRAM performance（DRAM性能），100
DRAMs and DIMMS（DRAM和DIMMS），**101**
Google WSC servers（Google WSC服务器），468~469
Intel Core i7，121
SDRAMs，101
Double data rate 2（DDR2），SDRAM timing diagram（双数据率2，SDRAM时序图），**139**
Double data rate 3（DDR3）（双数据率3）
DRAM internal organization（DRAM内部组织方式），**98**
GDRAM，102
Intel Core i7，118
SDRAM power consumption（SDRAM功耗），102，**103**
Double data rate 4（DDR4），DRAM（双数据率4，DRAM），99
Double data rate 5（DDR5），GDRAM（双数据率5，GDRAM），102
Double-precision floating point（双精度浮点）
add-divide（加除），**C-68**
AVX for x86（x86的AVX），**284**
data access benchmarks（数据访问基准测试），**A-15**
Fermi GPU architecture（Fermi GPU体系结构），306
floating-point pipeline（浮点流水线），C-65
GTX 280，325，328~330
IBM 360，171

MIPS，285，A-38~A-39
MIPS data transfers（MIPS 数据传输），A-34
MIPS registers（MIPS 寄存器），12，A-34
Multimedia SIMD vs. GPUs（多媒体 SIMD 对 GPU），312
operand sizes/types（操作数大小/类型），12
as operand type（作为操作数类型），A-13~A-14
operand usage（操作数使用），297
pipeline timing（流水线时序），**C-54**
Roofline model（屋顶轮廓线模型），**287**，**326**
SIMD Extensions（SMID 扩展），283
VMIPS，**266**，266~267
Double words（双字）
aligned/misaligned addresses（对齐/未对齐地址），**A-8**
data access benchmarks（数据访问基准测试），**A-15**
memory address interpretation（存储器地址解释），A-7~A-8
MIPS data types（MIPS 数据类型），A-34
operand types/sizes（操作数类型/大小），12，A-14
stride（步幅），278
DRAM, 见 Dynamic random-access memory
Driver domains, Xen VM（驱动程序域，Xen VM），111
DSM, 见 Distributed shared memory
DSS, 见 Decision support system
Dual inline memory modules（DIMMs）（双列直插存储器模块）
clock rates, bandwidth, names(时钟频率，带宽，名称)，**101**
DRAM basics（DRAM 基础），99
Google WSC server（Google WSC 服务器），**467**
Google WSC servers（Google WSC 服务器），468~469
graphics memory（图形存储器），322~323
Intel Core i7，118，121
SDRAMs，101
WSC memory（WSC 存储器），473~474
Dual SIMD Thread Scheduler, example（双 SIMD 线程调度程序，示例），305~306
DVFS, 见 Dynamic voltage-frequency scaling
Dynamically scheduled pipelines（动态调度流水线），C-56
Dynamically shared libraries, control flow instruction addressing modes(动态共享库，控制流指令寻址方式)，A-18
Dynamic energy, definition（动态能量，定义），23
Dynamic power（动态功率）
energy efficiency（能量效率），211
microprocessors（微处理器），23
static power（静态功率），26
Dynamic random-access memory（DRAM）（动态随机访问存储器 DRAM）
bandwidth issues（带宽问题），322~323
characteristics（特性），98~100
clock rates, bandwidth, names(时钟频率，带宽，名称)，**101**
cost vs. access time（成本对访问时间），**D-3**
cost trends（成本趋势），27
CUDA，290
dependability（可靠性），104
Flash memory（闪存），103~104
Google WSC servers（Google WSC 服务器），468~469
GPU SIMD instructions（GPU SIMD 指令），296
improvement over time（随时间的改进），17
integrated circuit costs（集成电路成本），28
Intel Core i7，121
internal organization（内部组织），**98**
memory hierarchy design（存储器层次结构设计），73，**73**
memory performance（存储器性能），100~102
multibanked caches（多组缓存），86
NVIDIA GPU Memory structures（NVIDIA GPU 存储器结构），305
performance milestones（性能里程碑），**20**
power consumption（功耗），**63**
real-world server considerations（现实服务器考虑事项），52~55
Roofline model（屋顶轮廓线模型），286
server energy savings（服务器节能），**25**
speed trends（速度趋势），**99**
technology trends（技术趋势），17
WSC efficiency measurement（WSC 效率测量），450
WSC memory costs（WSC 存储器成本），473~474
WSC memory hierarchy（WSC 存储器层次结构），444~445
WSC power modes（WSC 功率模式），472
yield（正品率），32
Dynamic scheduling（动态调度）
ILP，168
MIPS scoreboarding（MIPS 记分卡），C-79
SMT on superscalar processors（超标量处理器上的 SMT），230
unoptimized code（未优化代码），C-81
Dynamic voltage-frequency scaling（DVFS）（动态电压-频率调整）
energy efficiency（能耗效率），25
Google WSC，467
processor performance equation（处理器性能公式），52
Dynamo（Amazon），438，452

E

Early restart, miss penalty reduction（早期重启，降低缺失代价），86
EBS, 见 Elastic Block Storage
EC2, 见 Amazon Elastic Computer Cloud
ECC, 见 Error-Correcting Code
Economies of scale（规模经济），434~456
EEMBC, 见 Electronic Design News Embedded Microprocessor Benchmark Consortium
EEPROM（Electronically Erasable Programmable Read-Only Memory，（电子可擦除可编程只读存储器）
compiler-code size considerations（编译器代码大小考虑事项），**A-44**
Effective address（实际地址），A-9
Eight-way set associativity（八路组相联）
ARM Cortex-A8，114
cache optimization（缓存优化），B-29
conflict misses（冲突缺失），B-23
data cache misses（数据缓存缺失），B-10
Elapsed time, execution time（消逝时间，执行时间），36
Elastic Block Storage（EBS），MapReduce cost calculations (弹性分块存储，MapReduce 成本计算），458~460，**459**
Electronically Erasable Programmable Read-Only Memory, 见 EEPROM（Electronically Erasable Programmable Read-Only Memory）
Electronic Design News Embedded Microprocessor Benchmark Consortium（EEMBC）(电子设计新闻嵌入式处理器基准测试协会）
ISA code size（ISA 代码大小），**A-44**
performance benchmarks（性能基准测试），38
Element group, definition（元素组，定义），**272**
Encoding（编码）
control flow instructions（控制流指令），A-18
erasure encoding（删除编码），439
instruction set（指令集），A-21~A-24，**A-22**
ISAs，14，A-5~A-6
MIPS ISA，A-33
MIPS pipeline（MIPS 流水线），C-36
opcode（操作码），A-13
VLIW model（VLIW 模型），195~196
Energy efficiency, 另见 Power consumption
Energy proportionality, WSC servers（能量比例，WSC 服务器），462
EPIC approach（EPIC 方法）
VLIW processors（VLIW 处理器），**194**，196
ERA, 见 Engineering Research Associates
Erasure encoding, WSCs（消除编码，WSC），439
Error-Correcting Code（ECC）（纠错码）
fault detection pitfalls（错误检测易犯错误），58
Fermi GPU architecture（Fermi GPU 体系结构），307
memory dependability（存储器可靠性），104
WSCs，473~474
Ethernet switches（以太网交换机）
architecture considerations（体系结构考虑因素），16
Dell servers（Dell 服务器），53
Google WSC，464~465，469
historical performance milestones（历史性能里程碑），**20**
EX, 见 Execution address cycle
Exceptions（异常）
ALU instructions（ALU 指令），C-4
architecture-specific examples（体系结构特有示例），**C-44**
categories（类别），**C-46**
control dependence（控制相关性），154~155
hardware-based speculation（基于硬件的推测），190
imprecise（非精确），169~170，188
long latency pipelines（长延迟流水线），C-55
MIPS，**C-48**，C-48~C-49
out-of-order completion（乱序完成），169~170
precise（精确），C-47，C-58~C-60
return address buffer（返回地址缓冲区），207
ROB instructions（ROB 指令），190
speculative execution（推测执行），222
stopping/restarting（停止/重新启动），C-46~C-47
types and requirements（类型与需求），C-43~C-46
Execute step（执行步骤）
instruction steps（指令步骤），174
ROB instruction（ROB 指令），186
Execution address cycle（EX）（执行地址周期）
basic MIPS pipeline（基本 MIPS 流水线），C-36
data hazards requiring stalls（需要停顿的数据冒险），**C-21**
data hazard stall minimization（数据冒险停顿最小化），C-17
exception stopping/restarting（异常住/重新启动），C-46~C-47
hazards and forwarding（冒险与转发），C-56~C-57
MIPS FP operations, basic considerations（MIPS 浮点运算，基本考虑事项），C-51~C-53
MIPS pipeline（MIPS 流水线），**C-52**
MIPS pipeline control（MIPS 流水线控制），C-36~C-39
MIPS R4000，C-63~C-64，**C-64**
MIPS scoreboarding（MIPS 记分卡），C-72，C-74，C-77
out-of-order execution（乱序执行），C-71

pipeline branch issues（流水线分支发射），**C-40**，**C-42**
RISC classic pipeline（RISC经典流水线），C-10
simple MIPS implementation（简单 MIPS 实现），C-31~C-32
simple RISC implementation（简单 RISC实现），C-6
Execution time（执行时间）
Amdahl's law（Amdahl定律），46~47，406
application/OS misses（应用程序/OS缺失），**B-59**
cache performance（缓存性能），B-3~B-4，B-16
calculation（计算），36
commercial workloads（商业工作负载），369~370，**370**
energy efficiency（能量效率），211
integrated circuits（集成电路），22
loop unrolling（循环展开），160
multilevel caches（多级缓存），B-32~B-34
multiprocessor performance（多处理器性能），405~406
multiprogrammed parallel "make" workload（多重编程的并行"make"工作负载），**375**
multithreading（多线程），**232**
performance equations（性能公式），**B-22**
pipelining performance（流水化性能），C-3，C-10~C-11
PMDs，6
principle of locality（局域性原理），45
processor comparisons（处理器对比），**243**
processor performance equation（处理器性能公式），49，51
reduction（缩减），B-19
second-level cache size（第二级缓存大小），**B-34**
SPEC benchmarks（SPEC基准测试），42~44，**43**，56
stall time（停顿时间），B-21
vector mask registers（向量遮罩寄存器），276
vector operations（向量运算），268~271
Expand-down field（向下扩展字段），**B-53**
Explicit operands, ISA classifications（显式操作数，ISA分类），A-3~A-4
Explicit unit stride, GPUs vs. vector architectures（显式单位步幅，GPU对向量体系结构），310
Extended accumulator（扩展累加器）
flawed architectures（有缺陷体系结构），A-44
ISA classification（ISA分类），A-3

F

Facebook，460
Failures, 另见 Mean time between failures（MTBF）; Mean time to failure（MTTF）
False sharing（伪共享），366~367
FarmVille，460
Fault detection, pitfalls（故障检测，易犯错误），57~58
Faulting prefetches, cache optimization（故障预取，缓存优化），92
Feature size（特征尺寸）
dependability（可靠性），33
FEC, 见 Forward error correction
Fermi GPU
architectural innovations（体系结构创新），305-308
future features（未来特征），333
Grid mapping（网格映射），**293**
multithreaded SIMD Processor（多线程 SIMD 处理器），**307**
NVIDIA，291，305
SIMD，296~297
SIMD Thread Scheduler（SIMD 线程调度程序），**306**
Fermi Tesla GTX 280
GPU comparison（GPU比较），324~325，**325**
memory bandwidth（存储器带宽），**328**
raw/relative GPU performance（原始/相对 GPU 性能），328
synchronization（同步），329
weaknesses（弱点），330
Fermi Tesla GTX 480
floorplan（平面图），**295**
GPU comparisons（GPU比较），323~330，**325**
Fetch-and-increment（提取并递增）
synchronization（同步），388
Fetching, 见 Data fetching
Field-programmable gate arrays（FPGAs），WSC array switch），现场可编程阵列，WSC阵列交换机），433
Fine-grained multithreading（细粒度多线程），224~226
First-level caches, 另见 L1 caches
technology trends（技术趋势），18
First-reference misses, definition（首次引用缺失，定义），B-23
FIT rates, WSC memory（FIT率，WSC存储器），473~474
Fixed-field decoding, simple RISC implementation（固定字段解码，简单 RISC 实现），C-6
Fixed-length encoding（定长编码）
general-purpose registers（通用寄存器），**A-6**
instruction sets（指令集），**A-22**
ISAs，14
Fixed-length vector（定长向量），264~284
Flags（标志）
performance benchmarks（性能基准测试），37
performance reporting（性能报告），41
scoreboarding（记分卡），C-75
Flash memory（闪存）
characteristics（特性），102~104
dependability（可靠性），104
memory hierarchy design（存储器层次结构设计），**72**

technology trends（技术趋势），18
WSC cost-performance（WSC成本-性能），474~475
Flexible chaining（灵活链接）
vector execution time（向量执行时间），269
Floating-point（FP）operations（浮点运算）
arithmetic intensity（算术密度），285~288，**286**
branch condition evaluation（分支条件判断），A-19
branches（分支），**A-20**
cache misses（缓存缺失），83~84
control flow instructions（控制流指令），A-21
CPI calculations（CPI计算），50~51
data access benchmarks（数据访问基准测试），**A-15**
data dependences（数据相关性），151
data hazards（数据冒险），169
dynamic scheduling with Tomasulo's algorithm（用Tomasulo算法进行动态调度），171~172，**173**
exception stopping/restarting（异常停止/重新启动），C-47
ILP exploitation（ILP开发），197~199
ILP exposure（揭示ILP），157~158
ILP in perfect processor（完美处理器中的ILP），**215**
ILP for realizable processors（可实现处理器的ILP），216~218
independent（独立），**C-54**
instruction operator categories（指令运算符类别），**A-15**
Intel Core i7，**240**，241
ISA performance and efficiency prediction（ISA性能与效率预测），241
latencies（延迟），**157**
MIPS，A-38~A-39
Tomasulo's algorithm（Tomasulo算法），**173**
MIPS exceptions（MIPS异常），C-49
MIPS operations（MIPS操作），A-35
MIPS pipeline（MIPS流水线），**C-52**
MIPS precise exceptions（MIPS精确异常），C-58~C-60
MIPS R4000，C-65~C-67，**C-66~C-67**
MIPS scoreboarding（MIPS记分卡），C-77
MIPS with scoreboard（具有记分卡的MIPS），C-73
misspeculation instructions（推测错误指令），**212**
Multimedia SIMD Extensions（多媒体SIMD扩展），285
multiple lane vector unit（多车道向量单元），**273**
multiple outstanding（多条未完成指令），**C-54**
operand sizes/types（操作数大小/类型），12
parallelism *vs.* window size（并行度对窗口大小），**217**
pipeline hazards and forwarding（流水线冒险与转发），C-55~C-57
pipeline structural hazards（流水线结构冒险），C-16
ROB commit（ROB提交），187
SMT，398~400
SPEC benchmarks（SPEC基准测试），**39**
stalls from RAW hazards（RAW冒险导致的停顿），**C-55**
static branch prediction（静态分支预测），C-26~C-27
Tomasulo's algorithm（Tomasulo算法），**185**
VAX，**B-73**
vector sequence chimes（向量序列钟鸣），270
VLIW processors（VLIW处理器），**195**
VMIPS，264
Floating-point registers（FPRs）（浮点寄存器）
MIPS data transfers（MIPS数据传输），A-34
MIPS operations（MIPS操作），A-36
MIPS64 architecture（MIPS64体系结构），A-34
write-back（写回），**C-56**
Floating-point square root（FPSQR）（浮点平方根）
calculation（计算），47~48
CPI calculations（CPI计算），50~51
Flush, branch penalty reduction（冲刷，降低分支代价），C-22
FORTRAN
compiler types and classes（编译器类型与分类），A-28
loop-level parallelism dependences（循环级并行度相关），320~321
MIPS scoreboarding（MIPS记分卡），C-77
return address predictors（返回地址预测器），206
Forwarding，另见 Bypassing
Four-way conflict misses, definition（四路冲突缺失，定义），B-23
FP，见 Floating-point operations
FPRs，见 Floating-point registers
FPSQR，见 Floating-point square root
Freeze, branch penalty reduction（冻结，降低分支代价），C-22
FU，见 Functional unit
Fully associative cache（全相联缓存）
block placement（块放置），B-7
conflict misses（冲突缺失），B-23
direct-mapped cache（直接映射缓存），**B-9**
memory hierarchy basics（存储器层次结构基础），74
Functional hazards（功能冒险）
ARM Cortex-A8，233
microarchitectural techniques case study（微体系结构技术案例研究），247~254
Functional unit（FU）（功能单元）
FP operations（浮点运算），**C-66**
Function calls（功能调用）
GPU programming（GPU编程），289
NVIDIA GPU Memory structures（NVIDIA GPU存储器

结构），304~305
PTX assembler（PTX汇编程序），301
Function pointers, control flow instruction addressing modes（功能指针，控制流指令寻址方式），A-18
Future file, precise exceptions（未来文件，精确异常），C-59

G

Gather-Scatter（集中/分散），309
GCD，见 Greatest common divisor test
GDDR，见 Graphics double data rate（GDDR）
GDRAM，见 Graphics dynamic random-access memory
General-purpose registers（GPRs）（通用寄存器）
advantages/disadvantages（优点/缺点），**A-6**
ISA classification（ISA分类），A-3~A-5
MIPS data transfers（MIPS数据传输），A-34
MIPS operations（MIPS操作），A-36
MIPS64，A-34
VMIPS，265
Geometric means, example calculations（几何均值，示例计算），43~44
GFS, 见 Google File System
Giga Thread Engine, definition（Giga线程引擎，定义），**292，314**
Global address space, segmented virtual memory（全局地址空间，分段虚拟存储器），B-52
Global common subexpression elimination, compiler structure (全局公共子表达式消去法，编译器结构），A-26
Global data area, and compiler technology（全局数据区域，与编译器技术），A-27
Global load/store, definition（全局载入/存储，定义），**309**
Global Memory（全局存储器），292，314
Global miss rate（全局缺失率），B-31
Global optimizations（全局优化）
compilers（编译器），A-26，A-29
optimization types（优化类型），**A-28**
Global predictors（全局预测器）
Intel Core i7，166
tournament predictors（竞赛预测器），164~166
Global scheduling, ILP, VLIW processor（全局调度，ILP，VLIW处理器），194
Google
Bigtable，438，441
cloud computing（云计算），455
MapReduce，**437**，458~459，**459**
server CPUs（服务器CPU），**440**
server power-performance benchmarks（服务器功率-性能基准测试），439~441
WSCs，432，449
containers（集装箱），464~465，**465**
cooling and power（制冷与功率），465~468
monitoring and repairing（监控与修复），469~470
PUE，**468**
servers（服务器），**467**，468~469
Google Clusters
memory dependability（存储器可靠性），104
Google File System（GFS）（Google文件系统）
MapReduce，438
WSC storage（WSC存储），442~443
Google Goggles
PMDs，6
user experience（用户体验），4
Google search（Google搜索）
shared-memory workloads（共享存储器工作负载），369
workload demands（工作负载需求），439
GPRs，见 General-purpose registers
GPU（Graphics Processing Unit）banked and graphics memory，（图形处理单元）分组与图形存储器），322~323
GPU Memory（GPU存储器），292，309，314
GPU programming（GPU编程），288
Graph coloring，register allocation（图形着色，寄存器分派），A-26~A-27
Graphics double data rate（GDDR）（图形双数据率）
characteristics（特性），102
Fermi GTX 480 GPU，294，324
Graphics dynamic random-access memory（GDRAM）（图形动态随机访问存储器）
bandwidth issues（带宽问题），322~323
characteristics（特性），102
Graphics-intensive benchmarks, desktop performance（图形密集基准测试，桌面性能），38
Graphics Processing Unit, 见 GPU
Graphics synchronous dynamic random-access memory（GSDRAM），characteristics（图形同步动态随机访问存储器，特性），102
Greatest common divisor（GCD）test, loop-level parallelism dependences（最大公约数测试，循环级并行度相关性），319
Grid（网格），292，309，313
Guest definition（来宾定义），108
Guest domains, Xen VM（来宾域，Xen VM），111

