密级：_____

中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

多核系统内存资源管理优化技术的研究

作者姓名：刘　磊

指导教师：吴承勇　研究员，冯晓兵　研究员

中国科学院计算技术研究所

学位类别：工学博士

学科专业：计算机系统结构

研究所：中国科学院计算技术研究所

2014 年 5 月

An Investigation into Key Issues of Memory Management and Optimization in Multicore Systems

By

Lei Liu

A Dissertation Submitted to

University of Chinese Academy of Sciences

In partial fulfillment of the requirement

For the degree of

Doctor of Computer Science

ICT

May, 2014

书脊

声　　明

我声明本论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，本论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

作者签名：　　日期：

论文版权使用授权书

本人授权中国科学院计算技术研究所可以保留并向国家有关部门或机构送交本论文的复印件和电子文档，允许本论文被查阅和借阅，可以将本论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本论文。

（保密论文在解密后适用本授权书。）

作者签名：　　导师签名：　　日期：

摘　要

“内存墙”问题一直是制约计算机系统整体性能的重要因素。在共享内存储器的多核系统中，线程之间的“访存干扰”会降低主存储器（Main Memory）系统的响应速率进而加大访存延迟（Memory Latency），这使得计算单元与主存之间在处理速率上的差异又进一步增大了，即“内存墙”对性能的“负”影响又被扩大了。为了降低线程间访存干扰，学术界进行了广泛的研究工作，包括优化内存控制器中访存调度算法，通过软硬结合的方式优化存储资源分配，以及任务调度等方法。这些方法在有些情况下是有效的，但也存在着各方各面的问题。例如，对内存控制器的优化需要修改片上的相关硬件逻辑因此开发成本很大，而任务调度方法的适用范围又非常有限。关键的问题是，这些方法都没有在根本上消除线程间在DRAM Bank上的访存干扰。

本研究以业界广泛讨论的“页着色”（Page-Coloring）技术为出发点，将该技术应用于对主存储器的性能优化，进而在根源上消除了线程间的访存干扰。随后，本研究进一步提出了内存储资源（包括Cache和主存）“垂直”管理的概念，并基于此概念设计实现了“应用与体系结构敏感”的内存管理模型（Application-Architecture-Aware Memory Management），为未来多核、众核系统的内存资源的管理与优化提供了参考方案。本文的主要贡献如下：

（1）本研究首次在真实环境下实现了DRAM Bank的“页着色”划分机制（Bank Partitioning Mechanism，BPM），彻底消除了多道程序之间在DRAM Bank上相互之间的访存干扰。实验证明，在现有内存控制器的调度算法的基础上，BPM能使整机性能再提升平均5%左右，同时降低了主存的能耗。BPM支持动态的划分方式，扩展性良好，用户可根据工作集的特点和需求有针对性的部署不同的划分策略。

（2）本研究提出并首次在真实环境中实现了通道划分（Channel Partitioning）的机制，缓解了线程间在多通道上的访存干扰。基于此，本研究设计了一套利用PMU的支持动态划分框架BPM+（并实现其原型），该框架整合了DRAM Bank划分和Channel划分机制，形成了在主存储器体系上完整的划分优化机制。实验结果表明，BPM+的性能普遍优于BPM。基于BPM+，后继研究者可以根据自身研究需求实现其它的采用DRAM划分方式的优化机制。

（3）本文首次提出了多级存储器之间协同的“垂直”划分的方式，并揭示了相应的较为完整的划分空间（见第四章A-/B-/C-VP）。随后，本文采用数据挖掘的方式在大量的真实的实验结果中找到了工作集的特点与最优的划分方式之间的关系。本研究为后继的系统优化人员提供了更多的优化选择与参考。

（4）本文首次提出了内存储资源“垂直”管理的概念，并基于此概念设计了应用程序特征与体系结构敏感的内存管理模型（Application-Architecture-Aware Memory Management）。原型系统的实验结果显示，在真实环境下，对于动态变化的工作集，该系统能够根据工作集的访存行为特点选择恰当的内存资源管理和分配方式，使系统的性能始终处在较优的状

态。实验结果显示，与当前先进（state-of-the-art）的优化机制相比，在真实的评测环境中，最大性能提升为11%。

关键词：页着色；DRAM Bank；通道（Channel）；划分；垂直划分；内存资源管理

An Investigation into Key Issues of Memory Management and Optimization in Multicore Systems

Lei Liu (Computer Architecture)

Directed By Chengyong Wu and Xiaobing Feng

The “memory wall” problem has been a predominant factor that limits the overall performance of modern computing systems. In today’s prevalent multicore machines with shared memory, the gap between the processing core and the memory is enlarged by memory access interference among simultaneously running threads, which further amplifies the negative impact of the memory wall issue. To mitigate the inter-threads memory interference and alleviate the memory wall problem, extensive research has been conducted by numerous academic groups to explore a variety of mechanisms including scheduling optimizations in memory controllers, software-hardware cooperative resource allocation approaches, memory-access-aware task scheduling, etc. Although these mechanisms help in increasing memory access performance under different scenarios, they all have certain limitations and bring problems of various kinds. For example, optimizations on memory controller require on-chip hardware logic redesign or modifications that incur high development and manufacturing costs. A task scheduling approach eliminates hardware changes but has highly limited application scenarios. Additionally, these existing approaches cannot entirely eliminate inter-thread inference on parallel machines.

This dissertation work leverages the widely adopted page coloring technique to optimize memory accesses and entirely eliminate inter-thread inference. Additionally, a new concept of vertical memory management is proposed, based on which an application-architecture-aware memory management system is built as a solution for memory resource allocation and optimization for high-performance multiprocessors platforms. The primary contributions of this dissertation work include:

(1). This work proposes bank partitioning mechanism (BPM), a page-coloring based framework that is implemented on a real system, and, for the first time, completely addresses memory access interference issue at the DRAM bank level. Experiments demonstrate BPM brings an average of 5% system performance improvement while reducing the energy consumption of the main memory. BPM supports dynamic partitioning and can adapt its partitioning mechanisms based on different use cases, workload characteristics and memory requirements.

(2). A PMU (Performance Monitoring Unit) based framework, BPM+, is designed and prototyped to augment the baseline BPM system with the additional functionality of dynamically memory channel partitioning. BPM+ combines the benefits of the DRAM bank and channel partitioning and forms a complete system for memory partitioning and

optimization. BPM+ is the first effort that achieves DRAM bank-level and channel-level partitioning simultaneously in a real system to mitigate memory interference. Experimental results demonstrate that BPM+ outperforms BPM in general cases. BPM and BPM+ provide a rich space of memory partitioning space based on which many other DRAM partitioning mechanisms can be achieved for future potential studies.

(3).Additionally, this dissertation is the first to propose a "vertical" memory partitioning mechanism that provides a relatively complete partitioning space (see A-/B-/C-VP in Chapter 4) across multiple levels of the memory hierarchy. Using a data mining driven approach on a large set of experiment results in real systems, optimal memory allocation mechanisms are correlated with applications with certain memory/cache characteristics. This result provides valuable insights and choices for relevant work in selecting/designing memory system optimization mechanisms.

(4). Base on the concept of vertical memory management, an Application-Architecture-Aware memory management system is designed. The prototype of this system is demonstrated to be capable of appropriately selecting memory resource allocation methods according to dynamically changing workloads. In the system, relevant workloads characteristics can be captured and used to determine an optimal memory allocation approach in an adaptive manner. Experiments show the proposed memory management system can bring an 11% performance gain compared to the state-of-the-art memory allocation approach.

Keywords: Page-Coloring, DRAM Bank, Channel, Partitioning, Vertical Partitioning, Memory Management

目 录

图目录

表目录

第一章 引 言

1.1 内存储器系统

计算机的内存储器系统的层次结构自下而上的包括主存（Main Memory）、高速缓存（Cache）以及寄存器（Register）。目前主流计算机的主存储器是由DRAM Banks组成的片外系统，与中央处理器（CPU）芯片通过数据通道（Channel）相连接。在运算的过程中，主存通过通道为CPU提供所需的数据，衡量的指标是带宽（GB/sec）。计算单元的运算速率与主存相比存在较大的差异，理论上每次访存行为需要50-100个时钟周期，而在每个时钟周期CPU理论上能够运行若干条指令。但由于CPU运行时所需的指令和数据均需要从主存中读取，因此，理论上每个时钟周期CPU都有可能访问主存。也就是说，运行中的CPU在每个时刻都有可能被迫停止而经历较长时间的访存延迟（Memory Latency），这使得CPU频繁的处于空闲状态。因此，主存储器系统成为计算机系统的“瓶颈”，被业内形象的描述为“内存墙”（Memory Wall）[38]，并始终影响着计算机的整体性能。

表1.1 1980～2013年DRAM发展明细［38，46，72-76］

Standard	Clock rate(MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
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

为了缓解“内存墙”问题导致的系统性能的“瓶颈”，在过去的30多年里（从1980年至今），业界尝试通过提高主存的工作频率来提高访存效率降低访存延迟。如表1.1所示，从DDR-266到DDR4-3200，主存的峰值带宽已经从2.1GB/s提高到了25.6GB/s，即访存效率提高了8倍。但在“摩尔定律”依然有效的时代（准确的说目前我们已进入“后摩尔定律”时代），片上计算单元的数目与计算单元的频率依然在持续增加（但相对摩尔定律时代这个增长速率呈现缓和趋势），对带宽的需求也在以更高的速率增大，即“内存墙”的问题依然存在。例如，对于目前广泛使用的Intel-i7 3.2GHz的4核8线程的CPU，使这款CPU达到饱和运算状态时理论上所需的峰值带宽将高达409.6GB/sec，但是主流的DDR3-1600和DDR4-3200 DRAM主存的理论峰值带宽是12.8GB/sec和25.6GB/sec，仅仅是

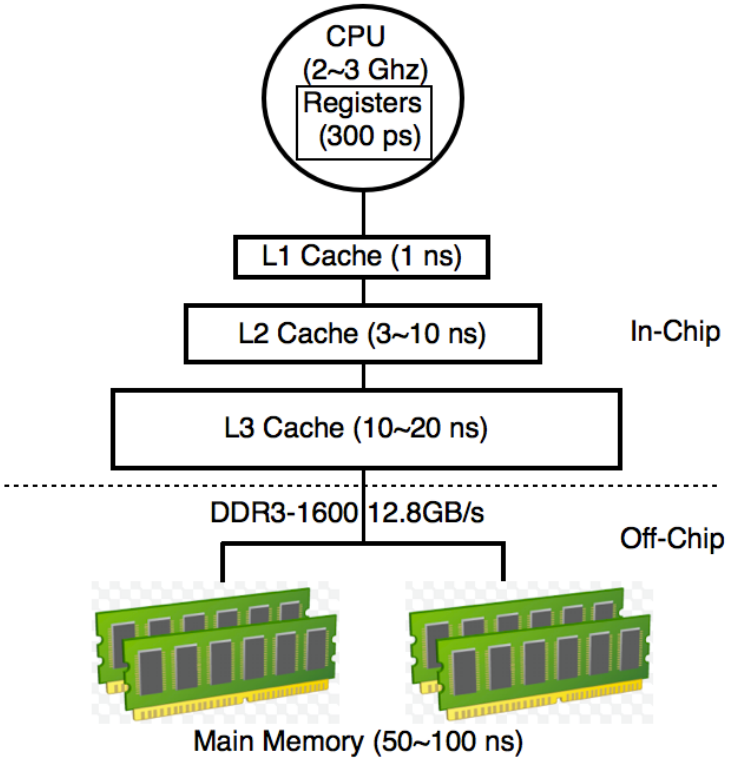


图 1.1 内存储器系统的的层次结构［38］

这一需求的3%和6%［38］。因此，从这些发展趋势和现象可以看出，从上个世纪80年代到今天，计算单元与主存之间的“瓶颈”问题始终伴随着计算机科学的发展［88，89］。

1.2 内存储器系统的层次结构

为了提高系统的整体性能，计算机体系结构的研究和设计者们提出了多级内存储器架构，如图1.1所示，即在芯片内部设置存储资源（Cache和寄存器），一些数据将从主存中被读取到片上的存储资源，部分的数据请求将不必访问主存，而仅访问这部分延迟较小片上资源。寄存器的访存延迟平均仅为300ps，L1/L2/L3 Cahce的访存延迟一般是1~2ns/3~10ns/10~20ns［38］。因此，访问这些片上资源避免了高代价的访存操作（理论值50～100ns，实际值对远超过理论值），有助于提高计算机的整体性能。然而，受芯片面积、材料规格、制造成本等问题的影响，片上存储资源的容量通常不可能做的很大［37,38］，因此对计算机性能来讲显得非常关键。例如，Xeon E5620 64bit处理器最后一级Cache的容量是12MB（已经算是比较大的片上存储资源），而其最大的寻址能力则可达2^64（16 EB）。系统运行时只有很少一部分的数据能够被读入Cache，因此，如何利用好这些宝贵的片上存储资源就变得至关重要，学术界和工业界也对此问题表示非常关注。为了利用好宝贵的Cache资源，优化Cache的性能，业界提出了很多方案，概括来说，主要集中在提高访问命中时间，Cache带宽，减少缺失率以及缺失代价［12,15,26,27,38,95,97］。

1.3 优化访存的必要性

无论以何种方式优化Cache的性能，除非程序的工作集的需求（Working Set Size）完全小于Cache容量，否则Cache缺失（Cache Miss）是一定会发生的［26，27，32，43，57，38］，每次缺失需要从主存（DRAM Bank）中返回64Byte的数据填充到一个Cache行（被称为Cache Line，或Cache Block）。在多核计算机上，通常最后一级Cache（Last Level Cache, LLC）和DRAM Bank是被多个计算单元所共享的［38］。在现代计算机体系结构中，资源的共享则意味着相互竞争和干扰［38，98］。线程间（进程间）[1] 在LLC上的相互干扰会引发Cache上的“冲突缺失”，不仅降低了Cache的效率还强迫程序访问主存，进而又会造成在主存资源上的“访存干扰”。如前文所述，主存与计算单元之间存在着“先天”的计算速率的差异，再加之线程之间的访存干扰，二者之间的差距再度被放大，给系统性能造成较大的负面影响。

从共享LLC的角度来说，线程间竞争导致的性能下降可达50%[32]；对于共享的DRAM来说，则有可能引发系统吞吐量（System Throughput）、服务质量（Quality of Service，Fairness）的严重下降。在DRAM上的访存干扰通常被学术界细化为Bank级的冲突（Bank Level Conflicts），这种“冲突”带有微体系结构的特点，主要表现为行缓冲（Row-Buffer）的“颠簸”（Row-Buffer Thrashing）。具体来讲，对于DRAM Bank，每一个被读取的数据都需要先从DRAM Bank中被读到行缓冲（Row-Buffer）里，再通过数据总线传送到Cache。被读到行缓冲中的数据必须是一整行（即Bank的一行），这种设计蕴含着“局部性”优化的原理，即在行缓冲中的数据应能够尽可能多的被使用。但是，在多核环境下，如果来自不同线程的访存请求被映射到同一个Bank上（通常不会是同一行），那么将不可避免的引发行缓冲的颠簸，破坏了局部性、增大了访存延迟，进而降低了系统的整体性能。例如，DDR3芯片的Row Access Strobe (RAS)是24～36ns ［72-76］，指的是从打开一个Row，到数据操作完成，允许将行缓冲中的内容写回Bank的最小时间间隔，标志着一次数据操作的完成，是DRAM操作中最为耗时的部分。较优的访存行为是能够连续命中行缓冲中的数据（即局部性），但如果频繁的发生行缓冲的颠簸，将会连续引入RAS操作，访存延迟就被扩大了。

图1.2显示了在行缓冲颠簸的情况下造成的访存延迟与行缓冲命中时的性能对比（具体参数见第二章背景介绍），如果行缓冲能够命中的话，仅三次访存即能节省1/3的访存时间。有数据显示，在8核8线程并发的情况下，访存密集型（Memory Intensive）的程序可能会对访存非密集型（Memory Non-Intensive）的程序造成11.35倍的性能下降［48］，意味着访存干扰已经对系统的服务质量已经造成了严重的影响；对于系统的吞吐量，［42，69］显示的数据说明线程间访存行为的干扰可能会引发2.9～14.6倍的访存延迟，在4～8核的计算平台上。**本研究即集中在解决由于线程间访存干扰引发的系统性能下降的问题。**

1.4 访存优化的方法

[1] 本文约定，除非特殊说明，本文所提及的“线程”即代表“进程”。在系统结构领域，国际学术界的文献中通常用“Thread”代替“Process”。

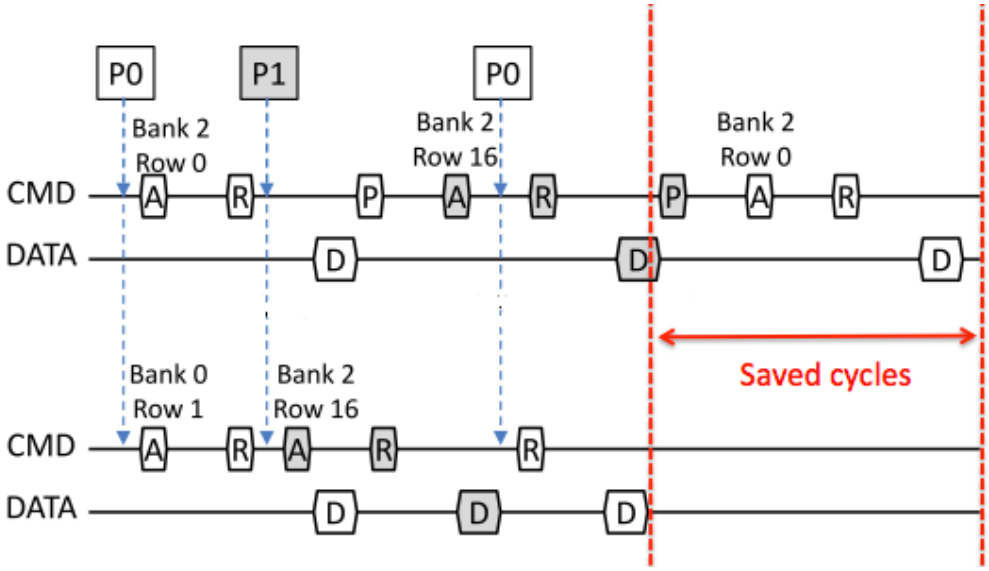


图1.2 行缓冲颠簸与命中的性能对比［85］。注：A（ACTIVE），R（READ），D（Data，数据块传输），P（PRECHARGE）分别代表DRAM的操作或行为，详见第二章的背景介绍。P0,P1分别代表两个不同程序在发出访存请求。

本文把通过优化存储体系或相关机制以降低访存延迟并提高计算机整体性能的研究工作归类为“访存优化”的范畴。研究访存优化是非常有意义的，学术届每年都进行很热烈的讨论，代表体系结构领域前沿研究成果的学术会议每年都有若干篇文章就此问题展开讨论；在计算单元性能已近极限的情况下，工业界为了能够提高系统的整体性能，也花费大量的人力物力研究此问题。总的来说，访存优化相关的工作可以被归纳为三个层次：

（1）硬件层（Hardware Level）——优化控制访存行为的核心部件与机制（如内存控制器）。内存控制器（Memory Controller，MC）是系统硬件级控制访存行为的核心部件（具体协议见第二章）。业界经常提及的“访存调度”，就是指MC所采用的对访存请求的调度策略。当前主流机器通常采用的是面向行缓冲局部性的调度优化（FR-FCFS，类似于“先来先服务”的策略），目的在于最大化行缓冲的命中率，减少行缓冲的颠簸。但由于此类方法在任意调度窗口内始终赋予即将访问已经打开的行的访存请求较高的优先级，因此会始终滞后其它的访存请求，故很可能会引发访存的不公平现象，从而造成服务质量的下降。另外，FR-FCFS虽然很容易在内存控制器中实现，但随着多核体系结构的发展，以及工作集的访存特点在近几年不断的变化，该方法已经越来越不能满足应用以及系统性能的需求。近期关于优化MC调度的研究成果证明了［48，68，70］，单一的访存调度策略已经不能满足当前的需求，取而代之的应该是兼顾线程访存特点和微体系结构特征的高效灵活的调度机制。另外，面向多核平台的访存调度不应该“呆板”的依据单一的原则进行优化，而应面向某一种性能上的特殊需求（如服务质量，Quality of Service），或者同时兼顾多种性能指标来进行优化。例如，TCM［48］能够依据线程访存特点同时兼顾吞吐量和公平性两个重要的指标，是目前较先进的优化机制。此外，业界也提出了很多面向单一性能指标的优化

方式（可能会牺牲其它性能指标），比如，仅面向公平性的服务质量敏感的调度机制（在数据中心服务器上尤其有效），等等，这里不再赘述。另一个在硬件层解决问题的思路是彻底摈弃现有的材质的存储体系，更换现有的DRAM系统，代之以新材料的存储介质，如非易失性存储材质（NVM）等，但该问题不在本文的主要研究范围之内。可参见本文关于未来工作的讨论。

（2）应用层（User Level）——软件优化的策略。通过在应用层编写包含控制或调度策略的软件来控制线程运行状态和对存储资源的使用。具有代表性的方法是通过系统调用来控制线程的运行速率，进而降低并发线程对共享存储资源的压力，实际是以另一种形式降低了访存竞争。该方式通常被学术界称为“节流”（Throttling）[29,92,111,115,121]。总的来讲，这种方法通常需要辅以PMU（Performance Monitor Unit）[124]硬件采样来汇总每个线程在运行时的信息，再根据预先设定的策略建模得出的结果作为调度的依据来控制系统的运行。例如，一种简单的，保证高优先级程序服务质量的调度策略，可以通过“节流”其它相对低优先级程序来达到目的。另外，这种方法对于带宽利用率相关的优化尤其有效[111]，通过节流资源的方式使带宽的利用率保持在目标范畴内，进而使得整个系统的性能可控。有些学者认为，“任务映射”（Task Mapping）也属于本层次工作的范畴[23]，这种方式通常考虑到了任务（或线程）之间的访存冲突（或线程之间的访存互扰），目的在于最小化资源竞争带来的性能下降。我们认为在应用层上的方法很多都是“殊途同归”的。

（3）系统层（System Level）——优化内存资源的管理与分配机制。这种方式主要指的是直接修改操作系统（Operating System, OS）对物理页面管理与分配的核心机制Buddy System，目的在于通过趋于合理的资源分配方式来缓解或者消除线程之间的访存干扰。比较有代表性的例子有两个，一个是采用“页着色”（Page-Coloring）的方式，仅分配给某个线程属于某种颜色的物理页面，进而在根源上消除线程间在共享Cache上的干扰[25,57]。在学术界，页着色通常等同于存储资源的“划分”。另一个是采用随机分配的方式（M^3）[83]来随机化物理页面的分配，使每个线程所使用的物理页面离散的分布在所有Bank上（或者说以物理页为单位粗粒度的“交替”在所有Bank上）。两种不同的分配方式，“划分”与“交替”，是背道而驰的资源分配的哲学。实验证明，这两种方式针对不同性质的工作集表现“各有千秋”。本文将在后继章节详细分析这些内容。

1.5 现有的访存优化方法的优势与劣势

在三个层次上的研究代表着三条不同的思路，虽然都能在不同程度上优化访存效率，但由于所处的层次不同，采用的设计原则不同，应用的场景不同，因此它们也各有其优势与劣势。计算机系统是一个复杂的系统工程[98]，因此，从系统设计原则的角度来讲，任何一个微小的改动，都有可能引发连锁反应（或描述为产生“蝴蝶效应”）[98]。所以，对计算机系统核心部件的任何改进都不能被视为是微小的。

硬件层次上的改进可谓是这三个层次中最为复杂的，成本也是最高的。改进之后的硬件逻辑需要流片才能真正看到效果（通常都只有采用模拟的方式进行前期的实验），这与纯软件的方式相比不占优势，因为流片的成本与周期是不可被忽视的问题。另外，复杂的硬件逻辑也无形中增加了芯片设计的成本、生产成本。即便流片成功芯片投入运行，新的调度算法能够在多大程度上发挥作用？复杂的调度逻辑的实用性怎么样？是否能够适用于真实的复杂的计算环境？等等。本文认为这种方式处在一个极端，要么真正的起到作用，使每台采用新调度算法的机器性能大幅度提升；要么就造成资源上的浪费，生产出的芯片可用性不高。因此，除非有稳妥的方案，工业界的芯片制造商对此类方案通常持谨慎的态度［60,61］。

在应用层部署优化软件的方式相对与纯硬件的方法较为可行，成本也相对低，因此更多的设计者愿意去尝试。但问题在于，在系统硬件及软件已经确定的情况下，应用层优化的方式能够发挥的空间往往是极为有限的。另外，从系统设计的原则来讲，软件系统的复杂度应该维持在一个较低的范畴内，否则会影响系统的可用性、可维护性和扩展性［98］。如前文所述，这种方法通常都需要依赖硬件采样的信息，为了准确起见，又往往引入复杂的建模和预测逻辑，以及调度逻辑算法组件，等等。凡此种种，会在原有系统外围形成一套复杂的辅助的软件机制。从软件工程的角度来讲，这必然会造成软件系统复杂度持续升高，违背了系统设计的原则。随着软件系统的生命周期的继续（继续开发与升级），其可用性会变的越来越差。另外，硬件计数器往往有赖于特定的平台，依平台而异。因此，这种方法的实用性又是一个问题。从实践的角度来讲，系统运行与维护工程师们最“头疼”的就是安装并维护复杂的软件系统。因此，在应用层部署辅助软件的方式有它的局限性。

相对而言，在系统层解决问题通常是一种折中的可被接受的方案。第一，比较容易将系统的改动限定在某个可接受的范围之内，不至于导致不可接受的系统复杂度的扩张。第二，系统级的改动往往涉及到核心机制，更触及问题的本质，因此更为有效。第三，软件的方式与硬件的方式相比更容易部署，投入使用的周期也更短，并且成本较低（没有生产成本，如果在开源社区则成本会更低），容易被业界接受。比如上文提及的“页着色”技术，就是纯系统方式的优化机制，通过修改操作系统内核的内存资源分配机制以隔离线程之间的访存，达到优化的目的。该方法及其扩展软件被广泛的应用在商业系统上，真实的给产业界带来利益。但系统层的挑战在于，对研发人员的要求较高。越是核心的机制往往越精密，逻辑更为复杂。因此“牵一发而动全身”，如果改动不当，非但达不到效果，还有可能使系统的功能性能受到影响，或者造成灾难性的后果。

1.6 本文研究的具体问题和出发点

本研究的目的在于优化内存储系统的性能。前文所提及的访存优化算法其核心无非就是为了提高行缓冲的命中率（局部性），其方法无非就是降低线程之间的访存干扰。因此，

本研究设想，**能否设计一种简单有效的机制在根源上消除线程之间在Bank上的相互干扰？**此问题即为本研究的出发点。

受“页着色”的启发，本文更倾向于在系统层寻找解决方案，即通过资源分配的方式消除或降低线程之间的访存干扰。能否同样利用页着色技术来划分DRAM系统？如果答案是肯定的，那么每个线程将只访问属于自己的那一组Bank（取决于分配给它的颜色），因此线程之间的访存干扰也就不复存在，即在根本上消除了访存干扰。其实，关于这个问题，在学术界曾经有过热烈的讨论［79,85,121］，但直到本研究出现为止，Bank划分的思想还没在真实的系统上被实现过，关于这种机制是否真实有效，性能提升多少，扩展性怎么样，适用性如何等问题都没有明确的答案。在存储器资源划分优化的问题背后，并不局限于访存优化，实际上有一个更为深邃的问题：**在多核体系结构下，采用什么样的资源分配方式是最合理的？或者近似最合理的？**

为了回答这些问题，本文以当今主流的多核计算平台为目标机，对“页着色”机制进行了相对完整的分析和研究。在此基础上，本文进一步提出并研究了对DRAM Bank的划分机制，多级存储器（包括LLC和DRAM）之间的“垂直”划分，提出了应用与体系结构敏感的内存管理概念模型（Application-Architecture-Aware Memory Management）。该模型整合了物理页的“垂直”划分管理、随机分配模型等较新的资源管理与分配机制，并希望根据工作集的特点选择合适的资源分配方案，目标是最大化现有体系结构的优势，最大化资源的利用率，最小化访存冲突带来的负面影响。

1.7 本研究的概况及主要贡献

本研究是在真实的主流多核计算环境中进行的。与采用模拟器进行的研究不同，本研究在进行过程中完整运行了由数千组真实工作集组成的实验（仅实验部分耗时即接近两年），获得了大量的真实数据。为了获取划分机制与工作集特点之间的敏感性关系，我们采用了数据挖掘来揭示划分方式、工作集特点与最优性能之间规律，并将这些规律总结成为指导资源管理与分配的重要原则。通常数据挖掘是应用在网络数据的研究领域，本文尝试在系统结构领域的研究中应用这方面的技术，从实际效果来看（见第四章），这种“小学科交叉”的尝试收到了较好的效果。

本文为在系统层解决访存优化问题提供了参考。为了适应复杂的真实运行环境，本文的系统原型支持动态的在线（on-the-fly）的优化机制，可以根据工作集的访存特点选择较优的资源分配方式，以获得更高的资源利用率和系统性能。本研究是一个纯粹的操作系统层次的工作，没有使用除操作系统之外的辅助硬件（如硬件计数器等）或者其它额外的辅助系统。本系统通过轻量的内核级的采样机制和支持多种分配方式的多层次x-Buddy System达到最优资源分配的目的，功能模块是在操作系统内核态运行的，从而避免了复杂的系统软件部署过程，只要目标机器能够被安装Linux操作系统，本系统即可被部署。

本研究是基于学术界丰厚的积淀而开展的。“页着色”技术曾经被广泛的应用于商业处理器（SUN UltraSPARC和Intel系列），和商业系统（如vmware的虚拟机机制），学术界也曾就该技术的核心机制和应用策略进行过深入的研究和讨论［16,25-27,50,84,91,93,99,103,109,116,123］。“页着色”技术被业界普遍认可，至今依然是业界研究的“热点”问题。但目前面临的问题有两个，第一，近几年Intel等芯片制造商普遍采用了XOR的机制来散列物理地址的映射（内存储器资源已经不能直接通过地址位来索引），因此，页着色的应用前景也晦暗不明，面临着严峻的挑战；第二，该技术在提出之初，是为了改善共享Cache的性能。但现在的工作集对Cache资源的需求越来越大，划分将会减少程序的可用资源，进而损害系统性能，这有可能会导致划分的“弊大于利”。因此，在当今主流的系统上，如何使用页着色技术也是一个难题。挑战往往蕴含着更大的机遇，眼下，数据中心的资源管理以及多核系统资源利用率的问题始终困扰着体系结构研究者，页着色技术恰好给我们提供了有效的参考，使得我们所从事的研究有了可靠的技术手段和扎实的理论基础。这是我们研究的另一个优势，这得益于前人的积累。