H

Hadoop, WSC batch processing（Hadoop，WSC批处理），437

Half words（半字）
aligned/misaligned addresses（对齐/非对齐地址），**A-8**
memory address interpretation（存储器地址解释），A-7~A-8
MIPS data types（MIPS 数据类型），A-34
operand sizes/types（操作数大小/类型），12
operand type（数据类型），A13~A14
Hard drive, power consumption（硬盘，功耗），**63**
Hardware（硬件）
architecture component（体系结构分量），15
cache optimization（缓存优化），**96**
energy/performance fallacies（能量/性能谬论），56
ILP approaches（ILP 方法），148，214~215
pipeline hazard detection（流水线冒险检测），**C-38**
Virtual Machines protection（虚拟机保护），108
WSC cost-performance（WSC 成本-性能），474
WSC running service（WSC 运行服务），434~435
Hardware-based speculation（基于硬件的推测）
basic algorithm（基本算法），**191**
data flow execution（数据流执行），184
FP unit using Tomasulo's algorithm（使用 Tomasulo 算法的浮点单元），185
key ideas（关键思想），183~184
Hardware prefetching（硬件预取）
cache optimization（缓存优化），131~133
miss penalty/rate reduction（降低缺失代价/缺失率），91~92
NVIDIA GPU Memory structures（NVIDIA GPU 存储器结构），305
SPEC benchmarks（SPEC 基准测试），**92**
Hardware primitivies（硬件原语）
basic types（基本类型），387~389
synchronization mechanisms（同步机制），387~389
Header（标头）
Heap, and compiler technology（堆和编译器技术），A-27~A-28
Heterogeneous architecture, definition（异构体系结构，定义），262
Hewlett-Packard ProLiant SL2x170z G6, SPECPower benchmarks(Hewlett-Packard ProLiant SL2x170z G6, SPECPower 基准测试），**463**
High-level optimizations, compilers（高级优化，编译器），A-26
Highly parallel memory systems, case studies（高并行存储器系统，案例研究），133~136
High-order functions, control flow instruction addressing modes(高阶函数，控制流指令寻址方式），A-18
High-performance computing（HPC）（高性能计算）
write strategy（写入策略），B-10
WSCs，432，435~436
History file, precise exceptions（历史文件，精确异常），C-59
Hit time（命中时间）
average memory access time（存储器平均访问时间），B-16~B-17
first-level caches（第一级缓存），79~80
memory hierarchy basics(存储器层次结构基础），77~78
reduction（缩减），78，B-36~B-40
way prediction（路预测），81~82
Home node, directory-based cache coherence protocol basics(本地节点，目录式缓存一致性协定基础），382
Host definition（主机定义），108，305
HPC, 见 High-performance computing
HP-Compaq servers（HP-Compaq 服务器）
price-performance differences（性价比差别），441
SMT，230
Hypertransport, AMD Opteron cache coherence（Hypertransport，AMD Opteron 缓存一致性），361
Hypervisor, characteristics（Hypervisor，特性），108

I

IBM
Chipkill，104
IBM 360
address space（地址空间），B-58
instruction operator categories(指令操作符类别），**A-15**
instruction set complications(指令集复杂性），C-49~C-50
protection and ISA（保护与 ISA），112
IBM 360/91
dynamic scheduling with Tomasulo's algorithm（用 Tomasulo 算法进行动态调度），170~171
IBM 370
protection and ISA（保护与 ISA），112
Virtual Machines（虚拟机），110
IBM CodePack, RISC code size（IBM CodePack, RISC 代码大小），A-23
IBM eServer p5 processor（IBM eServer p5 处理器）
performance/cost benchmarks（性能/成本基准测试），**409**
SMT and ST performance（SMT 和 ST 性能），**399**
speedup benchmarks(加速比基准测试），**408**，408~409
IBM J9 JVM
real-world server considerations（现实服务器考虑事项），52~55
WSC performance（WSC 性能），**463**
IBM PCs, architecture flaws vs. success（IBM PC，体系结构缺陷对成功），A-45

IBM Power processors（IBM Power处理器）
branch-prediction buffers（分支预测缓冲区），**C-29**
characteristics（特性），**247**
exception stopping/restarting（异常停止/重新启动），C-47
MIPS precise exceptions（MIPS精度异常），C-59
shared-memory multiprogramming workload（共享存储器多重编程工作负载），378
IBM Power 4
peak performance（峰值性能），**58**
IBM Power 5
manufacturing cost（制造成本），**62**
multiprocessing/ multithreading-based performance（基于多重处理/多线程的性能），398~400
IBM Power 7
Google WSC，436
ideal processors（理想处理器），214~215
multicore processor performance（多核处理器性能），400~401
multithreading（多线程），**225**
IBM servers, economies of scale（IBM服务器，规模经济），456
IC,见 Instruction count
I-caches（I-缓存）
case study examples（案例研究示例），B-63
way prediction（路预测），81~82
ID, 见 Instruction decode
Ideal pipeline cycles per instruction, ILP concepts（理想的每指令流水线周期，ILP概念），149
Ideal processors, ILP hardware model（理想处理器，ILP硬件模型），214~215，219~220
IF, 见 Instruction fetch cycle
IF statement handling（IF语句处理）
control dependences（控制相关性），154
GPU conditional branching（GPU条件分支），300，302~303
memory consistency（存储器一致性），392
vectorization in code（代码中的向量化），271
vector-mask registers（向量遮罩寄存器），267，275~276
ILP,见 Instruction-level parallelism
Immediate addressing mode（立即数寻址方式）
ALU operations（ALU运算），**A-12**
basic considerations（基本考虑事项），A-10~A-11
MIPS，12
MIPS instruction format（MIPS指令格式），A-35
MIPS operations（MIPS操作），A-37
value distribution（值分布），**A-13**
Implicit operands, ISA classifications（隐式操作数，ISA分类），A-3
Implicit unit stride，GPUs vs. vector architectures(隐式单位步幅，GPU对向量体系结构），310
Imprecise exceptions（非精确异常），167~170
Inactive power modes, WSCs（非活跃功率模式，WSC），472
Inclusion（包含）
cache hierarchy（缓存层次结构），397~398
implementation（实现），397~398
invalidate protocols（失效协议），357
Indexed addressing（变址寻址）
Indexes（索引）
address translation during（地址变换期间），B-36~B-40
AMD Opteron data cache（AMD Opteron数据缓存），B-13~B-14
ARM Cortex-A8，**115**
size equations（大小公式），B-22
Index field, block identification（索引字段，块识别），B-8
Index vector, gather/scatter operations（索引向量，集中/分散操作），279~280
In flight instructions, ILP hardware model（使用中的指令，ILP硬件模型），214
Information tables, examples（信息表，示例），176~177
Infrastructure costs（基础设施成本）
WSC，446~450，452~455，**453**
WSC efficiency（WSC效率），450~452
Initiation interval, MIPS pipeline FP operations（启动间隔，MIPS流水线浮点运算），C-52~C-53
Initiation rate（启动速率）
floating-point pipeline（浮点流水线），C-65~C-66
memory banks（存储器组），276~277
vector execution time（向量执行时间），269
In-order commit（循序提交）
hardware-based speculation(基于硬件的推测)，188~189
In-order execution（循序执行）
average memory access time（存储器平均访问时间），B-17~B-18
cache behavior calculations（缓存行为计算），B-18
cache miss（缓存缺失），B-2~B-3
dynamic scheduling（动态调度），168~169
IBM Power processors（IBM Power处理器），**247**
ILP exploitation（ILP开发），193~194
multiple-issue processors（多发射处理器），**194**
superscalar processors（超标量处理器），196
In-order floating-point pipeline, dynamic scheduling(循序浮点流水线，动态调度），169
In-order issue（循序发射）
ARM Cortex-A8，233
dynamic scheduling（动态调度），168~170，C-71
In-order scalar processors，VMIPS（循序标量处理器，

VMIPS），267
Instruction cache（指令缓存）
AMD Opteron example（AMD Opteron 示例），**B-15**
antialiasing（别名消失），B-38
application/OS misses(应用程序/操作系统缺失)，**B-59**
branch prediction（分支预测），C-28
commercial workload（商业工作负载），373
GPU Memory（GPU存储器），306
instruction fetch（指令提取），202~203，237
ISA，241
MIPS R4000 pipeline（MIPS R4000流水线），C-63
miss rates（缺失率），161
multiprogramming workload（多重编程工作负载），374~375
prefetch（预取），236
RISCs，A-23
Instruction commit（指令提交）
hardware-based speculation(基于硬件的推测)，184~185，187~188，**188**，190
instruction set complications（指令集复杂性），C-49
Intel Core i7，237
speculation support（推测支持），208~209
Instruction count（IC）（指令计数器）
addressing modes（寻址方式），A-10
cache performance（缓存性能），B-4，B-16
compiler optimization(编译器优化)，**A-29**，A-29~A-30
processor performance time（处理器性能时间），49~51
Instruction decode（ID）（指令译码（ID））
basic MIPS pipeline（基本MIPS流水线），C-36
branch hazards（分支冒险），C-21
data hazards（数据冒险），169
hazards and forwarding（冒险与转发），C-55~C-57
MIPS pipeline（MIPS流水线），C-71
MIPS pipeline control(MIPS流水线控制)，C-36~C-39
MIPS pipeline FP operations（MIPS浮点运算），C-53
MIPS scoreboarding（MIS记分卡），C-72~C-74
out-of-order execution（乱序执行），170
pipeline branch issues(流水线分支发射)，C-39~C-41，**C-42**
RISC classic pipeline（RISC经典流水线），C-7~C-8，C-10
simple MIPS implementation（简单MIPS实现），C-31
simple RISC implementation(简单RISC实现)，C-5~C-6
Instruction fetch（IF）cycle（指令提取周期）
basic MIPS pipeline（基本MIPS流水线），C-35~C-36
branch hazards（分支冒险），C-21
branch-prediction buffers（分支预测缓冲区），C-28
exception stopping/restarting(异常停顿/重启)，C-46~C-47
MIPS exceptions（MIPS异常），C-48
MIPS R4000（MIPS R4000），C-63
pipeline branch issues（流水线分支发射），**C-42**
RISC classic pipeline（RISC经典流水线），C-7，C-10
simple MIPS implementation（简单MIPS实现），C-31
simple RISC implementation（简单RISC实现），C-5
Instruction fetch units（指令提取单元）
integrated（集成），207~208
Intel Core i7，237
Instruction issue（指令发射），C-36
Instruction-level parallelism（ILP）（指令级并行），9，149~150
Instruction path length, processor performance time（指令路径长度，处理器性能时间），49
Instruction prefetch（指令预取）
integrated instruction fetch units（集成指令提取单元），208
miss penalty/rate reduction（降低缺失代价/缺失率），91~92
SPEC benchmarks（SPEC基准测试），**92**
Instruction register（IR）（指令寄存器（IR））
basic MIPS pipeline（基本MIPS流水线），C-35
dynamic scheduling（动态调度），170
MIPS implementation（MIPS实现），C-31
Instruction set architecture（ISA），另见 Intel 80x86 processors; Reduced Instruction Set Computer（RISC）
Instructions per clock（IPC）（每时钟周期的指令数）
ARM Cortex-A8，236
flawless architecture design（无缺陷体系结构设计），A-45
ICP for realizable processors（用于可实现处理器），216~218
MIPS scoreboarding（MIPS记分卡），C-72
multiprocessing/multithreading-based performance（基于多重处理/多线程的性能），398~400
processor performance time（处理器性能时间），49
Sun T1 multithreading unicore performance（Sun T1多线程单核性能），**229**
Sun T1 processor（Sun T1处理器），**399**
Instruction status（指令状态）
dynamic scheduling（动态调度），177
MIPS scoreboarding（MIPS记分卡），C-75
Integer operand（整数操作数）
flawed architecture（有缺陷体系结构），A-44
GCD，319
graph coloring（图形着色），A-27
instruction set encoding（指令集编码），A-23
MIPS data types（MIPS数据类型），A-34
as operand type（作为操作数类型），12，A-13~A-14
Integer operations（整数运算）

addressing modes（寻址方式），A-11
ALUs，**A-12**，C-54
ARM Cortex-A8，**116**，**232**，**235**，236
benchmarks（基准测试），**167**，C-69
branches（分支），A-18~A-20，**A-20**
cache misses（缓存缺失），83~84
data access distribution（数据访问分布），**A-15**
data dependences（数据相关性），151
dependences（相关性），322
desktop benchmarks（桌面基准测试），38~39
displacement values（位移值），**A-12**
exceptions（异常），C-43，C-45
hardware ILP model（硬件 ILP 模型），**215**
hardware vs. software speculation（硬件推测对软件推测），221
hazards（冒险），C-57
ILP，197~200
instruction set operations（指令集运算），A-16
Intel Core i7，238，**240**
ISA，242，A-2
longer latency pipelines（长延迟流水线），C-55
MIPS，C-31~C-32，C-36，C-49，C-51~C-53
MIPS64 ISA，14
MIPS FP pipeline（MIPS 浮点流水线），C-60
MIPS R4000 pipeline（MIPS R4000 流水线），C-61，C-63，C-70
misspeculation（错误预测），**212**
MVL，274
pipeline scheduling（流水线调度），157
precise exceptions（精确异常），C-47，C-58，C-60
processor clock rate（处理器时钟频率），244
R4000 pipeline（R4000 流水线），**C-63**
realizable processor ILP（可实现处理器 ILP），216~218
RISC，C-5，C-11
scoreboarding（记分卡），C-72~C-73，**C-76**
SIMD processor（SIMD 处理器），307
SPEC benchmarks（SPEC 基准测试），**39**
speculation through multiple branches（多分支推测），211
static branch prediction（静态分支预测），C-26~C-27
T1 multithreading unicore performance（T1 多线程单核性能），227~229
Tomasulo's algorithm（Tomasulo 算法），181
tournament predictors（竞赛预测器），164
VMIPS，265
Integer registers（整数寄存器）
hardware-based speculation（基于硬件的推测），192
MIPS dynamic instructions（MIPS 动态指令），A-41~A-42
MIPS floating-point operations（MIPS 浮点运算），A-39
MIPS64 architecture（MIPS 体系结构），A-34
VLIW，194
Integrated circuit basics（集成电路基础）
cost trends（成本趋势），28~32
dependability（可靠性），33~36
logic technology（逻辑技术），17
microprocessor developments（微处理器发展），2
power and energy（功率与能量），21~23
scaling（缩放），19~21
Intel Atom，230
Intel Atom processors（Intel Atom 处理器）
ISA performance and efficiency prediction（ISA 性能和效率预测），241~243
performance measurement（性能测量），405~406
SMT，231
WSC memory（WSC 存储器），474
WSC processor cost-performance（WSC 处理器成本-性能），473
Intel Core i7
Alpha processors（Alpha 处理器），**368**
architecture（体系结构），15
basic function（基本功能），236~238
"big and dumb" processors（"大而被动"处理器），245
branch predictor（分支预测器），166~167
clock rate（时钟速率），**244**
dynamic scheduling（动态调度），170
GPU comparisons（GPU 对比），324~330，**325**
hardware prefetching（硬件预取），91
ISA performance and efficiency prediction（ISA 性能与效率预测），241~243
L2/L3 miss rates（L2/L3 缺失率），**125**
memory hierarchy basics（存储器层次结构基础），78，117~124，**119**
memory hierarchy design（存储器层次结构设计），73
memory performance（存储器性能），122~124
MESIF protocol（MESIF 协议），362
microprocessor die example（处理器晶片示例），**29**
miss rate benchmarks（缺失率基准测试），**123**
multibanked caches（多组缓存），86
multithreading（多线程），**225**
nonblocking cache（无阻塞缓存），83
performance（性能），**239**，239~241，**240**
performance/energy efficiency（性能/能量效率），401~405
pipelined cache access（流水化缓存访问），82
pipeline structure（流水线结构），**237**
processor comparison（处理器对比），**242**
raw/relative GPU performance（原始/相对 GPU 性能），**328**