本研究揭示了多核系统中访存优化的一般性规律，即利用微体系结构的特点和依据工作集的访存特征进行资源的合理分配。本研究即利用了地址映射中存储资源索引位（Indexing Bits）的特殊性来根据应用的特征分配物理资源，进而有效的缓解了原始系统中由于随机分配而导致的资源利用率下降的问题。本文为多核体系结构下内存资源的管理与优化提出较为可行的方案。本研究的主要贡献包括：

（1）本研究首次在真实环境下实现了对DRAM Bank的“页着色”划分机制（Bank Partitioning Mechanism，BPM），进而彻底的消除了多道程序之间在DRAM Bank上的相互之间的访存干扰。BPM支持动态的划分方式，用户可根据工作集的“多样性”和具体需求有针对性的设计和部署不同的划分策略。

（2）本研究提出并首次在真实环境中研究了通道划分（Channel Partitioning）的机制，以进一步缓解线程间在多通道上的访存干扰。此外，本研究设计出了一套基于PMU的支持动态划分框架BPM+（并实现其原型），该框架结合了DRAM Bank划分和Channel划分，形成了在主存储器体系上完整的划分优化的机制。基于此机制，后继研究者可以根据自身研究需求实现其它的以划分为基础的优化机制。

（3）本文首次提出了多级存储器之间协同的“垂直”划分的方式，并揭示了相应的较为完整的划分空间（见第四章A-/B-/C-VP）。随后，本文采用数据挖掘的方式在大量的真实的实验结果中找到了工作集的特点与最优的划分方式之间的关系。本研究为后继的系统优化工作提供了更多的优化选择与参考。

（4）本文首次提出了内存储资源“垂直”管理的概念，并基于此概念设计了“应用特征与体系结构敏感的内存管理（Application-Architecture-Aware Memory Management）”的概念模型。原型系统的实验结果显示，在真实环境下，对于动态变化的工作集，该系统能够根据工作集的访存行为特点选择恰当的内存资源管理策略，通过合理的资源分配方式使系统的性能始终处在较优的状态。

1.8 本文的组织

本文的组织是，第二章介绍与本研究相关的背景知识与相关工作；第三章介绍对DRAM系统的划分机制，包括动态和静态的划分方法和策略；第四章介绍多级存储器之间的垂直划分机制；第五章介绍体系结构特征和应用程序特征敏感的资源管理和分配模型，以及支持该模型的x-Buddy System；第六章对本文进行总结，并展望未来内存储器体系结构的发展和潜在优化可能以及本文提出的系统的潜在应用场景。

第二章 研究背景与相关工作

为了展开后继内容，本章着重介绍与本研究直接相关的背景知识和相关工作。主要包括DRAM系统的结构与相关协议，优化原则，与近期较有影响力的相关研究成果。另外，本章还介绍了工作集访存行为的“多样性”，为后文研究多样化的资源管理策略做铺垫。

2.1 DRAM的结构与相关操作协议

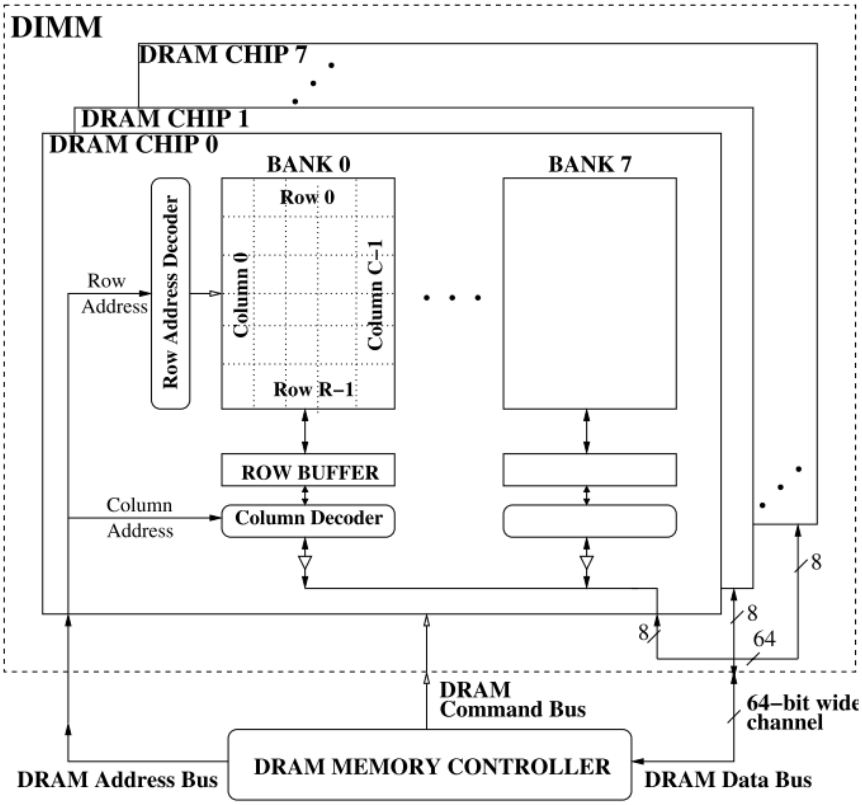


图2.1 DRAM系统的组织结构［68,69,70,72-76］。 注：任何数据的读、写操作都必须经过行缓冲（ROW BUFFER），很多访存优化工作都是围绕行缓冲的局部性进行的。

如图 2.1 所示，以单个晶体管作为存储单元，并以 N*M 个存储单元组成存储矩阵，再以若干个存储矩阵构成 Bank 的 DRAM（Dynamic Random-Access Memory）是目前主流主流计算机采用的内存结构［72-76］。这个结构有一个关于数据读取的特点，那就是如果需要从 Bank 中读取数据，则必须首先将 Bank 中的一行读入到“行缓冲”（Row buffer）之中，之后再打开 I/O 门（I/O Gating），将数据通过数据总线（Channel，通道）传送到芯片的高速缓存中。行缓冲是整个数据操作过程的核心部件，只有将数据读入到行缓冲才能进行读写操作，而每个 Bank 只有一个行缓冲，在一段时间内，只有一行的内容可

以被读或写，因此，对行缓冲的“使用权”，是支持和保证运算的关键。

具体来讲，DRAM 的操作包括以下三个:

1. ACTIVE：读取一整行数据到行缓冲中，只有当数据被读到行缓冲当中才能被使用。

2. READ/WRITE：从行缓冲中读 / 写入数据。

3. PRECHARGE：将行缓冲置为“不活跃”，等待接受新的数据内容从 Bank 中被读到行缓冲当中，此操作是 1，2 两步的先决条件。

由于材质和结构的原因，对 DRAM 的操作必须严格按照协议所规定的时序进行，整个运行过程环环相扣，不能出现差错，如图 2.2 所示。但因为每个具体操作的耗时不同，如果调度不善，很容易引发访存操作的延迟，进而影响系统整体性能。本文将对这些参数进行详细分析。

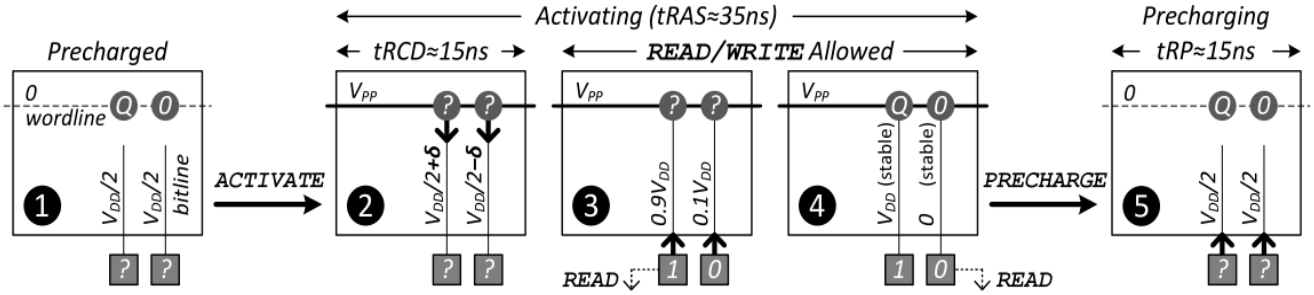


图2.2 DRAM系统的操作流程［39,47］。注：DRAM的每个操作都耗时十几到几十ns，但是CPU每个ns就可能执行若干条指令，多核情况下这个差距将会更大。

ACTIVE Row（行）：如前文所述，在操作数据之前，数据必须以整行的形式从 Bank 中被读取到行缓冲之中。但在这个操作之前，必须先进行 PRECHARGE 操作，以达到状态（1），在此状态下，所有的 bitline 都维持在 1/2 Vdd（如图 2.2 中所示的状态 1）。当接到 ACTIVE 命令的时候，Bank 中相应行的 wordline 的电压将发生变化，如果该行对应的数据位有数据（Cell bit 为 1，图 2.2 中表示为 Q），那么电压将升至 Vdd，否则，电压将降至 0。在这个过程中，由于电压的变化，数据将从 wordline 通过 bitline 开始传输，即开始从 Bank 中相应行的数据单元（Cell bit）向行缓冲中传输数据，状态从（1）向（2）转换。行缓冲的放大器单元（amplifies）在感受到电压的变化时，随即根据电压将数据位调整为 1 或 0，图 2.2 中对应的状态为（2）到（3）。到达状态（3）的时候，数据已经可以被操作了。这个过程中有一个问题，由于在传输过程中电压的变化破坏了 Bank 中对应位原来的电压，因此，Cell bit 处于一个不确定的状态，也就是说原来的在 Bank 中对应行的数据被破坏了。因此，一旦 bitline 上的电压达到稳定状态（Vdd 或者 0），Cell bit 的电压将恢复到原来的状态（restore），这个过程是（3）到（4）的状态，稳定状态是（4）。整个过程是整个 DRAM 操作中最耗时的部分（tRAS），有资料显，这个时间开销是 35ns 左右［39,47］。

READ/WRITE Column（列）：在 ACTIVE 之后，内存控制器（Memory Controller，MC）将发出读或写操作的命令，以及需要读取的列地址。在 ACTIVE 与 READ/WRITE 操作被允许的之间的这段时间被记为 tRCD（大约 15ns）[39,47]，这个时间反映了 DRAM 中数据能够被操作所需要的最短时间限制。虽然这个过程是耗时的（与处理器的速率相比），但是如果随后的访存行为能够继续操作已经在行缓冲中的数据（行缓冲命中），那么时间开销就会被降低，需要的数据能够在最短的时间内被处理，系统性能也会相对提高。[39,47]

PRECHARGE Bank: 如果需要访问一个新的行（该行不在行缓冲之中），内存控制器必须首先发出 PRECHARGE 命令，将整个 Bank 置为 PRECHARGE 状态，如图中（1）和（5）所表示的状态。具体来说包含两个操作，第一，当前访问的行对应的 wordline 电压归零，并将数据单元与 bitline 断开连接；第二，bitline 重新置为 1/2 Vdd，等待下一次 ACTIVE 的命令和操作。这个时间被记为 tRP，大约也是 15ns。[39,47]

为了保证 DRAM 系统的正常运行，内存控制器发出操作的命令必须遵守一些严格的时序（或称为协议），否则就会产生数据错误。例如，如果在数据恢复（restore）之前发出 PRECHARGE 命令，那么将会丢失新数据（写操作之后）或者破坏 Bank 中原始行中的数据（读操作之后）。在这些必须被遵守的时序和协议中，被业内普遍认为是关键因素的是 tRC 和 tWR。[39,47]

tRC 是 tRAS 和 tRP 的代数和，是连续的 ACTIVATE 操作（不同行）之间最小的时间间隔。在极端的情况下，来自 n 个线程的 n 个访存请求如果访问的目标是同一个 Bank（通常不同线程的访存不可能命中相同的行），那么至少要经历 n*tRC 的时间延迟，这有可能是成百上千个 ns。与之形成鲜明对比的是，计算单元在 1ns 内就可以发射若干条指令，主存就成为整个系统的“瓶颈”。因此，从系统优化的角度来讲，在 tRC 不能被降低的情况下（由于材料的因素），如何能够减少 ACTIVATE 操作的次数，进而降低 n*tRC 带来的时间延迟，就成了系统优化的关键。[39,47]

tWR 是指“写恢复”（write-recovery）的耗时。这是由于结构的因素引发的“特殊”问题。由于每次对行缓冲的写操作结束后都会破坏对应的数据位上原有的 bitline 的电压，因此，在写操作结束后，行缓冲需要重新将 bitline 上的电压恢复到稳定的状态（如状态（4）），这个过程就是写恢复，这个额外的时间开销就是 tWR。PRECHARGE 操作不能在写恢复完成之前进行，否则新写入行缓冲的数据将不能被安全的存入 Bank，有可能丢失数据。[39,47]

以上这些操作的命令和时间参数均由内存控制器发出并控制，内存控制器必须严格按照这些协议来运作，否则就会出现数据错误。可以看出，不同的操作有不同的时间开销，并且，这些时间开销是不可被忽略的。从根源上来讲，引发这些时间开销的根源是构成主存材料，如图 2.3 所示，Restore 和 Sensing 这两个耗时的部分是每次将数据读到行缓冲中（ACTIVE）和写数据后所必须的操作，都是由于 DRAM 特殊的材质而必须的操作。另外，DRAM 这种“易失性”材料，必须通过稳定的电压和定期的刷新（refresh）

来维护数据的正常存储。Jamei Liu 和 Onur 在 ISCA'12 上新的研究结果表明, 刷新操作给系统带来的在能耗浪费和性能损失方面的问题，将会随着存储数据量的增大给系统增加愈发沉重的负担［64,78］。尽管从某些程度上来讲，这些问题可以通过某些"折衷"的"智能"方案来缓解，但除非更换材料或结构，否则这些问题是很难被彻底解决的。因此，从系统结构的角度来讲，主存"先天"的劣势严重制约着系统的整体性能。

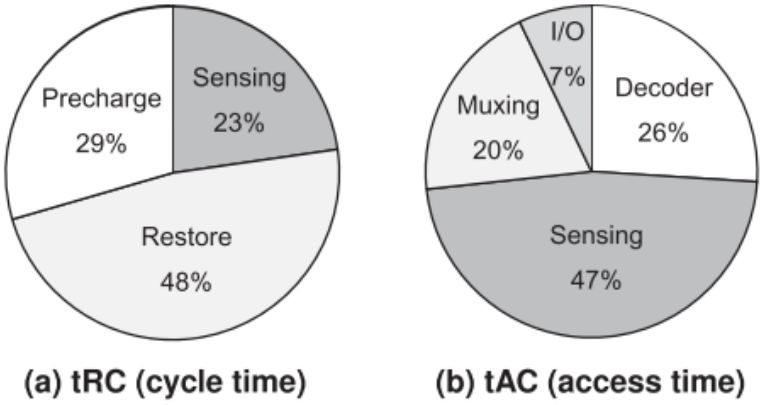


图2.3 DRAM系统操作的耗时百分比图例［39，77］。

2.2 新材料与新存储结构

关于新材料的研究与讨论是近些年学术界和工业界广泛讨论的话题。例如，Phase-change memory（PCM）采用硫化物玻璃（Chalcogenide Glass）作为存储介质，通过电流发热来切换于结晶（crystalline）和无定型（amorphous）的两种状态，以代表 0,1 的两种数据状态。这种材料是非易失的存储材质，不需要定期的刷新（功耗极低）可靠性高，而且具有更好的扩展性［63，66］。另外，研究工作者还将现在的 2D-DRAM 扩展到 3D（3D stacked）[7,63,66,94,102,111]。真正的以 3D 的形式组织存储单元的 3D DRAMs 是由 Tezzaron 公司首次发布的，理论环境下模拟器结果显示，最高可以提高 32％的 tRAS（5-layer 的 3D stacked）[42]。再之，带宽（Bandwidth）的极为有限让计算机的设计者势必要为高性能计算机拓宽带宽。而在传统的工艺下，带宽的扩展，又会导致 CPU 引脚数目(Pin-Count)的增加，这种进退两难的窘境让人感到非常的沮丧。IBM 成为了摆脱这个"带宽维谷"的先行者。她利用了近几年出现的高速串行信号作为带宽的介质，比如 Power 7 使用的是 5Gb/s 的高速串行信号，使得整个芯片可以达到 100GB/s 的访存带宽，当前 DDR3-1600 的 12.8GB/s 带宽已明显不可与其同日而语。然而，这些技术还没有被普及到我们所使用的每一台计算机上。每一个新技术本身也有其自身的问题[112,113]。比如 PCM 面临的造价问题，以及在高温情况下出现的漏电现象等基本材料方面的问题，等等，都是新技术需要面对的挑战，而走向成熟则更需时日，本文将不再展开。

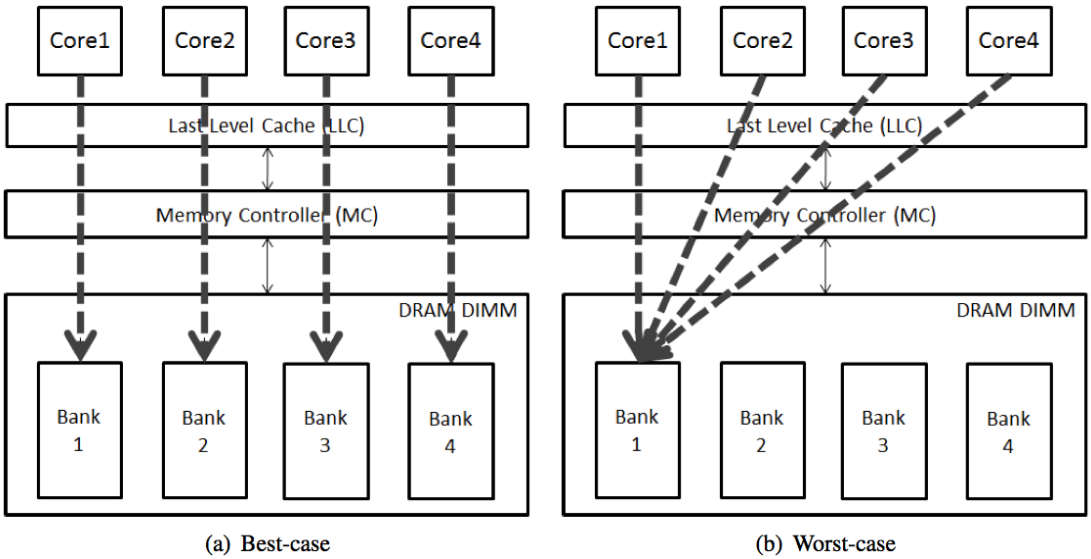


图 2.4 理想情况下的 Bank 并行模式与极端情况下的对比［18］。

2.3 DRAM 性能优化的原则

本文继续讨论的问题是，在现有系统中，如何提高 DRAM 的吞吐量？这主要涉及两个体系结构设计上的原则。第一，提高 Bank 并行（Bank Level Parallelism）。这是体系结构设计者能够想到的最直接有效的办法。如图 2.4 （a）所示，虽然每个 Bank 在一个时刻只能打开一个行，但是系统中所有的 Bank 可以同时工作，访问不同 Bank 的访存请求可以并行的被处理，如上文所提及的那些 DRAM 操作均在每个 Bank 上可以并发的运行。假设当前系统有 n 个 Bank，现在有 n 个来自不同程序并且访问不同 Bank 的请求，那么这 n 个 Bank 的 ACTIVE Row 的操作可以同时进行（虽然 cmd 命令信号还是串行的，但这些信号的响应时间极小且不造成延迟），那么 tRC 这个时间延迟就被掩盖了，理论上讲，只要经过一个 tRC 时间（实际会略高于这个时间），n 个请求均可被响应。如图 2.4(a)所示即为这种情况。第二个原则是提高行缓冲的“局部性”（Locality）。如前文所述，ACTIVE 行的操作是非常耗时的，那么系统需要尽可能的提高已打开行的利用率。

这两个原则启发计算机设计者一方面要从整个系统的角度出发，提高系统中 Bank 的并行度，让系统中所有的 Bank 都尽可能的处在利用率较高的状态；另一方面也要尽可能保证每个 Bank 处于服务状态（响应读 / 写操作），而非浪费大量的时间于 ACTIVE 或 PRECHARGE 操作。我们可以想像，理论的最优情况下，在每个节拍都会有数据（8B）从 DRAM 系统中发出，64B（1 个 Cache 块）的数据在 5ns 既可被输出（DDR3-1600，带宽峰值 12.8GB/s）[38]。但理论上的理想模型在实际系统中往往会被各种复杂情况所干扰。尤其在当代主流的多核计算平台上，由于线程之间的访存干扰，行缓冲的命中率已经下降到较低的数值［51,90］。如图 2.5 所示，随着计算单元数目的增多（并发的线程数目也相应增多），行缓冲的命中率在持续下降，带宽的利用率通常仅在理论峰值的 50% 以下［61］。

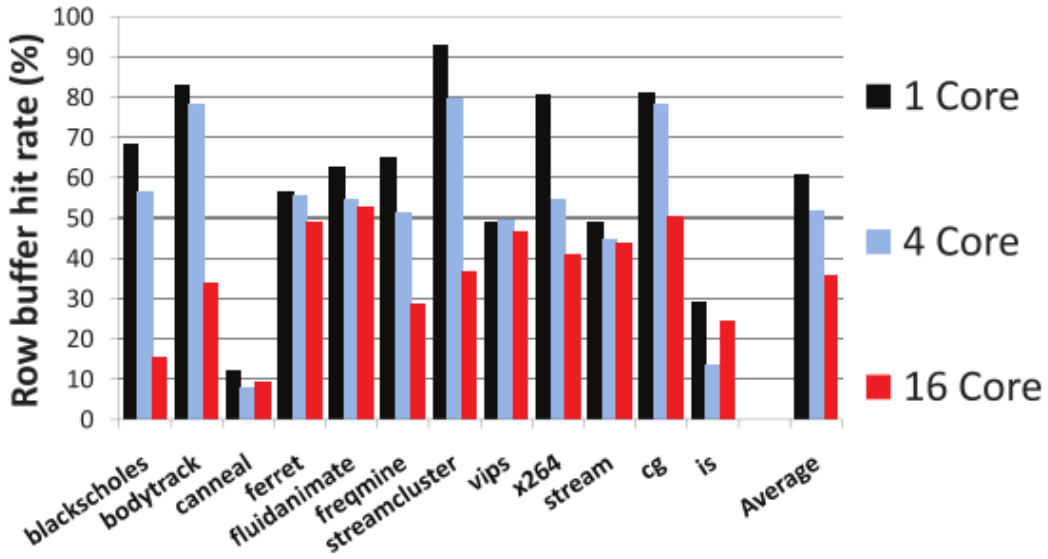


图 2.5 在 1～16 核的机器行缓冲命中率的对比［51,90］。注：下降趋势明显。

另一种情况是，经常会出现对某个或某几个 Bank 的访问相对集中的情况（“Hot Bank”），如图 2.4 中 b 情况所表示，而系统中其它的 Bank 则处在负载不均衡且利用率较低的状态，系统性能也因此处于这种较低的状态。本文以这些问题为背景，目的在于提出一种优化机制，提高系统性能，降低 DRAM 系统作为“瓶颈”的负面影响。

2.4 共享“内存储器”系统的计算机体系结构

现代计算机体系结构支持资源的共享。通常来说，如图2.6所示，整个“内存储器系统”都是被共享的。本文所指的“内存储器系统（或资源）”包括片上高速缓存和主存系统（由若干DRAM Bank构成）。对于当代多核结构的计算机来说，“共享内存”（Shared memory）通常指的是共享的最后一级Cache（LLC）和主存系统（Shared Main Memory System）。位于一个或多个芯片上的若干计算单元（Computing Unit, Core，核）能够共同访问这些资源，享有同等的使用权［30,38］。

资源的“共享”是现代计算机体系结构设计的重要原则，它使得系统富有“弹性”，能够潜在的为并发运行的程序提供高效能资源支持。在多核体系结构下，资源共享机制引发计算机体系结构设计者更为深入的思考。对于共享主存（Main Memory），其优势在于，在计算环境中，多道并发的程序能够同时访问由操作系统统一编址的全局存储空间，计算单元之间能够通过快速访问同一块可见的存储空间进行高效的通信，从而避免了冗余的数据拷贝。对于程序员来讲，由于存储资源对所有的计算单元都是唯一且全局可见的，因此，编程也相对简单（与分布式非共享存储相比）［38,106］。但共享主存带来的问题也不容忽视，我们前文提到的“内存墙”问题，即 CPU-Memory 之间的瓶颈，在共享的情况下显得更为严重，使得共享主存结构的计算机的扩展困难，尤其当计算单元的数量增

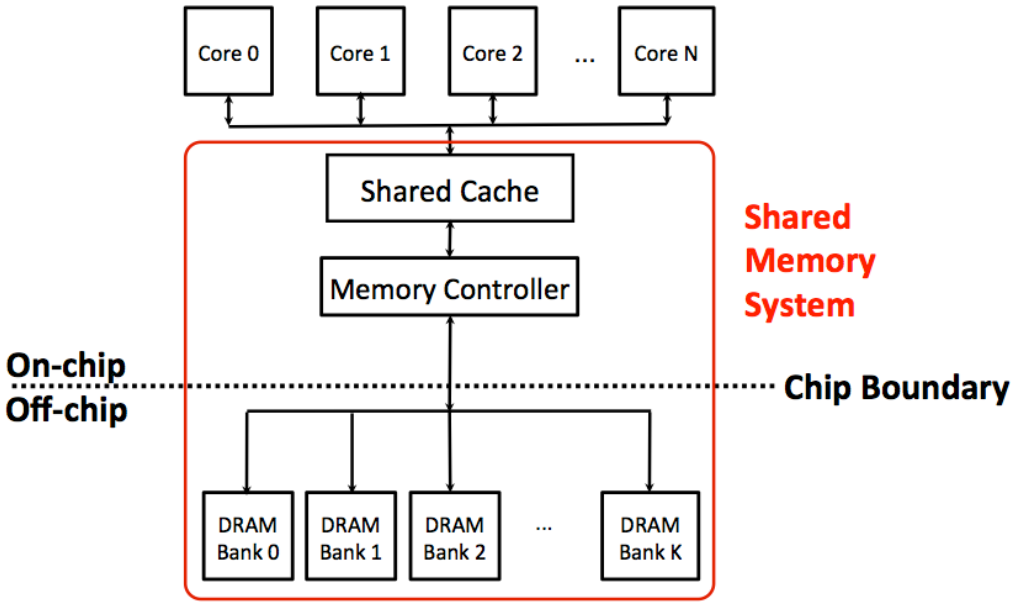


图2.6 多核计算机的共享内存储资源（Shared Memory System）。注：包括共享的LLC。

至几十甚至上百的时候（如 Many-core 结构），计算性能很难得以充分的发挥，实际上造成了严重的计算资源的浪费。

“资源共享”所带来的一个严重的问题是“竞争”。如前文所述，“内存墙”问题将会被“放大”，即本来就天然存在的计算单元与存储资源之间速率上的差异又被增大了。对于DRAM来说，线程间的竞争则意味着在资源使用上发生”冲突”，”冲突”则必然会导致相互干扰，以至于破坏行缓冲局部性（这是现代计算机体系结构设计的重要原则），造成行缓冲的“颠簸”（Thrashing），拉大了访存延迟。这个问题又不仅存在于DRAM Bank这一个层次，在共享LLC这一层级依然会由于线程之间的相互干扰而引发Cache块（Cache Block，Cache Line）的颠簸。

以主存为例，理论上，由于访存行为的“交替”性和Buddy System对资源分配的“盲目”性，任何一个进程使用的物理页面可能来自任意Bank，即任何进程都有可能访问到系统中任何一个Bank。此外，据上文的分析，由于通常一个物理页面即对应Bank中的一行，故除非特殊情况，不同进程的访存几乎不可能命中同样Bank的同一行。因此，从全局的角度来看，在任意Bank上均可能发生访存冲突。每发生一次行缓冲的颠簸大概要经历100ns的延迟，因此主存系统优化的重要目标是减少行缓冲的颠簸并提高行缓冲的利用率。

2.5 访存优化相关工作

针对优化行缓冲局部性的目标，国内外高校和研究机构展开了广泛的研究工作。总的说来，可以被概括为三类。（1）优化内存控制器中的硬件调度。S. Rixner［87］在 2000 年初首次提出了通过优化内存控制器调度逻辑的方式来提高访存效率，其核心思想是，在一个调度窗口内部，将能够命中已经打开的行缓冲的访存请求赋予较高的优先级。因

此系统可以达到连续的行缓冲命中的状态。这种做法看起来是非常合乎访存优化原则的，尤其在单核时代非常有效。在此文之后，学术届又相继提出了很多诸如此类的优化方法。比较有代表性的是卡内基梅隆大学的 Onur 教授的团队和康奈尔大学微系统实验室等提出的一系列访存调度算法［11,23,28,30,33-35,42,47-49,68-70］，这些方法可被概括为通过不同的策略调整 MC 调度队列中访存请求的序列，进而降低线程间的访存干扰，保证行缓冲的局部性，并同时兼顾公平性（Fairness）和吞吐量（Throughput），调度策略中有时也会带有细粒度的优先级上的考虑。他们的工作通常采用内部的模拟器进行。例如，PAR-BS［68］方法将一个时间段内的访存请求重新排列并调度，使得不同线程的访存请求不再无序的混杂在 MC 的调度队列里，取而代之的是按照线程的等级（Rank）在一个小的时间段里统一且无干扰的调度每个线程的所有请求。Rank 的制定则有赖于当时系统情况的综合考虑，包括此线程当时 Bank 的并行度，对每个 Bank 的负载等即时产生的参数。在整个的执行过程中，等级是可能因为访存行为的不同而发生变化的。PAR-BS 在降低线程间访存干扰的同时，也兼顾了 Bank 的并行性，并且可以随着系统负载的变化而自适应的调整优化方案。这一方法在 ISCA'08 提出，此后一直是学术界讨论的热点。Onur 的研究进一步指出，访存的干扰通常会发生在不同访存行为类型的线程之间。因此，Onur 团队的 Yoongu Kim 在 Micor'10 发表了学术论文 Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior［48］，这一研究在之前工作的基础上首次把区分不同线程的访存行为（Memory Access Behavior）作为优化的主要手段，在运行时将并行的线程划分为访存密集 (Memory-Intensive) 和访存非密集（Memory-non-Intensive）两类，并始终赋予后者较高的优先级。因此，与 PAR-BS 相比，TCM［48］更为有效的提高了系统的吞吐率，因为访存非密集型的程序对系统的吞吐率影响更为显著，并且 TCM 的公平性也更好。图 2.7 中展示了几种先进的调度算法的对比。"调度"是一个有效的方式［36,56,117,119］，但是此类方法通常需要修改硬件才能实现。

（2）任务级（或线程级）软件调度优化机制。研究表明，通过软件方式的任务调度也能达到优化存储器性能的目的［29,80,100,101,111,115］。这种方式通常需要调整线程的优先级，故通常通过编写／修改系统软件的方式来实现，而极少涉及硬件的修改。这一特性决定了虽然任务级调度的可用性要高于前者，但其所能发挥的空间则远不及前者，因此学术界对此研究的兴趣并不高涨。Di Xu［111］通过带宽敏感的线程调度的手段较为有效的缓解了带宽的不恰当竞争而引发的若干问题，进而提高了整体吞吐量，是目前对此类调度手段较新的研究成果。具体做法是，通过 PMU 监控并发的进程占有的带宽，并利用进程调度的手段使整个系统的带宽利用率趋于"均衡"（即当前工作集的平均带宽需求）。这一做法源于一个有趣的观察：带宽的利用率在极小的时间片内（1～10ms，Linux CFS 进程调度的默认时间片是 4ms）仍然有较大的波动，并且，如果调度算法在某一个时间段内使带宽的利用达到带宽峰值（Peak Bandwidth），那么将会引发超线性的性能下降。因此，调度算法将调度的时间片减小，并使得带宽利用率始终维持在某个"均衡"的状态下，就是合乎逻辑的做法了。这一作法虽然在真实的系统上可以取得平均近 5%的

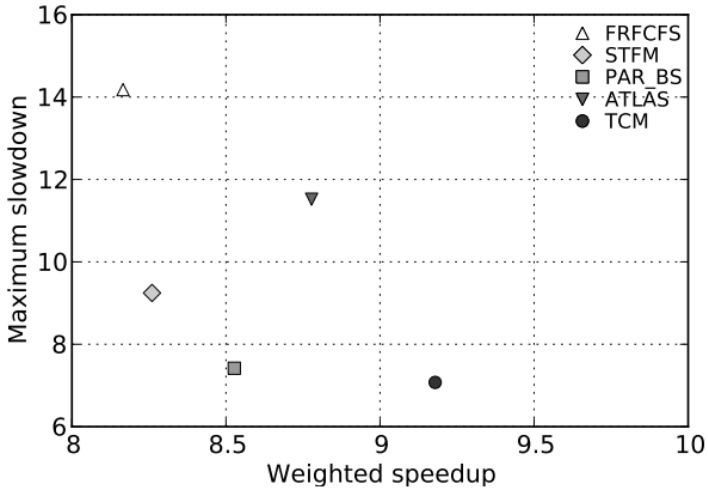


图 2.7 State-of-the-Art 的访存调度算法的对比。注：越向右下方的越好。

性能提升，然则又引发了新的问题。首先，调度器依赖的 PMU 是否在每一台计算机上都能被成功的部署？不同机器采用的处理器架构是不同的。其次，调度器只是部分缓解了带宽上的压力，对于存储器上的访存竞争则并没有被消除掉，性能提升并没有发挥到最大的程度。再者，针对带宽的调度是否可以与针对提高局部性的调度策略相结合，从而产生正相关的合力？局部性的优化是 DRAM 系统优化的原则，如果这两种方式能够有机的结合起来，系统性能可能会进一步被提升。另外，调度也可以用以针对特殊的服务需求来提高系统的公平性（Fairness, 或更概化为 QoS）[22，40]，本文则不再赘述。

（3）软硬结合资源分配的优化，如Cache, Channel, Bank上的划分和XOR技术的使用。为了降低多线程在Cache上的相互干扰，Jiang Ling和Xiaodong Zhang提出了采用“页着色（Page-Coloring）”的方式消除了Cache上的线程间干扰（如图2.8所示）。他们的工作在真实机器上开展，并支持动态的划分策略［57]，详见2.6节。实验结果表明，对于某些工作集，该方法能够达到较好的效果（10％左右整机性能提高），但同时也存在一些工作集会因为Cache划分导致性能的“负”增长。Wei Mi［79］首次提出了多道程序间Bank划分的思想，该方法依据物理地址位中的Bank索引位来着色，像划分Cache一样将Bank划分成不同的组，每个程序只访问自己所属的Bank，因此程序间在Bank上的竞争就消失了。实践证明，划分技术有利于维护和提高Cache和DRAM的局部性，对于DRAM Bank来说，本来相互干扰的来自多线程的访存现在被分别隔离在仅属于自己的一组Bank中，每个线程的访存都是相对独立的且互不干扰，因此每个程序内部的局部性可以不被破坏。但Wei Mi认为对Bank上的划分会增加在Cache上的竞争，因此又同时采用了XOR的方式来减少这些冲突［118]，这个方式需要涉及硬件的地址映射方式的修改。S. Prashanth和Onur在2011年［82］提出了将线程的访存行为划分到不同的Channel中，如果有多个线程被划分在一个Channel内部，则尽可能的在Bank上将其区分。我们认为这个工作是Bank划分另一种实验方式。Bank划分的思想在体系结构的顶级会议上多次出现。在HPCA’12上，Min Kyu Jeong［85］等学者再一次研究了这个问题。与前人不同的是，他们切入的角度是在多核环境下“平衡DRAM的

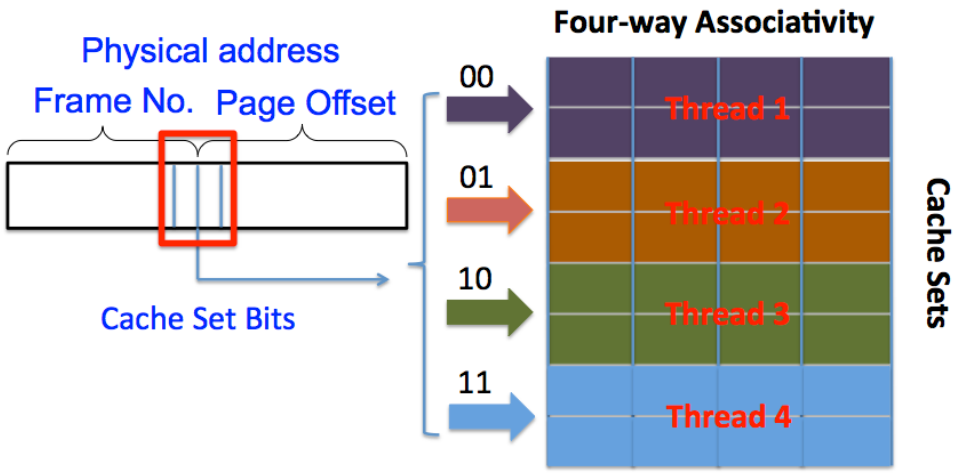


图2.8 “页着色”划分Cache的原理图。注：图中的4个线程分别仅使用属于自己的一部分Cache Sets，互相之间再无干扰。但每个线程能够使用的资源量却下降了。

局部性与并行度”，并且结合了Sub-Rank技术以提高并行度，他们的方法也是一个典型的软硬结合的方法。上述三者的工作都是在内部的模拟环境下进行实验的，实验环境并没有向外界公开。他们的工作对划分的研究并不是彻底的。比如说，在Bank上的划分是否真的会导致Cache的冲突？能否同时对多级存储器同时进行划分，从而使得程序的访问能够同时在Cache和Bank上同时保持局部性？这一技术该如何在真实的机器上被部署？对真实机器上的性能，能耗产生什么样的影响？与调度策略是正交的吗？等等。这些问题在我们的研究中会一步步的探讨和解开。

2.6“页着色”技术的背景与前景

业界很早就意识到了线程间在Cache上的访存冲突会造成系统性能的下降。因此，很自然的就想到通过隔离线程间在Cache上访存行为的方式来优化系统性能。研究者发现可以利物理地址映射中索引Cache的地址位来将物理页面分类，这一过程被形象的比喻为“着色”，将物理页面分成不同颜色的类别。比如，图2.8中，假设描述物理页的PFN（Physical Frame Number）中有2为是Cache Set的索引位，那么系统中所有的物理页面可被分为4中颜色（分别被表示成00、01、10和11）并分别将每种颜色赋予一个线程。在线程运行发成“页缺失”的时刻，系统在全部物理页面中为该线程选择并分配仅属于其对应颜色的页面，则该线程使用的Cache资源即被限定在某个范围内而不再与其它线程发生冲突。

对页着色技术的研究是传承有序的。如表2.1中所列举的在学术界十余年间公开发表的有代表性的学术文章，研究了从静态划分到动态策略的多种问题，研究方法也从采用模拟器到真实系统建模，无所不包。时过境迁，现在的多核系统上运行的工作集已经和十年前的情况有所不同，程序对Cache的需求量越来越大，因此，在Cache上的划分很有可能会损失系统性能。另外，现在的工作集展现的越来越明显的访存密集型的特征，对DRAM层造成很大的压力。这些原因也促使了研究人员将页着色技术应用于消除主存系统上的干扰，

在另一层次思考访存优化的问题。还有，在“DataCenter as Computer”的计算密集型大数据时代，资源管理显得尤为重要，页着色技术在资源管理方面的优势能够为数据中心的资源管理与服务性能优化提供必要的手段和方法，在新的计算环境下“焕发生机”[67]。

表2.1 “页着色”技术的发展历史与现状。注：关于C-/B-/O-bits的定义请参见第四章。

Research	Methodology	Used Color Bits	Brief Contributions
Sherwood et al.	Simulation w/o OS	C-bits	Proposed page-coloring based cache partitioning (PCCP)
Mi et al.	Simulation w/o OS	B-bits	Proposed page-coloring based bank partitioning (PCBP)
Joeng et al.	Simulation w/o OS	B-bits	Augmented PCBP with memory sub-ranking technology
Tam	Real Systems w/ Linux	C-bits	Implemented static PCCP on real systems
Lin et al.	Real Systems w/ Linux	C-bits + O-bits	Augmented PCCP with dynamic color adjustment
Soares et al.	Real Systems w/ Linux	C-bits	Augmented PCCP with isolating high-pollution pages
Zhang et. al.	Real Systems w/ Linux	C-bits	Augmented PCCP with coloring only hot pages
Liu et al.	Real Systems w/ Linux	B-bits + O-bits	Implemented PCBP on real systems

2.7 程序访存行为的多样性

由于人们不断增长和变化的计算需求，程序的访存行为也逐渐呈现多样性。在内存储器的范畴以体系结构的视角来看，我们可以观察的到现象是，程序对存储资源的敏感度是多样化的。如图2.9所示，随着分配给程序的Cache资源多少的变化，不同程序的性能变化也不同。这三个程序是非常具有代表性的，mgrid基本不变化（不敏感）；art的前半段变化激烈，但后半段持平；mcf则始终呈现上升的趋势，它对Cache的容量最敏感。

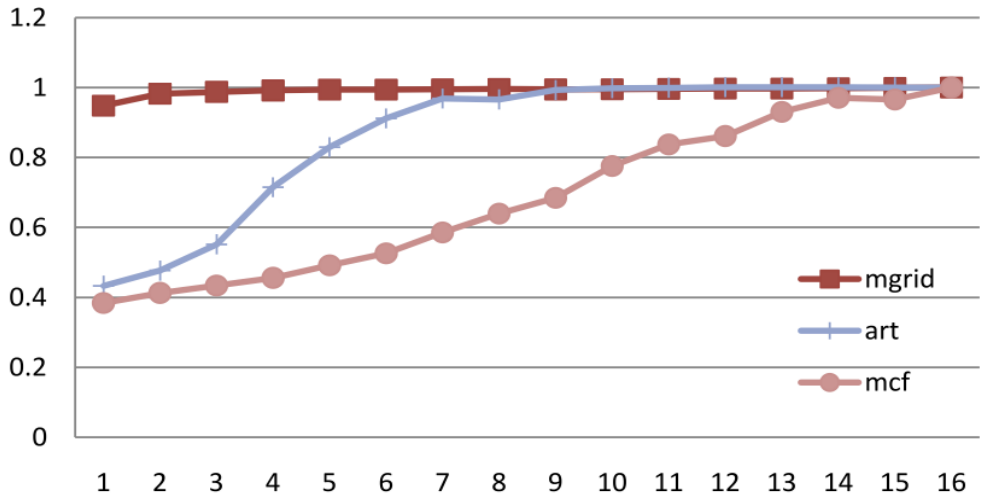


图2.9 三个SPEC基准测试程序对Cache资源的敏感度[122]。注：x轴表示分配给程序的Cache资源量，y轴表示系统性能的变化。这三个程序展示了完全不同的三种Cache行为（敏感、不敏感与条件敏感），在多线程并发的情况下，访存行为将会更加复杂。

对Cache的敏感性是多样性体现的一个方面。另一方面，多样性可以通过对主存资源的需求反应出来。如表2.2所示，通过更为具体和细化的参数，可以看出每个程序对资源的需求、访存特点都各自不同。例如，业界通常将MPKI>＝1的程序定性为访存密集型（Memory Intensive），将小与1的程序定性为访存非密集型（Memory Non-Intensive）或者计算密集型。这两类型程序对带宽的需求有很大差异，对系统吞吐量的影响也不同。因此，在多线程并发的多核、众核时代，包含若干个线程（众核情况下可能会达到上百个线程）的工作集的访存行为也将表现的更加复杂。

表2.2 SPEC2006中程序的访存行为相关参数［48］
注：(MPKI: Misses per kiloinstruction, RBL: Row-buffer locality, BLP: Bank-level parallelism)

#	Benchmark	MPKI	RBL	BLP	#	Benchmark	MPKI	RBL	BLP
1	429.mcf	97.38	42.41%	6.20	14	464.h264ref	2.30	90.34%	1.19
2	462.libquantum	50.00	99.22%	1.05	15	435.gromacs	0.98	89.25%	1.54
3	437.leslie3d	49.35	91.18%	1.51	16	445.gobmk	0.77	65.76%	1.52
4	450.soplex	46.70	88.84%	1.79	17	458.sjeng	0.39	12.47%	1.57
5	470.lbm	43.52	95.17%	2.82	18	403.gcc	0.34	70.92%	1.96
6	459.GemsFDTD	31.79	56.22%	3.15	19	447.dealII	0.21	86.83%	1.22
7	482.sphinx3	24.94	84.78%	2.24	20	481.wrf	0.21	92.34%	1.23
8	483.xalancbmk	22.95	72.01%	2.35	21	444.namd	0.19	93.05%	1.16
9	471.omnetpp	21.63	45.71%	4.37	22	400.perlbench	0.12	81.59%	1.66
10	436.cactusADM	12.01	19.05%	1.43	23	454.calculix	0.10	88.71%	1.20
11	473.astar	9.26	75.24%	1.61	24	465.tonto	0.03	88.60%	1.81
12	456.hmmer	5.66	34.42%	1.25	25	453.povray	0.01	87.22%	1.43
13	401.bzip2	3.98	71.44%	1.87					

2.8 本章小结

现代计算机系统访存优化问题的实质如何处理“多样性”的问题。但是，如前文所述，现代计算机目前采用的还是单一的类FR-FCFS的访存调度算法，并且，在应用层的优化空间也非常有限。在当今包含DRAM主存的计算环境中，通过“页着色”进行访存优化是一种可靠的有效的机制，也是本研究所采用的手段。

第三章 BPM/BPM+: DRAM 系统的划分机制与策略

目前，计算机的主存（Main Memory）系统是由若干DRAM Bank构成的。在多核体系结构下，这些Bank被运行于多核之上的多道程序所共享，如前文所述，这将引发在DRAM Bank上的访存干扰，增大访存延迟，进而影响系统的整体性能。本文采用“划分”技术隔离多道程序之间的访存行为，进而在根本上消除并发程序之间的访存干扰。这种做法的优势在于：第一，避免了很多复杂的访存调度算法所引入的开销；访存调度算法和机制是近些年学术界讨论的热门话题，但由于很多State-of-the-Art的调度算法存在着诸多不确定的因素而难于被产品化。例如，前文提及的TCM［48］调度策略，不但需要引入额外的4K（最小）的存储空间，还需要通过辅助硬件来分别记录很多关于进程在运行时的信息，并且在调度时也需要复杂的逻辑支持，因此，TCM引入的开销很难被衡量和优化，为产品化的设计与实现带来困难。相比而言，“划分”技术避免了上述问题，具有有较强的实用性。第二，无需辅助硬件设备支持。这将有利于实际环境中的部署和扩展。如前文所述，本文采用的方式基于页着色的扩展，是纯粹操作系统级的工作，只要目标机器能够被安装Linux操作系统即可。

3.1 DRAM 系统划分的意义

在详细介绍划分技术之前，本章首先讨论如下几个问题，以揭示存储器划分技术存在的意义：（1）资源的分配方式决定着系统的性能。在运行时，系统给每个程序如何分配Bank，将决定系统的整体性能。在系统软件层，分配的任务由操作决定［55］；在硬件层，分配方式由内存控制器等硬件设备控制内存的地址映射来完成［38］。为了提高Bank并发度进而最大限度的提高系统吞吐量，现代计算机一般采用“交替”（Interleaving）的方式将程序连续的访存请求分散到若干个不同Bank［60,61］，优势在于最大化了可并发输出数据的Bank的数量，即最大化Bank并行度（Bank Level Parallelism，BLP）。此方法在有些情况下是有效的，例如，对于某些由访存非密集程序组成的工作集，其总体的访存压力较小，具体在每一个Bank上的访存竞争也不会非常激烈，因此，提高Bank并行度将不会在全局造成强烈的线程间访存干扰，而系统总体性能将得益于Bank并行度的提高；但是，对于某些访存密集型的工作集，对整个DRAM系统的访存压力都很大，因此，完全盲目的“交替”方式将有可能在系统的每个Bank上都造成相对较大干扰，潜在的引发激烈的访存竞争，抵消BLP带来的好处。综上所述，现代多核计算机需要更优的资源分配方式。

（2）Bank的数量越多并不意味着性能越高。对于主流的计算机系统，在能够满足容量需求的情况下，程序的性能会不会随着Bank数量的增多而增高？一般情况下，程序需要多少个Bank就能够达到最优性能？回答这个问题的前提是我们认为现在的计算机系统采用的“交替”访存是一种合理的内存分方式，因此，我们将这种方式作为基准参考。我们进行了

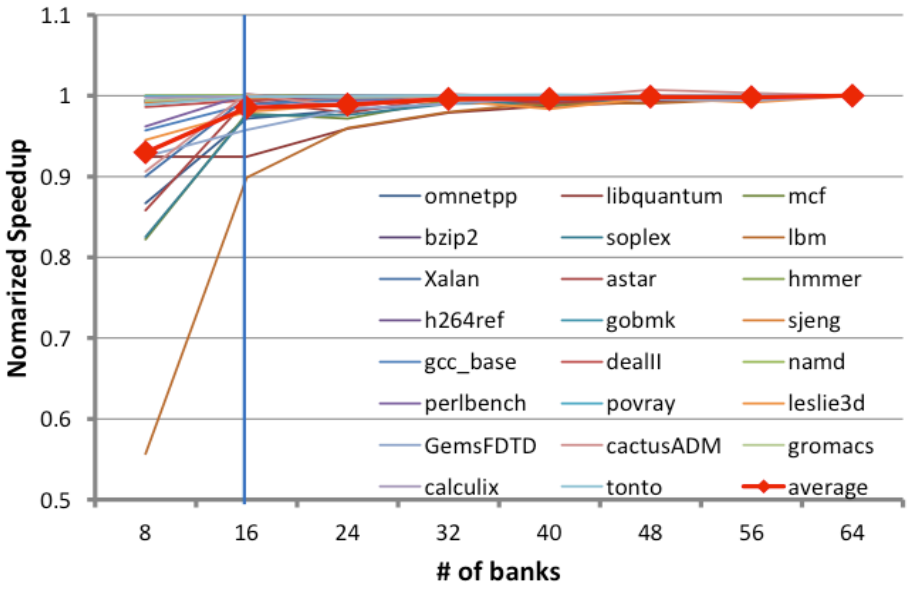


图3.1 单个程序的性能变化与所分配的DRAM Bank的数量的关系

一组实验（实验环境如27页所述），来揭示程序性能与实际所需Bank数目之间的关系。我们采SPEC2006［123］作为测试用例。在本实验中，对于每个应用程序，我们以递增的方式逐步赋予其不同的Bank数目，并分别观测每个程序在每种情况下所需的运行时间。例如，对于470.lbm，我们将其运行8次，每次递增的赋予其8个Bank，初次8个Bank，随后是16个Bank，32个Bank依此类推，最后是64个Bank，在8种情况下分别运行，并记录运行时间。我们用此方法测试了SPEC2006中的所有的基准测试程序。实验结果显示，对于90%分程序而言，16个Bank（每个Bank容量125MB）即能满足其需要，即程序的运行时间与其在所有Bank上“交替”访存的情况下无明显差距；对于大部分程序，在8个Bank的的情况下也能达到90%以上的性能，即与Bank“交替”访存的情况相比，仅有小于10%的性能损失。通过这组实验我们可以归纳出结论，程序对Bank数量的需求是有限的，在满足容量需求的情况下，性能并不随着这Bank数目的增多而持续提高。因此，如果盲目“交替”的话，将不会提高系统的性能，反而会引发线程之间的访存干扰，即“交替”的“弊大于利”。

（3）业界期望功能性强、实用性强的访存优化机制。目前被工业界广泛采用的内存控制器调度算法还比较简单，而学术界提出的很多较先进的访存优化算法由于种种原因还没有被工业界广泛采用，因此，对访存行为的优化依然有较大的改进空间。举例来说，目前被广泛使用的依然是FR-FCFS［87］调度算法。该算法的思想源于First Come First Service (FCFS), 即内存控制器优先调度先发出的内存请求。但这个方法的缺陷在于没有考虑到DRAM行缓冲的局部性。比如说，前一个访存请求已经在行缓冲中打开了一个行（如上文所述，打开一个行在DRAM操作中将引发较大的开销），那么应该尽可能的让调度窗口中潜在的，能够命中该行的后继的请求最大化的命中该行。FR-FCFS是对FCFS的改进，在保持先来先服务的基础上，在一个时间窗口内调度所有访存序列以尽量命中已经打开的行缓冲。但是这种“单纯”的最大化行缓冲利用率的调度方式固然容易实现但在两个重要的指

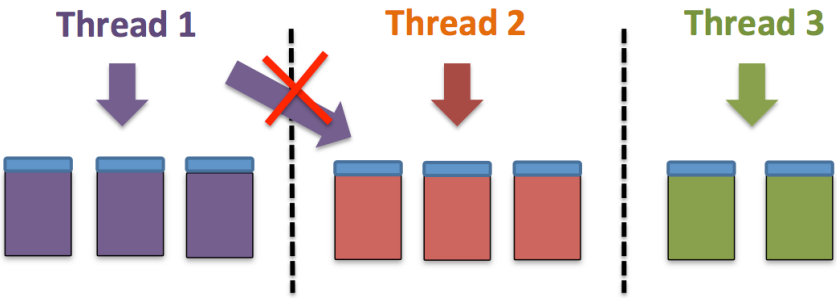


图3.2 BPM的原理示意图。注：线程不能访问不属于自己的Bank资源。

标公平性（Fairness）和吞吐量（Throughput）表现的却都不是很理想［48］。国内外研究团队对访存优化的研究都显示出浓厚的兴趣，各种方式层出不穷，但总的来讲都是围绕提高公平性和吞吐量这两个指标进行的。例如，前文介绍的PAR-BS和TCM，采用相对复杂的硬件逻辑在区分线程行为的基础上调度其访存请求，在模拟器上收到较好的效果。但是，真实的系统还没有采用如此复杂的调度算法，有信息表示，原因在于运行时的开销和复杂的硬件逻辑以及算法必须的额外存储，这使得对内存控制器芯片的设计更为复杂。而且，目前的内存控制器与处理器和高速缓存都被设计共存于同一个芯片之上，过于复杂的内存控制器构成了芯片内部的不稳定因素，降低良品率。

3.2 BPM—Bank-level Partitioning Mechanism 的核心思想

访存优化的方式各有利弊，但优化的目标有两个：吞吐量（Throughput）和公平性（Fairness）。在学术界，吞吐量用来衡量计算机的计算性能，通常用“加权加速比”（Weighted Speedup）来量化［48］；公平性用来表示系统所能提供的服务质量，通常由工作集中性能下降最大的程序来表示（MaxSlowdown）［48］。先进的调度算法虽然在性能上表现不俗，但由于硬件芯片设计与生产的特殊性，所以，工业界对这类算法的使用通常持谨慎态度［60,61,62］。本文的目的在于，提出一种简易的访存优化机制，该机制能最大化的降低程序间的访存干扰，并且需要简单易行，能够给工业界和学术界提供有前景的优化方案。

减少访存竞争，降低线程（进程）之间的访存干扰，是访存优化的核心。如前文所述，调度算法可以降低线程间的访存干扰，但是行缓冲的“颠簸”还依然存在，也就是说程序间的访存干扰还不能彻底的被消除。我们的研究希望找到一种方法能够将访存干扰在根源上彻底的消除。

本研究提出一种在根源上消除线程间访存干扰的机制：**BPM** (Bank level Partitioning Mechanism)，即DRAM Bank划分机制。其核心思想是，如图3.2所示，将系统所有的DRAM Bank划分成若干组（示以不同颜色），让每个运行的程序（线程）使用且仅使用属于它自身的那一组Bank，那么程序间的访存干扰也就自然消失了，程序自身的行缓冲局部性得以保证。更为形式化的表示为，BPM将全部的DRAM Bank划分为若干个互相没有交集的子组SUBi(bank)，并且，$\bigcup(SUB_i(bank))=Bank$。需要注意的是，在每个SUBi(Bank)内部，访

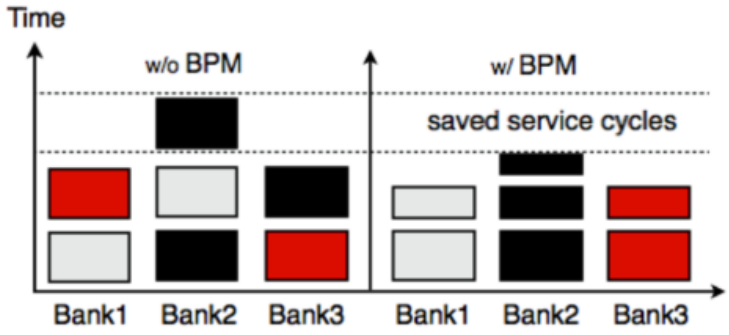


图3.3 BPM的使用效果图。注：不同颜色的色块代表来自不同线程的访存请求。

存行为依然满足“交替”的行为特性。

如图3.3所示，在不采用BPM的情况下，来自不同程序的访存请求被交替分散到所有的Bank上，由于线程间相互干扰引发的行缓冲颠簸将导致部分访存请求的延时较长；但相比较之下，由于采用了DRAM Bank划分技术（w/BPM），消除了程序之间的访存干扰，每个程序的行缓冲局部性都有所提高，因此，访存请求的响应延迟相对减小，有利于总体性能。从访存协议和DRAM内部操作等微体系结构的角度来看，Bank划分将有可能直接提高访存请求的响应速率，图3.3还向我们展示了在使用BPM的情况下（理想情况），每个Bank上的访存请求都节省了近1/3的响应时间，每个线程的运行速率都有可能因此而提高。

3.3 BPM 的设计与实现

3.3.1 Bank bits 和实验环境

如前文所述，“页着色”技术通常被用来划分共享LLC。对于主流的多核计算机来说，在物理页面的地址映射（或物理页帧）中有几位代表LLC Set的索引，由这几位构成的多种组合即形象的被看作为不同的“颜色”。例如，某系统如果有3个这样的索引位，则可以构成8种不同颜色，分别被记为000～111，LLC就因此被划分为分别由这8种颜色所代表的8份。在程序运行时，物理页面分配系统将且仅将属于某一种颜色的物理页面分配给一个进程，由于物理页面与LLC的关联性，该进程则仅使用属于某一种颜色的LLC Set。因此，当每个进程仅使用属于自己的那部分LLC资源而不干扰别的进程的时候，进程间在LLC上的竞争也就消失了。实验证明，在某些情况下，该方法能够提高系统的整体性能。

有鉴于此，为了消除进程间在DRAM Bank上的竞争（为了实现对DRAM Bank的划分），我们尝试将LLC划分的方法拓展到DRAM层。如果能够利用地址映射中的Bank bits，如图3.4所示，将Bank分为不同的颜色的类别，那么即可达到利用“页着色”划分Bank的目的。问题在于，在地址映射中的Bank bits是什么样的分布状态？由于Bank bits通常是不被芯片制造商公布的，那么我们该如何确定这些bits？

我们进一步研究了某些典型系统的地址映射（intel-i7 series），发现其中同样有几位能够能够索引DRAM Bank（如图3.5）。因此，通过“着色”方法，即能够将属于不同颜色的页

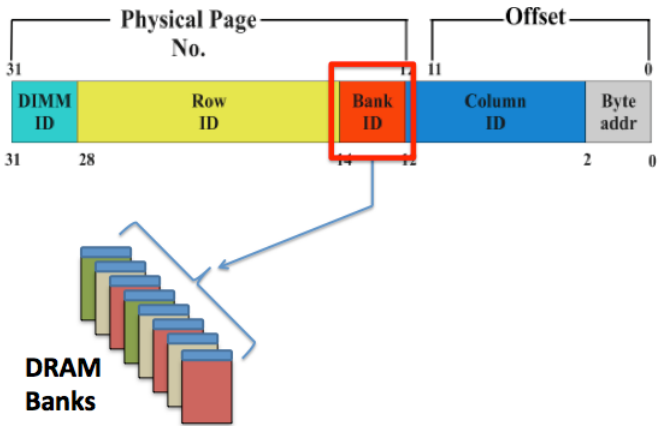


图3.4 利用Bank bit给DRAM Bank着色的示意图

面分配给不同的线程。从理论上来讲，将“页着色”技术应用在DRAM Bank上是可能的，只要按照线程对“颜色”的需求，在分配物理页面时有选择的将属于特定颜色的物理页面分配给特定的线程即可。该操作发生在线程出现“页缺失”的情况下，此时，操作系统将为该线程分配物理页面，并建立从虚页面到实际物理页面的虚实映射，同时创建一个页表项。以Linux操作系统为例，“伙伴系统”（Buddy System）负责管理系统所有的物理页面［106］，在每次发生“页缺失”的时候，操作系统都会在伙伴系统中返回一个实际的物理页面，并在该进程的页表中建立虚实映射关系。

我们的实验环境采用Linux CentOS 5.4，内核版本是2.6.32。硬件平台是Intel i7-860，2.8GHz，4核8线程中央处理器以及8MB LLC，系统配备8GB的内存／64 Banks。地址映射中表示Cache Set的地址位通常是固定的，而表示Bank的地址位则随着机器的结构、地址映射、内存配置等因素的不同而有所不同。例如，4GB内存的机器的Bank地址位要少于8GB内存时的地址位。为了准确的找出系统的Bank地址位，本文采用［60,61］提出的Bank地址检测算法来探测真实机器上的地址映射（见附录中算法的实际运行效果），并另辅以外部硬件HMTT卡［6］来校准探测结果。实验结果表明，针对真实环境下的多种机器配置，该算法能够准确的检测出Bank地位索引位。图3.5即展示真实机器下的完整地址映射，包括Bank，channel，rank的物理地址索引位。我们发现，Bank地址位并非完全连续，而是被分为两个连续的子部份，即图中所示的13、14、15位和21、22位。