Roofline model（屋顶轮廓线模型），286~288，**287**
Inter corei7(continued）
single-threaded benchmarks（单线程基准测试），**243**
SMP limitations（SMP 局限性），363
SMT，230~231
snooping cache coherence implementation（监听缓存一致性实现），365
three-level cache hierarchy（三级缓存层次结构），**118**
TLB structure（TLB 结构），**118**
write invalid protocol（写失效协议），356
Intel 80x86 processors（Intel 80x86 处理器）
address space（地址空间），B-58
architecture flaws vs. success（体系结构缺陷与不足），A-44~A-45
Atom，231
cache performance（缓存性能），B-6
common exceptions（常见异常），**C-44**
instruction set encoding（指令集编码），A-23
Intel Core i7，117
ISA，11~12，14~15，A-2
memory accesses（存储器访问），B-6
memory addressing（存储器寻址），A-8
process protection（进程保护），B-50
RISC，2，A-3
top instructions（使用较多的指令），**A-16**
variable encoding（变量编码），A-22~A-23
virtualization issues（虚拟化问题），**128**
Virtual Machines ISA support（虚拟机 ISA 支持），109
Virtual Machines and virtual memory and I/O（虚拟机与存储器、I/O），110
Intel IA-32 architecture（Intel IA-32 体系结构）
call gate（调用门），B-54
descriptor table（描述符表），B-52
instruction set complications（指令集复杂性），C-49~C-51
segment descriptors（段描述符），**B-53**
segmented virtual memory（分段虚拟存储器），B-51~B-54
Intel IA-64 architecture（Intel IA-64 体系结构）
multiple issue processor approaches（多发射处理器方法），**194**
Intel Itanium 2
"big and dumb" processors（"大而被动"处理器），245
clock rate（时钟频率），**244**
peak performance（峰值特性），**58**
SPEC benchmarks（SPEC 基准测试），**43**
Intel MMX, multimedia instruction compiler support（Intel MMX，多媒体指令编译器支持），A-31~A-32
Intel Nehalem，411
Intel Pentium 4
Intel Pentium III
pipelined cache access（流水化缓存访问），82
Intel Pentium Pro，82
Intel Pentium processors（Intel Pentium 处理器）
"big and dumb" processors（"大而被动"处理器），245
clock rate（时钟频率），**244**
Opteron memory protection(Opteron 存储器保护），B-57
pipelining performance（流水线性能），C-10
segmented virtual memory example（分段虚拟存储器示例），B-51~B-54
SMT，230
Intel processors（Intel 处理器）
early RISC designs（早期 RISC 设计），2
Intel Streaming SIMD Extension（SSE）（Intel 流式 SIMD 扩展）
basic function（基本功能），283
Multimedia SIMD Extensions（多媒体 SIMD 扩展），A-31
vector architectures（向量体系结构），282
Interactive workloads, WSC goals/ requirements（交互式工作负载，WSC 目标/需求），433
Interconnection networks（互连网络）
multicore single-chip multiprocessor（多核单芯片多处理器），**364**
Internal fragmentation, virtual memory page size selection(内部碎片，虚拟存储器页大小选择），B-47
Internal Mask Registers, definition（内部遮罩寄存器，定义），**309**
Interrupt，见 Exceptions
Invalidate protocol（使协议失效）
directory-based cache coherence protocol example（目录式缓存一致性协议示例），385~383
example（示例），359，**360**
implementation（实现），356~357
snooping coherence（监听一致性），**355**，355~356
Inverted page table, virtual memory block identification（反转分页表，虚拟存储器块识别），B-44~B-45
I/O bound workload, Virtual Machines protection（I/O 限工作负载，虚拟机保护），108
I/O cache coherency, basic considerations（I/O 缓存一致性，基本考虑事项），113
I/O devices（I/O 设备）
address translation（地址转换），B-38
average memory access time（存储器平均访问时间），B-17
cache coherence enforcement（缓存一致性实施），354
centralized shared-memory multiprocessors（集中共享存储器多处理器），351
future GPU features（未来 GPU 特征），332

inclusion（包含），B-34
Multimedia SIMD vs. GPUs（多媒体 SIMD 对 GPU），312
multiprocessor cost effectiveness（多处理器成本效率），407
Virtual Machines impact（虚拟机影响），110~111
write strategy（写策略），B-11
Xen VM，111
I/O latency, shared-memory workloads（I/O 延迟，共享存储器工作负载），368~369，371
I/O registers, write buffer merging（I/O 寄存器，写缓冲区合并），87
Issue logic（发射逻辑）
ARM Cortex-A8（ARM Cortex-A8），233
ILP，197
longer latency pipelines（长延迟流水线），C-57
multiple issue processor（多发射处理器），**198**
register renaming vs. ROB（寄存器重命名对 ROB），210
speculation support（推测支持），210
Issue stage（发射级）
ID pipe stage（ID 流水级），170
instruction steps（指令步骤），174
MIPS with scoreboard（具有记分卡的 MIPS），C-73~C-74
out-of-order execution（乱序执行），C-71
ROB instruction（ROB 指令），186

J

Java benchmarks（Java 基准测试）
Intel Core i7，401~405
SMT on superscalar processors（超标量处理器上的 SMT），230~232
without SMT（没有 SMT），**403~404**
Java language（Java 语言）
hardware impact on software development（硬件对软件开发的影响），4
return address predictors（返回地址预测器），206
SMT，230~232，402~405
SPECjbb，40
SPECpower，52
virtual functions/methods（虚拟函数/方法），A-18
Java Virtual Machine（JVM）（Java 虚拟机）
IBM，463
multicore processor performance（微处理器性能），400
multithreading-based speedup（基于多线程的加速比），**232**
SPECjbb，53
JBOD, 见 RAID 0
Jump prediction（Jump 预测）
hardware model（硬件模型），214
ideal processor（理想处理器），214
Jumps（跳转）
control flow instructions（控制流指令），14，A-16，**A-17**，A-21
GPU conditional branching（GPU 条件分支），301~302
MIPS control flow instructions（MIPS 控制流指令），A-37~A-38
MIPS operations（MIPS 操作），A-35
return address predictors（返回地址预测器），206
RISC instruction set（RISC 指令集），C-5
JVM, 见 Java Virtual Machine

K

Kernels（内核）
arithmetic intensity（算术密度），**286**，286~287，327
benchmarks（基准测试），56
bytes per reference, vs. block size（每次引用的字节数，与块大小），**378**
caches（缓存），329
commercial workload（商业工作负载），369~370
compilers（编译器），A-24
compute bandwidth（计算带宽），328
via computing（通过计算），327
EEMBC benchmarks（EEMBC 基准测试），38
FP benchmarks（浮点基准测试），C-29
Livermore Fortran kernels（Livermore Fortran 内核），331
multimedia instructions（多媒体指令），A-31
multiprocessor architecture（多处理器体系结构），408
multiprogramming workload（多重编程工作负载），375~378，377
performance benchmarks（性能基准测试），37，331
primitives（原语），A-30
protecting processes（保护进程），B-50
segmented virtual memory（分段虚拟存储器），B-51
SIMD exploitation（SIMD 开发），330
vector, on vector processor and GPU（向量，在向量处理器和 GPU 上），334~336
virtual memory protection（虚拟存储器保护），106
WSCs，438

L

L1 caches，另见 First-level caches
L2 caches，另见 Second-level caches

L3 caches，另见 Third-level caches
Lanes（车道）
GPUs vs. vector architectures（GPU 与向量体系结构），**310**
Sequence of SIMD Lane Operations（SIMD 车道操作序列），**292**，**313**
SIMD Lane Registers（SIMD 车道寄存器），**309**，**314**
SIMD Lanes（SIMD 车道），296~297，**297**，302~303，308，**309**，311~312，**314**
vector execution time（向量执行时间），269
vector instruction set（向量指令集），271~273
Vector Lane Registers（向量车道寄存器），**292**
Vector Lanes（向量车道），**292**，296~297，309，311
Latency，另见 Response time
Layer 3 network, array and Internet linkage（L3 网络，阵列与互联网连接），**445**
Layer 3 network, WSC memory hierarchy（L3 网络，WSC 存储器层次结构），445
Learning curve, cost trends（学习曲线，成本直接），27
Least recently used（LRU）（最近使用）
AMD Opteron data cache（AMD Opteron 数据缓存），B-12，B-14
block replacement（块替换），B-9
virtual memory block replacement（虚拟存储器块替换），B-45
Level 3, as Content Delivery Network（L3，作为内容交付网络），460
Limit field, IA-32 descriptor table（界限字段，IA-32 描述符表），B-52
Line, memory hierarchy basics（行，存储器层次结构基础），74
Linear speedup（线性加速比）
cost effectiveness（成本效率），407
IBM eServer p5 multiprocessor（IBM eServer p5 多处理器），**408**
multicore processors（多核处理器），400，**402**
performance（性能），405~406
Link register（链接寄存器）
MIPS control flow instructions（MIPS 控制流指令），A-37~A-38
procedure invocation options（过程调用选项），A-19
synchronization（同步），389
Linpack benchmark（Linpack 基准测试）
vector processor example（向量处理器示例），267~268
Linux operating systems（Linux 操作系统）
Amazon Web Services（Amazon Web 服务），456~457
architecture costs（体系结构成本），2
protection and ISA（保护与 ISA），112
WSC services（WSC 服务），441
Lisp
ILP，215
as MapReduce inspiration（为 MapReduce 启发），437
Literal addressing mode, basic considerations（直接数寻址方式，基本考虑事项），A-10~A-11
Little Endian（小端）
memory address interpretation（存储器地址解释），A-7
MIPS data transfers（MIPS 数据传输），A-34
Liveness, control dependence（活性，控制相关性），156
Livermore Fortran kernels, performance（Livermore Fortran 内核，性能），**331**
LMD，见 Load memory data
Load instructions（载入指令）
control dependences（控制相关性），155
data hazards requiring stall（需要停顿的数据冒险），**C-20**
dynamic scheduling（动态调度），177
ILP，**199**，201
loop-level parallelism（循环级并行），318
memory port conflict（存储器端口冲突），**C-14**
pipelined cache access（流水化缓存访问），82
RISC instruction set（RISC 指令集），C-4~C-5
Tomasulo's algorithm（Tomasulo 算法），182
VLIW sample code（VLIW 示例代码），**252**
Load linked（链接载入）
locks via coherence（通过一致性锁定），391
synchronization（同步），388~389
Load locked, synchronization（锁定载入，同步），388~389
Load memory data（LMD），simple MIPS implementation（载入存储器数据，简单 MIPS 实现），C-32~C-33
Load stalls, MIPS R4000 pipeline（载入停顿，MIPS R4000 流水线），C-67
Load-store instruction set architecture（载入-存储指令集体系结构）
basic concept（基本概念），C-4~C-5
Intel Core i7，**124**
ISA，11
ISA classification（ISA 分类），A-5
MIPS operations（MIPS 运算），A-35~A-36，**A-36**
simple MIPS implementation（简单 MIPS 实现），C-32
VMIPS，265
Load/store unit（载入/存储单元）
Fermi GPU，305
ILP hardware model（ILP 硬件模型），215
multiple lanes（多车道），**273**
Tomasulo's algorithm（Tomasulo 算法），171~173，182，197
vector units（向量单元），265，276~277

Load upper immediate（LUI），MIPS operations（载入高位立即数(LUI)，MIPS操作），A-37
Local address space, segmented virtual memory（载入地址空间，分段虚拟存储器），B-52
Locality，见 Principle of locality
Local Memory（本地存储器），292，314
Local miss rate, definition（局部缺失率，定义），B-31
Local node, directory-based cache coherence protocol basics（本地节点，目录式缓存—致性协议基础），382
Local optimizations, compilers（局部优化，编译器），A-26
Local predictors, tournament predictors（局部预测器，竞赛预测器），164~166
Local scheduling, ILP, VLIW processor（局部调度，ILP，VLIW处理器），194~195
Locks（锁）
  via coherence（通过一致性），389~391
  hardware primitives（硬件原语），387
  multiprocessor software development（多处理器软件开发），409
Lock-up free cache（自由查询缓存），83
Long integer（长整数）
  operand sizes/types（操作数大小/类型），12
  SPEC benchmarks（SPEC基准测试），A-14
Loop-carried dependences（循环间相关），315~316
Loop exit predictor, Intel Core i7（循环退出预测器，Intel Core i7），166
Loop interchange, compiler optimizations（循环交换，编译器优化），88~89
Loop-level parallelism（循环级并行），149~150
Loop stream detection, Intel Core i7 micro-op buffer（循环流检测，Intel Core i7微操作缓冲区），238
Loop unrolling（循环展开）
  basic considerations（基本考虑事项），161~162
  ILP exposure（揭示ILP），157~161
  ILP limitation studies（ILP限制性研究），220
  Tomasulo's algorithm（Tomasulo算法），179，181~183
  VLIW processors（VLIW处理器），**195**
LRU,见 Least recently used
Lucas
  compiler optimizations（编译器优化），**A-29**
  data cache misses（数据缓存缺失），B-10
LUI，见 Load upper immediate

M

Machine memory, Virtual Machines(机器存储器，虚拟机)，110
Macro-op fusion, Intel Core i7（微操作融合，Intel Core i7），237~238
Main Memory（主存储器），292，309
Manufacturing cost（制造成本）
  chip fabrication case study（芯片制造案例研究），61~62
  cost trends（成本趋势），27
  modern processors（现代处理器），**62**
  operation cost（运行成本），33
MapReduce
  cloud computing（云计算），455
  cost calculations（成本计算），458~460，**459**
  Google usage（Google应用），**437**
  reductions（减少），321
  WSC batch processing（WSC批处理），437~438
  WSC cost-performance（WSC成本性能），474
Mask Registers（遮罩寄存器），309
Matrix300 kernel（Matrix 300内核），56
Matrix multiplication（矩阵乘法）
  benchmarks（基准测试），56
  multidimensional arrays in vector architectures（向量体系结构中的多维数组），278
Maximum vector length（MVL）（最大向量长度）
Multimedia SIMD extensions（多媒体SIMD扩展），282
  vector vs. GPU（向量对GPU），311
  VLRs，274~275
M-bus，见 Memory bus
MCF
  compiler optimizations（编译器优化），**A-29**
  data cache misses（数据缓存缺失），B-10
  Intel Core i7，240~241
Mean time between failures（MTBF）（平均故障间隔时间）
  fallacies（谬论），56~57
  SLA states（SLA状态），34
Mean time to failure（MTTF）（平均无故障时间）
  computer system power consumption case study（计算机系统功耗案例研究），63~64
  example calculations（示例计算），34~35
  SLA states（SLA状态），34
  WSCs vs. servers（WSC对服务器），434
Memory access（存储器访问）
  ARM Cortex-A8 example（ARM Cortex-A8示例），**117**
  basic MIPS pipeline（基本MIPS流水线），C-36
  block size（块大小），**B-28**
  cache hit calculation（缓存命中计算），B-5
  data hazards requiring stall（需要停顿的数据冒险），C-19~C-21
  data hazard stall minimization(数据冒险停顿最小化)，C-17，C-19
  integrated instruction fetch units(集成指令提取单元)，208
  MIPS data transfers（MIPS数据传输），A-34

multimedia instruction compiler support（多媒体指令编译器支持），A-31
pipeline branch issues（流水线分支发射），**C-40**，**C-42**
RISC classic pipeline（RISC 经典流水线），C-7，C-10
shared-memory workloads（共享存储器工作负载），**372**
simple MIPS implementation（简单 MIPS 实现），C-32~C-33
simple RISC implementation（简单 RISC 实现），C-6
structural hazards（结构性冒险），C-13~C-14
Memory addressing（存储器寻址）
ALU immediate operands（ALU 立即数操作数），**A-12**
basic considerations（基本考虑事项），A-11~A-13
displacement values（位移值），**A-12**
immediate value distribution（立即数值分布），**A-13**
interpretation（解释），A-7~A-8
ISA，11
Memory banks，另见 Banked memory
Memory bus（M-bus）（存储器总线（M-总线）），351
Memory consistency（存储器一致性）
basic considerations（基本考虑事项），392~393
cache coherence（缓存一致性），352
compiler optimization（编译器优化），396
directory-based cache coherence protocol basics（目录式缓存一致性基础），382
multiprocessor cache coherency（多处理器缓存一致性），353
relaxed consistency models（宽松连贯性模型），394~395
single-chip multicore processor case study（单芯片多核处理器案例研究），412~418
speculation to hide latency（以推测隐藏延迟），396~397
Memory hierarchy（存储器层次结构）
address space（地址空间），B-57~B-58
basic questions（基本公式），B-6~B-12
block identification（块识别），B-7~B-9
block placement issues（块放置问题）B-7
block replacement（块替换），B-9~B-10
cache optimization（缓存优化），B-22~B-40
cache performance（缓存性能），B-3~B-6
case studies（案例研究），B-60~B-67
inclusion（包含），397~398
levels in slow down（减缓级别），**B-3**
Opteron data cache example（Opteron 数据缓存示例），B-12~B-15，**B-13**
Opteron L1/L2，**B-57**
OS and page size（OS 和页大小），B-58
overview（概述），**B-39**
Pentium vs. Opteron protection（Pentium 与 Opteron 保护），B-57
processor examples（处理器示例），B-3
process protection（进程保护），B-50
terminology（术语），B-2~B-3
virtual memory（虚拟存储器），B-40~B-54
write strategy（写策略），B-10~B-12
WSCs，**443**，443~446，**444**
Memory hierarchy design（存储器层次结构设计）
access times（访问时间），**77**
Alpha 21264 floorplan（Alpha 21264 平面图），**143**
ARM Cortex-A8 example（ARM Cortex-A8 示例），114~117，**115~117**
cache coherency（缓存一致性），112~113
cache optimization（缓存优化），81~133
cache performance prediction（缓存性能预测），125~126
cache size and misses per instruction（缓存大小与每指令的缺失数），**126**
DDR2 SDRAM timing diagram（DDR2 SDRAM 时序图），**139**
highly parallel memory systems（高度并行存储器系统），133~136
high memory bandwidth（高存储器带宽），126
instruction miss benchmarks（指令缺失基准测试），**127**
instruction simulation（指令模拟），126
Intel Core i7，117~124，**119**，**123~125**
Intel Core i7 three-level cache hierarchy（Intel Core i7 三级缓存层次结构），**118**
Intel Core i7 TLB structure（Intel Core i7 TLB 结构），**118**
Intel 80x86 virtualization issues（Intel 80x86 虚拟问题），**128**
memory basics（存储器基础），74~78
overview（概述），72~74
protection and ISA（保护与 ISA），112
server vs. PMD（服务器对 PMD），**72**
system call virtualization/ paravirtualizationperformance（系统调用虚拟/部分虚拟化性能），**141**
virtual machine monitor（虚拟机监视器），108~109
Virtual Machines ISA support（虚拟机 ISA 支持），109~110
Virtual Machines protection（虚拟机保护），107~108
Virtual Machines and virtual memory and I/O（虚拟机和虚拟存储器、I/O），110~111
virtual memory protection（虚拟存储器保护），105~107
VMM on nonvirtualizable ISA（不可虚拟化 ISA 上的 VMM），128~129
Xen VM example（Xen VM 示例），111
Memory Interface Unit（存储器接口单元）
NVIDIA GPU ISA，300

vector processor example（向量处理器示例），**310**
Memory mapping（存储器映射）
memory hierarchy（存储器层次结构），B-48~B-49
segmented virtual memory（分段虚拟存储器），B-52
TLBs，323
virtual memory definition（虚拟存储器定义），B-42
Memory-memory instruction set architecture, ISA classification,（存储器-存储器指令集体系结构，ISA分类），A-3，A-5
Memory protection（存储器保护）
control dependence（控制相关性），155
Pentium vs. Opteron（Pentium对Opteron），B-57
processes（进程），B-50
safe calls（安全调用），B-54
segmented virtual memory example（分段虚拟存储器示例），B-51~B-54
virtual memory（虚拟存储器），B-41
Memory stall（存储器停顿），B-4~B-5
Memory technology basics（存储器技术基础）
DRAM，**98**，98~100，**99**
DRAM and DIMM characteristics(DRAM和DIMM特性），**101**
DRAM performance（DRAM性能），100~102
Flash memory（闪存），102~104
overview（概述），96~97
performance trends（性能趋势），**20**
SDRAM power consumption（SDRAM功耗），102，**103**
SRAM，97~98
MESI, 见 Modified-Exclusive-Shared-Invalid（MESI）protocol
Messages（消息）
coherence maintenance（一致性保持），**381**
Microarchitecture（微体系结构）
architecture component（体系结构组件），15~16
ARM Cortex-A8，241
data hazards（数据冒险），168
ILP exploitation（ILP开发），197
Intel Core i7，236~237
Nehalem，411
out-of-order example（乱序示例），**253**
PTX vs. x86（PTX对x86），298
study，47~254
Microfusion, Intel Core i7 micro-op buffer（微合并，Intel Core i7微操作缓冲区），238
Microinstructions（微指令）
Complications（复杂性），C-50~C-51
x86，298
Micro-ops（微操作）
Intel Core i7，**237**，238~240，**239**
processor clock rates（处理器时钟频率），244
Microprocessor overview（微处理器概述）
clock rate trends（时钟频率趋势），**24**
cost trends（成本趋势），27~28
desktop computers（台式计算机），6
embedded computers（嵌入式计算机），8~9
energy and power（能量与功率），23~26
integrated circuit improvements（集成电路改进），2
Moore's law（摩尔定律），3-4
performance trends（性能趋势），19~20，**20**
power and energy system trends（功率与能量系统趋势），21~23
technology trends（技术趋势），18
Microprocessor without Interlocked Pipeline Stages，见 MIPS（Microprocessor without Interlocked Pipeline Stages）
Microsoft（微软）
cloud computing（云计算），455
Intel support（Intel支持），245
WSCs，464~465
Microsoft Azure，465
Microsoft Windows
benchmarks（基本测试），38
multithreading（多线程），223
time/volume/commoditization impact（时间/数量/大众商品化影响），28
WSC workloads（WSC工作负载），441
Microsoft Windows 2008 Server
real-world considerations（现实考虑事项），52~55
SPECpower benchmark（SPECpower基准测试），463
Migration, cache coherent multiprocessors（迁移，缓存一致性多处理器），354
MIMD(Multiple Instruction Streams, Multiple Data Streams）and Amdahl's law（多指令流，多数据流）与Amdahl定律），406~407
Minicomputers, replacement by microprocessors（小型计算机，被微处理器代替），3~4
Minniespec benchmarks（Minniespec基准测试）
ARM Cortex-A8，**116**，235
ARM Cortex-A8 memory（ARM Cortex-A8存储器），115~116
MIPS（Microprocessor without Interlocked Pipeline Stages）（没有互连流水级的多级互连）
addressing modes（寻址方式），11~12
basic pipeline（基本流水线），C-34~C-36
branch predictor correlation（分支预测器相关），163
cache performance（缓存性能），B-6
control flow instructions（控制流指令），14
data dependences（数据相关），151