3.3.2 支持 BPM 的索引系统和 HASH 算法

五个Bank地址位（13、14、15、21、22）将构成2^5=32种颜色。按照Bank划分的设计思路，对于某个线程仅分配给它特定的一种或者几种颜色的物理页面。因此，需要在运行时从Buddy System中检索出所需求的颜色的页面。由于Buddy System是操作系统的核心模块，运行时需要高效和稳定，原始的Buddy System在O(1)时间内即可返回所需的物理页面，但这个操作是顺序的遍历整个free list，并返回每个order free list中最顶端的页面。但是划分机制需要返回的某种颜色的页面则不一定在最顶端，如果按照传统Buddy System所采用的

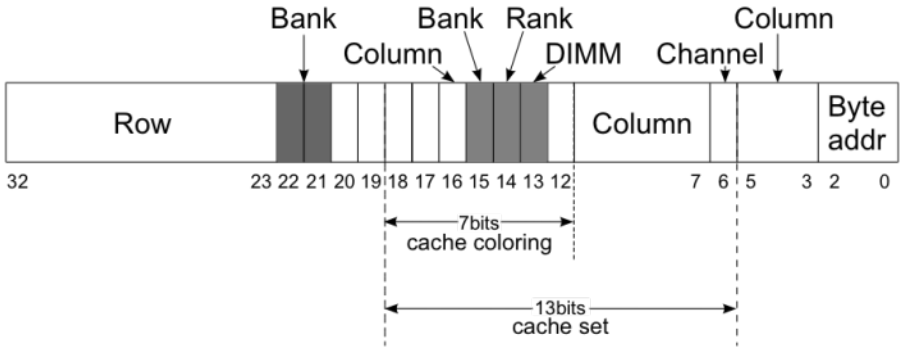


图3.5 Intel i7-860的详细地址映射。注：由灰色示出的是Bank的索引位。

顺序遍历free list的检索方法，那么返回所需页面的时间复杂度将会上升到O(N)。为了在O(1)时间内返回所需颜色的页面，我们按照页面的颜色排布规则创建了一套buddy system的索引机制（如图3.6所示）。

颜色的排布以页面的地址映射为基础。现代操作系统普遍采用4KB的虚拟页面，对于我们的实验平台而言（如图3.5），从0 bit～11 bit表示页内的偏移，第12 bit～32 bit即为物理页帧号（Physical Frame Number, PFN）。由于从第13 bit开始才是Bank bit，因此，地址连续的物理页面有可能属于同一种颜色。另外，由于Bank bit被分为两个部分，之间相隔16 bit～20 bit的6个bit，因此，由13、14、15 bit的组合所表示的页面将重复出现2^6次之后才会出现由21和22 bit所代表的页面（如图3.6中所示的页面排布规律）。建立索引机制的目的是为了以O(1)的时间复杂度返回所需的任意颜色的页面，其原则是利用了重复页面的出现规律，通过2的倍数作为参数即可在大块中跳转到目标页面的位置。基于这个索引结构的Hash算法如下，本文中以伪代码形式表示：

seudocode 1: Hashing algorithm for selecting pages

Input: (1) order; (2) target_color Output: one page of target color

```
BEGIN
/*Physical pages organized based on bits 13~ 15, 21~22*/
IF using 13, 14, 15, 21, 22 THEN
    SWITCH (order)
```

case	0~1	2	3	4~9	10
colors_per_block =	1	2	4	8	16

```
    END SWITCH
    block_color = (target_color / colors_per_block) × colors_per_block;
    IF order is 10 AND the color bits are x1xxx THEN //The 4th bit is 1
        page_index = (target_color - block_color - 8) × 2 + (1 << 9);
        // As shown in Figure 3.6: 32 blocks × 8 colors = 1024 blocks
```

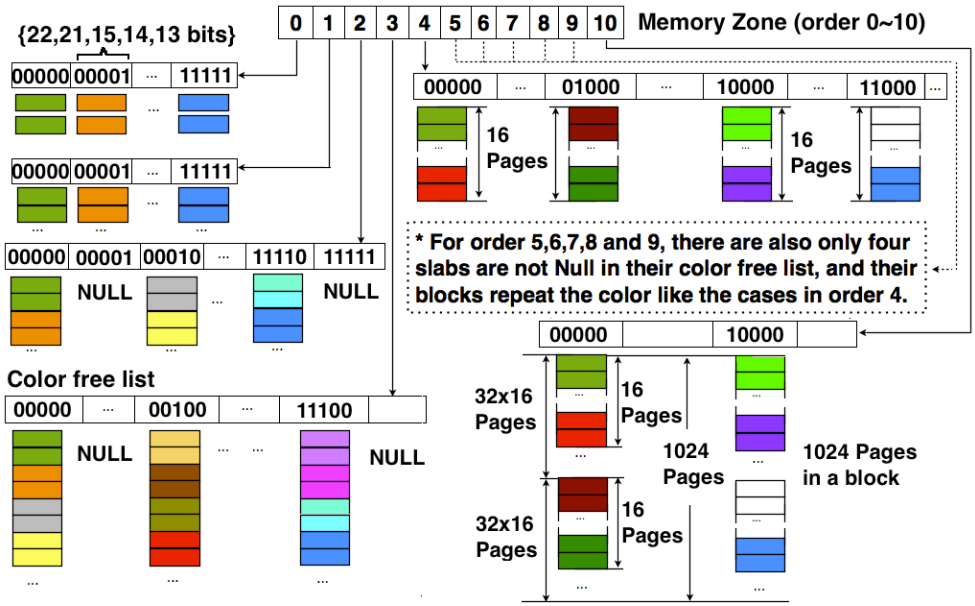


图3.6 支持Bank划分的索引系统。注：Bank bits是5位22、21、15、14、13。

ELSE page_index = (target_color - block_color) × 2;

END IF

END IF

Expand color block (page_index, order)

// physical pages represented by "struct page" are in page[] array in Linux kernel.

RETURN page[page_index] and remove this page from free list.

END

---

* target_color is the color of the requested page.

* block_color is the color of the first page in a block.

* colors_per_block is the number of colors in a block.

算法的理论时间复杂度是O(1)，返回目标页面仅需要很低的开销（基本上与原Buddy System的时间开销相仿）。例如，若需要在order-3的free list中返回一个“浅黄色”的页面，则在00000的颜色列表中自上而下的遍历6个页面，而只需根据算法跳过前6个非目标颜色的页面，直接通过hash获得第一个属于所需颜色页面的地址。移走了所需的页面之后，这个包含8个页面的大块将会被拆散成两个分别包括6个页面和1个页面的子块，并挂在相应的order的free list下。

为了支持“着色”操作，需要修改操作系统中代表进程（线程）的数据结构task_struct，添加代表颜色的变量color_mask，以作为在Buddy System中检索页面的依据。在进程被创建之初，该变量可由用户指定，在运行过程中也可被更改。第五章将详述这部分内容。

3.4 实验结果与分析（静态 BPM）

3.4.1 实验平台与量化指标

本研究的实验平台采用的是主流的多核Intel i7 4/8-threaded处理器，2.8GHz，并装备有8MB的16路LLC。主存系统系统装配有2个通道，4个DIMM插槽，总计包括64个DRAM Bank，每个Bank 125MB，总共8GB容量。基于该平台，我们的实验包括4/8-programmed的工作集，也包括真正多multi-threaded多线程工作集。按照本文的约定，除非特别说明，本文所指的线程是从系统与体系结构的角度出发的，通常指的就是运行时的一道程序（进程）。另外，如前所述，我们量化研究的的指标有两个（也是业界广泛任认可和采用的方式），吞吐量（Throughput）和公平性（Fairness）。吞吐量用来衡量计算机的计算性能，通常用“加权加速比”（Weighted Speedup）来衡量；公平性用工作集中性能下降最大的程序来表示（MaxSlowdown）[48]。对于现代计算机系统来说，这两个指标都非常重要且都是越高越好。它们的公式表达如下：

$$\text{Weighted Speedup(WS)} = \sum \frac{RUNTIME_{ALONE}}{RUNTIME_{SHARED}};$$

$$\text{Maximum Slowdown(MS)} = Max\left\{\frac{RUNTIME_{SHARED}}{RUNTIME_{ALONE}}\right\}$$

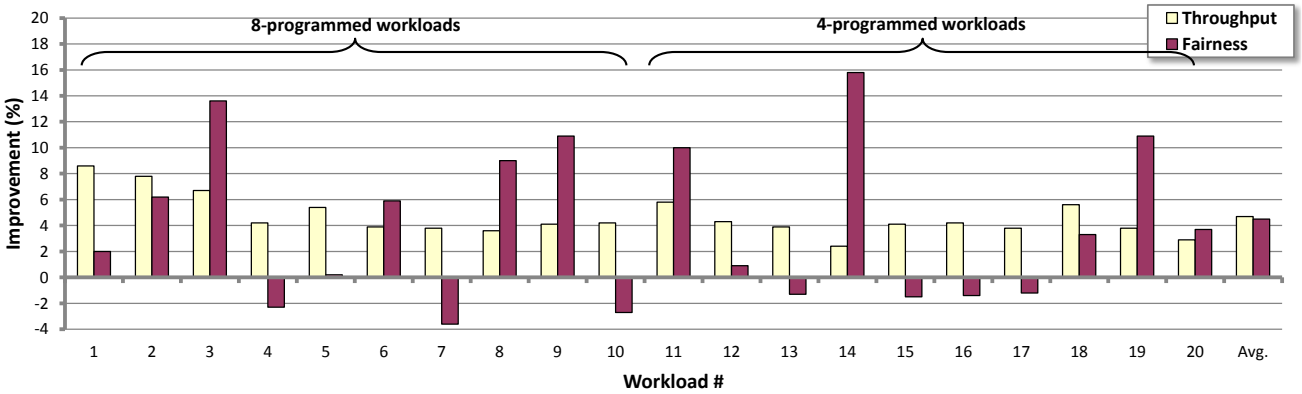


图3.7 系统整体性能的提升

3.4.2 BPM 整体性能的评测

对DRAM Bank的划分能够在根源上彻底消除进程间的访存干扰，因此，从理论上讲，可以同时提高系统的吞吐量和服务质量（公平性）。我们首先测试了静态的划分方式，将64个Bank平均分配给所有线程，在8线程的情况下，每个线程8个Bank，4线程的情况下，每个进程单独使用16个Bank。这里需要说明的是，采用均分的方式在大部分情况下是合理的，因为现在主流系统能够支持的Bank数量在持续上升，平均每个计算单元能够使用8～16个Bank（每个Bank 125MB），这样的配置能够满足绝大部分工作集对主存资源的需求。

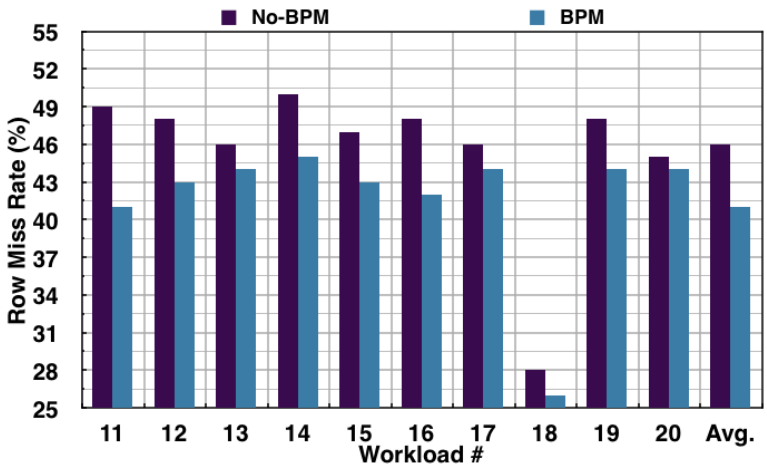


图3.8 行缓冲命中率的对比

为了验证静态划分的有效性，我们随机生成了20组工作集（10组由8线程组成，另外10组由4线程组成）。

总体来看，如图3.7所示，系统吞吐量都有提高，平均达到4.8%，最大能够达到8.6%。对于系统的公平性，平均能够提高5.2%，最高能达到15.6%，但同时也发现了若干工作集有公平性下降的情况。首先本文对包含4个程序的工作集（从图3.7中编号11～20）进行详细的考察和分析。对于工作集11，BPM能够提升高于平均值的5.7%的整体性能。该工作集中包含了462.libquantum (MPKI = 50, RBL = 99.22%), 403.gcc (MPKI = 0.4), 447.dealII (MPKI = 0.5) 和 444.namd (MPKI = 0.3)这4个程序。在这些程序之中，462是一个典型的访存密集型的程序，而且具备“流”（stream-like）式访存的特点（其行缓冲命中率高达99%），在并发运行的过程中，这类型的程序极易对其它程序产生干扰（因为现在主流的内存控制器的优化原则是提高行缓冲命中率）。相比较而言，这个工作集中的其它程序都是访存非密集类型的（MPKI在1%以下），因此很容易受到有强烈“侵略性”程序的干扰（如462.libquantum），在“交替”访存的情况下，462产生的干扰可以蔓延至每一个Bank。从理论上来讲，这种干扰是非常严重，不仅影响整机的吞吐量，而且访存的公平性也会受到影响，因为总是462的访存优先被服务。从图3.8上也可以看出，对于11号工作集，在没有BPM的情况下，行缓冲的缺失率（Row-buffer miss rate）接近50%。BPM在根源上消除了程序之间的访存干扰，它让每个程序能够“独享”仅属于自己的一组Bank（在4进程的情况下，每个进程独享16个Bank），根据图3.1所示，虽然这将造成462自身8%左右的下降，其它程序不到1%的性能下降，但是总体来说，系统的整体性能将提高接近6%，同时消除了8%左右的行缓冲缺失。这个性能提升对于真实的访存密集型的系统来说是很有价值的。对于8-programmed的工作集，我们发现包含了462的1号工作集同样也是性能提高最多的（8.6%），本文认为这个原因是与上文相同的。从另一个角度来将，我们的观察发现，8-programmed的工作集的性能提升要普遍高于4-programmed的工作集，原因在于普遍来讲

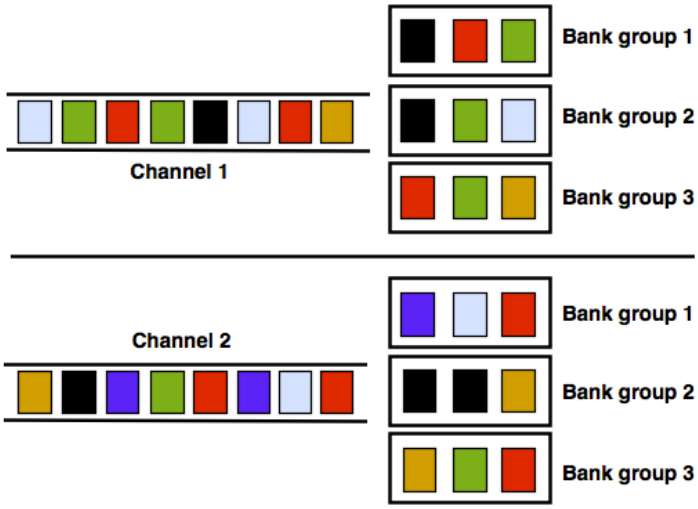


图3.9 线程间通道上的访存“交替”。注：细粒度的以Cache Line为单位的通道间“交替”。

前者能够发出更多的访存请求，也就是可能会产生更大的进程间的相互干扰，因此BPM也就更为有效。

公平性（Fairness）的优化是衡量BPM有效性的另一个关键指标。对于公平性，第11号工作集的提高是10%，第14号工作集的提高接近16%。这两个工作集都包含462这个程序，如前文所述，这是一个会对其它程序产生强烈干扰的程序，这对公平性是很大的影响。这两个工作集的差异在于它们分别包含了447.dealll（MPKI = 0.5）和456.hmmer（MPKI = 5.7），因此第14号工作集程序之间的干扰会更加严重。从之前提到的访存行为“多样性”的角度来说，整个工作集的访存行为会由于个别程序的行为变化而变化，实验结果表明，虽然整体的平均性能有所提升，但是很难按照线性回归的方式来完美预测BPM的功效。后继工作将继续展开研究这方面的问题。

值得注意的是，有几个工作集的公平性是下降的（第13，15，16，17号工作集）。我们发现这些工作集都包含429.mcf（MPKI = 99.8，RBL = 42.41%）。429.mcf也是一个极端的访存密集的程序，并且有较高的行缓冲命中率，图3.1显示在将它使用的Bank数量从64降低到16的时候，性能下降仅为2%，这说明在16个Bank的时候内存控制器赋予这个程序更高的优先级（或者始终赋予该程序较高的优先级），才使得它在Bank数目减少的情况下性能不下降（与462.lib不同）。这个程序也是访存密集型程序中性能变化相对较小的程序之一。因此，在使用BPM的情况下，由于429.mcf始终享有较高的访存优先级，因此导致其它程序相对略微不公平的现象，但幸运的是公平性受到的影响并不大。

3.5 通道（Channel）的划分技术研究

3.5.1 BPM+: 消除通道间的访存干扰

当前计算机普遍采用多通道（Multi-Channel）技术，目的是为了增加系统的总体带宽。

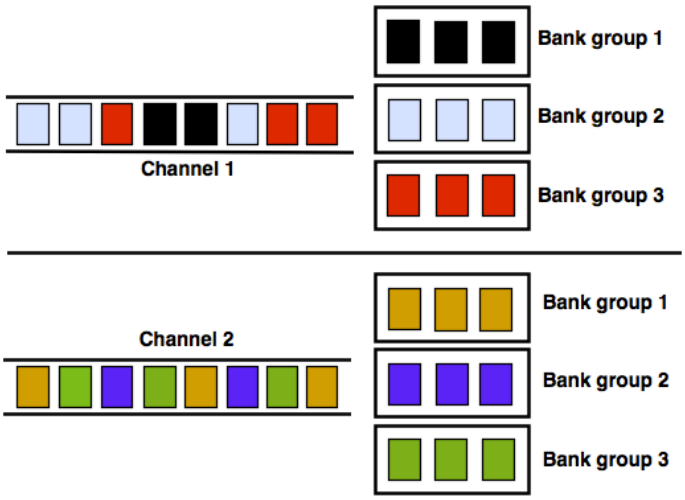


图3.10 BPM+的实用效果图

例如，DDR3-1600的峰值带宽是12.8GB/S, 如果系统有两条通道，那么系统的整体峰值带宽将是25.6GB/S （虽然实际带宽不能达到这个数字）。装备双通道的计算机已经非常普遍，某些高性能的服务器还采用四通道、八通道技术。我们实验的机器i7-860即采用了双通道技术，并结合以Cache Line (64B)为单位的“交替”访存，将访存平均分配在两个通道内，希望能够以简单的方式充分利用带宽。然而，“盲目”交替访存实际上会加剧在所有通道内对带宽的竞争，同时也可能会引发更多的DRAM上的冲突，因此在很多情况下未必会产生收益。如图3.9所示，不同颜色的色块代表来自不同程序的访存请求，在任意一个通道内部都包含了所有程序发出的访存请求，也就是说，每一个通道均要向所有程序提供服务；再加上DRAM上的冲突，那么由于资源竞争和干扰而引发的访存延迟就有可能更为严重。本文将这种在全部通道、全部Bank上的竞争和干扰理解为all-to-all的DRAM系统上的资源竞争和访存干扰。

结合前文讨论过的DRAM Bank划分机制（BPM），如果我们能够进一步降低在通道上的访存竞争，即限定任意程序仅访问某一个通道，就可以避免这种（all-to-all）的访存干扰，那么有可能使系统性能进一步提高。如图3.10所示，我们期望，每个程序仅使用一个通道并且在每个通道内部使用Bank划分，因此，在每个channel的竞争被降低的同时，Bank上的冲突也消除了，本文将这种机制命名为BPM+。双通道计算机地址映射中有一位表示的是channel，四通道机器则需要有两位来表示通道，依此类推。如前文所述，由于现在的计算机多采用XOR的方式将访存分散在多个通道之间，这样一来就没有办法实现channel划分。幸运的是，有些平台的BIOS为我们提供了可配置的选项，我们可以通过调整这个选项取消“交替”模式，以我们的实验平台为例，在调整BIOS之后，第32bit即为channel位。着色的方式与上文提到的方式类似，差别在于用来索引物理页面的是12，13，14，20，21，32 这6个bit，因此，“页着色”的方法同样可被用于BPM+的实现。

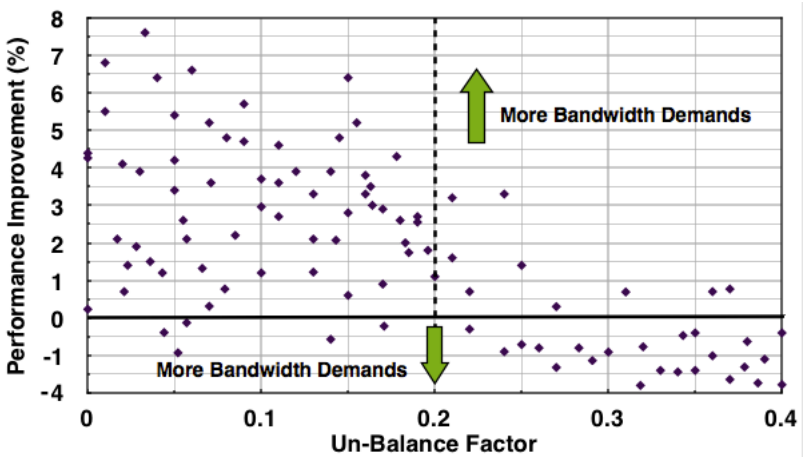


图3.11 通道间带宽的均衡程度与通道划分性能之间的关系。注：图中每个点即代表一个工作集。

3.5.2 通道间带宽均衡的研究

在进行通道划分之前，我们还要深入研究一个关于通道间带宽利用率平衡的问题。交替访存方式虽然被认为是“盲目”的，但是有利于在多通道之间扯平带宽，不会导致某个通道的带宽利用率远低于（或远高于）其它通道，因此不会导致严重的资源利用率不足的情况，这是交替方式的优势所在。然而，划分方式则有可能会造成这种资源利用率不高的情况。因为程序对带宽的需求是不同的，限定某个程序的访存仅在一个通道有可能导致通道间带宽利用率的明显差异。问题是，在真实情况下，如果很难保证两个通道的带宽利用率是完全相同的，那么这种差异在多大程度上会影响系统性能？为了说明这个问题，我们进行了如下实验。实验采用近100组工作集（4/8-Programmed），采用通道划分技术将不同程序划分在不同的channel，通过PMU分别监控每条通道上的实际带宽，并计算系统的整体吞吐量，基准线（base-line）是“交替”方式。我们定义了一个反映通道间带宽差异的指标，不均衡因子：Un-balance Factor＝|BUC1 – BUC2| / MIN (BUC1, BUC2)，BUC1是Bandwidth Utilizations of Channel（通道上带宽使用情况）的简写，阿拉伯数字表示通道的编号。图中的每一个点代表一个工作集，横坐标表示两个通道上带宽的差异，纵坐标表示在此情况下对应的系统吞吐量的变化。如图3.11中的实验结果为我们展示了几个重要的趋势：（1）通道之间的的带宽差异在小于20%的情况下，系统吞吐量不会受到影响；（2）带宽越大，通道之间的带宽差异越小，划分效果越好；（3）带宽越大，通道间带宽差异越大，划分效果越差。实验结果表明，在带宽的不平衡因子大于20%的情况下，通道划分会造成系统性能下降。

另一个值得注意的现象是，我们观察到存在部分工作集，无论不均衡因子是否超过0.2，它们的性能波动在[-1,+1]之间，亦即受通道划分的影响较小。深入观察这些工作集，我们发现它们的实际带宽需求较小，因此引发的访存竞争无论是在通道上还是DRAM Bank上都相对较弱，而划分所带来的开销则有可能导致系统性能的下降。

问题是，在什么情况下，通道的划分是有效的？由此延伸，在什么情况下需要划分通道？与“交替”方式相比，划分通道带来怎样的性能变化趋势？为了回答这些问题，我们进

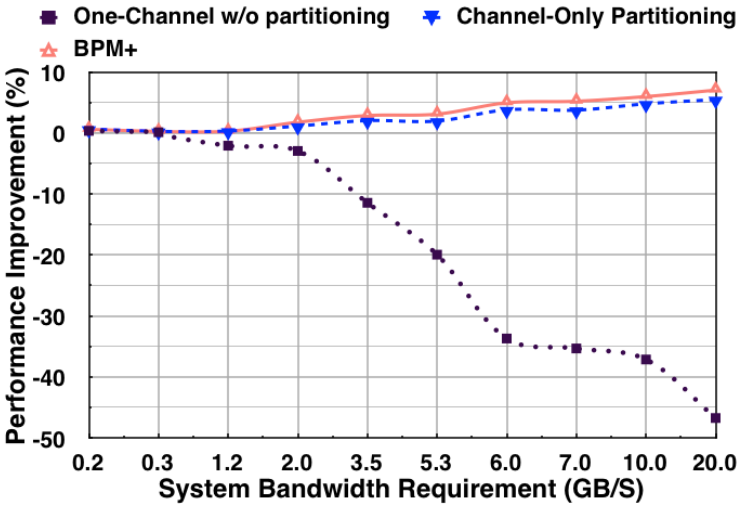


图3.12 带宽需求量与通道划分性能之间的关系

行了一组带宽变化的压力实验，以揭示带宽，划分，和性能之间的关系，基准线是通道间的“交替”方式。如图3.12所示，在工作集总的带宽需求小于2GB/S的时候，让所有程序仅使用一条通道不会引发严重的性能下降（在2GB/s时候，最大只有-3.7%）。从整个趋势来看，在小于2GB/S这个范围之内，带宽需求越小，性能下降越小。而在带宽需求超过2GB/S的时候，系统性能的变化呈现非线性的急剧下降状态，在20GB/S的时候，低于-45%。通道划分的性能增长曲线也说明了这个2GB/S阈值的存在，带宽需求在小于这个值的时候，几乎观察不到性能的提高（小于2%），但随着带宽需求的增大，通道划分和通道划分并结合DRAM Bank划分的两种方式都逐渐展现出优势。这个现象说明了只有在总的带宽需求超过一定阀值的时候通道的划分才能有效，否则，没有必要使用多条通道，因为使用一条通道所达到的性能基本上与多通道“交替”访存的效果相同。本实验在i7-860 DDR3-1600的平台上平均了几十组实验的结果。在减少了通道上的冲突之后，BPM又进一步的消除了通道内部在DRAM Bank上的相互干扰，因此，BPM+与单纯的通道划分相比，还有稳定的额外性能提高，并随带宽需求的提高而逐渐增大。从大量的实验结果可以看出，最大有3%左右的性能提高。

综上所述，本文至此得出的结论也是通道划分的应用原则是，第一，通道划分的前提是在一个阈值内拉平带宽（Un-balance Factor小于0.2）；第二，工作集总的带宽需求要大于一个阈值（至少是2GB/s）。这为后文的动态划分机制的研究提供了指导。

3.5.3 BPM+的静态实验结果

实验结果证明了BPM+在BPM之上另有额外的性能提升，平均来看（如图3.13），对于8-/4-programmed的工作集，分别有的1.2%和1.1%的增益（这个增益是稳定的）。总的来讲，BPM+和BPM一样都是在较高负载的情况下有较好的表现，对于8-programmed的工作集的性能提升要高于4-programmed的工作集。BPM+有效的原因在于它降低了通道上的线程间

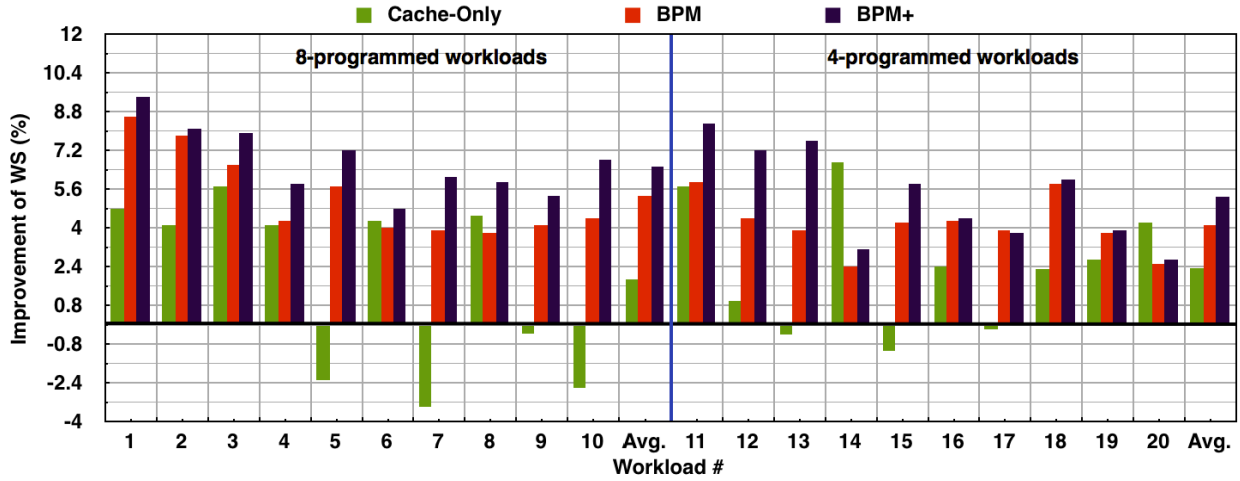


图3.13 BPM+的性能提升及与其它划分机制的对比

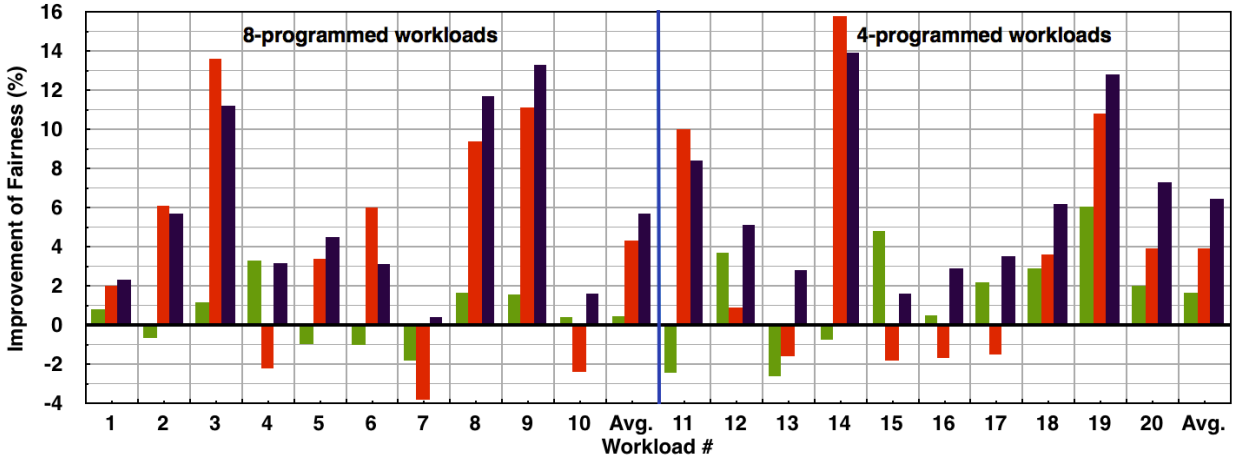


图3.14 BPM+对公平性的提升及与其它划分机制的对比。注：图例与3.13相同。

访存竞争，让每个通道不在为全部的的线程服务，从而避免了全局的资源竞争。对于公平性（如图3.14），BPM+不会在某些情况下损害系统的服务质量，这点与BPM不同。对于上文讨论的公平性变差的工作集，BPM+支持将429.mcf与其它易受干扰的程序分别映射到不同的通道，进而避免了不公平的现象，所以实验结果显示BPM+的这项指标要优于BPM。综合来开，本文认为，虽然在双通道的机器上BPM+的性能提升有限，但从长远来讲，对于未来装备更多通道的高性能服务器计算环境，BPM+应该能够获得更好的性能收益。本文还将Cache划分的结果与BPM/BPM+进行了对比，总的来说Cache划分会造成很多情况下的系统性能下降（在两个衡量指标上均如此），平均性能也处于较低的状态，但在少数情况下会有较好的表现。Cache划分的工作集敏感性较高，本论文将在第四章深入分析这些问题。

3.6 动态的划分机制的研究

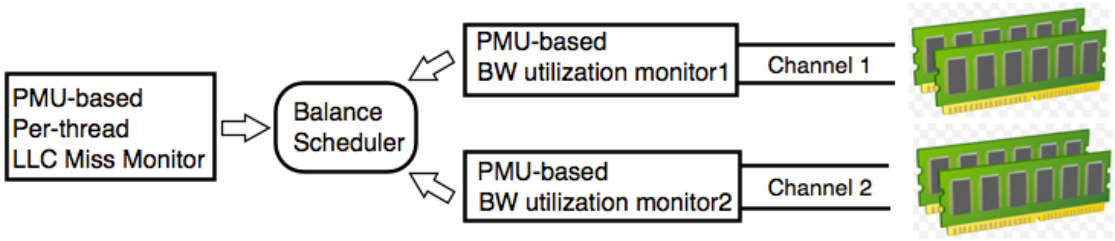


图3.15 基于PMU的BPM+调度框架原理图

前文所讨论的是静态的划分方法，一旦划分结束，资源的分配方式将不再改变。先前的实验也证明了这种方式在大多数实验情况下是有效的，但是，静态划分方法通常不能适应真实环境下相对复杂的需求以及程序访存行为的变化所引发的各种后继反应。这些问题主要被概括为以下几点：（1）在运行过程中，程序的访存足迹（Memory Footprint）可能会变大，那么先前分配的资源就可能不能满足当前的需求；（2）在系统资源有限的情况下，静态的方法不能有效、高效的利用现有资源；（3）由于资源利用率的不均衡，很可能会引发昂贵的交换区（SWAP）I/O的磁盘操作，进而造成不可预知的系统性能下降。因此，在实践中我们需要动态优化的方式在对资源的理想需求量与有限资源供应量这对矛盾之间寻求平衡。

3.6.1 动态划分机制的基本工作方式

动态机制的目的在于，当程序的需求超过静态初始分配的资源量时，会根据程序的需求动态的调整。我们形式化这个调整的依据：BNP (Number of Pages in allocated Banks)，即表示分配的Bank中包含多少个可以被使用的物理页面；RNP_i (Required Number of Pages), 表示一个程序在当前运行时刻所需要的物理页面的最大数目。每个物理页面4KB。$SUM(RNP_i)$即表示系统当前全部线程（进程）所需要的物理页面的数量，如果$SUM(RNP_i)$x4KB小于全部的当前剩余的物理内存容量，那么，从理论上讲，即存在一种能够满足当前工作集需求的划分方式。在这种分配方式中，虽然每个程序所分配的资源数量可能是不同的，但是不会出现资源利用率不均的情况也不会引发涉及交换分区的I/O操作。动态机制的基本原则是，从资源利用率不高的程序所属的资源中划拨适量的资源分配给需要资源的进程。我们定义了一个变量NICE=BNP-RNP，即分配给当前进程的资源数量与实际所需资源数量的差值，这个值越大表示资源的利用率越低，即有相对多的资源可以让别的进程使用。调度系统将当前运行进程按照NICE值的大小排序。当某个进程的BNP-RNP<=0的时候（当前资源不能满足需求），即会触发调度。

如上所述的调度方式需要在线的实时关注每个程序的RNP，这个指标可能会随时发生变化（因为程序的访存行为随时有可能变化）。基于硬件性能计数器（performance counter）的方式不能的得出这个指标。我们设计了一种操作系统级的动态低开销采样方式，动态扫描进程的VMA (Virtual Memory Area) 数据结构。每个进程可能包含多个VMA区间，通过指针结构串联在一起；每个VMA区间都有表示范围的start和end两个变量，它们的差值即

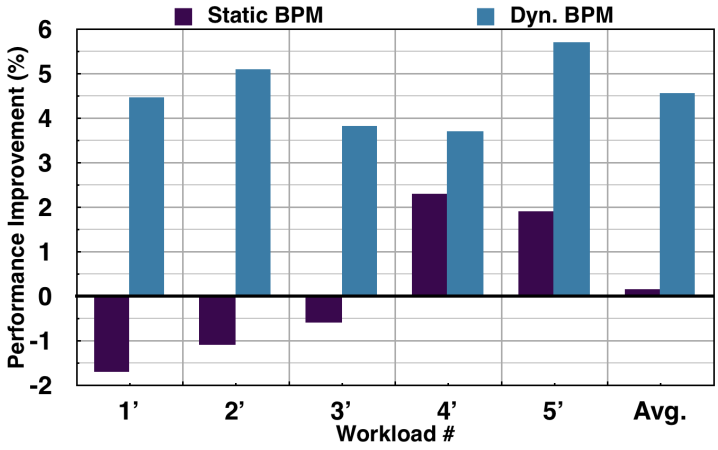


图3.16 在资源有限的情况下静态BPM与动态BPM的功效对比

表示这块虚拟空间的最大容量；将一个程序的若干个虚拟空间的容量相加，即可估算出运行时程序所需的最大内存量（RNP）。为了确保运行正常，当这个值大于BNP的时候，就要进行重新分配。

3.6.2 动态 BPM+的框架与实现

在“通道”这个层次上的调度也不能仅局限于静态的方式，因为动态的机制更具“生命力”。如前文所述，动态方式的目的在于如何平衡多个通道之间的带宽利用率以使它们之间的差异在某个阈值之内（例如，我们的实验平台i7-860可允许的通道间差异在20%以内）。如果能够达到这个阈值，则通道划分就可能会有利于整体性能的提高。为达到这个目的，必须首先计算出每条通道上的实时带宽。最简单的方式是利用硬件计数器 (Performance Counter)实时收集每条通道的带宽信息，例如，Intel的处理器支持统计每条通道的读写请求事件（UNC_QMC_NORMAL_READS.CH0，表示0号通道的“读”请求事件，相应的也有统计“写”的事件［125］）可以被用来计算带宽。基于此，我们设计了一个简单实用的通道层调度器，如图3.15所示，我们的系统有两条通道，我们分别监控每条通道上的带宽，并将信息传送给重要调度器并计算Un-balance Factor (不均衡因子)；此外，由于PMU无法获得每个程序的带宽使用量，因此，我们只有通过采样来获得每个程序的Cache缺失率以估计每个程序的带宽使用情况（Cache缺失率通常与带宽大小成正比）。这种做法虽然不能准确的得出每个程序的带宽的具体数值，但是可以获得所有程序带宽使用量从大到小的排序，这是一个当前系统每个程序的带宽使用的相对值。调度器总是选择Cache缺失率最小的程序进行调度（即带宽最小），将这个程序从带宽高的通道调度到低带宽利用率的通道。随后，调度器重新计算通道间的差异，继续上述调度过程，直到Un-balance Factor进入到0.2的阈值范围为止。

BPM+动态调度是一个迭代的过程，这个过程需要依赖操作系统的“页面迁移”机制。本研究是基于页着色机制实现的，因此，当遇到资源调整（或调度）的时候需要将当前线程所使用的页面更换成其它颜色的页面，这个过程被称作页面的“重着色”。操作系统对

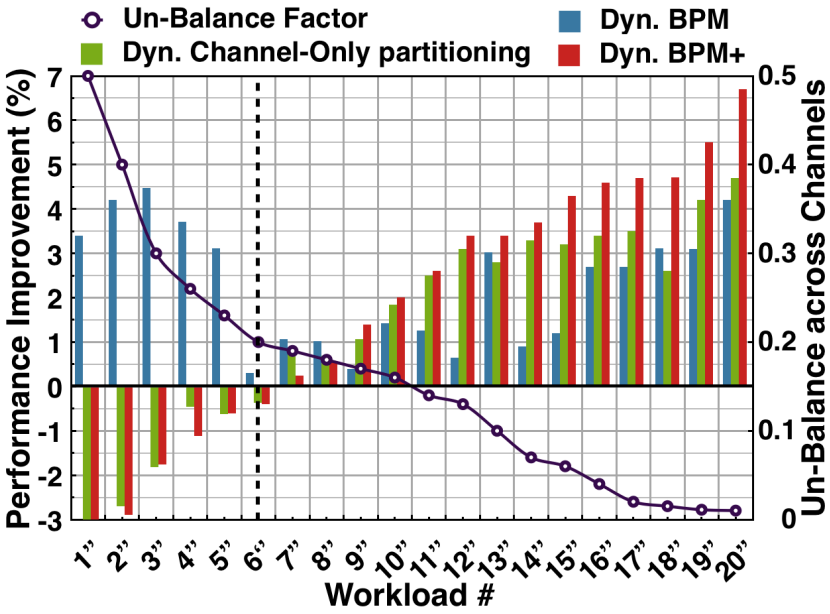


图3.17 随机的情况下三种动态划分方式的功效对比。注：这些工作集与之前3.13/3.14中使用的工作集不同。这些工作集是随机产生的，每个工作集包含4～8个程序。

页面的迁移采用的是页面内存拷贝的方式进行的，在大量的页面拷贝的过程中会产生开销，有可能会抵消优化带来的好处。本文后继的内容将会逐步分析和介绍开销与性能优化之间的平衡关系，以及降低开销的策略与方法。

3.7 BPM+动态机制的实验结果

3.7.1 避免低资源利用率和额外的 I/O 操作

本节首先评测动态BPM的效果，即在高负载的情况下保证主存资源的利用率而不引入交换区的I/O操作。本实验生成5组8-programmed的工作集，每个工作集中至少包含一个对内存需求超过500MB的程序，实验平台总的内存容量是4GB。

图3.16展示了这组实验的结果。静态的BPM将DRAM平均划分并分配给各个线程，对于内存资源需求大的进程如果其运行时的需求超过了预先分配的份额，就需要使用SWAP，从而引发磁盘I/O操作。由于磁盘操作的开销是内存操作开销的100倍左右［38］（足以抵消划分带来的好处），因此引入这类操作无疑会导致系统性能的下降。从图3.16中可以看出，在这种内存需求大的高负载情况下，BPM的静态均分造成3个工作集的性能产生“负”影响，另外两个工作集的性能提升也不高。

相比较而言，动态的划分更有“弹性”，在动态的过程中，动态BPM根据程序的实际需求能够调整资源的分配，以避免磁盘I/O的问题，对于长时间运行的工作集来说，系统性能将会有明显的改善，平均能达到4.6%的性能提升，这与静态不引入磁盘I/O的情况相近。

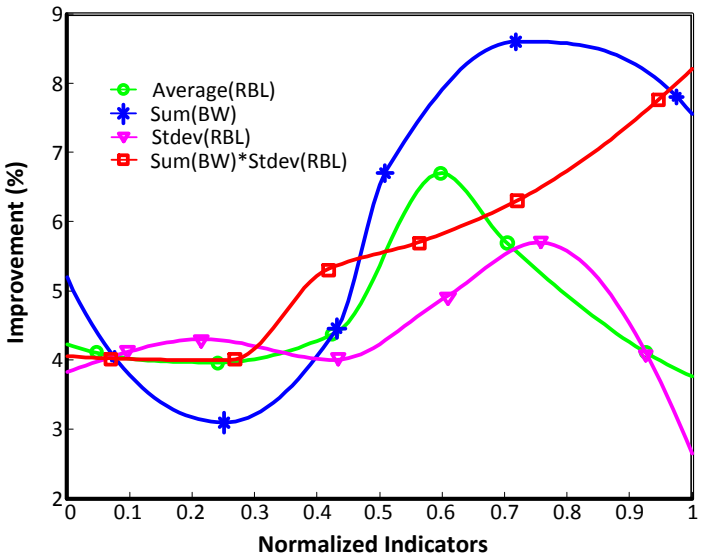


图3.18 四个指标与BPM/BPM+性能提升的关系

3.7.2 随机情况下动态机制的功效

图3.17展示了动态的Channel-Only划分, BPM和BPM+在20组随机产生的工作集下的性能表现，实验结果按照BPM+的性能的升序来排序。这组实验的工作集是随机产生的（与之前实验的工作集不重合），每个工作集至少包括4个程序，由于程序访存行为的多样性，这些工作集的Unbalance Factor也不相同。实验结果可以看出，从1号到6号工作集，由于它们的Unbalance Factor大于0.2，因此Channel划分会导致系统性能的下降。由于本文实验中的划分和调度是以线程为单位的粗粒度优化行为，因此在这种情况下是无法达到通道间的均衡的。但是BPM依然是有效的，因为这种机制采用的是细粒度的通道间的交替访存模式，通道间的带宽不会不平衡。本实验为将来的系统优化者提供了一个优化机制的选择空间和参考依据。第7，8工作集看不出BPM+在性能上的优势，它们的性能变化在±1%以内。这类工作集包含的程序主要是访存非密集类型的程序，因此本身优化的空间也不大，上文对这个问题已经进行了详细的讨论。从第9号工作集开始，BPM+的优势就逐步显示出来，随着由曲线表示的Unbalance Factor的不断减少（从0.2到接近0），即通道间的带宽利用率越发趋于均衡，BPM+的性能也在逐渐提高。本实验也说明了基于通道划分机制要比仅对Bank的划分方式更有优势，原因在于现代的多核计算机对带宽的竞争更为激烈，相比较Bank而言，通道的数量也更为有限。通道划分技术能够缓解线程间在通道上产生的访存竞争，BPM+又能够在可能的情况下避免资源使用率低的问题，故本文认为BPM+的实践方法对未来高性能服务器的设计提供了较有价值的参考。

这里需要针对BPM+的工作方式作出说明，动态的调度器首先通过在通道间调度线程的方式尽力去平衡通道间的带宽利用率，它将若干线程的访存重新映射到带宽利用率低的通道里，之后再根据具体的需求动态的调整通道内部的Bank的分配情况。实践证明，我们的方法是一个性价比较高实践方案，它能够被相对容易的部署在多通道的服务器上，并且

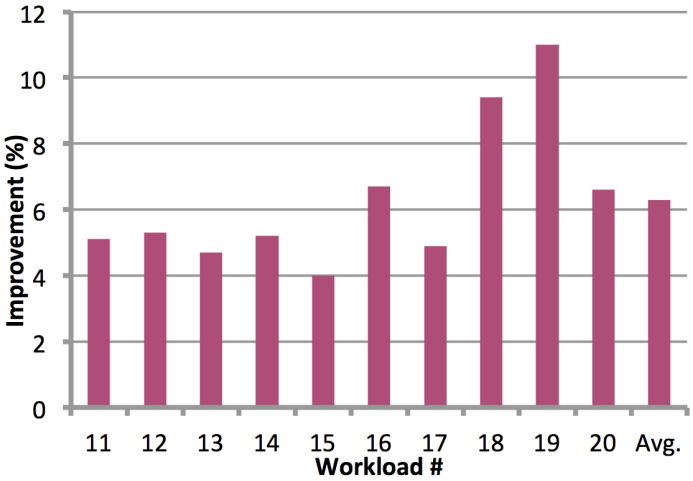


图3.19 Open-Page的情况下使用BPM的性能提升。注：Base-line是Close-Page w/o BPM。

调度的逻辑也相对简单有效。在实际环境中，能够高效且有效的完成任务的方法是最好的，我们的机制同时支持BPM/BPM+这两种方式，两种方式各有利弊，对于有些情况（实验中的1～6号工作集），BPM是有效的，因为这些工作集中可能存在某个程序的资源需求量远远超过其它程序的总和，BPM+采用的线程级的调度不可能在通道间保持带宽的平衡。但是BPM+也同样支持跨通道的资源分配，对于这个例子，BPM+可以将另一个利用率低的通道中的资源分配给需求高的进程以求得通道间的平衡。这种做法在我们现有的双通道的机器上是会引发通道上的干扰，未必能有好处，但在装备4或8通道的机器上则有可能会收到良好的效果，此问题留给系统优化的从业者根据具体情况酌情处理。

3.8 哪些因素影响 BPM/BPM+的性能？

这一节我们将初步探究工作集的特点与BPM/BPM+产生的性能提升之间的关系。我们拟定了四个指标，工作集的平均行缓冲局部性 (Average(RBL))，工作集总的带宽需求(Sum(BW))，工作集中程序的行缓冲局部性的加权标准差(Stdev(RBL))，以及Sum(BW)*Stdev(RBL)。图3.18展示这四个指标与BPM/BPM+带来的性能提升之间的关系，每个曲线上的点都是取平均值之后标准化的结果。我们可以看出，只有Sum(BW)*Stdev(RBL)与BPM所带来的性能提升的趋势是正相关的，这实际说明了系统中访存干扰的强度越高，访存的密度也就越高，同时程序潜在的局部性的需求也越高，因此BPM也就会越有效。本文用同样的方法验证了BPM+，发现这个指标也是有效的，因为在通道间带宽平衡的情况下，BPM/BPM+有效的根本原因是相同的。

3.9 Page-policy和能耗

众所周知，Open-Page策略被主流体系结构所采用［51］，即行缓冲中的内容只有在发生冲突的时候才会进行PRECHARGE操作，其目的是为了更好的应对有较高行缓冲命中率

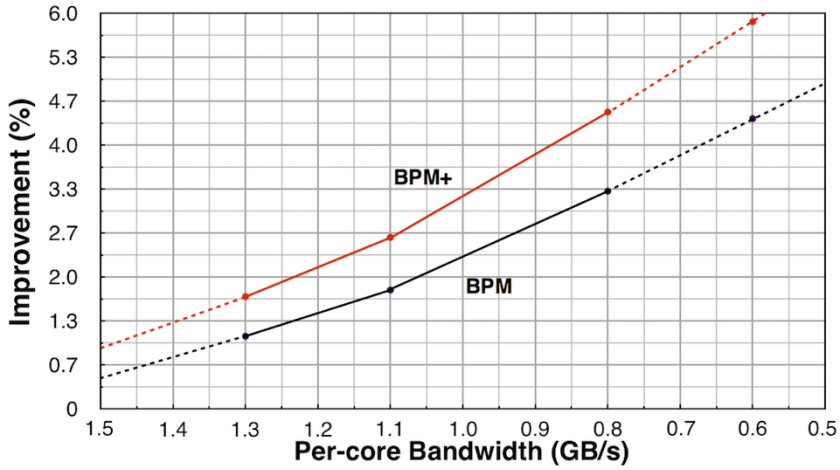


图3.20 带宽变化时BPM/BPM+的功效图

的访存行为情况。不同的是Close-Page在该行被访问结束之后就立刻进行PRECHARGE操作。但是近期的研究表明，在多核体系结构下，由于程序之间访存行为的相互干扰，行缓冲的命中率已经变的非常低，再继续使用Open-Page策略已经没有意义，因此很多体系结构设计者更倾向于使用Close-Page。但在使用BPM/BPM+的情况下，由于消除了Bank上的访存干扰，Open-Page依然能发挥出其优势。我们的实验平台能够通过调整BIOS来选择使用Open-Page或者Close-Page。我们使用上文实验中的4-Programmed工作集验证了我们的设想，如图3.19所示，基准线（Base-line）是使用Close-Page的系统，我们使用的Open-Page结合BPM的方式展示了更优的系统性能（平均6.3%的性能提升）。该实验证明了，如果未来采用恰当的划分策略，Open-Page是依然能够被使用的。另外，主存能耗问题受到业内的普遍关注［3-5，8，58，71］。ACTIVE操作是DRAM操作中最耗时耗能的操作［1，72］。实验证明，BPM能够减少此操作的次数，因此，直接降低了DRAM系统在运行时的功耗。本文采用［14］设计的真实的测能耗的硬件系统来收集功耗数据，实验结果表明，BPM能够有效节省5.2%的主存系统的能耗。

3.10 BPM/BPM+的性能与每个核（core）能够使用的带宽之间的关系

现有的装配DRAM的多核计算机的趋势是，均摊在每个核上的带宽资源随核数目的增多而减少。如前文所述，由于CPU引脚的数目（pin）是有限的，因此扩展现有带宽非常困难，这也间接影响了计算资源的扩展［47］。为了验证BPM/BPM+在这种趋势下的性能变化，我们通过调整BIOS来改变主存DRAM系统的工作频率（从1333到800 MHz），因此带宽的实际使用量也会呈现出递减的状态（在8个核的情况下，每个核的带宽从1.3GB/s降至0.8GB/s）。这个过程恰好模拟了每个计算单元能够使用的带宽在逐步减少的趋势。在每个频率下我们均分别使用BPM/BPM+，并且在每个频率上平均了数十组工作集的性能。实验结果显示（图3.20），在每个核能够使用的带宽逐步减少的情况下，BPM/BPM+能够提高的平均性能却在不断增加。我们前期的实验也佐证了这个结论，因为在8核的情况下（每个

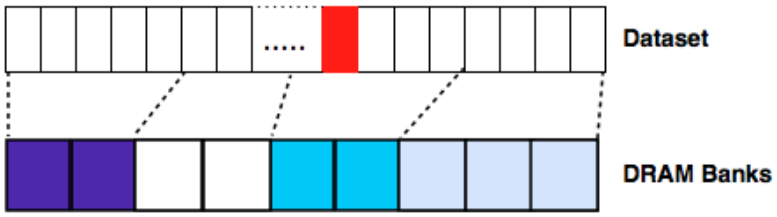


图3.21 多线程工作集的划分示意图

核的均摊带宽相对小）系统性能要优于4核的情况。因此本文得出结论，BPM/BPM+在未来多核、众核的情况下能够有更好的表现，因为每个核的均摊带宽将会持续减小。本实验也证明了在带宽变化的情况下BPM+始终优于BPM，并且代表BPM+的曲线的上升率要高于BPM，这进一步说明了在访存竞争越发激烈的环境下，BPM+同时缓解、消除了Channel／Bank两个层次上的访存干扰，是更为有效的优化方式。

3.11 划分技术对多线程（Multi-threaded）工作集的有效性

截止目前的实验结果都集中在说明BPM/BPM+对与Multi-programmed工作集的有效性，当前有很多多核服务器上的应用都是Multi-threaded的工作集。为了验证本文提出的机制对于这类应用的性能，本文选用了多线程基准测试程序集PARSEC［10］中有代表性的多线程聚类应用StreamCluster作为测试用例，采用最大的工作集1000000个离散点作为输入。实验中将这些点平均分为N个大块，分别对应不同的处理线程，前N-1个块包含相同数目的点，最后一个块包含剩下的所有点。StreamCluster创建的N个线程分别处理这些数据块，实验中的划分也是按照这些数据块进行着色的，每个数据块即对应一个线程，映射进独立的一组Bank，如图3.21所示。实验结果证明BPM仅能带来微小的性能提升，分别是1.7%/2.3％在4/8-threaded的情况下。这个性能的提升不如多进程工作的集性能提升大，因为，如图3.21所示，在每个块中实际都存在共享数据（红色表示），因此划分技术不能完全消除在这种情况下线程之间的干扰，故性能提高也不明显。从本实验可以得出结论，由于程序访存行为多样性的差异，应该采用不同的资源分配机制或优化方式来应对不同访存特征的工作集。

3.12 “重着色”和开销

动态的机制依赖于对物理页面的“重着色”，具体是指，将属于某个颜色的物理页面的内容迁移到属于另一个颜色的物理页面中。迁移的过程实质是数据的拷贝，在内核采用memcpy函数来完成的。从目前操作系统的设计与实现的原则来看，这个过程是必须的。动态调整DRAM Bank数量时需要触发“重着色”，将原先属于一个进程的Bank上物理页面中的内容迁移至另一个Bank；同理，在通道之间调度也需要将物理页面的内容在通道和Bank间迁移。页面的迁移将引发系统的开销，在我们的实验平台上时3～8us/per page，也就是

每迁移一个4KB的物理页面需要3～8us的时间，在迁移大量页面的时候，这个开销是不可被忽略的，在某些情况下甚至可以抵消划分带来的好处。

为了降低开销，本文采用的策略有：（1）“懒惰”迁移（Lazy-Migration）[13]，迁移只发生在必要的时刻。我们观察到并不需要将所有待迁移的页面一次性的处理完毕，有些页面的访问频率很低并且在程序运行的后期才被访问。那么，[57]提出了仅在程序出现对该页面的请求的时候才迁移此页面。平均来看，这种做法要优于盲目的一次性迁移机制。（2）“最小”调度。我们力求以最小的迁移代价来完成通道之间迁移的任务，因此，总是先在带宽利用率较高的通道中选择带宽需求最小的程序（Cache缺失率最小）进行迁移。另一个引发开销的因素是由PMU采样产生的。为了能够精准的获得程序运行时的信息，往往需要密集的采样频率，但这样做同时又会带来相对大的采样开销，因此，我们需要在采样的精准度与采样的开销之间作出权衡。在目前的实验阶段，我们的工作集来自于SPECCPU2006，在我们的系统上，我们发现以10秒作为调度周期是一个比较合适的选择，因为能够收集到足够的调度信息并且不至于引发过于频繁的调度开销。当然，这个调度周期可能会随计算环境的不能而变化，我们建议用户在使用我们的设计时能够首先根据自身的环境验证这个调度周期，以期选择出更为合适的参数。

3.13 若干问题的讨论

（1）如果不能拉平带宽的情况怎么办？从通道划分的角度来讲，我们采用的调度是“粗粒度”的线程（进程）级的调度，这也就意味着有可能无论如何也不能将通道间的带宽差异调整到“阈值”之内。对此问题本文提出两种讨论：（1）通道划分的“禁用”。也就是说，仅在能够拉平带宽的情况下才启用通道划分，而在不能平衡的环境中采用的是动态或静态的Bank划分。另外，如果能够辅以预先静态采样（static profiling）的方式，那么在真实环境下会有更强的实用性。（2）允许部分程序的通道“交替”。这是一种折衷的办法，该方法让带宽需求高的程序分散一部分的访存需求到别的通道中，目的是为了让通道间的带宽趋于平衡。通过控制颜色分配方式可以达到分散带宽需求的目的，比如说，实验中为了拉平带宽，我们需要让某个程序分散部分访存请求到另一个通道，那么只要重新给该程序着色，并且分配属于另一通道的部分页面即可，至于分配的数量需要根据具体需求确定。这个过程需要“重着色”和页面迁移机制的支持，正如如前文所讨论。

（2）避免引发I/O操作。当程序的总内存需求不超过分配给他的内存大小时，BPM和BPM+都有可能带来较大的性能提升。然而，一旦程序对内存的需求超过了系统能够提供的大小时， BPM和BPM+就会因为引发了I/O操作而造成性能上的损失。我们选取了SPEC2006中的一些程序，在内存受限的条件下进行了实验，结果表明一旦使用了I/O Swap，BPM/BPM+带来的性能会受I/O事务的延迟而减少。在这样的情况下，总体的性能提升只有1%左右（低于正常情况下5%的性能提升），有的时候甚至小于0。所以，我们建议在采用我们提出的策略时，系统中应该避免产生的I/O操作。

（3）BPM+支持多种分配与调度方式。虽然BPM+限制一个线程只能访问一个特定的通道，我们也提供API支持用户为线程在别的通道分配更多的内存资源。当一个通道的带宽显著高于其他通道时，这个API能够有效地均衡多个通道之间的带宽。事实上，我们在工作集16～20中使用了这个API（如图3.13所示），实验发现在双通道平台上，很难每次都获得性能收益，主要原因有两个：首先，均衡通道间的带宽又导致了通道之间的干扰，类似于前文提到的在多核平台上MC之间的干扰；其次，由于访存行为的多样性，一个线程的带宽需求在某些情况下不能被恰当地转移到其它通道上。但是，这个API在四通道以上的机器上将更为有意义，它能够使一个带宽需求较高的线程使用多个通道，同时通过其他通道消除干扰。因此，我们认为在未来的计算环境中，如何更好的利用这个API将是我们将来继续研究的问题之一。

（4）Bank并行度(Bank Level Parallelism, BLP)与BPM/BPM+之间的关系。BLP是我划分机制性能的影响因素之一。我们发现如果程序470.lbm在工作集中，采用BPM和BPM+策略反而会带来性能下降。从图3.1中可以看到，当470.lbm程序可以使用的Bank数量小于16时，性能下降十分明显。比如，当给这个程序分配的bank数量在8到16之间时，性能损失在10%到40%之间，明显大于比其他程序的性能下降（平均在1%到8%之间）。这是由于470.lbm比其他程序具有更高的Bank并行度。基于这项研究，我们根据经验设计了一个“10%规则”用于指导正确的使用BPM/BPM+策略：当工作集中包含一个以上的程序由于Bank划分自身受到了10%以上的性能损失时，应该更谨慎的使用我们的方法。除了上面两个因素，前面章节提到的不均衡的因素同样也会影响BPM+策略。

（5）通道的划分是否会给对带宽需求高的程序造成影响？直觉上，通道层次的划分可能会使原本通过“交替”方式使用多个通道的线程的带宽使用量下降。但是，我们从实验中得到了一些反直觉、但能够被解释的现象，本文概括为两点：第一，允许每个线程使用多个通道不可避免地会带来通道和 Bank 上的严重冲突，从而在很多情况下减弱了通道层次上“交替”访存带来的好处；第二，我们发现在真实情况下，每个线程真实的平均带宽使用量通常都在 6GB/s 以下（SPEC2006 中的程序平均在 1～2GB/s，而每条通道的峰值带宽是 12.8GB/s），理论上一条通道即可满足需求。所以，我们的结论是线程之间的通道划分能够在满足性能需求的情况下消除干扰，实际上能够带来总体的性能提升。然而，很自然就会想到下一个问题：通道层次的划分总是有效的吗？为了回答这个问题，我们引入了一个新的参数 Unbalance Factor 来判断划分方式是否能够提升性能。在实际运行中，一个程序有可能比其他程序需要更大的带宽，我们的系统能够动态的为这样的程序分配更多的通道或 Bank 资源。