data hazards（数据冒险），169
dynamic scheduling with Tomasulo's algorithm（采用 Tomasulo 算法的动态调度），171，173
encoding（编码），14
exceptions（异常），C-48，C-48~C-49
exception stopping/restarting（异常停止/重启），C-46~C-47
FP pipeline performance（浮点流水线性能），C-60~C-61，**C-62**
FP unit with Tomasulo's algorithm（采用 Tomasulo 算法的浮点单元），**173**
hazard checks（冒险检测），C-71
ILP，149
ILP exposure（揭示 ILP），157~158
ILP hardware model（ILP 硬件模型），215
instruction set complications（指令集复杂性），C-49~C-51
ISA class（ISA 类），11
ISA example（ISA 示例），A-22~A-42
Livermore Fortran kernel performance（Livermore Fortran 内核性能），**331**
memory addressing（存储器寻址），11
basic considerations（基本考虑事项），C-51~C-54
hazards and forwarding（冒险与转发），C-54~C-58
precise exceptions（精确异常），C-58~C-60
operands（操作数），12
pipeline branch issues（流水线分支发射），C-39~C-42
pipeline control（流水线控制），C-36~C-39
pipe stage（流水级），C-37
processor performance calculations（处理器性能计算），218~219
registers and usage conventions（寄存器与使用约定），**12**
RISC code size（RISC 代码大小），A-23
scoreboard components（记分卡组件），**C-76**
scoreboarding（记分卡），C-72
scoreboarding steps（记分卡步骤），**C-73**，C-73~C-74
simple implementation（简单实现），C-31~C-34，**C-34**
unpipelined functional units（非流水化功能单元），**C-52**
write strategy（写策略），B-10
MIPS16
instruction set architecture formats（指令集体系结构结构），**14**
instruction subset（指令子集），**13**，**A-40**
in MIPS R4000（在 MIPS R4000 中），C-61
RISC instruction set（RISC 指令集），C-4
MIPS R4000
FP pipeline（浮点流水线），C-65~C-67，**C-66**
integer pipeline（整数流水线），**C-63**
pipeline overview（流水线概述），C-61~C-65
pipeline performance（流水线性能），C-67~C-70
pipeline structure（流水线结构），C-62~C-63
MIPS R8000 precise exceptions（MIPS R8000 精确异常），C-59
MIPS R10000，81
latency hiding（隐藏延迟），397
precise exceptions（精确异常），C-59
Misalignment, memory address interpretation（非对齐，存储器地址解释），A-7~A-8，**A-8**
Misprediction rate（错误预测率）
branch-prediction buffers（分支预测缓冲区），**C-29**
predictors on SPEC89（关于 SPEC 89 的预测器），**166**
profile-based predictor（基于一览数据的预测器），**C-27**
SPECCPU2006 benchmarks（SPECCPU2006 基准测试），**167**
Mispredictions（错误预测）
ARM Cortex-A8，**232**，235
branch predictors（分支预测器），164~167，240，C-28
branch-target buffers（分支目标缓冲区），205
hardware-based speculation（基于硬件的推测），190
hardware vs. software speculation（硬件推测与软件推测），221
integer vs. FP programs（整数程序与浮点程序），**212**
Intel Core i7，**237**
prediction buffers（预测缓冲区），**C-29**
static branch prediction（静态分支预测），C-26~C-27
Misses per instruction（每指令缺失数）
application/OS statistics（应用程序/操作系统统计数字），**B-59**
cache performance（缓存性能），B-5~B-6
cache protocols（缓存协议），359
cache size effect（缓存大小影响），**126**
L3cache block size（L3 缓存块大小），**371**
memory hierarchy basics（存储器层次结构基础），75
performance impact calculations（性能影响计算），B-18
shared-memory workloads（共享存储器工作负载），**372**
SPEC benchmarks（SPEC 基准测试），**127**
strided access-TLB interactions（步骤访问-TLB 交互），323
Miss penalty（缺失代价）
average memory access time（存储器平均访问时间），B-16~B-17
cache optimization（缓存优化），79，B-35~B-36
cache performance（缓存性能），B-4，B-21
compiler-controlled prefetching（编译器控制预取），92~95
critical word first（关键字优化），86~87
hardware prefetching（硬件预取），91~92
ILP speculative execution（ILP 推测执行），223
memory hierarchy basics（存储器层次结构基础），75~76

nonblocking cache（非阻塞缓存），83
out-of-order processors（乱序处理器），B-20~B-22
processor performance calculations（处理器性能计算），218~219
reduction via multilevel caches（通过多级缓存来降低），B-30~B-35
write buffer merging（写缓冲区合并），87
Miss rate（缺失率）
AMD Opteron data cache（AMD Opteron 数据缓存），B-15
ARM Cortex-A8，**116**
average memory access time（存储器平均访问时间），B-16~B-17，**B-29**
basic categories（基本类别），B-23
block size（块大小），**B-27**
cache optimization（缓存优化），79
cache performance（缓存性能），B-4
cache size（缓存大小），**B-24~B-25**
compiler-controlled prefetching（编译器控制预取），92~95
compiler optimizations（编译器优化），87~90
example calculations（示例计算），B-6，B-31~B-32
hardware prefetching（硬件预取），91~92
Intel Core i7，**123**，**125**，241
memory hierarchy basics（存储器层次结构基础），75~76
multilevel caches（多级缓存），**B-33**
processor performance calculations（处理器性能计算），218~219
shared-memory multiprogramming workload（共享存储器多重编程工作负载），**376**，376~377
shared-memory workload（共享存储器工作负载），370~373
single vs. multiple thread executions（单线程与多线程执行），**228**
Sun T1 multithreading unicore performance（Sun T1 多线程单核性能），228
virtual addressed cache size（虚拟寻址缓存大小），**B-37**
Mixed cache（混合式缓存）
AMD Opteron example（AMD Opteron 示例），B-15
commercial workload（商业工作负载），373
MMX，见 Multimedia Extensions
Mobile clients（移动客户端）
data usage（数据使用），3
GPU features（GPU 特征），**324**
server GPUs（服务器 GPU），323~330
Modified-Exclusive-Shared-Invalid（MESI） protocol（修改–排除–共享–废弃（MESI）协议）
characteristics（特性），362
Modified-Owned-Exclusive-Shared-Invalid（MOESI）protocol
characteristics（修改–拥有–排除–共享–废弃(MOESI）协议特性），362
Modified state（经修改状态）
coherence protocol（一致性协议），362
directory-based cache coherence（目录式缓存一致性）
protocol basics（协议基础），380
snooping coherence protocol（监听一致性协议），358~359
Module availability, definition（模块可用性，定义），34
Module reliability, definition（模块可靠性，定义），34
Moore's law（摩尔定律）
DRAM，100
flawed architectures（有缺陷体系结构），A-45
microprocessor dominance（处理器优势），3~4
RISC，A-3
software importance（软件重要性），55
technology trends（技术趋势），17
Mortar shot graphs, multiprocessor performance measurement（迫击炮射击曲线，多处理器性能测量），405~406
Multibanked caches（多组缓存）
cache optimization（缓存优化），85~86
example（示例），**86**
Multicomputers（多计算机），345，2-59
Multicore processors（多核处理器）
architecture goals/requirements（体系结构目标/需求），15
cache coherence（缓存一致性），361~362
centralized shared-memory multiprocessor structure（集中式共享存储器多处理器结构），**347**
directory-based cache coherence（目录式缓存一致性），**380**
directory-based coherence（目录式一致性），381，419
DSM architecture（DSM 体系结构），**348**，379
multichip（多芯片），419
multiprocessors（多处理器），345
performance（性能），400~401，**401**
performance gains（性能增益），398~400
performance milestones（性能里程碑），**20**
single-chip case study（单芯片案例研究），412~418
SMT，404~405
snooping cache coherence implementation（监听缓存一致性实现），365
SPEC benchmarks（SPEC 基准测试），**402**
uniform memory access（一致存储器访问），**364**
write invalidate protocol implementation（写失效协议实现），356~357
Multicycle operations, MIPS pipeline（多周期操作，MIPS 流水线）
basic considerations（基本考虑事项），C-51~C-54
hazards and forwarding（冒险与转发），C-54~C-58

precise exceptions（精确异常），C-58~C-60
Multidimensional arrays（多维数组）
dependences（相关性），318
in vector architectures（向量体系结构中），278~279
Multilevel caches（多级缓存）
cache optimizations（缓存优化），B-22
centralized shared-memory architectures（集中式共享存储器体系结构），351
memory hierarchy basics（存储器层次结构基础），76
miss penalty reduction（降低缺失代价），B-30~B-35
miss rate vs. cache size（缺失率与缓存大小），**B-33**
Multimedia SIMD vs. GPU（多媒体 SIMD 与 GPU），312
Multilevel exclusion, definition（多级互斥，定义），B-35
Multilevel inclusion（多级包含），397，B-34
Multimedia applications（多媒体应用程序）
GPUs，288
ISA support（ISA 支持），A-46
MIPS FP operations（MIPS 浮点运算），A-39
vector architectures（向量体系结构），267
Multimedia Extensions（MMX）（多媒体扩展（MMX））
compiler support（编译器支持），A-31
SIMD history（SIMD 历史），262
vector architectures（向量体系结构），282~283
Multimedia instructions（多媒体指令）
ARM Cortex-A8，236
compiler support（编译器支持），A-31~A-32
Multimedia SIMD Extensions（多媒体 SIMD 扩展）
basic considerations（基本考虑事项），262，282~284
compiler support（编译器支持），A-31
DLP，322
GPUs，312
MIMD, vs. GPU（MIMD 与 GPU），324~330
parallelism classes（并行分类），10
programming（编程），285
Roofline visual performance model（屋顶轮廓线虚拟性能模型），285~288，**287**
256-bit-wide operations（256 位宽操作），**282**
vector（向量），263~264
Multimedia user interfaces, PMDs（多媒体用户接口，PMD），6
Multiple Instruction Streams, Multiple Data Streams, 见 MIMD
Multiple Instruction Streams, Single Data Stream（MISD），definition（多指令流，单数据流（MISD），定义），10
Multiple-issue processors（多发射处理器）
basic VLIW approach（基本 VLIW 方法），193~196
dynamic scheduling and speculation（动态调度与推测），197~202
instruction fetch bandwidth（指令提取带宽），202~203
integrated instruction fetch units（集成指令提取单元），207
loop unrolling（循环展开），162
microarchitectural techniques case study（微体系结构技术案例研究），247~254
primary approaches（主要方法），**194**
SMT，224，226
speculation（推测），**198**
Tomasulo's algorithm（Tomasulo 算法），183
Multiple lanes technique（多车道技术）
vector instruction set（向量指令集），271~273
Multiple paths, ILP limitation studies（多路径，ILP 局限性研究），220
Multiply operations（乘法运算）
unfinished instructions（未完成指令），**179**
Multiprocessor basics（多处理器基础），354
Multiprogramming（多重编程），345
Multithreaded SIMD Processor（多线程 SIMD 处理器），292，309，313-314
Multithreaded vector processor（多线程向量处理器），292
Multithreading（多线程），223~225
MVL，见 Maximum vector length

N

Name dependences（名称相关性）
ILP，152~153
locating dependences（定位相关性），318~319
loop-level parallelism（循环级并行），315
scoreboarding（记分卡），C-79
Tomasulo's algorithm（Tomasulo 算法），171~172
Nameplate power rating, WSCs（名牌额定功率），449
NAND Flash, definition（NAND Flash，定义），103
Natural parallelism（自然并行）
multiprocessor importance（多处理器重要性），344
multithreading（多线程），223
NEC Earth Simulator, peak performance（NEC Earth 模拟器，峰值性能），**58**
NEC SX-9
Roofline model（屋顶轮廓线模型），286~288，**287**
Nested page tables（嵌套页表），129
NetApp，见 Network Appliance
Netflix, AWS，460
Network Appliance（NetApp）（网络设备）
Network attached storage（NAS）（网络连接存储）
WSCs，442
Network File System（NFS）（网络文件系统）
server benchmarks（服务器基准测试），40
Networking costs, WSC vs. datacenters（联网成本，WSC

与数据中心），455
Network interface card（NIC）（网络接口卡(NIC)）
Google WSC servers（Google WSC服务器），469
Network technology, 另见 Interconnection networks
NFS, 见 Network File System
NIC, 见 Network interface card
NMOS, DRAM，**99**
Nodes（节点）
coherence maintenance（一致性维护），**381**
directory-based cache coherence(目录式缓存一致性)，**380**
parallel（并行），**336**
Nonatomic operations（非原子操作）
cache coherence（缓存一致性），361
directory protocol（目录协议），386
Nonbinding prefetch, cache optimization（非绑定预取，缓存优化），93
Nonblocking caches（非阻塞缓存）
cache optimization（缓存优化），83~85，131~133
effectiveness（有效性），**84**
ILP speculative execution（ILP推测执行），222~223
Intel Core i7，**118**
Nonfaulting prefetches, cache optimization（无错误预取，缓存优化），92
Nonuniform memory access（NUMA）（非一致存储器访问）
DSM as，348
snooping limitations（监听局限性），363~364
Non-unit strides（非单位步幅）
multidimensional arrays in vector architectures（向量体系结构中的多维数组），278~279
vector processor（向量处理器），**310**，310~311
No-write allocate（非写入分派），B-11
Nullifying branch, branch delay slots（废弃分支，分支延迟时隙），C-24~C-25
NUMA,见 Nonuniform memory access
NVIDIA systems（NVIDIA系统）
fine-grained multithreading（细粒度多线程），224
GPU comparisons（GPU对比），323~330，**325**
GPU computational structures（GPU计算结构），291~297
GPU ISA，298~300
GPU Memory structures（GPU存储器结构），**304**，304~305
GPU programming（GPU编程），289
terminology（术语），313~315
N-way set associative（*N*路组相联）
block placement（块放置），B-7
conflict misses（冲突缺失），B-23
memory hierarchy basics（存储器层次结构基础），74
TLBs，B-49

O

Observed performance, fallacies（观测性能，谬论），57
Offset（偏移量）
addressing modes（寻址方式），12
AMD64 paged virtual memory（AMD64分页虚拟存储器），B-55
block identification（块识别），B-7~B-8
cache optimization（缓存优化），B-38
call gates（调用门），B-54
control flow instructions（控制流指令），A-18
directory-based cache coherence protocols（目录式缓存一致性协议），381~382
example（示例），B-9
gather-scatter（集中-分散），280
IA-32 segment（IA-32分段），B-53
instruction decode（指令译码），C-5~C-6
main memory（主存储器），B-44
memory mapping（存储器映射），B-52
MIPS，C-32
MIPS control flow instructions（MIPS控制流指令），A-37~A-38
misaligned addresses（非对齐地址），A-8
Opteron data cache（Opteron数据缓存），B-13~B-14
pipelining（流水化），**C-42**
PTX instructions（PTX指令），300
RISC，C-4~C-6
RISC instruction set（RISC指令集），C-4
TLB，B-46
Tomasulo's approach（Tomasulo方法），176
virtual memory（虚拟存储器），B-43~B-44，B-49，B-55~B-56
OLTP,见 On-Line Transaction Processing
OMNETPP, Intel Core i7，240~241
On-chip cache（片上缓存）
optimization（优化），79
SRAM，98~99
One-way conflict misses, definition（1路冲突缺失，定义），B-23
On-Line Transaction Processing（OLTP）（联机事务处理）
commercial workload（商业工作负载），**369**，**371**
server benchmarks（服务器基准测试），41
shared-memory workloads（共享存储器工作负载），368~370，373~374
OpenCL
GPU programming（GPU编程），289
GPU terminology（GPU术语），**292**，313~315
NVIDIA terminology（NVIDIA术语），291

processor comparisons（处理器对比），323
Open source software（开源软件）
Amazon Web Services（Amazon Web 服务），457
WSCs，437
Operands（操作数）
Forwarding（转发），**C-19**
instruction set encoding（指令集编码），A-21~A-22
ISA，12
ISA classification（ISA 分类），A-3~A-4
MIPS data types（MIPS 数据类型），A-34
MIPS pipeline（MIPS 流水线），C-71
MIPS pipeline FP operations（MIPS 流水线浮点运算），C-52~C-53
NVIDIA GPU ISA，298
per ALU instruction example（每 ALU 指令示例），**A-6**
type and size（类型与大小），A-13~A-14
vector execution time（向量执行时间），268~269
Operating systems（general）（操作系统（通用））
address translation（地址转换），B-38
architecture development（体系结构开发），2
memory protection performance（存储器保护性能），B-58
miss statistics（缺失统计数字），**B-59**
multiprocessor software development（多处理器软件开发），408
page size（页大小），B-58
segmented virtual memory（分段虚拟存储器），B-54
server benchmarks（服务器基准测试），40
shared-memory workloads（共享存储器工作负载），374~378
Operational costs（操作成本）
basic considerations（基本考虑事项），33
WSCs，434，438，452，456，472
Operational expenditures（OPEX）（运行费用）
WSC costs（WSC 成本），452~455，**454**
WSC TCO case study（WSC TCO 案例研究），476~478
Oracle database（Oracle 数据库）
commercial workload（商业工作负载），368
miss statistics（缺失统计数字），**B-59**
multithreading benchmarks（多线程基准测试），**232**
single-threaded benchmarks（单线程基准测试），**243**
WSC services（WSC 服务），441
Organization（组织）
cache, performance impact（缓存，性能影响），B-19
cache blocks（缓存块），B-7~B-8
cache optimization（缓存优化），B-19
coherence extensions（一致性扩展），362
computer architecture（计算机体系结构），11，15~16
DRAM，**98**
MIPS pipeline（MIPS 流水线），**C-37**
multiple-issue processor（多发射处理器），197，**198**
Opteron data cache（Opteron 数据缓存），B-12~B-13，**B-13**
Pipelines（流水线），152
processor history（处理器历史），2~3
processor performance equation（处理器性能公式），49
shared-memory multiprocessors（共享存储器多处理器），346
TLB，B-46
Orthogonality, compiler writing-architecture relationship（正交性，编译器编写-体系结构关系），A-30
Out-of-order execution and cache miss（乱序执行与缓存缺失），B-2~B-3
cache performance（缓存性能），B-21
data hazards（数据冒险），169~170
hardware-based execution（基于硬件的执行），184
ILP，245
memory hierarchy（存储器层次结构），B-2~B-3
microarchitectural techniques case study（微体系结构技术案例研究），247~254
MIPS pipeline（MIPS 流水线），C-71
miss penalty（缺失代价），B-20~B-22
performance milestones（性能里程碑），**20**
power/DLP issues（功能/DLP 问题），322
processor comparisons（处理器对比），323
R10000，397
SMT，246
Tomasulo's algorithm（Tomasulo 算法），183
Out-of-order processors（乱序处理器）
DLP，322
Intel Core i7，236
multithreading（多线程），226
vector architecture（向量体系结构），267
Out-of-order write, dynamic scheduling（乱序写，动态调度），171
Output dependence（输出相关），152~153
Overclocking（超频模式）
microprocessors（微处理器），26
processor performance equation（处理器性能公式），52
Oversubscription（超额认购）
array switch（阵列交换机），443
Google WSC，469
WSC architecture（WSC 体系结构），441，461

P

Packed decimal, definition（压缩十进制，定义），A-14
Page coloring, definition（页面着色，定义），B-38
Paged segments, characteristics（分页段，特性），B-43~B-44

Paged virtual memory（分页虚拟存储器）
Opteron example（Opteron示例），B-54~B-57
protection（保护），106
segmented（分段），**B-43**
Page faults（页面错误）
cache optimization（缓存优化），A-46
exceptions（异常），C-43~C-44
hardware-based speculation（基于硬件的推测），188
memory hierarchy（存储器层次结构），B-3
MIPS exceptions（MIPS 异常），C-48
Multimedia SIMD Extensions（多媒体 SIMD 扩展），284
stopping/restarting execution（停止/重启执行），C-46
virtual memory definition（虚拟存储器定义），B-42
virtual memory miss（虚拟存储器缺失），B-45
Page offset（页偏移）
cache optimization（缓存优化），B-38
main memory（主存储器），B-44
TLB，B-46
virtual memory（虚拟存储器），B-43，B-49，B-55~B-56
Pages（页），B-3
Page size（页大小），B-56
Page Table Entry（PTE）（页表项）
AMD64 paged virtual memory（AMD64 分页虚拟存储器），B-56
IA-32 equivalent（IA-32 等价），B-52
Intel Core i7，120
main memory block（主存储器块），B-44~B-45
paged virtual memory（分而虚拟存储器），B-56
TLB，B-47
Page tables（页表）
address translation（地址转换），B-46~B-47
AMD64 paged virtual memory（AMD64 分页虚拟存储器），B-55~B-56
descriptor tables as（描述符表），B-52
IA-32 segment descriptors（IA-32 段描述符），B-53
main memory block（主存储器块），B-44~B-45
multiprocessor software development（多处理器软件开发），407~409
multithreading（多线程），224
protection process（保护进程），B-50
segmented virtual memory（分段虚拟存储器），B-51
virtual memory block identification（虚拟存储器块识别），B-44
virtual-to-physical address mapping（虚拟地址-物理地址映射），**B-45**
Palt, definition（页错误，定义），B-3
Parallelism（并行）
cache optimization（缓存优化），79
challenges（挑战），349~351
classes（类），9~10
computer design principles（计算机设计原理），44~45
loop-level（循环级），149~150，215，217~218，315~322
MIPS scoreboarding（MIPS 记分卡），C-77~C-78
multiprocessors（多处理器），345
natural（自然），223，344
request-level（请求级），4~5，9，345，434
RISC development（RISC 开发），2
speedup（加速比），**263**
task-level（任务级），**9**
window size（窗口大小），**217**
WSCs vs. servers（WSC 与服务器），433~434
Parallel programming（并行编程）
Parallel Thread Execution（PTX）（并行线程执行）
basic GPU thread instructions（基本 GPU 线程指令），**299**
GPU conditional branching（GPU 条件分支），300~303
GPUs vs. vector architectures（GPU 对向量体系结构），308
NVIDIA GPU ISA，298~300
NVIDIA GPU Memory structures（NVIDIA GPU 存储器结构），305
Parallel Thread Execution（PTX）Instruction（并行线程执行指令），292，309，313
Paravirtualization（部分虚拟化）
system call performance（系统调用性能），**141**
Xen VM，111
Parity（奇偶校验）
fault detection（错误检测），58
memory dependability（存储器可靠性），104~105
WSC memory（WSC 存储器），473~474
PARSEC benchmarks（PARSEC 基准测试）
Intel Core i7，401~405
SMT on superscalar processors（超标量处理器上的 SMT），230~232，**231**
speedup without SMT（没有 SMT 的加速比），**403~404**
Partial store order, relaxed consistency models（部分存储顺序，宽松的一致性模型），395
Partitioning（划分）
Multimedia SIMD Extensions（多媒体 SIMD 扩展），282
virtual memory protection（虚拟存储器保护），B-50
WSC memory hierarchy（WSC 存储器层次结构），445
Pascal programs（Pascal 程序）
compiler types and classes（编译器类型和类），**A-28**
PC, 见 Program counter
PDP-11，2-10，2-17~2-19，2-56
Peak performance（峰值性能）
DLP，322

fallacies（谬论），57~58
multiple lanes（多车道），273
multiprocessor scaled programs（多处理器缩放程序），**58**
Roofline model（屋顶轮廓线模型），**287**
vector architectures（向量体系结构），331
WSC operational costs（WSC 运行成本），434
Perfect Club benchmarks（Perfect Club 基准测试）
vector architecture programming（向量体系结构编程），**281**，281~282
Perfect processor，ILP hardware model（完美处理器，ILP 硬件模型），214~215，**215**
Performance，另见 Peak performance
Personal mobile device（PMD）（个人移动设备）
characteristics（特性），6
computer class（计算机类），**5**
embedded computers（嵌入式计算机），8~9
Flash memory（闪存），18
integrated circuit cost trends（集成电路成本趋势），28
ISA performance and efficiency prediction（ISA 性能与效率预测），241~243
memory hierarchy basics（存储器层次结构基础），78
memory hierarchy design（存储器层次结构设计），**72**
power and energy（功耗与能量），25
processor comparison（处理器比较），**242**
Phase-ordering problem, compiler structure（阶段排序问题，编译器结构），A-26
Physical addresses（物理地址）
address translation（地址变换），B-46
AMD Opteron data cache（AMD Opteron 数据缓存），B-12~B-13
ARM Cortex-A8，**115**
directory-based cache coherence protocol basics（目录式缓存一致性协议基础），382
main memory block（主存储器块），B-44
memory hierarchy（存储器层次结构），B-48~B-49
memory hierarchy basics（存储器层次结构基础），77~78
memory mapping（存储器映射），B-52
paged virtual memory（分页虚拟存储器），B-55~B-56
page table-based mapping（基于分页表的映射），**B-45**
safe calls（案例调用），B-54
segmented virtual memory（分段虚拟存储器），B-51
sharing/protection（共享/保护），B-52
translation（转换），B-36~B-39
virtual memory definition（虚拟存储器定义），B-42
Physical cache, definition（物理缓存，定义），B-36~B-37
Physical memory（物理存储器）
centralized shared-memory multiprocessors（集中式共享存储器多处理器），347
directory-based cache coherence（目录式缓存一致性），354
future GPU features（未来 GPU 特征），332
GPU conditional branching（GPU 条件分支），303
main memory block（主存储器块），B-44
memory hierarchy basics（存储器层次结构基础），B-41~B-42
multiprocessors（多处理器），345
paged virtual memory（分页虚拟存储器），B-56
processor comparison（处理器比较），323
segmented virtual memory（分段虚拟存储器），B-51
unified（统一），333
Virtual Machines（虚拟机），110
Pipeline bubble, stall as（流水线气泡，作为停顿），C-13
Pipeline cycles per instruction（每条指令的流水线周期）
basic equation（基本公式），148
ILP，149
processor performance calculations（处理器性能计算），218~219
R4000 performance（R4000 性能），C-68~C-69
Pipeline delays（流水线延迟），228
Pipeline interlock（流水线互锁）
data dependences（数据相关性），151
data hazards requiring stall（需要停顿的数据冒险），C-20
MIPS R4000，C-65
MIPS vs. VMIPS（MIPS 与 VMIPS），268
Pipeline latches（流水线闩锁），C-35
Pipeline organization（流水线组织）
dependences（相关性），152
MIPS，**C-37**
Pipeline registers（流水线寄存器），C-35
Pipeline scheduling（流水线调度）
basic considerations（基本考虑事项），161~162
dynamic scheduling（动态调度），168~169
ILP exploitation（ILP 开发），197
ILP exposure（揭示 ILP），157~161
microarchitectural techniques case study（微体系结构技术案例研究），247~254
MIPS R4000，C-64
Pipeline stall cycles（流水线停顿周期）
branch scheme performance（分支方案性能），C-25
pipeline performance（流水线性能），C-12~C-13
Pipelining（流水化），C-2~C-3
Pipe segment, definition（流水段，定义），C-3
Pipe stage（流水级），C-3
PMD，见 Personal mobile device
Point-to-point multiprocessor, example（点对点多处理器，示例），**413**

Point-to-point networks（点对点网络）
directory-based coherence（目录式一致性），418
directory protocol（目录协议），421~422
SMP limitations（SMP局限性），363~364
Portable computers（便携式计算机）
processor comparison（处理器对比），**242**
Position independence, control flow instruction addressing modes（位置独立性，控制流指令寻址方式），A-17
Power（电力）
distribution for servers（服务器配电），**490**
distribution overview（配电概述），**447**
DLP，322
first-level caches（一级缓存），79~80
Google server benchmarks（Google服务器基准测试），439~441
Google WSC，465~468
PMDs，6
real-world server considerations（现实服务器考虑事项），52~55
WSC infrastructure（WSC基础设施），447
WSC power modes（WSC功率模式），472
WSC resource allocation case study（WSC资源分配案例研究），478~479
WSC TCO case study（WSC TCO案例研究），476~479
Power consumption，另见 Energy efficiency
Power distribution unit（PDU），WSC infrastructure（配电单元，WSC基础设施），447
Power failure（电力故障）
exceptions（异常），C-43~C-44，C-46
utilities（公用设施），435
WSC storage（WSC存储），442
Power gating, transistors（电源闸控，晶体管），26
Power modes, WSCs（功率模式，WSC），472
PowerPC
AltiVec multimedia instruction compiler support（AltiVec多媒体指令编译器支持），A-31
consistency model（一致性模型），395
IBM Blue Gene/L
multimedia compiler support（多媒体编译器支持），A-31
precise exceptions（精确异常），C-59
RISC architecture（RISC体系结构），A-2
RISC code size（RISC代码大小），A-23
Power-performance（功率-性能）
low-power servers（低功率服务器），**477**
servers（服务器），**54**
Power Supply Units（PSUs），efficiency ratings（电源单元（PSU），额外效率），**462**
Power utilization effectiveness（PUE）（电力利用效率（PUE））
datacenter comparison（数据中心对比），**451**
Google WSC，**468**
Google WSC containers（Google WSC集装箱），464~465
WSC，450~452
WSCs vs. datacenters（WSC与数据中心），456
WSC server energy efficiency（WSC服务器能量效率），462
Precise exceptions（精确异常），C-47
Predicate Registers（断言寄存器），309
Predicted-not-taken scheme（预测未选中方案）
branch penalty reduction（降低分支代价），**C-22**，C-22~C-23
MIPS R4000 pipeline（MIPS R4000流水线），C-64
Predictions, 另见 Mispredictions
Prefetching（预取）
integrated instruction fetch units（集成指令提取单元），208
Intel Core i7，122，**123~124**
NVIDIA GPU Memory structures（NVIDIA GPU存储器结构），305
parallel processing challenges（并行处理挑战），351
Present bit, IA-32 descriptor table（存在位，IA-32描述符表），B-52
Price vs. cost（价格与成本），32~33
Price-performance ratio（性价比）
cost trends（成本趋势），28
Dell PowerEdge servers（Dell PowerEdge服务器），**53**
desktop comptuers（台式计算机），6
processor comparisons（处理器对比），55
WSCs，8，441
Primitives（原语）
architect-compiler writer relationship（架构师-编译器编写人员关系），A-30
basic hardware types（基本硬件类型），387~389
compiler writing-architecture relationship（编译器编写人员-架构师关系），A-30
CUDA Thread（CUDA线程），289
dependent computation elimination（消除相关计算），321
GPU vs. MIMD（GPU与MIMD），329
locks via coherence（通过一致性锁定），391
operand types and sizes（操作数类型与大小），A-14~A-15
synchronization（同步），394
Principle of locality（局域性原理），45，B-2
Private data（私有数据）
cache protocols（缓存协议），359
centralized shared-memory multiprocessors（集中共享存储器多处理器），351~352
Private Memory（专用存储器），292，314

Private variables, NVIDIA GPU Memory(私有变量，NVIDIA GPU存储器)，304
Procedure calls（过程调用）
compiler structure（编译器结构），A-25~A-26
control flow instructions（控制流指令），**A-17**，A-19~A-21
dependence analysis（相关性分析），321
high-level instruction set（高级指令集），A-42~A-43
invocation options（调用选项），A-19
ISAs，14
MIPS control flow instructions（MIPS控制流指令），A-38
return address predictors（返回地址预测器），206
VAX，**B-73~B-74**
VAX swap（VAX交换），B-74~B-75
Process concept（进程概念），160，B-49
Process-identifier（PID）tags, cache addressing（进程识别符标签，缓存寻址），B-37~B-38
Process IDs, Virtual Machines（进程ID，虚拟机），110
Processor consistency（处理器一致性）
latency hiding with speculation（用推测隐藏延迟），396~397
relaxed consistency models（宽松连贯性模型），395
Processor cycles（处理器周期），C-3
Processor-dependent optimizations（处理器相关优化）
compilers（编译器），A-26
performance impact（性能影响），A-27
types（类型），**A-28**
Processor-intensive benchmarks, desktop performance（处理器密集基准测试，桌台性能），38
Processor performance（处理器性能）
average memory access time（存储器平均访问时间），B-17~B-20
cache performance（缓存性能），B-16
clock rate trends（时钟频率趋势），**24**
desktop benchmarks（桌面基准测试），38，40
historical trends（历史趋势），3，3~4
multiprocessors（多处理器），347
uniprocessors（单处理器），344
Processor performance equation, computer design principles（处理器性能公式，计算机设计原理），48~52
Processor speed（处理器速度）
clock rate（时钟频率）
CPI，244
snooping cache coherence（监听缓存一致性），364
Process switch（进程开关），106，B-49
Productivity（生产能力）
CUDA，290~291
NVIDIA programmers（VNIDIA程序员），289
software development（软件开发），4
virtual memory and programming（虚拟存储器与编程），B-41
WSC，450
Profile-based predictor, misprediction rate（基于一览数据的预测器，错误预测率），C-27
Program counter（PC）（程序计数器）
addressing modes（寻址方式），A-10
ARM Cortex-A8，**234**
branch hazards（分支冒险），C-21
branch-target buffers（分支目标缓冲区），**203**，203~204，206
control flow instruction addressing modes（控制流指令寻址方式），A-17
dynamic branch prediction(动态分支预测)，C-27~C-28
exception stopping/restarting(异常停止/重启)，C-46~C-47
GPU conditional branching（GPU条件分支），303
Intel Core i7，120
MIPS control flow instructions（MIPS控制流指令），A-38
multithreading（多线程），223~224
pipeline branch issues（流水线分支问题），C-39~C-41
pipe stages（流水级），**C-35**
precise exceptions（精确异常），C-59~C-60
RISC classic pipeline（RISC经典流水线），C-8
RISC instruction set（RISC指令集），C-5
simple MIPS implementation（简单 MIPS 实现），C-31~C-33
TLP，344
virtual memory protection（虚拟存储器保护），106
Program counter-relative addressing（程序计算机相对寻址），A-10
Programming primitive, CUDA Thread（编程原语，CUDA线程），289
Program order（程序顺序）
cache coherence（缓存一致性），353
control dependences（控制相关性），154~155
data hazards（数据冒险），153
dynamic scheduling（动态调度），168~169，174
hardware-based speculation（基于硬件的推测），192
ILP exploitation（ILP开发），200
name dependences（名称相关），152~153
Tomasulo's approach（Tomasulo方法），182
Protection schemes（保护机制）
control dependence（控制相关性），155
ISA，112
Pentium vs. Opteron（Pentium与Opteron），B-57
processes（进程），B-50
safe calls（安全调用），B-54

segmented virtual memory example（分段虚拟存储器示例），B-51~B-54
Virtual Machines（虚拟机），107~108
virtual memory（虚拟存储器），105~107，B-41
Pseudo-least recently used（LRU）（伪最近使用(LRU)）
block replacement（块放置），B-9~B-10
Intel Core i7，118
PSUs,见 Power Supply Units
PTX, 见 Parallel Thread Execution
PUE, 见 Power utilization effectiveness
Python language, hardware impact on software development（Python 语言，硬件对软件开发的影响），4