3.14 相关工作与本研究的对比

DRAM Bank的划分：业内对基于硬件和软件的划分以消除线程间访存干扰的研究非常丰富。Wei Mi［79］第一次提出了Bank划分的方法，之后又陆续出现了进一步的工作［60,85］

介绍了动态的Bank划分机制，其中BPM［60］是首次在真实多核系统上实现了Bank划分，并揭示了物理地址的重叠位（O-bits）能够同时索引LLC和DRAM Bank, 可以被用来同时划分LLC和DRAM Bank。在文献［82］中首次提出了通道划分的方法，它按照线程的内存访问行为将不同线程的数据映射到不同通道上。但是，他们没有对通道和DRAM Bank的划分方法以及多通道的负载均衡问题做深入讨论。另外，这个方法也没有在真实机器上实现。

Memory Controller（MC）的优化：学术界对这方面的研究展现了浓厚的兴趣，近5年来的研究成果层出不穷。[68-70］设计了能够区分线程访存行为的MC，在运行时通过监控到的线程的访存行为来进行调度，而且调度策略会基于应用特征的变化而调整。TCM策略［48］依据公平性和吞吐量的指标动态地将线程分为两类（访存密集的和非密集的）并采用不同的调度策略针对不同类型的程序。在文献［81］中，作者基于网络公平队列调度算法（为了确保服务质量）在MC中设计了一个内存调度器，对系统整体性能有较好的表现。最前沿的研究工作MISE［92］通过计算被响应的带有优先级关系的访存请求在所有请求中的比例来预测程序的性能变化，实验证明该方法的准确率较高，以此为依据的调度方法也表现不俗。这些MC的优化方法都需要修改硬件才能实现，并且都不同程度的存在着开销，随算法复杂度的升高这些开销也随之增大。

用户级的软硬件协同的内存调度优化：文献［119］提出了DI和DIO调度算法，该算法通过线程映射的方式以达到避免缓存、MC、总线和预取硬件发生严重冲突。另外，文献［29,115］采用线程执行节流（Throttling）和内存资源利用节流的方法来获得高吞吐量或者公平性。这些用户级的方法常常采用硬件计数器来采样，我们认为BPM/BPM+可以与这些方法相结合以产生“合力”。

行缓冲利用率的优化：行缓冲的利用率是访存优化的关键，也是近些年研究的热点。为了克服行缓冲颠簸的问题，［90］提出了一种将位于Bank中不同行的但经常被访问的数据块汇总到一行的做法（每行只移动部分“热”数据到行缓冲之中），在这种情况下，运行时的行缓冲实际汇总了某些热点数据，进而减少了颠簸的次数，也节省了能耗。另一个有代表性的工作是［51］，该方法重新定义了行缓冲的工作协议，在行缓冲被访问4次之后就写回，希望达到局部性与数据使用之间的平衡。另外，他们的工作也修改了物理地址映射的方式，使得在Bank上有较好的并发度。

物理页管理的优化：这方面的工作不是特别多，但都很有代表性。近期的研究工作［83］观察到，随机地访问物理页在很多情形下都优于规则页访问模式。因为随机的页访问模式有可能会减少行缓冲区上的线程间干扰。基于此，他们提出了M^3方法，对来自多线程程序的内存访问尽可能地随机分布在所有Bank上，该方法对于具有“流”式内存访问模式的多线程工作集尤其有效。他们的实验表明，对于StreamCluster，M^3方法在八线程下获得超过15%的性能加速。然而，从他们报告的结果来看，该方法的适用范围是有限的。另一项近期工作［23］提出了从应用程序到核的映射策略（Application-to-Core）以减少多核平台上的内存干扰，该工作需要修改操作系统的物理页分配和替换策略来保证系统正常工作。这些工作的内容都涉及在操作系统内采用新的物理页管理机制来减少线程间访存干扰。

BPM/BPM+与相关工作的比较：直接相关的工作是［82］和［85］。文献［82］提出了一种应用特征敏感的通道级划分方法。［85］提出了Bank级划分结合Sub-Ranking机制来消除Bank级的干扰，并通过提高能够独立的工作的Bank的数量来补偿减少的BLP带来的损失。我们的工作与上述工作的差别主要体现在以下两个方面。首先，BPM/BPM+是首次在真实的主流多核计算机上实现的Bank划分和通道划分，我们获得的实验数据和性能评估更有价值。其次，我们的方法不需要修改硬件并能用于任何Linux系统。M^3方法对于一些多线程工作集是有效的，但在一些情况下其表现并不尽如人意，有些时候甚至会造成系统的性能下降（-3%左右）。而且，他们的工作没有测试多道程序（multi-programmed）。相比而言，BPM和BPM+则是普遍有效的，在我们的大部分实验中并不导致系统吞吐量的下降（例如，在极端的情况下引发I/O操作）。而且，BPM+的“弹性”机制能够支持不同的动态划分和调度策略，更适应实际需求。

3.15 本章小节

本章介绍了 Bank 划分的核心机制，核心算法，以及相关的实验数据。本章证明了 BPM 是一种有效的消除线程间 DRAM Bank 上干扰的机制，并且具备较好的扩展性和可用性。本工作还进一步还研究了基于通道的划分方式，进而形成了在整个 DRAM 内存系统上的划分优化的框架，为高性能、多通道的服务器性能优化提供了参考。

第四章 Going Vertical: 内存储资源的“垂直”划分及其敏感性

4.1 与划分相关的问题

在第三章本文着重讨论了对DRAM Bank系统的划分，加之学术界广泛讨论的对LLC（Last Level Cache）的划分，可以说前期研究分别对这两级内存储器的“水平”（Horizontal）划分的研究已经比较广泛。详细内容可参考学术界有代表性的论文。

但是，关于划分技术还有很多问题值得深入研究并亟待解决：

问题（一），对Cache的划分会对DRAM的利用率造成什么样的影响？或者，更进一步说，两个层次的水平划分机制之间有没有相互的影响？

问题（二），有没有可能将两种水平划分机制结合起来，“垂直”的划分DRAM Bank和Cache以同时消除Bank和Cache上的线程间访存相互干扰，进而使划分的效果形成叠加？

问题（三），所有可能被用来划分的地址位（地址映射中的bits）对于划分效果来说都是等同的吗？不同的地址映射方式会对划分效果产生什么样的影响？

问题（四），划分的效果是不是对工作集敏感的？不同的划分方法与机制和工作集的多样性之间存在什么样的联系？

回答这些问题是必要的，因为这将有助于工业界和学术界更加深入的理解“页着色”技术，为该技术在未来多核 / 众核平台上的应用提供理论和实践上的依据。本章我们希望尝试为这些问题寻求答案。

4.2 研究的动机

如前文所述，本研究的一个优势即在于采用真实系统而非模拟器，因此，大量的真实数据的采集是本文开展分析和研究的前提，我们可以通过大量的实验来观察结果并归纳出结论。

为了回答4.1中的这些问题，以进一步揭示划分机制之间的关系，我们随机产生了214组工作集（8-programmed或者4-programmed），其中包含的程序来自于SPECCPU2006。我们对每一组工作集都验证仅划分Cache (Cache-Only) 和仅划分Bank (Bank-Only) 时的有效性。如下图4.1所示，实验结果分布在一个由四象限构成的点图中，每一个点代表一个工作集，横坐标表示由Bank-Only划分带来的性能提升，同理，纵坐标代表由Cache-Only划分所提高的性能比率。性能的变化通过加权加速比 (Weighted Speedup)的变化来衡量。我们可以看到，几乎所有的数据点都分布在第一和第四两个象限中。位于第一象限的数据点说明这些工作集即能够得益于Cache划分同时也能够被Bank划分所优化；第二象限表示Cache划分有效但Bank划分无效，但在我们的实验中几乎没有发现这样的有代表性的工作集（除了少数性能波动的工作集之外）；第三象限中有很少的点，这说明Cache划分和Bank划分都引

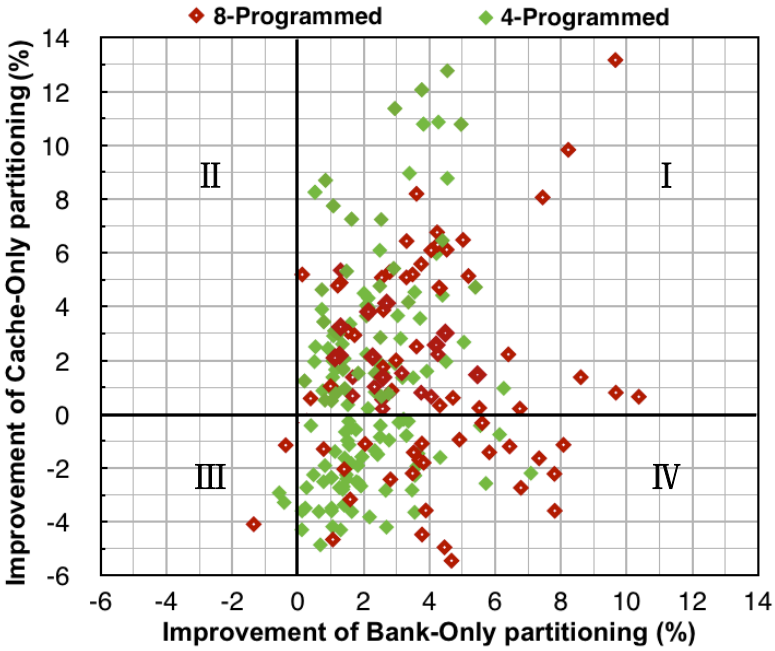


图4.1 Cache-Only和Bank-Only两种划分机制的结果散点分布图

发系统性能下降的情况也为数不多；最后，有几乎一半的点落在第四象限，这说明，Cache划分在很多情况下会引发性能性能下降，而Bank划分则在绝大多数的情况下都是有益于系统整体性能的。

从这个图上我们还观察到，对于第一象限中的工作集，大部分的性能提升仅在2%～4%之间。虽然对于真实的计算环境来说这并不能被忽略，但是我们希望能够进一步的挖掘出潜在的性能收益。回到我们前面提到的问题，对于这些Bank划分和Cache划分都有效的工作集，有没有可能将这两种划分机制带来的好处进行叠加？如果有可能的话，系统性能对于第一象限中的这些工作集可以进一步被提升；第一象限中的工作集几乎占总数的一半，那么也就是说，系统的整体性能在近乎50%的情况下可以被进一步提升。本研究认为，从工业界的角度来讲，在现代体系结构下处理器速度已接近极限，能对近50%的随机工作集有增强的优化效果，则这个动机是有价值的；从学术研究的观点来看，如第二章背景中2.6节所述，截至本文之前，学术界还没有就此问题展开过深入的研究和讨论，对“页着色”技术的认识一直是不全面的。

4.3 多级存储器之间的“垂直”划分（Vertical Partitioning）

我们进一步观察和研究了主流计算机的地址映射机制（包括i7-系列，xeon系列等），发现Bank bits与Cache set bits有一部分是重合的，我们将这部分地址位定义为重合位（Overlapped bits, O-bits），O-bits能够同时索引DRAM Bank的物理地址和Cache set的地址。同时，系统中还存在另外两种地址位，即仅能索引Bank的地址位（Bank bits, B-bits）和仅能索引Cache的地址位（Cache bits, C-bits）。如图4.2所示。

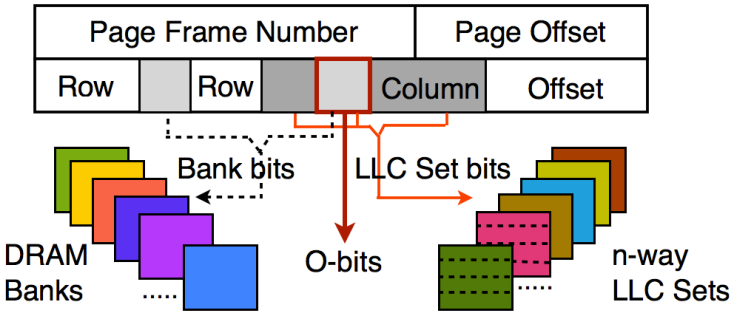


图4.2 主流体系结构中地址映射中的O-bits

因此，从技术的角度来讲，如果我们对O-bits进行“着色”，即按照这些重合的地址位进行划分，那么就可以同时的划分DRAM Bank和LLC。我们形象的将这种划分方式称之为**“垂直”划分**（Vertical Partitioning, VP）。以我们的主要实验平台及环境配置为例（i7-860，8GB主存，64 Banks），如图4.3所示，O-bits有3位（13，14，15位），但由于第13位同时也是L2 Cache（第二级Cache）的索引位，将它作为着色位将会划分L2 Cache，因此，实际的着色机制的O-bits仅使用14，15两位，也就是可以同时将DRAM Bank和Cache划分为等量的四份。此外，B-bits包含两位，21和22位。C-bits由16，17和18三位组成。

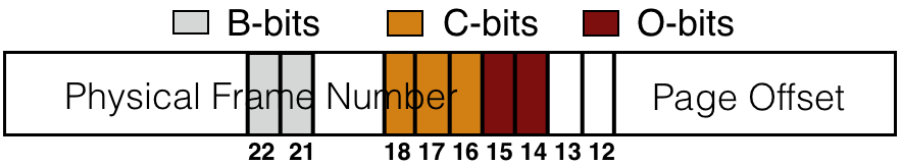


图4.3 i7-860中地址映射中的O-bits

4.4 “垂直”划分体系

本文将内存资源物理地址映射中的索引位划分为三类，B-bits、O-bits和C-bits，分别索引不同内存资源。从图4.1中已经可以得出结论：采用不同索引位的划分对系统性能的影响是不同的。如果划分方式包含了O-bits，那么将垂直的划分Cache和DRAM Bank。而结果也无非是两种可能，要么多级存储器上的划分效果相叠加，系统性能进一步提高；要么则正负相抵消，产生含糊的、从水平划分的角度难以解释的实验结果。

基于O-bits，采用垂直划分，内存储器划分的方式（或划分空间）可以进一步扩展到多维。以O-bits为核心，进行两级的水平拓展，即O-bits + B-bits，在DRAM层次上划分的粒度更细；以及O-bits + C-bits，在Cache上划分的更细。由于我们的实验平台最多只有8核，因此，有意义的最细粒度的划分是将存储资源划分为8份。在其它平台上，根据地址映射的不同，划分方式可能更为多样化。综合来看，如表4.1所示，我们得到多种划分机制，A-/B-/C-VP，加之原有的Cache-Only和Bank-Only划分机制，本文将现有体系结构下存在的所有的可能的划分方式全部揭示了出来，这是以前关于划分的研究从来没涉及过的工作。

垂直划分是本研究的一个贡献。其意义在于两点，第一，揭示了多级存储器之间存在

表4.1 两种水平划分机制和三种垂直划分机制的综合情况

Policy	Coloring Bits	Description	Target Cores
Cache-Only	C-bits {16~18}	LLC → 8 groups	4/8-core
Bank-Only	B-bits {21~22}	Banks → 4 groups	4-core
Bank-Only	B-bits {21~22} O-bits {15}	LLC → 2 groups Banks → 8 groups	8-core
A-VP	O-bits {14~15}	LLC → 4 groups Banks → 4 groups	4-core
B-VP	B-bits {22} + O-bits {14~15}	LLC → 4 groups Banks → 8 groups	8-core
C-VP	C-bits {16} + O-bits {14~15}	LLC → 8 groups Banks → 4 groups	8-core

垂直划分的可能，这是一种协同划分的资源管理方式；第二，解释了Bank和Cache之间在本质上相互关联和影响的内在原因，在采用地址映射中资源索引位作为HASH方式的计算机系统中，这种影响始终存在，也就是说资源“垂直”管理的方式始终有效（划分也是一种资源分配与管理的方式）。这为本文后继研究应用与体系结构敏感的存储管理机制奠定了基础。

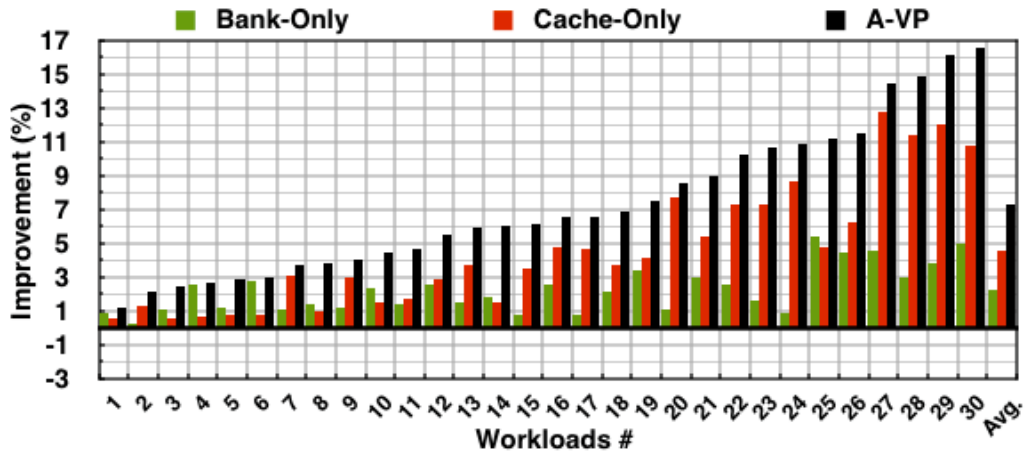
4.5 实验结果及其分析

如前文所述，我们希望垂直划分能够优化第一象限中的工作集的性能，即在当前体系结构下，能够使这部分工作集的运行速率更快，最大化整机的性能。但垂直划分对性能的影响还不在我们的认知范围内。

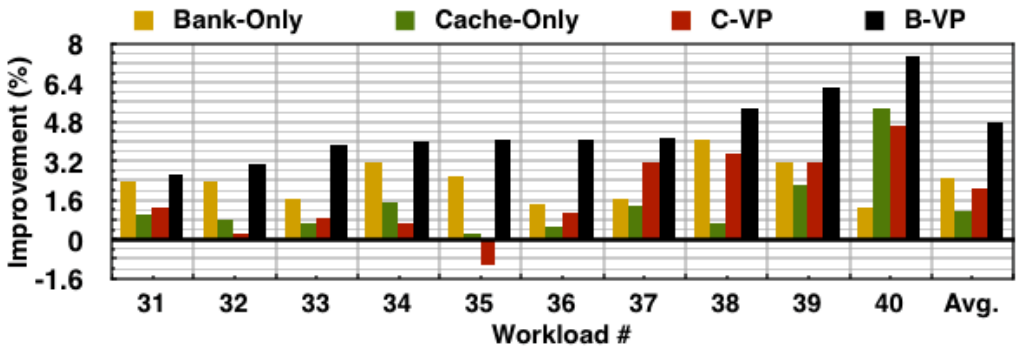
为了验证垂直划分的功效，我们进行了五十次随机的实验。在图4.1的第一象限中我们随机选择了五十组工作集（包括三十组4-programmed工作集和20组8-threaded的工作集）。根据表4.1中的规则与分配方式，本实验对于每一组4-programmed工作集采用A-VP, 对于每一组8-programmed的工作集分别采用B-VP和C-VP。

图4.4综合展示了本次以及前期的Bank-/Cache-Only的实验结果，如我们所料，对于这些随机产生的工作集，与任意一种“水平”划分相比，”垂直”划分都能够更进一步的提高性能。并且，初步的实验结果即展示出不同的“垂直”划分机制的工作集敏感性（我们将这种敏感性定义为“垂直”划分的“友好”性，亦即X-VP Friendly）。显然，从图4.4我们可以看到对于工作集编号31～40，B-VP的性能要优于其它机制；对于工作集编号41～50，C-VP的性能是最好的；A-VP也如是。

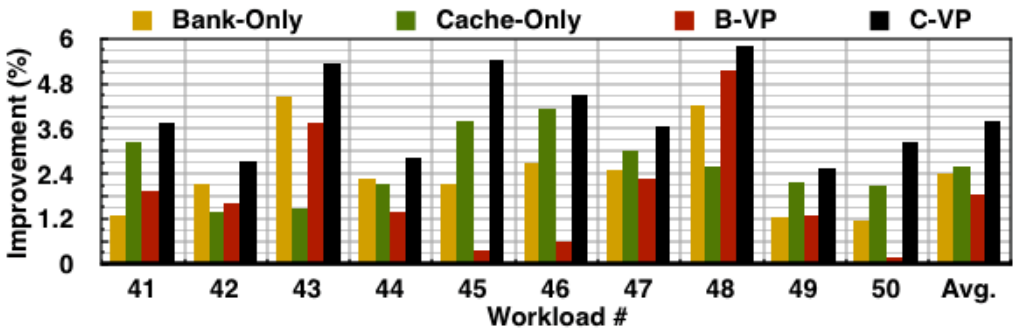
由于VP同时消除了LLC和DRAM Bank两级存储上的线程间访存干扰，因此，在适用的情况下，对性能的提升也要优于水平划分的方式。图4.4（a）中显示了A-VP的最大性能提升将近17%，实际上，从对很多组工作集的观察来看，均形成了划分效果的叠加。平均来看，也要高于传统的单级存储资源划分的方式2～4个百分点。4.1节中提出的关于性能累



(a) 4-programmed 工作集(A-VP performs best) .



(b) 8-programmed 工作集(B-VP performs best).



(c) 8-programmed 工作集 (C-VP performs best).

图4.4 A-/B-/C-VP的性能提升。注：每种VP都有其适用的工作集。

加的问题至此可有定论。另外，本实验也证明了先前的关于划分的研究是不完整的，从机制的角度来讲，先前的研究没有完整的挖掘出潜在的性能优化空间。

垂直划分是普遍有效的。这点从我们随机选择的工作集中即可反应出结果。从图4.5（与图4.1包含相同工作集）中可以看出，第一象限中标出了4.1节中提及的50个随机工作集的分布，基本上覆盖了图4.5中第一象限的范围。因此，本文从理论和实践的两个方面得

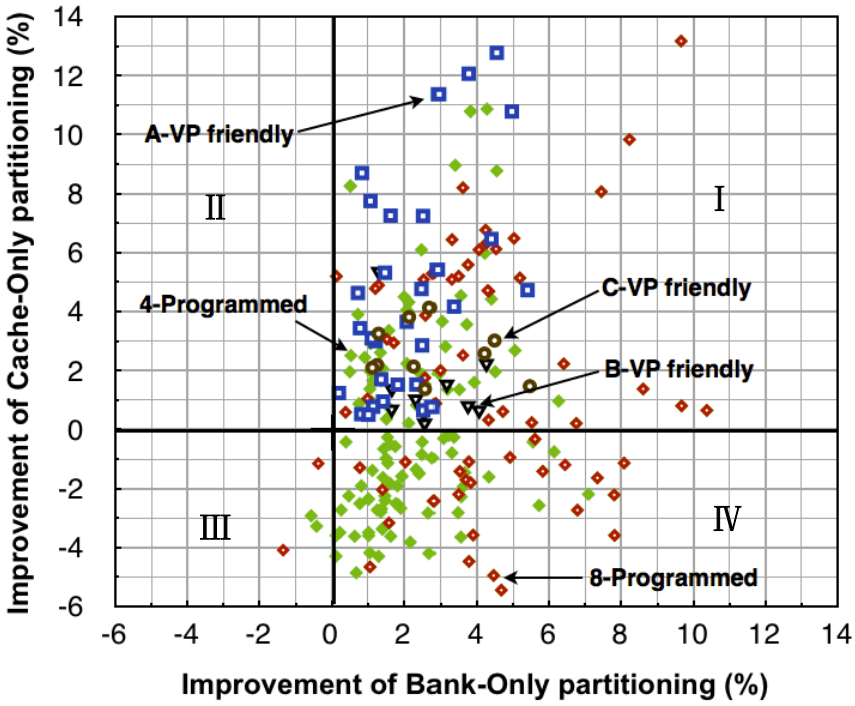


图4.5 A-/B-/C-VP的“友好”性与划分性能总体分布图。注：由不同图案标出了图4.4中的50个随机产生的工作集的分布。蓝色矩形代表A-VP friendly的工作集，黑色倒三角代表B-VP friendly的工作集，棕色圆圈代表C-VP的工作集。

出结论：在Bank和Cache划分均为有效的情况下，垂直划分很有可能会有效，即产生划分效果的叠加。垂直划分所得到的性能提升是稳定的，虽然在有些情况下的性能提升不大（C-VP的综合性能提升不是很明显），但也不能被忽略。

本实验还尝试了对第四象限中的工作集采用VP机制的结果。如图4.6所示，如我们所预料的一样，由于在Cache上的划分的性能的“负”影响很大，VP的性能表现含混不清，不具有说服力。本章后文将进一步分析这些问题。

另一个问题是，如第三章所讨论的带宽变化的影响因素是否也会影响VP？本节还是采用如第三章的调整带宽变化的方式来观察VP的性能变化。此外，公平性也是需要考察的重要指标，本节也会对此进行考量，量化的标准采用第三章介绍的MaxSlowdown。

图 4.7 汇总了不同划分机制下的系统吞吐量和公平性的变化。为了说明这些划分机制在不同内存带宽下的表现情况，我们的实验方法是改变内存频率（与第三章中的实验方法相同），使其在 1333MHz 到 800MHz 范围内变化（相当于减少了系统的带宽），进而观察各种下性能变化的趋势。从图 4.7 中的数据看出，三种垂直划分的平均值优于水平的单独 cache 划分或者单独 bank 划分。特别是，A-VP 比单独 cache 划分提升性能近 6%，比单独 bank 划分提升 5%。而且，当带宽下降导致更严重的冲突时（在我们调整带宽的情况下），三种垂直带来更多的性能提升。因此，我们可以得出结论，垂直划分策略能带来比水平划分更好的性能，而且随着片上计算单元数目的增多和平均带宽下降的趋势，优化的机会也会更多。图 4.7 还向读者展示了公平性与吞吐量相同的变化趋势。

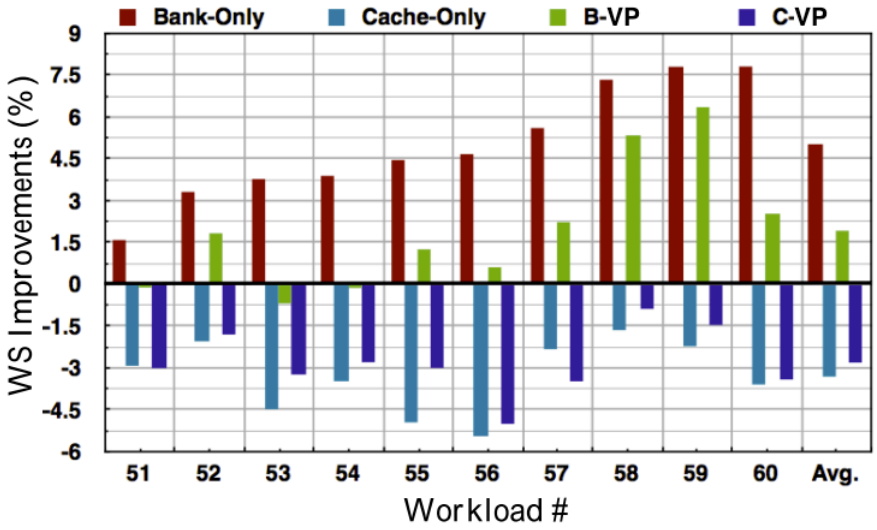


图 4.6 在第四象限中的 10 组工作集的 VP 性能表现

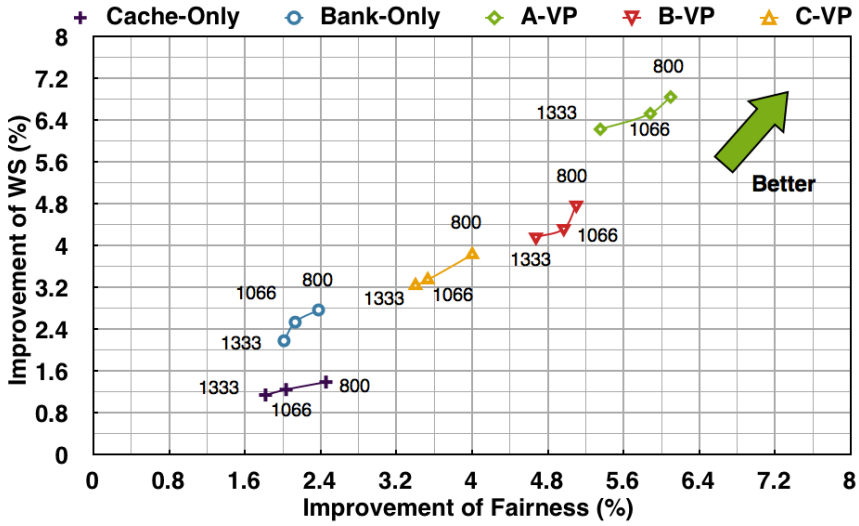


图4.7 由带宽变化引发的VP性能变化图

本文反复谈到未来计算平台的问题，我们希望本研究能够为未来计算机系统的优化提供可靠的参考。目前的研究表明，未来片上计算单元的数量将会持续增长，由多核发展到众核，由几个计算单元发展到几十个计算单元。这会导致两个问题，第一，共享 Cache 的作用下降，因为每个计算单元（线程）能够分到的片上存储资源会逐步减少，竞争反而会加剧。那么 Cache 划分后能使用的资源会更少。第二，平均每个计算单元能够使用的带宽会下降。而且，对于工作集来说，对内存资源的需求量会不断增大。在这种情况下，Bank 划分的优势会不断凸显出来，如图 4.8 所示，在 8-core 的机器上，Cache 划分的效果平均来看已经微乎其微了，因为在很多情况下产生了“负”增长。

如前所论，有可能在未来的系统中 DRAM 已经全部被 PCM、STT 这样的非易失存储介质，数据通道也采用新型的光互联等技术。在这种新型的体系结构下，内存资源的管理与优化将走向另一个范畴。本文第六章展望了新型的计算机架构。