Q

Quickpath（Intel Xeon），cache coherence（Intel Xeon），Quickpath，缓存一致性，361

R

Race-to-halt, definition（竞相暂停，定义），26
Rack units（U），WSC architecture（机架单元，WSC 体系结构），441
RAID（Redundant array of inexpensive disks）（RAID(廉价磁盘冗余阵列)）
data replication（数据复制），439
memory dependability（存储器可靠性），104
WSC storage（WSC 存储），442
Random replacement（随机替换），B-9
RAR, 见 Read after read
RAS, 见 Row access strobe
RAW, 见 Read after write
Ray casting（RC）（射线造型）
GPU comparisons（GPU 比较），329
throughput computing kernel（吞吐量计算内核），327
RDMA,见 Remote direct memory access
Read after read（RAR），absence of data hazard（读后读，没有数据冒险），154
Read after write（RAW）（写后读）
data hazards（数据冒险），153
dynamic scheduling with Tomasulo's algorithm（用 Tomasulo 算法进行动态调度），170~171
hazards, stalls（冒险、停顿），C-55
hazards and forwarding（冒险与转发），C-55~C-57
instruction set complications（指令集复杂性），C-50
microarchitectural techniques case study（微体系结构技术案例研究），253
MIPS FP pipeline performance（MIPS 浮点流水线性能），C-60~C-61
MIPS pipeline control（MIPS 流水线控制），C-37~C-38
MIPS pipeline FP operations（MIPS 流水线浮点运算），C-53
MIPS scoreboarding（MIPS 记分卡），C-74
ROB，192
Tomasulo's algorithm（Tomasulo 算法），182
unoptimized code（非优化代码），C-81
Read miss（读取缺失）
AMD Opteron data cache（AMD Opteron 数据缓存），B-14
cache coherence（缓存一致性），357，358，359~361
coherence extensions（一致性扩展），362
directory-based cache coherence（目录式缓存一致性）
protocol example（协议示例），380，382~386
memory hierarchy basics(存储器层次结构基础)，76~77
memory stall clock cycles（存储器停顿时钟周期），B-4
miss penalty reduction（缩小缺失代价），B-35~B-36
Opteron data cache（Opteron 数据缓存），B-14
write-through（直写），B-11
Read operands stage（读操作数级）
ID pipe stage（ID 流水级），170
MIPS scoreboarding（MIPS 记分卡），C-74~C-75
out-of-order execution（乱序执行），C-71
Realizable processors, ILP limitations（可实现处理器，ILP 局限性），216~220
Real memory, Virtual Machines（实际存储器，虚拟机），110
Real-time performance, PMDs（实时性能，PMD），6
RECN, 见 Regional explicit congestion notification
Reduced Instruction Set Computer, 见 RISC（Reduced Instruction Set Computer）
Reductions（降低）
commercial workloads（商业工作负载），371
cost trends（成本趋势），28
loop-level parallelism dependences(循环级并行相关)，321
multiprogramming workloads（多重编程工作负载），377
T1 multithreading unicore performance（T1 多线程单核性能），227
WSCs，438
Redundancy（冗余）
Amdahl's law（Amdahl 定律），48
chip fabrication cost case study（芯片制造成本案例研究），61~62
computer system power consumption case study（计算机系统功耗案例研究），63~64
index checks（索引检查），B-8
integrated circuit cost（集成电路成本），32
integrated circuit failure（集成电路故障），35

simple MIPS implementation（简单 MIPS 实现），C-33
WSC，433，435，439
WSC bottleneck（WSC 瓶颈），461
WSC storage（WSC 存储），442
Redundant power supplies, example calculations（冗余电源，示例计算），35
Reference bit（引用位）
memory hierarchy(存储器层次结构），B-52
virtual memory block replacement（虚拟存储器块放置），B-54
Register addressing mode（寄存器寻址方式）
MIPS，12
Register allocation（寄存器分配）
compilers（编译器），396，A-26~A-29
Register definition（寄存器定义），**314**
Register fetch（RF）（寄存器提取）
MIPS data path（MIPS 数据路径），C-34
MIPS R4000，C-63
pipeline branches（流水线分支），C-41
simple MIPS implementation（简单 MIPS 实现），C-31
simple RISC implementation（简单 RISC 实现），C-5~C-6
Register file（寄存器堆）
data hazards（数据冒险），C-16，C-18，C-20
dynamic scheduling（动态调度），172，**173**，175，177~178
Fermi GPU，306
field（字段），176
hardware-based speculation（基于硬件的推测），184
longer latency pipelines（长延迟流水线），C-55~C-57
MIPS exceptions（MIPS 异常），C-49
MIPS implementation（MIPS 实现），C-31，C-33
MIPS R4000（MIPS R4000），**C-64**
MIPS scoreboarding（MIPS 记分卡），C-75
Multimedia SIMD Extensions（多媒体 SIMD 扩展），282，285
multiple lanes（多车道），272，**273**
multithreading（多线程），224
precise exceptions（精确异常），C-59
RISC classic pipeline（RISC 经典流水线），C-7~C-8
RISC instruction set（RISC 指令集），C-5~C-6
scoreboarding（记分卡），C-73，C-75
speculation support（推测支持），208
structural hazards（结构性冒险），C-13
Tomasulo's algorithm（Tomasulo 算法），**180**，182
vector architecture（向量体系结构），264
VMIPS，265，308
Register-memory instruction set architecture（寄存器-存储器指令集体系结构）
architect-compiler writer relationship（架构师编译器编写人员关系），A-30
dynamic scheduling（动态调度），171
ISA classification（ISA 分类），11，A-3~A-6
Register prefetch, cache optimization（寄存器预取，缓存优化），92
Register renaming（寄存器重命名）
dynamic scheduling（动态调度），169~172
hardware vs. software speculation（硬件与软件推测），222
ideal processor（理想处理器），214
ILP hardware model（ILP 硬件模型），214
ILP limitations（ILP 局限性），213，216~217
ILP for realizable processors（可实现处理器的 ILP），216
instruction delivery and speculation（指令提交与推测），202
microarchitectural techniques case study（微体系结构技术案例研究），247~254
name dependences（名称相关性），153
ROB，208~210
ROB instruction（ROB 指令），186
sample code（示例代码），**250**
SMT，225
speculation（推测），208~210
superscalar code（超标量代码），**251**
Tomasulo's algorithm（Tomasulo 算法），183
WAW/WAR hazards（WAW/WAR 冒险），220
Register result status, MIPS scoreboard（寄存器结果状态，MIPS 记分卡），C-76
Registers（寄存器）
instructions and hazards（指令与冒险），**C-17**
pipe stages（流水级），**C-35**
VAX swap（VAX 交换），B-74~B-75
Register tag example（寄存器标签示例），177
Regularity（规则）
compiler writing-architecture relationship（编译器编写与体系结构的关系），A-30
Relative speedup, multiprocessor performance（相对加速比，多处理器性能），406
Relaxed consistency models（宽松连贯性模型）
basic considerations（基本考虑事项），394~395
compiler optimization（编译器优化），396
WSC storage software（WSC 存储软件），439
Release consistency, relaxed consistency models（释放连贯性，宽松连贯性模型），395
Reliability（可靠性）
Amdahl's law calculations（Amdahl 定律计算），56
example calculations（示例计算），48
modules, SLAs（模块，SLA），34

MTTF，57
redundant power supplies（冗余电源），34~35
transistor scaling（晶体管缩放），21
Relocation, virtual memory（重定位，虚拟存储器），B-42
Remote node, directory-based cache coherence protocol basics（远程节点，目录式一致性协议基础），381~382
Reorder buffer（ROB）（重排序缓冲区）
dependent instructions（相关指令），**199**
dynamic scheduling（动态调度），175
FP unit with Tomasulo's algorithm（采用 Tomasulo 算法的浮点单元），**185**
hardware-based speculation（基于硬件的推测），184~192
ILP exploitation（ILP 开发），199~200
ILP limitations（ILP 局限性），216
Intel Core i7，238
register renaming（寄存器重命名），208~210
Repeat interval, MIPS pipeline FP operations（重复间隔，MIPS 流水线浮点运算），C-52~C-53
Replication（复制）
cache coherent multiprocessors（缓存一致性多处理器），354
centralized shared-memory architectures（集中式共享存储器体系结构），351~352
coherence enforcement（一致性实施），354
R4000 performance（R4000 性能），C-70
RAID storage servers（RAID 存储服务器），439
TLP，344
virtual memory（虚拟存储器），B-48~B-49
WSCs，438
Reproducibility, performance results reporting（再生能力，性能结果报告），41
Requested protection level, segmented virtual memory（所请求的保护级别，分段虚拟存储器），B-54
Request-level parallelism（RLP）（请求级并行），9
Reservation stations（保留站）
dependent instructions（相关指令），199~200
dynamic scheduling（动态调度），178
example（示例），177
fields（字段），176
hardware-based speculation（基于硬件的推测），184，186，189~191
ILP exploitation（ILP 开发），197，199~200
Intel Core i7，238~240
loopiteration example（迭代示例），181
microarchitectural techniques case study（微体系结构技术案例研究），253~254
speculation（推测），208~209
Tomasulo's algorithm（Tomasulo 算法），172，**173**，174~176，179，**180**，180~182
Resource allocation（资源分配）
computer design principles（计算机设计原理），45
WSC case study（WSC 案例研究），478~479
Response time，另见 Latency
performance considerations（性能考虑事项），36
performance trends（性能趋势），18~19
server benchmarks（服务器基准测试），40~41
user experience（用户体验），4
WSCs，450
Responsiveness（响应），C-45
Restorations, SLA states（恢复，SLA 状态），34
Resume events（恢复事件）
control dependences（控制相关性），156
exceptions（异常），C-45~C-46
hardware-based speculation（基于硬件的推测），188
Return address predictors（返回地址预测器）
instruction fetch bandwidth（指令提取带宽），206~207
prediction accuracy（预测精度），**207**
Returns（返回）
Amdahl's law（Amdahl 定律），47
cache coherence（缓存一致性），352~353
compiler technology and architectural decisions（编译器技术与体系结构决策），A-28
control flow instructions（控制流指令），14，A~17，A-21
hardware primitives（硬件原语），388
invocation options（调用选项），A-19
procedure invocation options（过程调用选项），A-19
return address predictors（返回地址预测器），206
RF，见 Register fetch
Rings（环）
process protection（进程保护），B-50
RISC（Reduced Instruction Set Computer）（精简指令集计算机）
architecture flaws vs. success（体系结构缺陷与成功），A-45
basic concept（基本概念），C-4~C-5
cache performance（缓存性能），B-6
classic pipeline stages（经典流水级），C-6~C-10
code size（代码大小），A-23~A-24
development（发展），2
ISA performance and efficiency prediction（ISA 性能与效率预测），241
ROB，见 Reorder buffer
Roofline model（屋顶轮廓线模型）
GPU performance（GPU 性能），**326**
memory bandwidth（存储器带宽），332
Multimedia SIMD Extensions（多媒体 SIMD 扩展），285~288，**287**
Row access strobe（RAS），DRAM（行访问选通），98

Row major order, blocking（行主序，分块），89
Ruby on Rails, hardware impact on software development（Ruby on Rails，硬件对软件开发的影响），4

S

S3, 见 Amazon Simple Storage Service
SaaS, 见 Software as a Service
Sandy Bridge dies, wafter example（Sandy Bridge 晶片，晶圆示例），**31**
SATA（Serial Advanced Technology Attachment）disks（串行高级技术连接）磁盘）
  Google WSC servers（Google WSC 服务器），469
SAXPY, GPU raw/relative performance（SAXPY，GPU 原始/相对性能），**328**
Scalability（可扩展性）
  cloud computing（云计算），460
  coherence issues（一致性问题），378~379
  Fermi GPU，295
  Java benchmarks（Java 基准测试），402
  multicore processors（多核处理器），400
  multiprocessing（多重处理），344，395
  parallelism（并行度），44
  server characteristic（服务器特性），7
  transistor performance and wires（晶体管性能与连线），19~21
  WSCs，8，438
  WSCs vs. servers（WSC 与服务器），434
Scalar expansion, loop-level parallelism dependences（标量扩展，循环级并行相关性），321
Scalar Processors, 另见 Superscalar processors
Scalar registers（标量寄存器）
  GPUs vs. vector architectures（GPU 与向量体系结构），311
  loop-level parallelism dependences（循环级并行度相关性），321~322
  Multimedia SIMD vs. GPUs（多媒体 SIMD 与 GPU），312
  sample renaming code（示例重命名代码），251
  vector vs. GPU（向量与 GPU），311
  vector performance（向量性能），331~332
  VMIPS，265~266
Scaled speedup, Amdahl's law and parallel computers（缩放加速，Amdahl 定律与并行计算机），406~407
Scaling（缩放）
  Amdahl's law and parallel computers（Amdahl 定律和并行计算机），406 ~ 407
  cloud computing（云计算），456
  DVFS，25，52，467
  dynamic voltage-frequency（动态电压-频率），25，52，467
  Intel Core i7，404
  multicore vs. single-core（多核与单核），402
  processor performance trends（处理器性能趋势），3
  transistor performance and wires（晶体管性能与连线），19~21
  VMIPS，267
SCCC, 见 Intel Single-Chip Cloud Computing
Scoreboarding（记分卡）
  ARM Cortex-A8，233，**234**
  components（分量），**C-76**
  definition（定义），170
  dynamic scheduling（动态调度），171，175
  example calculations（示例计算），C-77
  MIPS structure（MIPS 结构），**C-73**
  NVIDIA GPU，296
  results tables（结果表），**C-78~C-79**
  SIMD thread scheduler（SIMD 线程调度程序），296
Scripting languages, software development impact（脚本语言，软件开发影响），4
SDRAM，见 Synchronous dynamic random-access memory
Second-level caches，另见 L2 caches
Segment descriptor, IA-32 processor（分段描述符，IA-32 处理器），B-52，**B-53**
Segmented virtual memory（分段虚拟存储器）
  bounds checking（界限检查），B-52
  Intel Pentium protection（IntelPentium 保护），B-51~B-54
  memory mapping（存储器映射），B-52
  paged（分页），**B-43**
  safe calls（安全调用），B-54
  sharing and protection（共享与保护），B-52~B-53
Semantic clash, high-level instruction set（语义冲突，高级指令集），A-41
Semantic gap, high-level instruction set（语义鸿沟，高级指令集），A-39
Semiconductors（半导体）
  DRAM technology（DRAM 技术），17
  Flash memory（闪存），18
  GPU vs. MIMD（GPU 与 MIMD），325
  manufacturing（制造），3~4
Sequence of SIMD Lane Operations, definition（SIMD 车道操作序列，定义），**292**，**313**
Sequential consistency（顺序一致性）
  latency hiding with speculation（通过推测隐藏延迟），396~397
  programmer's viewpoint（程序员的观点），394
  relaxed consistency models（宽松连贯性模型），

394~395
requirements and implementation（需求与实现），392~393
Sequential interleaving, multibanked caches（顺序交错，多组缓存），86
Serial Advanced Technology Attachment disks, 见 SATA（Serial Advanced Technology Attachment）disks
Serialization barrier synchronization（序列化屏障同步）
coherence enforcement（一致性实施），354
directory-based cache coherence（目录式缓存一致性），382
hardware primitives（硬件原语），387
multiprocessor cache coherency（多处理器缓存一致性），353
page tables（页表），408
snooping coherence protocols（监听一致性协议），356
write invalidate protocol implementation（写失效协议实现），356
Servers,另见 Warehouse-scale computers（WSCs）
Server side Java operations per second（ssj_ops）（每秒执行的服务器端 Java 运算（ssj_ops））
example calculations（示例计算），439
power-performance（功率-性能），**54**
real-world considerations（现实考虑事项），52~55
Service accomplishment, SLAs（服务完成，SLA），34
Service Health Dashboard, AWS（服务健康仪表板，AWS），457
Service interruption, SLAs（服务中断，SLA），34
Service level agreements（SLAs）（服务级别协议）
Amazon Web Services（Amazon Web 服务），457
dependability（可靠性），33
WSC efficiency（WSC 效率），452
Service level objectives（SLOs）（服务级目标）
dependability（可靠性），33
WSC efficiency（WSC 效率），452
Set associativity（集相联度）
access time（访问时间），77
address parts（地址部分），**B-9**
AMD Opteron data cache（AMD Operon 数据缓存），B-12~B-14
ARM Cortex-A8，114
block placement（块放置），B-7~B-8
cache block（缓存块），B-7
cache misses（缓存缺失），83~84，**B-10**
cache optimization（缓存优化），79~80，B-33~B-35，B-38~B-40
commercial workload（商业工作负载），**371**
energy consumption（能耗），**81**
memory access times（存储器访问时间），77
memory hierarchy basics（存储器层次结构基础），74，76
nonblocking cache（非阻塞缓存），84
performance equations（性能公式），**B-22**
pipelined cache access（流水化缓存访问），82
way prediction（路预测），81
Set basics（集基础），B-7
SFF，见 Small form factor SFF disk
SGI，见 Silicon Graphics systems
Shadow page table, Virtual Machines（影子页表），110
Sharding, WSC memory hierarchy（分片，WSC 存储器层次结构），445
Shared Memory（共享存储器），292，314
Shared-memory multiprocessors（共享存储器多处理器），346-351
Shared state（共享状态）
cache block（缓存块），357，359
cache coherence（缓存一致性），**360**
cache miss calculations（缓存缺失计算），366~367
coherence extensions（一致性扩展），362
directory-based cache coherence protocol basics（目标式缓存一致性协议基础），380，385
private cache（专用缓存），358
Sharing addition, segmented virtual memory（共享加法，分段虚拟存储器），B-52~B-53
Short-circuiting,见 Forwarding
Sign-extended offset, RISC（符号扩展偏移量，RISC），C-4~C-5
Silicon Graphics systems（SGI）（硅图系统）
economies of scale（规模经济），456
miss statistics（缺失统计数字），**B-59**
multiprocessor software development（多处理器软件开发），407~409
SIMD（Single Instruction Stream, Multiple Data Stream）（单指令流，多数据流），10
SIMD Instruction（SIMD 指令），292，313
SIMD Lane Registers, definition（SIMD 车道寄存器，定义），309，314
SIMD Lanes（SIMD 车道），**292，296，309**
SIMD Processors, 另见 Multithreaded SIMD Processor
SIMD Thread（SIMD 线程）
GPU conditional branching（GPU 条件分支），301~302
Grid mapping（网络映射），**293**
Multithreaded SIMD processor（多线程 SIMD 处理器），294
NVIDIA GPU，296
NVIDIA GPU ISA，298
NVIDIA GPU Memory structures（NVIDIA GPU 存储器结构），305

scheduling example（调度示例），**297**
vector vs. GPU（向量与 GPU），308
vector processor（向量处理器），310
SIMD Thread Scheduler（SIMD 线程调度程序），292，314
SIMT（Single Instruction, Multiple Thread）（单指令，多线程）
GPU programming（GPU 编程），289
SIMD，314
Warp（交换），**313**
Simultaneous multithreading（SMT）（同时多线程），224~225
Single Instruction, Multiple Thread,见 SIMT（Single Instruction, Multiple Thread）
Single Instruction Stream, Single Data Stream,见 SISD（Single Instruction Stream, Single Data Stream）
Single-level cache hierarchy, miss rates vs. cache size（单级缓存层次结构，缺失率与缓存大小），B-33
Single-precision floating point（单精度浮点）
GPU examples（GPU 示例），325
GPU vs. MIMD（GPU 与 MIMD），328
MIPS data types（MIPS 数据类型），A-34
MIPS operations（MIPS 操作），A-36
Multimedia SIMD Extensions（多媒体 SIMD 扩展），283
operand sizes/types（操作数大小/类型），12，A-13
operand type（操作数类型），A-13~A-14
Single-thread（ST）performance（单线程性能）
IBM eServerP5575，399，399
Intel Core i7，239
ISA，242
processor comparison（处理器对比），**243**
SLAs，见 Service level agreements
SLOs，见 Service level objectives
SM,见 Distributed shared memory（DSM）
Small Computer System Interface,见 SCSI
Smartphones（智能手机）
ARM Cortex-A8（ARM Cortex-A8），114
mobile vs. server GPUs（移动与服务器 GPU），323~324
SMP,见 Symmetric multiprocessors
SMT,见 Simultaneous multithreading
Snooping cache coherence（监听缓存一致性），354~355
Soft errors, definition（软错误，定义），104
Software as a Service（SaaS）（软件即服务）
clusters/WSCs（集群/WSC），8
software development（软件开发），4
WSCs，438
WSCs vs. servers（WSC 与服务器），433~434
Software development multiprocessor architecture issues（软件开发多处理器体系结构问题），407~409
Software prefetching, cache optimization（软件预取，缓存优化），131~133
Software speculation（软件推测），156
Software technology（软件技术）
ILP approaches（ILP 方法），148
Virtual Machines protection（虚拟机保护），108
WSC running service（WSC 运行服务），434~435
Solid-state disks（SSDs）（固态硬盘）
processor performance/price/power（处理器性能/价格/功率），52
server energy efficiency（服务器能量效率），462
WSC cost-performance（WSC 成本-性能），474~475
Sort primitive, GPU vs. MIMD（排序原语，GPU 与 MIMD），329
Sparse matrices（稀疏矩阵）
loop-level parallelism dependences（循环级并行相关性），318~319
vector architectures（向量体系结构），279~280
vector execution time（向量执行时间），271
vector mask registers（向量遮罩寄存器），275
Spatial locality（空间局域性），45，B-2
SPEC benchmarks（SPEC 基准测试）
branch predictor correlation（分支预测器相关），162~164
desktop performance（桌面性能），38~40
evolution（演进），**39**
fallacies（谬论），56
operands（操作数），A-14
performance（性能），38
performance results reporting（性能结果报告），41
processor performance growth（处理器性能增长），3
static branch prediction（静态分支预测），C-26~C-27
tournament predictors（竞赛预测器），164
two-bit predictors（两位预测器），**165**
SPEC89 benchmarks（SPEC89 基准测试）
branch-prediction buffers（分支预测缓冲区），C-28~C-30，**C-30**
MIPS FP pipeline performance（MIPS 浮点流水线性能），**C-61~C-62**
misprediction rates（错误预测率），**166**
tournament predictors（竞赛预测器），165~166
SPEC92 benchmarks（SPEC92 基准测试）
hardware vs. software speculation（硬件推测与软件推测），221
ILP hardware model（ILP 硬件模型），**215**
MIPS R4000 performance（MIPS R4000 性能），C-68~C-69，**C-69**
misprediction rate（错误预测率），**C-27**

SPEC95 benchmarks（SPEC95基准测试）
return address predictors（返回地址预测器），206~207，**207**
way prediction（路预测），82
SPEC2000 benchmarks（SPEC2000基准测试）
ARM Cortex-A8 memory（ARM Cortex-A8存储器），115~116
cache performance prediction（缓存性能预测），125~126
cache size and misses per instruction（缓存大小与每条指令的缺失率），**126**
compiler optimizations（编译器优化），**A-29**
compulsory miss rate（强制缺失率），B-23
data reference sizes（数据引用大小），**A-44**
hardware prefetching（硬件预取），91
instruction misses（指令缺失），**127**
SPEC2006 benchmarks（SPEC2006基准测试）
evolution（演进），**39**
SPECCPU2000 benchmarks（SPECCPU2000基准测试）
displacement addressing mode（放置寻址方式），**A-12**
Intel Core i7，122
server benchmarks（服务器基准测试），40
SPECCPU2006 benchmarks（SPECCPU2006基准测试）
branch predictors（分支预测器），**167**
Intel Core i7，**123~124**，**240**，240~241
ISA performance and efficiency prediction（ISA性能与效率预测），241
Virtual Machines protection（虚拟机保护），108
SPECfp benchmarks（SPECfp基准测试）
hardware prefetching（硬件预取），91
ISA performance and efficiency prediction（ISA性能和效率预测），241~242
MIPS FP pipeline performance（MIPS浮点流水线性能），C-60~C-61
nonblocking caches（非阻塞缓存），**84**
tournament predictors（竞赛预测器），164
SPECfp92 benchmarks（SPECfp92基准测试）
nonblocking cache（非阻塞缓存），83
SPECfp2000 benchmarks（SPECfp2000基准测试）
hardware prefetching（硬件预取），**92**
MIPS dynamic instruction mix（MIPS动态指令混合比），**A-42**
Sun Ultra 5 execution times（Sun Ultra 5执行时间），**43**
SPECfp2006 benchmarks（SPECfp2006基准测试）
Intel processor clock rates（Intel处理器时钟频率），**244**
nonblocking cache（非阻塞缓存），83
SPECfpRate benchmarks（SPECfpRate基准测试）
multicore processor performance（多核处理器性能），400
multiprocessor cost effectiveness（多处理器成本效率），407
SMT，398~400
SMT on superscalar processors（超标量处理器上的SMT），230
Special-purpose register（专用寄存器）
compiler writing-architecture relationship（编译器编写-体系结构关系），A-30
ISA classification（ISA分类），A-3
VMIPS，267
SPECINT benchmarks（SPECINT基准测试）
hardware prefetching（硬件预取），**92**
ISA performance and efficiency prediction（ISA性能与效率预测），241~242
nonblocking caches（非阻塞缓存），**84**
nonblocking cache（非阻塞缓存），83
SPECINT2000 benchmarks, MIPS dynamic instruction mix（SPECINT2000基准测试，MIPS动态指令混合比），**A-41**
SPECINT2006 benchmarks（SPECINT2006基准测试）
Intel processor clock rates（Intel处理器时钟频率），**244**
nonblocking cache（非阻塞缓存），83
SPECintRate benchmark（SPECintRate基准测试）
multicore processor performance（多核处理器性能），400
multiprocessor cost effectiveness（多处理器成本效率），407
SMT，398~400
SMT on superscalar processors（超标量处理器上的SMT），230
SPEC Java Business Benchmark（JBB）（SPEC Java业务基准测试）
multicore processor performance（多核处理器性能），400
multicore processors（多核处理器），**402**
multiprocessing/ multithreading-based performance（基于多重处理/多线程的性能），398
server（服务器），40
Sun T1 multithreading unicore performance（Sun T1多线程单核性能），227~229，**229**
SPECJVM98 benchmarks, ISA performance and efficiency prediction（SPECJVM98基准测试，ISA性能与效率预测），241
SPECPower benchmarks（SPECPower基准测试）
Google server benchmarks（Google服务器基准测试），439~440，**440**
multicore processor performance（多核处理器性能），400

real-world server considerations（现实服务器考虑事项），52~55
WSCs，**463**
WSC server energy efficiency（WSC 服务器能量效率），462~463
SPECRate benchmarks （SPECRate 基准测试）
Intel Core i7，**402**
multicore processor performance （多核处理器性能），400
multiprocessor cost effectiveness（多处理器成本效率），407
server benchmarks （服务器基准测试），40
SPECRate2000 benchmarks （SPECRate2000 基准测试），398~400
SPECRatios
execution time examples （执行时间示例），**43**
geometric means calculations（几何均值计算），43~44
SPECSFS benchmarks （SPECSFS 基准测试）
servers（服务器），40
Speculation，另见 Hardware-based speculation； Software speculation SPECvirt_Sc2010 benchmarks, server （SPECvirt_Sc2010 基准测试，服务器），40
SPECWeb benchmarks （SPECWeb 基准测试）
parallelism （并行度），44
server benchmarks （服务器基准测试），40
SPECWeb99 benchmarks （SPECWeb99 基准测试）
multiprocessing/ multithreading-based performance（基于多重处理/多线程的性能），398
Sun T1 multithreading unicore performance （Sun T1 多线程单核性能），227，**229**
Speedup （加速比）
Amdahl's law （Amdahl 定律），46~47
linear （线性），405~407
via parallelism （通过并行度），**263**
pipeline with stalls（具有停顿的流水线），C-12~C-13
relative （相对），406
scaled （缩放），406~407
true （真），406
Spin locks （自旋锁）
via coherence （通过一致性），389~390
SPLASH parallel benchmarks, SMT on superscalar processors （SPLASH 并行基准测试，超标量处理器上的 SMT），230
Split, GPU vs. MIMD （分割，GPU 与 MIMD），329
SRAM,见 Static random-access memory （SRAM）
SSDs, 见 Solid-state disks
SSE, 见 Intel Streaming SIMD Extension
ssj_ops, 见 Server side Java operations per second
Stack architecture （栈体系结构）
compiler technology （编译器技术），A-27
flaws vs. success （缺点与成功），A-44~A-45
operands （操作数），A-3~A-4
Stack or Thread Local Storage, definition（栈或线程本地存储，定义），**292**
Stale copy, cache coherency（旧副本，缓存一致性），112
Stall cycles （停顿周期），B-4~B-5
Stalls （停顿）
AMD Opteron data cache （AMD Opteron 数据缓存），B-15
ARM Cortex-A8，**235**，235~236
branch hazards （分支冒险），**C-42**
data hazard minimization （数据冒险最小化），C-16~C-19，**C-18**
data hazards requiring （数据冒险需求），C-19~C-21
delayed branch （延迟分支），**C-65**
Intel Core i7，239~241
microarchitectural techniques case study（微体系结构技术案例研究），252
MIPS FP pipeline performance （MIPS 浮点流水线性能），C-60~C-61，**C-61~C-62**
MIPS pipeline multicycle operations（MIPS 流水线多周期操作），C-51
MIPS R4000，**C-64**，**C-67**，C-67~C-69，C-69
miss rate calculations （缺失率计算），B-31~B-32
necessity （必要性），**C-21**
nonblocking cache （非阻塞缓存），84
pipeline performance （流水线性能），C-12~C-13
from RAW hazards, FP code（自 RAW 冒险，浮点代码），**C-55**
structural hazard （结构性冒险），**C-15**
VLIW sample code （VLIW 示例代码），**252**
VMIPS，268
Stardent-1500, Livermore Fortran kernels （Stardent-1500，Livermore Fortran 内核），**331**
Start-up overhead, vs. peak performance （启动开销，与峰值性能），331
Start-up time （启动时间）
memory banks （存储器组），276
page size selection （页大小选择），B-47
peak performance （峰值性能），331
vector architectures （向量体系结构），331
vector execution time （向量执行时间），270~271
State transition diagram （状态转换图）
director vs. cache （目录与缓存），**385**
directory-based cache coherence（目录式缓存一致性），**383**
Static power （静态功率）

basic equation（静态公式），26
SMT，231
Static random-access memory（SRAM）（静态随机访问存储器（SRAM））
characteristics（特性），97~98
dependability（可靠性），104
fault detection pitfalls（错误检测易犯错误），58
power（功率），26
yield（正品率），32
Static scheduling（静态调度），C-71
Storage systems（存储系统）
WSC vs. datacenter costs（WSC与数据中心成本），455
WSG，442~443
Store conditional（存储条件）
locks via coherence（通过一致性锁定），391
synchronization（同步），388~389
Store instructions，另见 Load-store instruction set architecture
Streaming Multiprocessor（流式多处理器）**292，313~314**
Strided accesses（步幅访问）
Multimedia SIMD Extensions（多媒体 SIMD 扩展），283
Roofline model（屋顶轮廓线模型），**287**
TLB interaction（TLB 交互），323
trides（步幅）
gather-scatter（集中－分散），280
highly parallel memory systems（高度并行存储器系统），133
multidimensional arrays in vector architectures（向量体系结构中的多维数组），278~279
NVIDIA GPU ISA，300
VMIPS，**266**
Strip mining（条带挖掘）
GPU conditional branching（GPU 条件分支），303
GPUs vs. vector architectures（GPU 与向量体系结构），311
NVIDIA GPU，291
vector（向量），**275**
VLRs，274~275
Strong scaling, Amdahl's law and parallel computers（强扩展，Amdahl 定律和并行计算机），407
Structural hazards（结构冒险），C-11
Structural stalls, MIPS R4000 pipeline（结构性停顿，MIPS R4000 流水线），C-68~C-69
Subset property, and inclusion（子集特性与包含性），397
Sun Microsystems
cache optimization（缓存优化），B-38
fault detection pitfalls（错误检测易犯错误），58
memory dependability（存储器可靠性），104
Sun Microsystems Niagara（T1/T2）processors（Sun Microsystems Niagara（T1/T2）处理器）
characteristics（特性），**227**
CPI and IPC（CPI 和 IPC），**399**
fine-grained multithreading（细精度多线程），224，**225**，226~229
manufacturing cost（制造成本），**62**
multicore processor performance（多核处理器性能），400~401
multiprocessing/ multithreading-based performance（基于多重处理/多线程的性能），398~400
T1 multithreading unicore performance（T1 多线程单核性能），227~229
Sun Microsystems SPARC
ALU operands（ALU 操作数），**A-6**
branch conditions（分支条件），A-19
ISA，A-2
precise exceptions（精确异常），C-60
Sun Microsystems Ultra 5, SPECfp2000 execution times（Sun Microsystems Ultra 5，SPECfp2000 执行时间），**43**
Supercomputers（超级计算机）
WSCs，**8**
Superlinear performance, multiprocessors（超线性性能，多处理器），406
Superpipelining（超级流水线），C-61
Superscalar processors（超级处理器）
ideal processors（理想处理器），214~215
ILP，192~197，**246**
microarchitectural techniques case study（微体系结构技术案例研究），250~251
multithreading support（多线程支持），**225**
register renaming code（寄存器重命名代码），**251**
rename table and register substitution logic（重命名步及寄存器替换逻辑），**251**
SMT，230~232
VMIPS，267
Superscalar registers（超标量寄存器）
sample renaming code（示例重命名代码），251
Supervisor process, virtual memory protection（管理员进程，虚拟存储器保护），106
SVM，见 Secure Virtual Machine
Swap procedure, VAX（交换进程，VAX）
register preservation（寄存器保留），B-74~B-75
Swim, data cache misses（Swim，数据缓存缺失），B-10
Switches（交换机）
array, WSCs（阵列，WSC），443~444
context（上下文），307，B-49
Ethernet switches（以太网交换机），16，**20**，53，441~444，464~465，469

process switch（进程交换），224，B-37，B-49~B-50
WSC hierarchy（WSC层次结构），441~442，**442**
WSC infrastructure（WSC基础设施），446
WSC network bottleneck（WSC网络瓶颈），461
Switch statements（Switch语句）
control flow instruction addressing modes（控制流指令寻址方式），A-18
GPU，301
Symmetric multiprocessors（SMP）（对称多处理器（SMP））
communication calculations（通信计算），350
directory-based cache coherence（目录式缓存一致性），354
limitations（局限性），363~364
snooping coherence protocols（监听一致性协议），354~355
TLP，345
Symmetric shared-memory multiprocessors，另见 Centralized shared-memory multiprocessors
Synchronization（同步），375
Synchronous dynamic random-access memory（SDRAM）（同步动态随机访问存储器）
ARM Cortex-A8，117
DRAM，**99**
*vs.*Flash memory（闪存），103
Intel Core i7，121
performance（性能），100
power consumption（功率消耗），102，**103**
SDRAM timing diagram（SDRAM时序图），**139**
Synchronous event, exception requirements（同步事件，异常需求），C-44~C-45
Synonyms（同义词）
address translation（地址转换），B-38
dependability（可靠性），34
Synthetic benchmarks（合成基准测试），37
System calls（系统调用）
CUDA Thread（CUDA线程），297
multiprogrammed workload（多重编程工作负载），378
virtualization/paravirtualization performance（虚拟化/部分虚拟化性能），**141**
virtual memory protection（虚拟存储器保护），106
System Performance and Evaluation Cooperative，见 SPEC benchmarks
System Processor（系统处理器），309
Systems on a chip（SOC），cost trends（片上系统，成本趋势），28
System Virtual Machines, definition（系统虚拟机，定义），107

T

Tag（标签）
AMD Opteron data cache（AMD）Opteron数据缓存，B-12~B-14
ARM Cortex-A8，**115**
cache optimization（缓存优化），79~80
dynamic scheduling（动态调度），**177**
invalidate protocols（失效协议），357
memory hierarchy basics（存储器层次结构基础），74
virtual memory fast address translation（虚拟存储器快速地址转换），B-46
write strategy（写入策略），B-10
Tag check（TC）（标签核对）
MIPS R4000，C-63
R4000 pipeline（R4000流水线），B-62~B-63
R4000 pipeline structure（R4000流水线结构），C-63
write process（写入过程），B-10
Tag fields（标签字段）
block identification（块识别），B-8
dynamic scheduling（动态调度），**173**，175
Target address（目标地址）
branch hazards（分支冒险），C-21，**C-42**
branch penalty reduction（降低分支代价），C-22~C-23
branch-target buffer（分支目标缓冲区），206
control flow instructions（控制流指令），A-17~A-18
GPU conditional branching（GPU条件分支），301
Intel Core i7 branch predictor（Intel Core i7分支预测器），166
MIPS control flow instructions（MIPS控制流指令），A-38
MIPS implementation（MIPS实现），C-32
MIPS pipeline（MIPS流水线），C-36，**C-37**
MIPS R4000（MIPS R4000），C-25
pipeline branches（流水线分支），C-39
RISC instruction set（RISC指令集），C-5
Target instructions（目标指令）
branch delay slot scheduling（分支延迟时隙调度），C-24
branch-target buffer variation（分支目标缓冲区变化），206
GPUconditional branching（条件分支），301
Task-level parallelism（TLP）definition（任务级并行定义），9
TB，见 Translation buffer
TC，见 Tag check
TCO，见 Total Cost of Ownership
TDP，见 Thermal design power
Technology trends（技术趋势）

basic considerations（基本考虑事项）17~18
performance（性能）18~19
Temporal locality（时间局域性），45，B-2
Terminate events（终止事件）
exceptions（异常），C-45~C-46
hardware-based speculation（基于硬件的推测），188
loop unrolling（循环展开），161
Test-and-set operation, synchronization（“测试并置位”操作，同步），388
Texas Instruments ASC（德州仪器 ASC）
peak performance vs. start-up overhead（峰值性能与启动开销），331
Thermal design power（TDP），power trends（散热设计功率，功率趋势），22
Third-level caches,另见 L3 cachesILP，245
ILP，245
SRAM，98~99
Thrash, memory hierarchy（摆动，存储器层次结构），B-25
Thread Block（线程块），**292，313**
Thread Block Scheduler（线程块调度程序）**292，309，313~314**
Thread-level parallelism（TLP）（线程级并行），9
Thread Processor（线程处理器），**292，314**
Thread Processor Registers, definition（线程处理器寄存器，定义），**292**
Thread Scheduler in a Multithreaded CPU, definition（多线程 CPU 中的线程调度程序，定义），**292**
Thread of SIMD Instructions（SIMD 指令线程），**292，313**
Thread of Vector Instructions, definition（向量指令线程，定义），**292**
Three-level cache hierarchy（三级缓存层次结构）
commercial workloads（商业工作负载），368
ILP，245
Intel Core i7，118，**118**
Throughput，另见 Bandwidth
Ticks（嘀嗒）
cache coherence（缓存一致性），**391**
processor performance equation（处理器性能公式），48~49
Time-cost relationship, components（时间-成本关系，组件），27~28
TLB,见 Translation lookaside buffer
TLP,见 Task-level parallelism
Tomasulo's algorithm（Tomasulo 算法）
advantages（优势），177~178
dynamic scheduling（动态调度），170~176
FP unit（浮点单元），**185**
loop-based example（基于循环的示例），179,181~183
MIP FP unit（MIP 浮点单元），**173**
register renaming vs. ROB（寄存器重命名与 ROB），209
step details（步骤细节），178，**180**
Top Of Stack（TOS）register, ISA operands（栈顶寄存器，ISA 操作数），**A-4**
TOS，见 Top Of Stack register
Total Cost of Ownership（TCO），WSC case study（总拥有成本，WSC 案例研究），476~479
Total store ordering, relaxed consistency models（总存储排序，宽松连贯性模型），395
Tournament predictors（竞赛预测器）
ILP for realizable processors（可实现处理器的 ILP），216
local/global predictor combinations（局部/全局预测器组合），164~166
Toy programs, performance benchmarks（玩具程序，性能基准测试），37
TP,见 Transaction-processing
TPC, 见 Transaction Processing Council
Tradebeans benchmark, SMT on superscalar processors（Tradebens 基准测试，超标量处理器上的 SMT），230
Transaction-processing（TP）（事务处理）
server benchmarks（服务器基准测试），41
Transaction Processing Council（TPC）（事务处理委员会）
parallelism（并行度），44
performance results reporting（性能结果报告），41
server benchmarks（服务器基准测试），41
TPC-B, shared-memory workloads（共享存储器工作负载），368
TPC-C
IBM eServer p5 processor（IBM eServer p5 处理器），**409**
multiprocessing/ multithreading-based performance（基于多重处理/多线程的性能），398
multiprocessor cost effectiveness（多处理器成本效率），407
single vs. multiple thread executions（单线程与多线程执行），**228**
Sun T1 multithreading unicore performance（Sun T1 多线程单核性能），227~229，**229**
WSC services（WSC 服务），441
TPC-D, shared-memory workloads（共享存储器工作负载），368~369
TPC-E, shared-memory workloads（共享存储器工作负载），368~369
Transfers，另见 Data transfers
Transistors（晶体管）
clock rate considerations（时钟频率考虑事项），244
dependability（可靠性），33~36