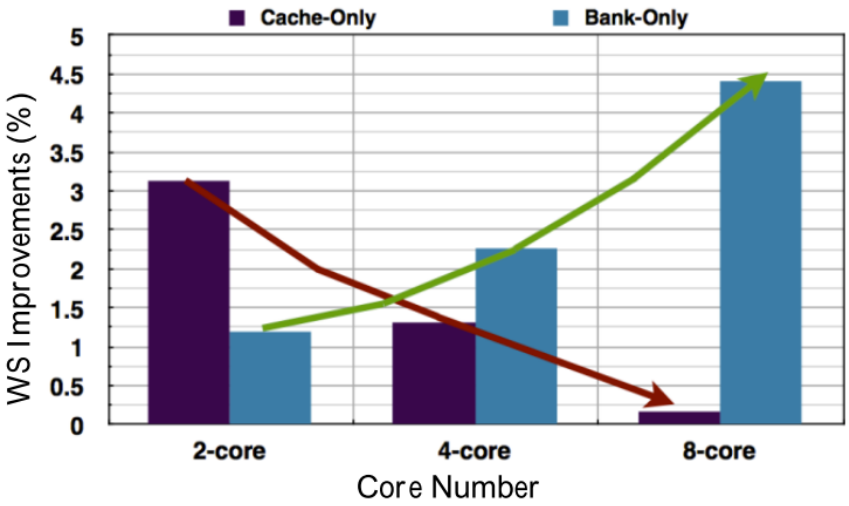


图4.8 计算单元数量和划分机制之间的关系。注：随着计算单元数目的增多，Bank划分机制将会越来越有效。本图平均了随机产生的30余组工作集。

4.6 划分机制与工作集敏感性

4.6.1 导致敏感的主要原因与程序分类

从214组工作集的分布图上看（图4.1，4.5），Cache-Only划分对工作集的敏感度最大，给近一半的工作集的性能带来“负”增长；截然相反的是，Bank-Only划分几乎对所有的工作集都是有效的。对于第一象限的工作集，从初步的实验结果来看，都可以通过某一种垂直划分方式来加速。接下来我们需要进一步探究划分机制与工作集特点之间的亲和性的本质原因。研究这个问题对于如何正确恰当的选择和使用划分方式老说是必要的。而这个问题的难点在于，工作集通常由不止一个程序构成，而每个程序各有各的行为特点、访存方式、资源需求也呈现出多样性，因此，由这些不同程序组成的工作集所表现出的访存特征将更为复杂。

为了把握程序与工作集的特点为调度或者其他优化机制提供依据，前期很多学者的研究从运行时或程序静态的细节入手（如带宽，Cache缺失率, 局部性等参数），采用分析程序参数的方法对运行时环境进行建模，通常还要辅以机器学习和预测的方法，进行任务级或者访存级的调度（如第二章中介绍的方法）。这些方法固然有其优势所在，但缺点也不可忽视。问题在于两点：第一，建模的复杂度和准确性；第二，运行时采样（profiling）的开销。这二者的辩证关系又非常复杂，很难找到恰当的平衡点。例如，如果希望建模的准确性较高，那么势必加大采样的频率，而这将毫无疑问的引发运行时的高开销；但是，如果缺少了建模的准确性，那么优化机制又将会缺乏可信的依据，则必然导致优化的失效，或者不理想的优化效果。另外，如果建模的复杂度太高，则在真实环境下通常会相对“脆弱”，即可用性会受到影响；但是建模的准确性又往往依赖于相对精巧严密的建模逻辑，而这又会增加建模的复杂性。综上所述，我们认为应该采用一种简单可靠的机制来引导优化，并且这种机制应该兼顾可用性和准确性。

4.6.2 基于 Cache 行为的程序分类

与前期的大部分研究工作不同，本文不采用上述的复杂的建模方法，代之以基于分类（Classification）的优化方案与机制。从图4.1和4.5中可以看出，由于Cache-Only划分对系统影响最大（近乎一半的工作集遭受了性能下降），因此，我们首先以Cache资源的敏感性来分类程序。我们对SPECCPU2006中的程序分别进行了资源需求的验证，具体做法是，通过“页着色”技术控制程序能够使用的Cache资源，在每种资源配置下测试并记录程序的运行时间，测试多种情况之后，即可获得程序对资源敏感性的性能变化。如前文所述，C-Bits由三位（16，17，18 bit）组成，我们可据此将Cache划分为8份，实验中分别逐次的赋予程序从8/8（全部Cache资源）～1/8 的Cache资源。性能的变化如图4.9所示。

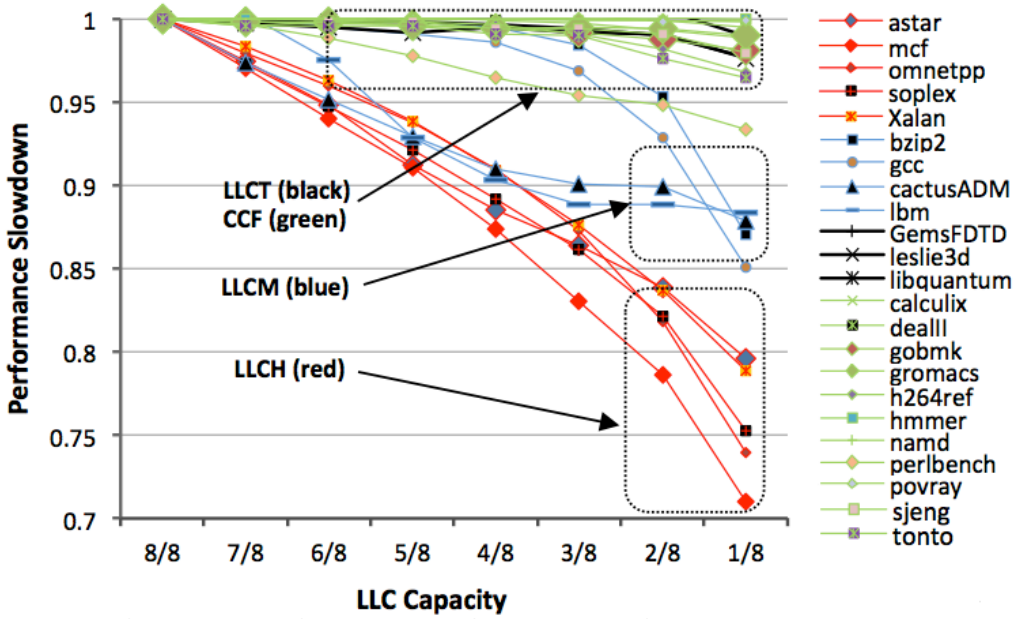


图4.9 SPEC2006中程序的LLC需求量与性能变化之间的关系

红色的线条展示的几个应用程序随着Cache资源的减少而导致的性能下降最为明显，在递减过程中，每减少1/8的资源即引发5%左右的性能下降。在整个实验过程中，这些程序的性能变化呈现出非线性的剧烈下降，如图所示，在仅分配1/8 Cache时，这些程序的性能损失在20%～30%不等。本文将这些程序定义为对Last Level Cache (LLC)需求量高(High)，简记为LLCH。另一类程序，如图中蓝色调示出的曲线，它们的性能下降没有LLCH程序明显，下降趋势也相对缓慢，在1/8 Cache的时候性能损失在10%～15%。我们将这些程序定义为Last Level Cache (LLC)需求量中等(Middle)，记做LLCM。这两类程序都会因Cache资源的变化而受到影响，与它们表现不同的是，如图中所示的绿线和黑线表示的程序，在我们的实验中它们受到的影响最大在5%左右，有些甚至不会受到影响。我们把这些程序归为Cache不敏感类型。深入考察这些程序，我们发现又可以将它们进一步分为两类，一部分程序的带宽使用很少，例如456.hmmer（实测MPKI为0.01，带宽实测为0），444.named

（MPKI为0.05，带宽实测为0.02 GB/S），这些程序是典型的访存非密集型的程序，再加之它们的性能又不受最后一级Cache（LLC）资源变化的影响，我们将这类程序归类为Core Cache Fitting (CCF)，亦即每个计算单元（Core）私有的Cache即可满足程序运行时需求，因此，这类程序将不受LLC资源变化产生的影响；另一类程序，如462.libquantum，在本实验中和CCF程序的表现相同（在图中上部表现为黑色的线条），但是却有很高的带宽需求（462.lib的实测带宽达到6.12GB/S，MPKI为18.5）。综合来看，它们属于”刷”Cache的程序，即时间局部性（Temporal Locality）很差，表现为一个Cache块（Cache Block）被使用之后在相对长的时间内将不会被访问（甚至在程序生存周期都不再会被访问），因此，紧缩Cache资源不会明显降低这类程序的性能，但是这类程序的带宽需求很高，程序在运行过程中需要密集的访存，这种行为特征通常会引发Cache Block的频繁替换（“刷” Cache），本文将这中程序命名为LLC Thrashing（最后一级Cache“颠簸”），简写为LLCT。至此，本文的四个程序分类已经形成，即CCF，LLCM，LLCH和LLCT，真实运算环境中的任意应用程序均可划归为上述分类中的一个。

表4.2中详细列举了SPECCPU2006中的程序程序分类以及对应的MPKI。从程序访存行为分析的角度来讲，MKPI是一个用以区分程序访存行为密集或者非密集的重要指标，描述的是程序对DRAM Bank产生的压力，Cache缺失，也可以间接反映出程序对带宽的需求（本文第三章就是采用这个指标来裁决调度）。在判断和预测VP能够产生的功效的时候，可以参考MPKI。MPKI越大，访存越密集，Bank划分的效果一般会越好。

表4.2 SPECCPU2006中程序分类与MKPI明细

Benchmark	MPKI	Type	Benchmark	MPKI	Type
libquantum	18.50	LLCT	sjeng	0.44	CCF
mcf	15.89	LLCH	tonto	0.28	CCF
lbm	15.25	LLCM	calculix	0.22	CCF
GemsFDTD	12.80	LLCT	gobmk	0.18	CCF
soplex	11.75	LLCH	bzip2	0.11	LLCM
leslie3d	10.74	LLCT	gromacs	0.11	CCF
omnetpp	8.74	LLCH	perlbench	0.07	CCF
gcc	4.72	LLCM	h264ref	0.05	CCF
astar	4.43	LLCH	namd	0.05	CCF
cactusADM	2.59	LLCM	hmmer	0.01	CCF
dealII	2.28	CCF	povray	0	CCF
Xalan	1.68	LLCH			

以上研究的都是单个程序的情况，由若干个程序组成的工作集的访存行为将比上述单个程序的情况更为复杂。我们如何分析一个包含了若干个不同类型程序的工作集的访存行为？工作集的访存特点是什么决定的？这些特点又与划分方式产生什么样的关系？后文将回答继续研究并回答这些问题。

4.7 数据挖掘（Data Mining）—发现划分与工作集“多样性”之间的关系

虽然本文的分类机制已经大大简化了建模的难度和复杂度（仅有4个分类），但是，如第二章所述，访存“多样性”的问题是难以回避的。每个程序的资源需求、访存特征等指标各不相同，若干个程序并发，则情况会更为复杂。逐个分析工作集细节特点往往会将研究引向非常复杂的数学建模问题而距离简单可行的实用办法越来越远。

本文决定采用新的思路进行研究：采用数据挖掘技术建立工作集访存特征与划分机制亲和性之间的关系。对于本研究来说，采用数据挖掘技术是必要的手段，也是我们研究的优势与特点。原因在于，第一，我们的实验结果不是模拟结果，而是真实系统下产生的大量实验结果。这些数据相对具有较高的参考价值，并为发现规律提供了坚实的数据支撑。第二，数据挖掘是分析大量数据并获取知识的必要手段。我们的研究将涉及大量数据，比如，每一个程序都有若干个对其进行描述的访存参数，每一个由若干个程序组成的工作集就有更多的描述参数，再加之大量的实验结果数据，如果不采用数据挖掘的方式而仅凭经验性的分析和建模，则很发现数据背后的规律和知识。第三，结构领域的研究较少与数据挖掘等方式相结合，但却蕴含着机遇，我们期望能够通过计算机技术领域的交叉为体系结构的优化提供指导和揭示潜在的优化空间。第四，为了充分体现工作集“多样性”的特点，本研究收集了近两年的实验结果，包含大量的有价值的信息，采用数据挖掘的手段能够全面的分析这些数据。

4.7.1 基于关联规则挖掘的机制

我们的目的是为了建立划分机制与工作集“多样性”之间的关系，即发现什么样的划分方式对具有什么特点的工作集最有效。这是一个典型的A→B的关联规则问题，从数据挖掘的角度来讲。因此，本研究部分类似采用发现关联规则的数据挖掘Apriori算法的数据挖掘过程，目的在于建立“有A则有B”的关联关系（A→B）。在数据挖掘领域为人所熟知的“啤酒”与“尿布”之间的关联规则即可用该算法获得。由于本研究不是数据挖掘领域的研究论文，因此关于Apriori算法的详细信息请参考相关文献。

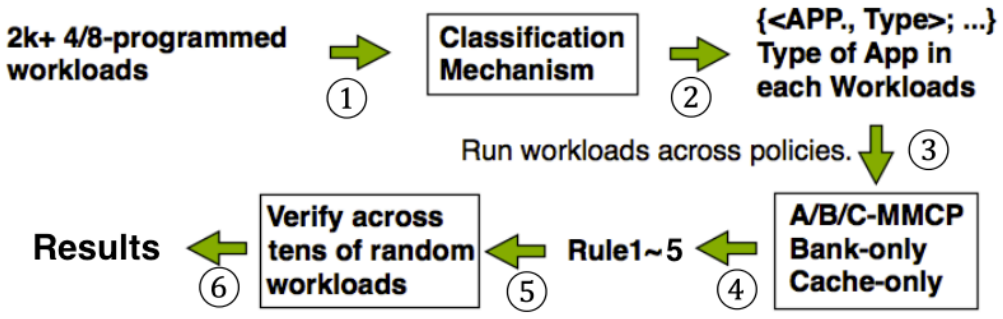


图4.10 本文所采用的数据挖掘的六个步骤

图4.10向读者展示了本文所采用的数据挖掘的全过程。我们收集了2000余组工作集，并对它们进行建模和分类（第一步与第二步），并对这些工作集分别运行本文提及的5种划分策略，收集性能结果。第四步在这些结果中进行挖掘，找到工作集特点分类与最优划分策略之间的关系，并在第五步进行验证和通过专业知识进行解释。

具体过程是：首先，我们对工作集进行建模。基于分类，我们将工作集建立类型向量（表示为程序名称和类型的二元组）。例如，现在有由四个程序组成的工作集{lib, mcf, bzip, hmmer}，这个工作集被表示成类型向量的组合{<lib, LLCT>, <mcf, LLCH>, <bzip, LLCM>, <hmmer, CCF>}。其次，针对每组工作集都运行表4.1中的所有划分方式，并将划分方式与相应的性能提升记录为二元组，例如，<cache-only, x%>, <bank-only, y%>, <A-VP, z%>, 等。x, y, z%即为性能提升的比率。最后，建立挖掘模型的关系表示，即获得最大性能提升的划分方式（Opt. partitioning）与工作集所包含的程序类型（Classification）之间的关系，表示为Classification → Opt. Partitioning，此结果即可揭示划分机制与工作集访存特征之间亲和性的关系，即何种类型的工作集适合采用什么样的划分机制。

4.7.2 数据挖掘结果——“划分规则”

如前文所述，我们的实验除了上述200余组工作集之外又另外使用了2000余组工作集，每个工作集由4或8个SPECCPU 2006中的程序随机组成而成，整个实验过程耗时接近17个月。发现的规则包括两部分，第一部分是三条划分规则，本文称之为**“划分规则”**：

规则（1）：如果包含一个或若干个LLCT程序的工作集，无论工作集中的其它程序属于什么分类，则是A-/C-VP友好型的工作集，即通过A-/C-VP可以获得最佳性能。（支持度：37.1%，置信度：94.4%）

规则（2）：如果工作集中不包含LLCT，但包含LLCH或同时包含LLCH和LLCM，则是Bank-Only友好型的工作集，即采用Bank-Only划分可获得最佳性能。（支持度：34.3%，置信度：83.3%）

规则（3）：如果工作集中包含LLCM，但不包含LLCT和LLCH，则此工作集是A-/B-VP友好型，即通过A-/B-VP可以获得最佳性能。（支持度：23.8%，置信度：87.9%）

注：A-VP适用于4程序（或小于）组成的工作集，C-/B-VP适用于8程序（或小于8个程序但多余4个程序）组成的工作集。

按照数据挖掘的步骤，得到的规则需要依据专业知识的解读和分析 (polishing) 才能形成知识。因此我们通过体系结构相关的知识“抛光”这些规则，而对于本文来说，这种分析可以解释VP对于工作集敏感性的原因。规则（1）之所以有效的本质原因在于，LLCT程序在运行时会干扰其它任何程序（也可以理解为LLCT对Cache的“压力”是最大的），而它自己又不会得益于更多的Cache资源。因此，A-/C-VP把Cache资源划分为与程序数量相同的等份，能够完全隔离开LLCT对其它类型的程序的干扰，也就对于包含LLCT的工作集能够起到较好的效果。规则（2）有效的原因是Bank-Only划分没有人为的限制任何程序对Cache的使用，但却消除了包括访存密集型程序在内的所有程序在Bank上的访存干扰，因此对于

Cache敏感性的工作集（图4.1、4.5中第四象限中的工作集）最为有效。规则（3）有效的原因是LLCM类型的程序本身对Cache的资源的需求量不高（1/4左右的Cache资源恰好能满足这类程序的需求），并且划分又能消除程序间的资源竞争。

仔细分析这些规则，可以进一步获得引导内存资源分配的策略。根据规则（2），包含LLCH的工作集在Cache划分情况下会损失性能。这说明在不包含LLCT的工作集中，LLCH处于主导地位并决定着工作集的访存属性，即对Cache资源的需求较高。如果不满足其需求，那么LLCH性能的下降将会影响整个系统运行时的吞吐量。将规则（2）与规则（1）相结合，如果工作集中同时包含了LLCT和LLCH，Cache划分就会提高系统吞吐量。这说明了LLCT对其它程序的干扰很大，并且，这种干扰所带来的性能损失要大于在减少任何程序的Cache资源时所引发的负面效应，只有这样，在Cache划分的情况下将消除LLCT对其它同时运行的程序所产生的干扰，系统性能才会得以提高。基于以上分析，本文得到一个知识，即对程序“侵略性”的强弱关系的排序：LLCT > LLCH > LLCM > CCF。这个关系说明了LLCT类型的程序会干扰与其并发运行的任何类型的程序，而LLCH和LLCM类型紧随其后，CCF通常不会干扰其它程序，但最容易受到其它程序的干扰。这条知识给我们提供了一个划分的原则：一定要将LLCT程序与其它类型的程序隔离开。该原则将在动态优化的过程中起到指导作用。

4.7.3 数据挖掘结果——“融合规则”

以上我们讨论的划分效果都是在均分的情况下产生的。但是，对于真实环境而言，完全平均的划分未必能获得最好的优化结果。为了应对真实环境下程序复杂的访存行为，以前的研究通常采用动态调整每个线程的资源使用量（结合采样、建模、预测等手段），但是效果并不理想［122］。那么对于“垂直”划分机制而言，我们应该采用什么样的资源管理方式以应对真实环境下的复杂情况？

为了回答这个问题，本文依然诉诸于数据挖掘方法。我们定义了新的数据挖掘模型，将每一个工作集表示为：<(n×LLCT, m×LLCH, p×LLCM, q×CCF), x%>, n, m, p, q表示在此工作集中某类程序的数量，x%表示Cache-Only划分能够提高的系统性能。因为Cache划分的波动很大，而Bank划分基本上都会有收益，因此我们主要讨论如何提高Cache划分的收益并避免其不良影响。以此模式挖掘的结果是：对于n=0, m+p>0的情况，性能总是降低的（x%＜0）；对于m=p=q=0, n>1的情况，性能提升不高（x%<1%）；对于CCF为主的工作集(n=m=p=0, q>1)，几乎观察不到性能提升。

实验结果反复证明，在Cache上完全采用划分策略对于包含LLCH的工作集是行不通的，对于{LLCH, LLCM, LLCT}类型的工作集也未必能达到最好的效果，这点和规则（2）的隐含意义吻合。我们可以理解为，基于主流计算机采用的LRU替换算法的Cache能够较好的处理类似LLCH和LLCM类型程序对资源的需求，但容量问题对于这类程序来说则是首要的。在这种情况下，允许Cache共享（不进行划分）实际上等于满足了LLCH和LLCM类型程序的基本需求，而Cache替换算法能够缓解Cache上的压力，因此，我们只需要消除Bank

上的访存竞争即可获得性能提升。至此，本文得到另一个原则，称之为“融合”原则：LLCM和LLCH程序应该融合（使用同样的颜色），这样它们可以使用更大的Cache空间和资源。关于多个LLCT程序同时运行的情况，它们应该融合在一个小的Cache份额中（1/8 Cache在我们的实验平台上）。与LLCT的融合类似，多个CCF也应该融合在一个小的份额中，因为它们自身对Cache资源需求很少，产生的干扰也少，但为了保证它们的性能，还应该将它们与属于其它分类的程序隔开。

这里本文总结一下能够指导划分的所有原则：第一，属于同一个分类的程序之间可以融合（共享同一种或几种颜色）；第二，不同分类之间需要隔离（LLCM和LLCH除外）；第三，在资源有限的情况下，LLCT类型的程序共享最小份额的资源即可。CCF的程序也类似。如图4.11所示。

数据挖掘的分析结果也同样支持“融合”规则：

规则（4）：LLCH和LLCM程序可以共享划分的颜色（融合在同一块较大的空间里）。（支持度：39.5%， 置信度：87.2%）

规则（5）：LLCT和CCF程序分别可以融合在一块较小的Cache资源里。（支持度：7.8%，置信度：90.5%）

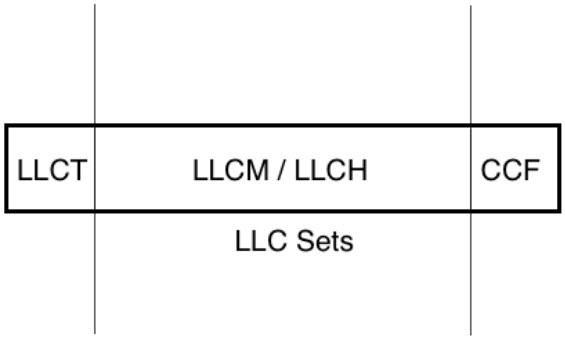


图 4.11 理想状态下的划分与融合

4.8 本章小节

本章是本研究中至关重要要的一章。本章的内容主要涉及到内存储资源的“垂直”划分机制，工作集敏感性问题的研究，以及相应的资源管理策略。本章将数据挖掘中的技术引入到体系结构的研究中来并收到了良好的效果，我们希望这中方式能够启发其他学者和我们自身将来的研究。

在本章的开始为本文归纳并设计了几个问题，并分别在本章的不同章节给予了答案。总的来说，第一，DRAM 和 Cache 的划分带来的性能变化是可以相互影响的，根据工作集的不同特点，影响不同，有时候会出现叠加的现象，而有些时候则会相互抵消。第二，“垂直”划分能够给系统的性能带来额外的好处，对于本章节中提到的第一象限中的工作集，总的来讲都能带来好处。第三，现有的参考文献中还没有彻底的对页着色技

术进行过透彻全面的研究，因此很多数据结果都是含混不清的，即不能分辨，也无法参考。比如，很多研究 Cache 划分的工作就将 LLC Set 的索引位全部作为可划分的范畴，而实际上引入了对 Bank 的划分，而在文章中又没有这方面的讨论，因此，参考价值不大。另外，对 DRAM Bank 划分的研究也如此，都没有考虑或深入的分析 O-bits 带来的性能影响。而本章则对此做了一个总结。第四，本文基于对划分机制系统化的研究，较为彻底的分析了划分机制与工作集敏感性之间的本质关系，为未来关于访存调度的研究提供了一些参考。

第五章 x-Buddy System: 用多种策略处理访存“多样性”

无论何种形式的“划分”，其实质都是一种内存资源管理与分配的机制。本文认为，访存优化的问题的实质在于如何处理好访存行为的“多样性”。本研究已经证明，内存资源的管理与分配方式影响程序的访存性能，因此，本研究最终的落脚点是，通过优化系统资源管理与分配的核心机制来优化多核系统中访存行为，即用多种资源的管理与分配策略来处理访存的“多样性”。

第四章的实验已经证明了VP（垂直划分）的有效性，即VP利用了计算机微体系结构上的特征（O-bits）,有助于提高系统的整体性能。但VP同样也是工作集敏感的，前期研究证明了其敏感性主要是因为程序对Cache资源的敏感。我们将这种敏感理解为程序对资源需求的“多样性”。第四章的研究工作向我们刨析了多样性的根源，并且证明了采用不同的资源管理策略以应对多样性能够更好的发掘计算资源的潜力，但还没有设计并实现出一套完整的运行时系统，以应对多样性，这正是本章需要解决的问题。

本章我们提出一种完全系统级的“应用与体系结构特征敏感的内存管理模型”（Application-Architecture-Aware Memory Management，AAA-Memory Management），这是一个访存优化的概念模型，即针对工作集访存行为的特点选择最优的资源分配方式。其核心是支持多种分配策略的x-Buddy System，是对Linux内核存储管理Buddy System“伙伴系统”的扩展。本研究希望通过优化资源管理与分配的方式在根源上提高访存效率，出发点即与细粒度的访存优化的调度策略不同，但本文认为，本研究更接近访存优化问题的本质。

5.1 AAA-Memory Management概念模型的运行时逻辑

我们的目标是，在真实的运行环境下，识别程序的分类为当前运行的工作集选择最合适的资源管理策略，并根据程序的访存特点调整资源分配方式。具体的工作流程分为6步，如图5.1所示，1～4步骤将工作集的程序分类，第5步选出最优的资源分配策略（在第四章根据数据挖掘的经验经结果），第6步是基于x-Buddy System的资源分配。

要使这一整套机制能够运转，需要解决以下的难点：

（1）低开销的系统级动态获取程序分类。我们希望设计一整套操作系统级的机制，而不是采用类似与硬件计数器的辅助硬件来协助系统的运行。硬件计数器通常会引入较高的采样开销，很难在开销与收益之间权衡。我们的系统已经引入了复杂的逻辑，如果再继续采用高开销的操作将会得不偿失。更为重要的是，我们不希望通过违反操作系统设计原则的方法来达到目的，系统级的问题即在系统级解决，如果引入了硬件计数器或其它的辅助设备，则破坏了操作系统设计的原则。（2）需要内核支持多种不同的物理页分配方式（A-/B-/C-VP等均需要不同的分配方式），并且能够低开销的在各种机制之间动态的切换。（3）需要一套高效的物理页检索机制。原始的Buddy System能够在O(1)的时间内返回一个

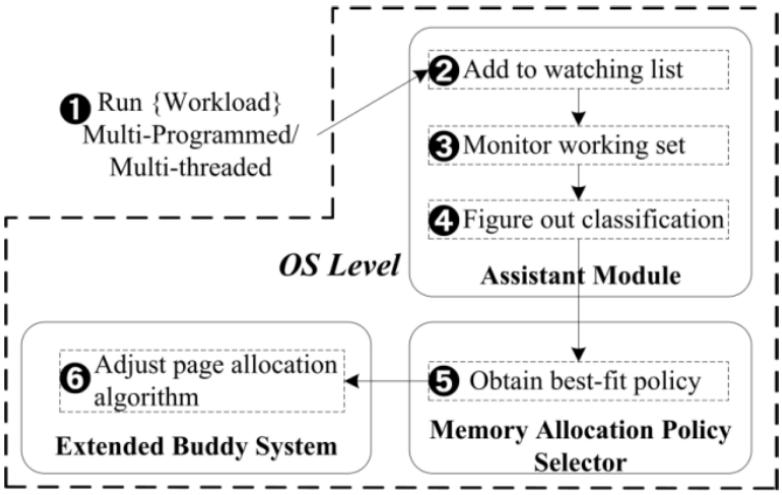


图5.1 AAA-MM的运行时逻辑框架

随机的物理页面，采用本文的系统之后也不能增加系统的时间复杂度，需要在O(1)的时间复杂度内返回属于任何颜色的页面。

5.2 基于页表项（PTE）信息的采样

5.2.1 程序的访存行为与PTE

如第四章所述，我们将程序分为四类（LLCH, LLCM, LLCT, CCF）的依据是程序在Cache上的需求量与行为，而我们在第四章所采用的分类方式需要在程序运行前做采样。在真实的应用场景中，始终采用静态采样或分析的方式分类程序是不受欢迎的，而通常需要在线的（on-the-fly）动态的分类机制来应对真实的应用。本文中我们反复提到了通过PMU的采样方式并不适合本研究，我们始终采用的是操作系统级的解决办法，不希望通过硬件计数器来辅助系统运行。此观点是从系统设计的原则出发来考虑的，一个典型的问题是，硬件计数器通常会因为平台的变化而变化，导致适用于某个平台的机制在别的平台上不使用（即不能适应硬件系统的多样性），我们的机制希望尽可能的降低这种体系结构差异带来的影响［20,24,31,53,110］。

主流操作系统对于页表的设计与实现给我们提供了机遇，启发了我们通过监控页表采样程序行为的灵感。在操作系统运行的过程中，任何一个被使用的物理页面都有一个对应的PTE来描述其与虚拟页面的映射关系，即建立虚实映射（注：在现代操作系统的设计中，每个进程都有自己的地址空间和页表）。在PTE中，有一个__access_bit，用来描述这个物理页是否被近期访问过。如果被访问过，将这个位置1，否则这个位的值是0。有鉴于此，我们将__access_bit为1的页面定义为“热页面”（Hot Page），因为该页面在近期被线程访问过；相对的，将__access_bit为0的页面定义为“冷页面”（Cold Page）。__access_bit是由系统硬件来控制的，每次CPU访问某个物理页面的时候机器自动将它置1，表示已访问过该页面［59，65，120］。