energy and power（能量与功率），23~26
ILP，245
performance scaling（性能扩展），19~21
processor comparisons（处理器对比），324
processor trends（处理器趋势），2
RISC instructions（RISC 指令），A-3
shrinking（收缩），55
static power（静态功率），26
technology trends（技术趋势），17~18
Translation buffer （TB）（转换缓冲区）
virtual memory block identification（虚拟块识别），B-45
virtual memory fast address translation（虚拟存储器快速地址转换），B-46
Translation lookaside buffer （TLB）（转换旁视缓冲区）
address translation（地址转换），B-39
AMD64 paged virtual memory（AMD64 分页虚拟存储器），B-56~B-57
ARM Cortex-A8，114~115
cache optimization（缓存优化），80，B-37
Intel Core i7（Intel Core i7），**118**，120~121
memory hierarchy（存储器层次结构），B-48~B-49
memory hierarchy basics（存储器层次结构基础），78
Opteron，**B-47**
Opteron memory hierarchy（Opteron 存储器层次结构），**B-57**
RISC code size（RISC 代码大小），A-23
shared-memory workloads（共享存储器工作负载）369~370
speculation advantages/ disadvantages（推测优缺点）210~211
strided access interactions（步幅访问交互），323
Virtual Machines（虚拟机），110
virtual memory block identification（虚拟存储器块识别），B-45
virtual memory fast address translation（虚拟存储器快速地址转换），B-46
virtual memory page size selection（虚拟存储器页大小选择），B-47
virtual memory protection（虚拟存储器保护），106~107
Trojan horses（特洛伊木马），B-51
True dependence（真相关）
loop-level parallelism calculations（循环级并行计算），320
name dependence（名称相关），153
True sharing misses（真共享缺失），366~367
True speedup, multiprocessor performance（真实加速比，多处理器性能），406
Turbo mode（Turbo 模式）
hardware enhancements（硬件增强），56
microprocessors（微处理器），26
Two-level branch predictors（两级分支预测器）
branch costs（分支成本），163
Intel Core i7（Intel Core i7），166
tournament predictors（竞赛预测器），165
Two-level cache hierarchy（两级缓存层次结构）
cache optimization（缓存优化），B-31
ILP（ILP），245
Two-way conflict misses, definition（两种冲突缺失，定义），B-23
Two-way set associativity（两路组相联）
ARM Cortex-A8，233
cache block placement（缓存块放置），B-7，**B-8**
cache miss rates（缓存缺失率），**B-24**
cache miss rates vs. size（缓存缺失率与大小），B-33
cache optimization（缓存优化），B-38
cache organization calculations（缓存组织计算），B-19~B-20
commercial workload（商业工作负载）370~373，**371**
multiprogramming workload（多重编程工作负载），374~375
nonblocking cache（非阻塞缓存），84
Opteron data cache（Opteron 数据缓存），B-13~B-14
2:1 cache rule of thumb（2:1 缓存经验规则），B-29
virtual to cache access scenario（虚拟至缓存访问情景），**B-39**
"Typical" program, instruction set considerations（"典型"程序，指令集考虑事项），A-43

U

U，见 Rack units
UMA，见 Uniform memory access
Uncached state, directory-based cache coherence protocol basics（未缓存状态，目录式缓存一致性协议基础），380，384~386
Unconditional branches（无条件分支）
branch folding（分支折合），206
branch-prediction schemes（分支预测方案），C-25~C-26
Unicode character（Unicode 字符）
MIPS data types（MIPS 数据类型），A-34
operand sizes/types（操作数大小/类型），12
popularity（普及性），A-14
Unified cache（统一缓存）
AMD Opteron example（AMD Opteron 示例），**B-15**
performance（性能）B-16~B-17
Uniform memory access （UMA）（一致存储器访问）
multicore single-chip multiprocessor（多核单芯片多处理器），**364**

SMP，346~348
Uninterruptible instruction（不可中断指令）
hardware primitives（硬件原语），388
synchronization（同步），386
Uninterruptible power supply（UPS）（不间断电源）
Google WSC（Google WSC），467
WSC calculations（WSC 计算），435
WSC infrastructure（WSC 基础设计），447
Uniprocessors cache protocols（单处理器缓存协议），359
development views（开发视角），344
linear speedups（线性加速比），407
memory hierarchy design（存储器层次结构设计），73
memory system coherency（存储器系统一致性），353，**358**
misses（缺失），371，373
multiprogramming workload（多重编程工作负载），376~377
multithreading（多线程），223~229
Unpacked decimal（非压缩十进制），A-14
UPS，见 Uninterruptible power supply
Use bit（使用位），B-46，B-52
User maskable events, definition（用户可遮罩事件，定义），C-45~C-46
User nonmaskable events, definition（用户不可遮罩事件，定义），C-45
User-requested events, exception requirements（用户请求事件，异常需求），C-45
Utility computing（公用计算），455~461

V

Valid bit（有效位）
address translation（地址转换），B-46
block identification（块识别），B-7
Opteron data cache（Opteron 数据缓存），B-14
paged virtual memory（分页虚拟存储器），B-56
segmented virtual memory（分段虚拟存储器），B-52
snooping（监听），357
symmetric shared-memory multiprocessors（对称共享存储器多处理器），366
Value prediction（值预测），202
Variable length encoding（变长编码）
control flow instruction branches（控制流指令分支），A-18
instruction sets（指令集），**A-22**
ISAs，14
Variables（变量）
compiler technology（编译器技术），A-27~A-29
CUDA，289
Fermi GPU，306
ISA，A-5，**A-12**
locks via coherence（通过一致性锁定），389
loop-level parallelism（循环级并行），316
memory consistency（存储器一致性），392
NVIDIA GPU Memory（NVIDIA GPU 存储器），304~305
procedure invocation options（过程调用选项），A-19
register allocation（寄存器分配），A-26~A-27
in registers（在寄存器中），A-5
synchronization（同步），375
TLP programmer's viewpoint（TLP 程序员的观点），394
VCs，见 Virtual channels
Vector architectures（向量体系结构），9
Vector Instruction（向量指令），**292，309**
Vectorizable Loop（可向量化循环），268，**292，313**
Vectorized code（向量化代码）
multimedia compiler support（多媒体编译器支持），A-31
vector architecture programming（向量体系结构编程），280~282
vector execution time（向量执行时间），271
VMIPS（VMIPS），268
Vectorized Loop,另见 Body of Vectorized Loop
Vector Lane Registers, definition（向量车道寄存器，定义），**292**
Vector Lanes（向量车道），**292，309**
Vector-length register（VLR）（向量长度寄存器）
basic operation（基本操作），274~275
VMIPS，267
Vector load/store unit（向量载入/存储单元）
memory banks（存储器组），276~277
VMIPS，265
Vector loops（向量循环）
NVIDIA GPU（NVIDIA GPU），294
processor example（处理器示例），267
strip-mining（条带挖掘），303
vector vs. GPU（向量与 GPU），311
vector-length registers（向量长度寄存器），274~275
vector-mask registers（向量遮罩寄存器），275~276
Vector-mask control, characteristics（向量遮罩寄存器，特性），275~276
Vector-mask registers（向量遮罩寄存器）
basic operation（基本操作），275~276
VMIPS，267
Vector Processor（向量处理器），292，309
Vector Registers（向量寄存器），309
Very Long Instruction Word（VLIW）（超长指令字）

clock rates（时钟频率），244
ILP，193~196
loop-level parallelism（循环级并行），315
multiple-issue processors（多发射处理器），**194**，
sample code（示例代码），**252**
Video（视频）
Amazon Web Services（Amazon Web 服务），460
application trends（应用趋势），4
PMDs，6
WSCs，8，432，437，439
Virtual address（虚拟地址）
address translation（地址转换），B-46
AMD64 paged virtual memory（AMD64 分页虚拟存储器），**B-55**
AMD Opteron data cache（AMD Opteron 数据缓存），B-12~B-13
ARM Cortex-A8，**115**
cache optimization（缓存优化），B-36~B-39
GPU conditional branching（GPU 条件分支），303
Intel Core i7，120
mapping to physical（映射到物理地址），B-45
memory hierarchy（存储器层次结构），**B-39**，**B-48**，B-48~B-49
memory hierarchy basics(存储器层次结构基础)，77~78
miss rate vs. cache size（缺失率与缓存大小），**B-37**
Opteron mapping（Opteron 映射），**B-55**
Opteron memory management（Opteron 存储器管理），B-55~B-56
page size（页大小），B-58
page table-based mapping（基于页表的映射），**B-45**
translation（转换），B-36~B-39
virtual memory（虚拟存储器），B-42，B-49
Virtual address space（虚拟地址空间）
example（示例），**B-41**
main memory block（主存储器块），B-44
Virtual caches（虚拟缓存），B-36~B-37
Virtual functions, control flow instructions（虚拟功能，控制流指令），A-18
Virtualizable architecture（可虚拟化体系结构）
Intel 80x86 issues（Intel 80x86 问题），**128**
system call performance（系统调用性能），**141**
Virtual Machines support（虚拟机支持），109
VMM implementation（VMM 实现），128~129
Virtualizable GPUs, future technology（可虚拟化 GPU，未来技术），333
Virtual machine monitor（VMM）（虚拟机监视器）
characteristics（特性），108
nonvirtualizable ISA（不可虚拟化 ISA），126，128~129
requirements（需求），108~109
Virtual Machines ISA support（虚拟机 ISA 支持），109~110
Virtual Machines（VMs）（虚拟机（VM））
Amazon Web Services（Amazon Web 服务），456~457
cloud computing costs（云计算成本），471
ISA support（ISA 支持），109~110
protection（保护），107~108
protection and ISA（保护与 ISA），112
server benchmarks（服务器基准测试），40
virtual memory and I/O（虚拟存储器与 I/O），110~111
WSCs，436
Xen VM，111
Virtual memory（虚拟存储器），B-3
Virtual Machines impact（虚拟机影响），110~111
Virtual methods, control flow instructions（虚拟方法，控制流指令），A-18
VLIW,见 Very Long Instruction Word
VLR, 见 Vector-length register
VMIPS
basic structure（基本结构），**265**
DLP，265~267
double-precision FP operations（双精度浮点运算），**266**
gather/scatter operations（集中/分散操作），280
ISA components（ISA 分量），264~265
multidimensional arrays（多维数组），278~279
Multimedia SIMD Extensions（多媒体 SIMD 扩展）282
multiple lanes（多车道），271~272
vector execution time（向量执行时间），269~270
vector vs. GPU（向量与 GPU），308
vector-length registers（向量长度寄存器），274
vector load/store unit bandwidth（向量载入/存储单元带宽），276
vector processor example（向量处理器示例），267~268
VLR，274
Voltage regulator modules（VRMs），WSC server energy efficiency（电压调节模块，WSC 服务器能量效率），462
Volume-cost relationship, components（体积-成本关系，分量），27~28

W

Wafers（晶圆）
example（示例），31
integrated circuit cost trends（集成电路成本趋势），28~32
Wafer yield（晶圆正品率），30
Wall-clock time（壁钟时间）
execution time（执行时间），36
WAR, 见 Write after read

Warehouse-scale computers（WSCs）（仓库级计算机），345
Warp Scheduler（Warp 调度程序），**292**，**314**
WAW, 见 Write after write
Way prediction, cache optimization（路预测，缓存优化），81~82
Way selection（路选择），82
WB, 见 Write-back cycle
Weak ordering, relaxed consistency models（弱排序，宽松连贯性模型），395
Weak scaling, Amdahl's law and parallel computers（弱扩展，Amdahl 定律和并行计算机），406~407
Web index search, shared-memory workloads（Web 索引搜索，共享存储器工作负载），369
Web servers（Web 服务器）
  ILP for realizable processors（可实现处理器的 ILP），218
  performance benchmarks（性能基准测试），40
Weitek（Weiteck），364
Wet-bulb temperature（湿球温度）
  Google WSC（Google WSC），466
  WSC cooling systems（WSC 制冷系统），449
Window
  latency（延迟），B-21
  processor performance calculations（处理器性能计算），218
  scoreboarding definition（记分卡定义），C-78
Window size（窗口大小）
  ILP limitations（ILP 局限性），221
  ILP for realizable processors（可实现处理器的 ILP），216~217
  parallelism（并行度），**217**
Wires（连线）
  energy and power（能量与功率），23
  scaling（缩放），19~21
Within instruction exceptions（指令内异常），C-45
Word count（词数），**B-53**
Word offset, MIPS（字偏移量，MIPS），C-32
Words（字）
  aligned/misaligned addresses（对齐/未对齐地址），**A-8**
  AMD Opteron data cache（AMD Opteron 数据缓存），B-15
  memory address interpretation（存储器地址解释），A-7~A-8
  MIPS data transfers（MIPS 数据传输），A-34
  MIPS data types（MIPS 数据类型），A-34
  operand sizes/types（操作数大小/类型），12
  operand type（操作数类型），A-13~A-14
Workloads（工作负载）
  execution time（执行时间），37
  Google search（Google 搜索），439
  Java and PARSEC without SMT（Java 与 PARSEC，不采用 SMT），**403~404**
  symmetric shared-memory multiprocessor performance（系统共享存储器多处理器性能），367~374
  WSC goals/requirements（WSC 目标/需求），433
  WSC resource allocation case study（WSC 资源分配案例研究），478~479
  WSCs，436~441
Write after read（WAR）（读后写）
  data hazards（数据冒险），153~154，169
  dynamic scheduling with Tomasulo's algorithm（采用 Tomasulo 算法的动态调度），170~171
  hazards and forwarding（冒险与转发），C-55
  ILP limitation studies（ILP 局限性研究），220
  MIPS scoreboarding（MIPS 记分卡），C-72，C-74~C-75，C-79
  register renaming vs. ROB（寄存器重命名与 ROB），208
  ROB，192
  Tomasulo's advantages（Tomasulo 的优点），177~178
  Tomasulo's algorithm（Tomasulo 算法），182~183
Write after write（WAW）（写后写（WAW））
  data hazards（数据冒险），153，169
  dynamic scheduling with Tomasulo's algorithm（采用 Tomasulo 算法的动态调度），170~171
  execution sequences（执行序列），C-80
  hazards and forwarding（冒险与转发），C55~C-58
  ILP limitation studies（ILP 局限性研究），220
  microarchitectural techniques case study（微体系结构技术案例研究），253
  MIPS FP pipeline performance（MIPS 浮点流水线性能），C-60~C-61
  MIPS scoreboarding（MIPS 记分卡），C-74，C-79
  register renaming vs. ROB（寄存器重命名与 ROB），208
  ROB，192
  Tomasulo's advantages（Tomasulo 的优势），177~178
Write allocate（写分配），B-11
Write-back cache（写回缓存），B-11
Write-back cycle（WB）（写回周期）
  basic MIPS pipeline（基本 MIPS 流水线），C-36
  data hazard stall minimization（数据冒险停顿最小化），C-17
  execution sequences（执行序列），C-80
  hazards and forwarding（冒险与转发），C-55~C-56
  MIPS exceptions（MIPS 执行），C-49
  MIPS pipeline（MIPS 流水线），**C-52**
  MIPS pipeline control（MIPS 流水线控制），C-39
  MIPS R4000（MIPS R4000），C-63，C-65

MIPS scoreboarding（MIPS 记分卡），C-74
pipeline branch issues（流水线分支问题），**C-40**
RISC classic pipeline（RISC 经典流水线），C-7~C-8，C-10
simple MIPS implementation（简单 MIPS 实现），C-33
simple RISC implementation（简单 RISC 实现），C-6
Write broadcast protocol, definition（写广播协议，定义），356
Write buffer（写缓冲区）
AMD Opteron data cache（AMD Opteron 数据缓存），B-14
Intel Core i7，**118**，121
invalidate protocol（无效协议），356
memory consistency（存储器一致性），393
memory hierarchy basics（存储器层次结构基础），75
miss penalty reduction（降低缺失代价），87，B-32，B-35~B-36
write merging example（写合并示例），**88**
write strategy（写策略），B-11
Write hit（写命中）
cache coherence（缓存一致性），**358**
directory-based coherence（目录式一致性），424
single-chip multicore multiprocessor（单芯片多核心处理器），414
snooping coherence（监听一致性），359
write process（写进程），B-11
Write invalidate protocol（写失效协议）
directory-based cache coherence protocol example（目录式缓存一致性协议示例），382~383
example（示例），359，**360**
implementation（实现），356~357
snooping coherence（监听一致性），355~356
Write merging（写合并）
example（示例），**88**
miss penalty reduction（降低缺失代价），87
Write miss（写缺失），385
Write result stage（写结果级）
data hazards（数据冒险），154
dynamic scheduling（动态调度），174~175
hardware-based speculation（基于硬件的推测），192
instruction steps（指令步骤），175
ROB instruction（ROB 指令），186
scoreboarding（记分卡），C-74~C-75，C-78~C-80
status table examples（状态表示例），C-77
Tomasulo's algorithm（Tomasulo 算法），178，180，190
Write serialization（写入序列化）
hardware primitives（硬件原语），387
multiprocessor cache coherency（多处理器缓存一致性），353
snooping coherence（监听一致性），356
Write stall, definition（写停顿，定义），B-11
Write strategy（写策略）
memory hierarchy considerations（存储器层次结构考虑事项），B-6，B-10~B-12
virtual memory（虚拟存储器），B-45~B-46
Write-through cache（直写缓存）
average memory access time（存储器平均访问时间），B-16
coherency（一致性），**352**
invalidate protocol（失效协议），356
memory hierarchy basics（存储器层次结构基础），74~75
miss penalties（缺失代价），B-32
optimization（优化），B-35
snooping coherence（监听一致性），359
write process（写过程），B-11~B-12
Write update protocol, definition（写更新协议，定义），356
WSCs，见 Warehouse-scale computers

X

Xen Virtual Machine（Xen 虚拟机）
Amazon Web Services（Amazon Web 服务），456~457
characteristics（特性），111

Y

Yahoo!, WSCs，465
Yield（正品率）
chip fabrication（芯片制造），61~62
cost trends（成本趋势），27~32
Fermi GTX 480（Fermi GTX 480），324

Z

Zynga, FarmVille，460