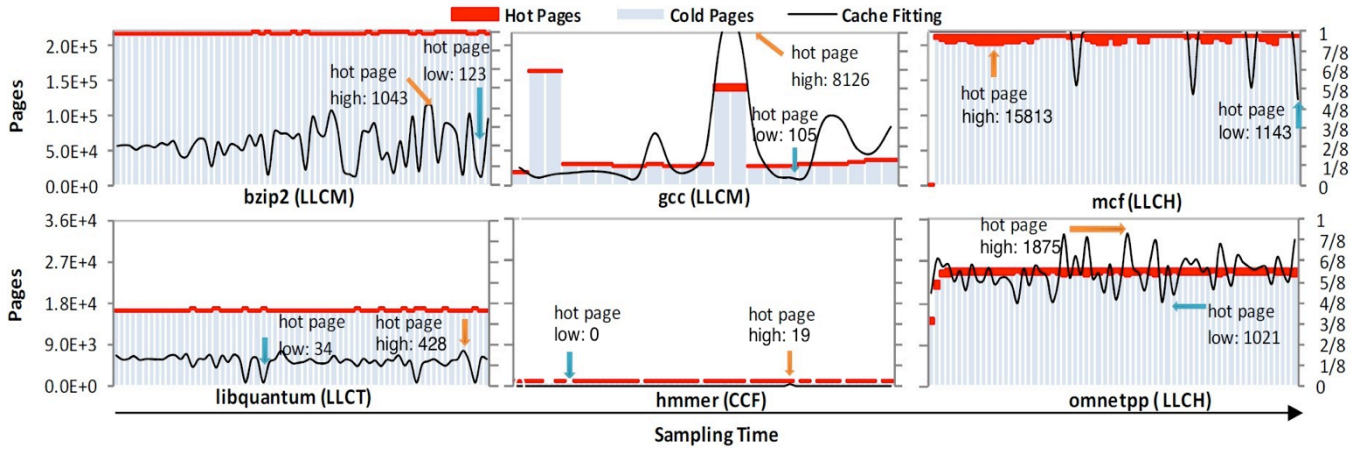


图5.2 “冷”/“热”页面与访存行为特征

针对程序访存行为的研究表明，一般情况下，某个程序频繁使用的物理页面（在某个“采样窗口”或者时间区间）仅占分配给该程序的全部的物理页面（allocated pages）的一部分。并且，根据局部性原理（DRAM的行缓冲局部性），一个4KB大小的页面通常是被连续访问的［120］。基于此，本文提出一种运行时Cache使用量的估算方法，该方法利用在某个时间区间内被访问的物理页面的数量来估算对Cache资源需求量。具体做法的描述是，定义一个时间区间（或称作采样窗口），采样这个时间区间之内的程序行为，该区间的长度可根据系统具体环境的情况来设定。在采样窗口的起点，我们将待采样程序的PTE的__access_bit全部置0。在采样窗口的终点统计被置为1的PTE的数量，我们将这个数量看作“热页面”的数量（Number of Hot Page, NHP），即在这个时间间隔内被访问过的所有物理页面的数量。本文采用放大估算的方式，对Cache的需求即为将这些“热页面”全部载入Cache时的资源使用量（NHPx4KB），真实的需求会小于这个值。类似的，以物理页面为单位可以将Cache的容量表示为NPC（Number of Pages in LLC）。整个采样的过程，通常，可能会包含若干个采用窗口，也可能会是整个程序的运行生命周期。具体情况见后文关于开销的讨论。

到目前为止，通过简单采样并估算容量的方法可以区分LLCH, LLCM, 和CCF类型的程序。如图5.2所示，用NHP/NPC（上取整）表示对Cache的需求量，本实验我们采样了所有SPECCPU2006中程序的整个生命周期，采样窗口为10us。在图5.2中，我们列举出6个有代表性的程序。以hmmer为例，它属于CCF类型程序，在我们的实验平台上，整个生命周期中NHP最大值仅为19，最小值为0（在采用窗口内没有访问主存的行为）。从图5.2中也可以看出Cache Fitting曲线（黑色线条）非常低（几乎在我们的坐标中不能被识别），对应的Cache容量相应的也仅在1/8以内。因此，这类程序对LLC的需求一定是不高的，他们对高速缓存的需求在L1和L2即可满足。明显形成鲜明对比的是LLCH程序，以mcf为例，在一个采样周期内NHP最大为15813，最小为1143。从图上可以看出，绝大多数的采样周期mcf均需要占据整个Cache（与图4.9的情况相同），即使在几个产生最小值的几个采样周期也需要不低于4/8的Cache资源。同样属于LLCH的omnetpp程序也展示出了平均Cache需求不低于

3/4的程序行为。对于LLCM类型的程序，从图上可以看出他们的Cache Fitting曲线远高于CCF，但低于LLCH，基本平均维持在［2/8，4/8］的区间内，这个现象同样也符合我们第四章的实验结果。

5.2.2 WPD: 页面的加权分布

在真实的应用场景下，我们需要考虑例外的情况。第一种情况是，用统计“热页面”数量的方式不能识别LLCT类型的程序，原因是有些LLCT程序虽然对其它程序的干扰很大但是其访问的热页面数量不算多，例如462.libquantum，最高的NHP的数量仅是428（如图5.2所示），如果用热页面的数量作为依据会将这类程序误归类为LLCM，因此我们需要通过其它特性来识别LLCT类型的程序，我们将在后文介绍。另一种例外情况是，如果程序的DRAM的行缓冲局部性很低，在一个采样的时间窗口中虽然访问了很多页面（出现很多“热页面”）但是读取的数据并不多（每一个页面可能仅读取64B或者几个Cache line的数据），如果按照上述的最大值估算方法会将这种程序错误的归类为LLCM。此时，需要一种简单的方法反映出程序的这种对大量页面进行非密集型访问的行为。本文定义了一个指标来描述对页面访问的权重，加权访问页面分布（Weighted Page Distribution, WPD）。WPD越大，说明这个程序对“热页面”的访问频度越高，从另一个角度也反映了程序的行缓冲局部性较高；反之，程序对“热页面”的访问频度就相对低，说明了对页面是“蜻蜓点水”式的访问，行缓冲局部性也相对低。

为了计算WPD, 我们需要重新设计相关的数据结构及算法。我们需要统计每个页面被访问的频度，这需要一个辅助的计数器。对于每个进程，我们为它的页表添加了一个“影子”数组（Shadow），每一个页表项对应Shadow中的一项，Shadow的大小与进程所需的页面数量相等（实验中我们通过VMA来计算这个数量）。使用一个循环来不断的将__access_bit清0，如果该页被访问则又会被置为1，在Shadow数组中将对应项的数值增1。在若干次循环之后，Shadow中的数据即可反映出这段采样区间内页面被访问的频率，这是我们计算WPD的依据。根据我们平台的具体情况，我们选择两次采样的间隔为3秒，循环次数为200次，即每3s进行200次的循环，每次循环等待10us。200次循环之后，根据统计到的访问次数，我们将这些页面分为5类，VH [150, 200], H [100, 150], M [64, 100], L [10, 64] and VL [1, 10]。all_used_pages_num表示为至少被访问了一次的页面的总和（200次循环后）。

WPD= (2xVH + 1.5xH +1xM + 0.5xL + 0.1xVL) / all_used_pages_num

这里要特别说明，通过页表Shadow可以帮助我们识别LLCT。在我们的实验中我们发现了LLCT程序总是展现两个重要的特性，第一，LLCT属于访存密集型程序，在运行过程中这类程序访问的物理页面数量通常要比CCF多，与LLCM或LLCH接近。第二，由于这类程序在行缓冲和Cache行上的时间局部性都很差（类似于Stream的“流”式访存）但是会持续的逐个访问连续的页面（访问结束后即丢弃），因此，它们在多个采样区间内访问页面的数量往往差异不大，这点和其它类型的程序明显不同。从我们对SPECCPU2006中的程序分析来看，LLCT的变化不会超过5%（462.libquntum的差异在1％以内）。

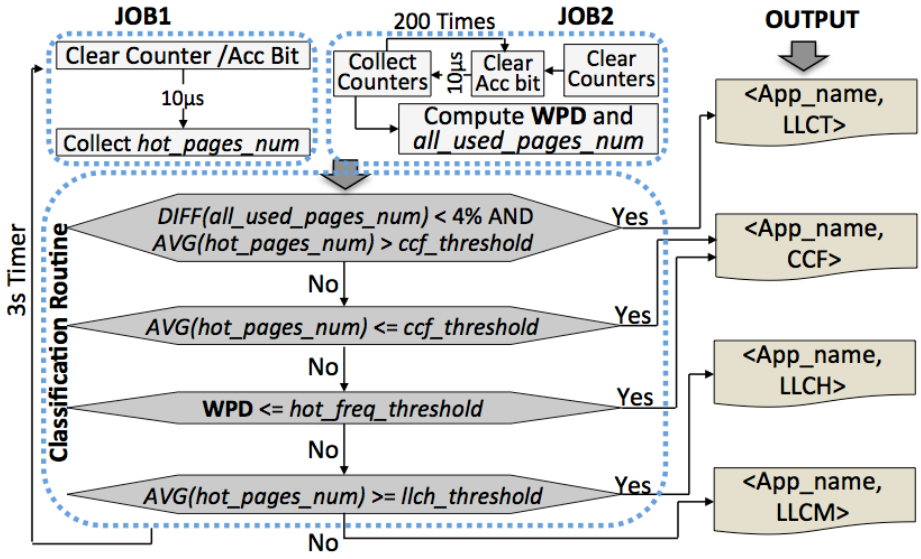


图5.3 内核级动态程序采样与分类的逻辑图

5.2.3 采样机制的完整逻辑与实现

基于以上的分析与设计，我们开发了一套以内核模块形式存在的机制来完成采样，在运行时插入内核。如图5.3，这套机制包括两个任务（JOB1和JOB2），JOB1用来识别LLCM，LLCH和CCF；JOB2用来辅助识别CCF和LLCT。整个过程的流程如下图所示（ccf_threshold，hot_freq_threshold，llch_threshold 分别是 100, 10%和1000，时间窗口是10us）。

我们在算法中使用了一些经验值作为参数，这些数据是比较可靠的，因为它们源自于对全部SPECCPU2006中的程序的分析以及在大量的混合运行工作集上的测试与验证。在真实系统环境中，由于系统之间的种种差异，我们推荐用户根据具体需求校准这些参数。另一个问题是关于我们采样方法的开销，主要来源于几个方面：（1）对页表的遍历。采样是针对每个进程分别进行的，进程页表大小不同即决定着采样的开销不同。对于一般程序来说，遍历整个页表的时间开销在微秒（us）级，但对于总容量在1GB左右的程序来说，这个开销将会上升到毫秒（ms）级。（2）对Shadow数组的统计以及计算WPD。这个过程将引入200次的循环，每隔3s即进行一次，因此这个开销有可能会影响系统性能。但这些开销并不是不能降低的，例如，虽然JOB1和JOB2的采样周期都是3s，但随着运行的稳定（分类不会频发的发生变化）这个采样周期的频率可以逐步被减缓，开销也就相对的被减小。我们将在本章节的后文详细讨论开销的数据和降低开销的方法。另一个能够降低采样开销的方式是以随机的方式遍历页表，随机选择PTE，这样就避免了遍历整个页表带来的开销。对于大部分程序来说，随机选取10%左右的PTE即可达到反应整个程序行为的目的，因此开销可以进一步降低。

对于本研究来讲，采用这种采样机制是由任务需求和系统要求两方面决定的。从任务需求的角度来讲，本研究需要量化程序运行时对Cache资源的实际需求和物理页的使用情况（冷热度，访存足迹），这点通过PMU采样的方式是不能直接做到的，而我们的方法可

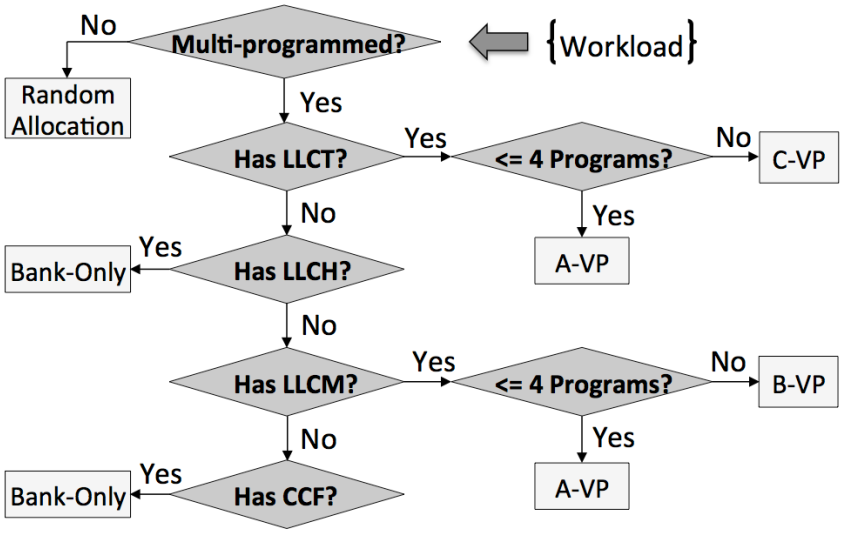


图5.4 决策树的构成

以高效准确的完成任务；从系统要求的角度来看，如前文反复提到的，我们不希望违反操作系统的设计原则。

5.3 资源分配决策树：选择最优的资源管理方式

将工作集中的程序分类结束之后就该为工作集选择最优的资源分配策略。分配的依据是第四章数据挖掘之后规则与结果。一个工作集的访存行为特点往往是由其中占“主导地位”的程序所决定的，在第四章本文介绍了这个关系LLCT > LLCH > LLCM > CCF。例如，如果工作集中出现LLCT类型的程序，它就是处在“主导地位”，对其它的程序影响最大，因此要采用规则中针对这类程序的划分机制（A-/C-VP）。详见4.7.2节的详细规则介绍。

真实的计算环境中往往会运行真正的多线程（Multi-threaded）的工作集。第三章曾经讨论了划分技术对这类工作集的性能提升不高，并不是最有效的方法。H.Park等学者在ASPLOS’13中介绍了M^3机制，将多线程程序的访存打散，以物理页为单位在所有Bank上做交替。实验证明这种方法对于多线程工作集的性能提高要明显优于划分机制。本文中的系统也支持这种随机分配，而且实现方式更为简单和高效。

如图5.4向我们展示了一颗决策树，根据工作集的特点选择最优的内存资源管理与分配方式。可以看出，这个决策树包含了有效的5种分配机制（Bank-Only，A-/B-C-VP,以及类似于M^3的随机页面交替方式）分别针对各种类型的“多样性”的工作集。如本文之前所分析的情况，LLCT类型的程序对其它任何类型的程序都会产生干扰，因此，一旦工作集包含这类程序则必须将它们与其它程序隔开。Bank-Only划分则在不划分Cache资源的前提下消除了DRAM层次的干扰，对于LLCH/LLCM占主导地位的工作集来说会有利于整体性能。需要注意的是，这个决策树中没有包含Cache-Only的划分，因为本文在第四章已经证明了这种机制会导致在很多情况下的性能下降（在工作集仅包含LLCH或LLCM类型的程序），

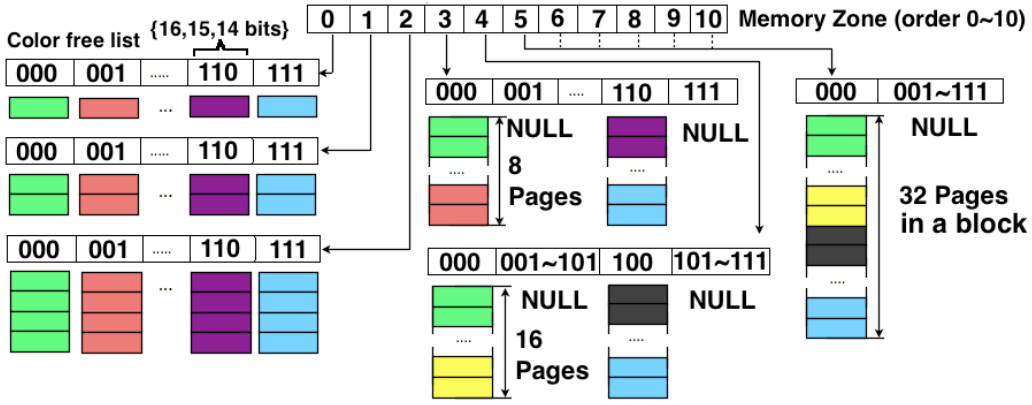


图5.5 基于2个O-bits (bits 14,15)和1个C-bit (bit 16)构建的Sub-system A

而在Cache-Only划分有效的情况下（图4.1/4.5中的第一象限），都可以使用VP来获得更好的系统性能。

完美的情况是系统上运行与计算单元数量相等的线程（4或8线程），但在现实环境中往往不一定出现这种情况。图5.4中的内存资源分配决策树也考虑这种情况，必要的时候将会启用资源的融合，原则是采用4.7.3章节中提到的“融合规则”。例如，如果系统中仅运行了3个线程，那么将会在最优的机制下调整资源分配（并不是均匀的分配资源）。随后的章节将会详细的介绍这部分并进行评测。

5.4 x-Buddy System的实现

本节我们详细介绍如何对Linux的内核的Buddy System进行修改以支持各种上述的资源分配机制。我们将这种能够在线的支持多种资源分配方式的Buddy System命名为x-Buddy System (简记为x-Buddy)。如上文所述，x-Buddy必须能够同时支持Bank-Only，A-/B-/C-AV，以及Random-Interleaved的分配方式，并且，运行过程中必须尽可能的将开销控制在最小，如本章的前文所描述的情况，必须在O(1)的时间复杂度返回属于任何颜色的页面。为了达到这个目的，我们扩展了Linux原始的伙伴系统，加入了两套协同并行的索引机制（Sub-system A和Sub-system B），分别用来支持A-/C-VP，B-VP和Bank-Only划分。

5.4.1 Sub-system A

此索引子系统通过2个O-bits（bit 14,15）和1个C-bits（bit 16）将物理页面重新组织。如图5.5所示，将Buddy System重组包括以下几个重要的改进：（1）在每个对应的order下创建一个以颜色为区别的“颜色列表”（Colored Free List），每个颜色列表又包含对应的“颜色槽”（Color Slab），物理页面按照其自身颜色被挂在相应的槽中。例如，如图所示，在order为0的颜色列表里，从000～111分别排布着对应的8种颜色的页面，每种颜色槽中的页面都是物理地址不连续的单个页面（2^0=1）；再如，order为1的颜色列表里，每个块包括2个连续的物理页面（2^1=2），并且颜色相同。这是因为真正有效的着色位是从14位开始的，12

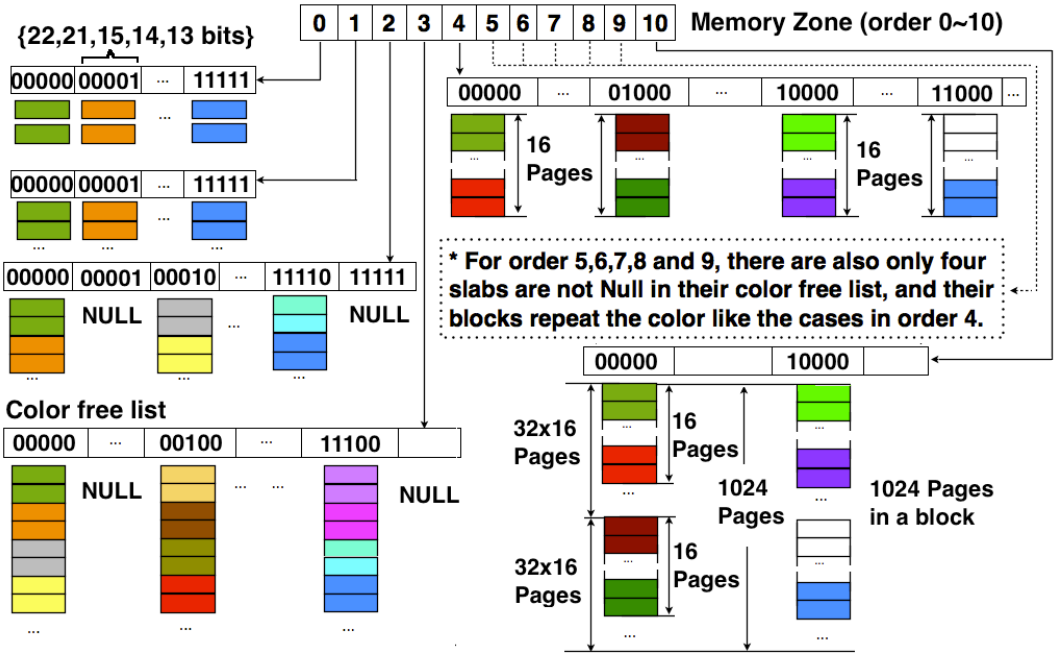


图5.6 基于3个O-bits (bits 13,14,15)和2个B-bits (bits 21,22)构建的Sub-system B。注：本图与图3.6是一样的，为了方便阅读与对比，将该图重置于本章。

和13位不是着色位但却是物理页面的有效地址位，因此，连续的4个页面的颜色是相同的，在这种着色机制下。Order为3的时候需要每个块包含连续的8个物理页，刚好包含了2种不同的颜色（每块的第一页面的颜色决定了本块的颜色属性），但同时相邻的颜色槽里则没有页面。以此类推，在随后的颜色列表中，每块所包含的页面的数量成2倍增长，32个页面即可涵盖全部的颜色种类。需要注意的是，在order为6之后的颜色列表中只有000代表的颜色槽不空。

如前所述，Sub-system A支持A-VP（4-programmed），在分配页面的时候，对于任意线程，在order为0的颜色列表中选择14，15位相同的两种颜色的页面一并分配给此线程，16位不作为区分位。例如，将100和000代表的颜色的页面统一分配给一个线程，将010和110代表的颜色的页面都分配给另一个进程，因此，这个两个进程在00和10索引的存储资源（LLC和DRAM Bank）上均不发生共享，从而不会产生竞争并导致系统性能的损耗。

5.4.2 Sub-system B

因为着色位的差别，同样的页面在不同的索引系统里被表示为不同的颜色。Sub-system B通过5个B-bits（13，14，15，21，22 bit）组织页面，其中14，15位是O-bits。Sub-system B的每个颜色列表都包含32个颜色槽，对应32种不同颜色的页面。不同之处在于，由于从第13位开始着色，那么有可能连续的且属于同一种颜色的页面数量最多只可能有两个。因此，我们从图5.6上可以看出，在任何一个包含4个以上页面的块中，每两个页面即出现一次颜色变化。这种页面颜色变化的规律（连同Sub-system A中的规律）将被我们用来进行目标页面的检索（见后文的hash算法）。Sub-system B可以支持Bank-Only划分和B-VP。与A系统中提到的方式类似，对于Bank-Only，只要在这32种颜色中不重叠的选择不同的颜色

的页面分配给进程即可；但对于B-VP，分配是一定要以14，15和22位这三位为区别位（如第四章所述），即对不同进程所分配的物理页面的这三位必须是不同的。

除了Bank-Only划分和B-VP，Sub-system B还能支持类似于M^3的随机访存。这里需要重新回顾一下M^3的核心思想，该机制将伙伴系统中所有的连续页面都打散，尽可能的将物理页面的分配随机化。那么，对连续的物理页的访问即有可能被分散到不同的Bank上，因此，降低了在线程之间在Bank上的冲突。可以看出，M^3的核心思想是希望尽可能的将连续的访存行为（页面级）分散在不同的Bank上。

我们的机制仅需轻微修改即可支持这种页面级的随机化分配。在Sub-system B中，只要在order为1的颜色列表中随机选择物理页面分配给发生页缺失的线程即可。或者，近似的来看，对于任意线程，可以直接使用Round-Robin策略在order为1的颜色列表中分配页面，避免出现访存过于集中在一个或少数几个Bank中的情况。这种做法与划分优化的方法是两种“哲学”，各有优势。如前文所提及的，由于共享数据的原因，划分的效果会受到影响。如果存在很大量的共享数据，如PARSEC中的StreamCluster，那么划分的优势几乎就消失了（可见第三章中关于多线程工作集的讨论）。在这种有很多共享数据的情况下，随机的分配方案将有利于系统的性能。

从理论上来讲，本文提出的方法与M^3是能够达到同等效果的；从实现上来讲，M^3的实现过于复杂，并且系统中不再保留由连续的物理页面组成块，因此，内核态的应用（如Driver，DMA）,需要大量的连续页面块，则不能在此环境下正常工作，而本文设计的机制则不会出现这种问题。

综上所述，我们可以看出，在每个子系统中，页面都会遵循一定的排布规律，这个规律可被利用以建立索引高效的索引机制。核心思想是，作为目标的属于某种颜色的页面一定会出现在某种颜色之后，并且颜色渐变的跨度是一定的。如前文所述，由于不同子系统的参数是不同的，因此，hash算法也不同。本文给出了在这两种情况下hash算法的伪代码。在我们实验中，该算法被实现在Linux Buddy System中。CASE2中的算法与3.3.2中的算法相同，3.3.2中是为了支持Bank-Only的划分机制，根据输入参数的不同，同样也能O(1)的时间开销内支持其它的分配方式。

seudocode 2: Hashing algorithm for selecting pages

Input: (1) order; (2) target_color Output: one page of target color

```
BEGIN
/*CASE 1: Physical pages organized based on bits 14~16*/
IF using 14,15,16 bits THEN
    SWITCH (order)
```

case	0~2	3	4	5~10
colors_per_block =	1	2	4	8

END SWITCH

block_color = (target_color / colors_per_block) × colors_per_block;

page_index = (target_color - block_color) × 4;

END IF

/*CASE 2: Physical pages organized based on bits 13~ 15, 21~22*/

IF using 13, 14, 15, 21, 22 THEN

SWITCH (order)

case	0~1	2	3	4~9	10
colors_per_block =	1	2	4	8	16

END SWITCH

block_color = (target_color / colors_per_block) × colors_per_block;

IF order is 10 AND the color bits are x1xxx THEN //The 4th bit is 1

page_index = (target_color - block_color - 8) × 2 + (1 << 9);

// As shown in Figure 5.6: 32 blocks × 8 colors = 1024 blocks

ELSE page_index = (target_color - block_color) × 2;

END IF

END IF

Expand color block (page_index, order)

// physical pages represented by "struct page" are in page[] array in Linux kernel.

RETURN page[page_index] and remove this page from free list.

END

* target_color is the color of the requested page.

* block_color is the color of the first page in a block.

* colors_per_block is the number of colors in a block.

5.5 两套索引机制的协同及实现

本文的x-Buddy能够同时支持A-/B-/C-VP、Bank-Only以及随机化的物理页面分配。在实现中，系统中每个物理页面都会同时被映射在两个子系统中（Sub-system A和B），因此，如果有页面被分配给了某个线程，传统的Linux内核需要在Buddy System中移除（remove）这个页面，而对于本文的x-Buddy来说，则需要在两个子系统中同时移除该页面，并相应的维护两个子系统中颜色的排布。在内核态，将一个较大的块分割成为若干个子块的过程被称作“扩展”（expanding），对于x-Buddy来说，扩展过程需要考虑由不同颜色带来的影响。如图5.7所示，一个由8个连续页面组成的块，在两个子系统中的颜色是完全不同的，该图向我们动态的展示了在B中每次移除页面时与A系统协同工作的典型过程。

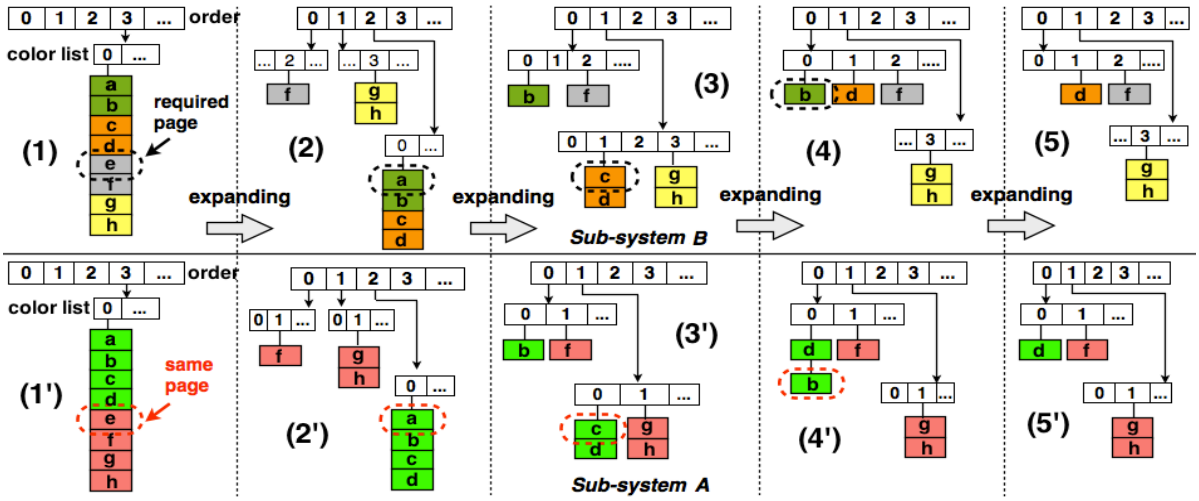


图5.7 Sub-system A与B的同步过程实例（order为3的情况下每块有2^3个页面）

现在假设有一个页面需求需要返回B系统中的一个灰色的页面e（此页面在A系统中是洋红色），在B中移除这个页面之后将会生成3个块（f，gh和abcd），分别将这三个小块挂载到对应的颜色槽中，同时在A系统中也要同时进行类似操作。在运行过程中，由于颜色的差异，两个子系统的颜色列表的状态始终是不同的，比如，从状态3～4转换的过程中发生的页面d挂载方式的差异就是由这个原因造成的。

看似相对复杂的逻辑在具体实现过程中并不复杂而且在运行中也不造成过高的开销。在内核的内存管理模块，每个物理页面都由一个page_struct表示并通过list_head结构链接进Buddy System。对于，x-Buddy系统，本工作给page_struct添加了两个list_head类型的指针lruA和lruB，分别将该页面链接在两个子系统中。理论上，分配一个物理页面的开销是O(1)，实际运行中，x-Buddy的开销略高于原始内核的Buddy System（但仍处于1/100秒～几秒的范畴内）。优化具体的实现方式可以进一步降低这个开销，例如，我们在具体实现过程中采用的bitmap来追踪需要被迁移的页面，也可通过三次散列的方式代替x-Buddy的两级子系统（这种方式在页面迁移的时候引入更复杂的迁移逻辑），这些问题都是可以公开讨论的。

本文所提出的这种方式有两个优势：第一，使用比较直观。我们现在的着色过程是通过向内核中写入颜色信息的方式进行的，颜色与划分之间的关系是直接映射的，区别仅在于不同的子系统采用不同的颜色而已。第二，基于简单的颜色规则，页面迁移的过程相对简单，因此开销也会相应减小。但如果采用了三次散列的方式，系统现有优势就没有了，而且页面迁移的过程将相对复杂。就本文现在的研究来看，页面迁移是系统开销的主要来源（通常会由于动态过程的“重着色”引发），因此，本文不想给这个过程再增加负担[9]。

5.6 基于AAA-Memory Management概念模型的HVR框架

至此，本文基于AAA-Memory Management的概念设计并完成了一套采用多种资源管理与分配的方式处理访存行为多样性的机制，或者，我们认为是一套框架，本文将其命名

为**HVR**，是该框架或机制能够支持的水平（**H**orizontal）、垂直（**V**ertical）和随机（**R**andom）三种资源管理与分配机制的简写。HVR工作于内核态，从收集程序行为并分类开始到选择恰当的资源分配方式并动态调整资源利用率，HVR包含了AAA-Memory Management概念模型中所需求的全部要素，是一个应用特征与体系结构敏感的内存资源管理模型的实例。

HVR通常要处理很多在真实运行环境中的复杂情况，从根源上讲，即需要管理线程的“颜色”。在本文中，我们总是用线程代替进程，因为从系统与结构的角度来看，这二者没有区别（体系结构研究中一贯如此）。例如，在操作系统“眼里”，无论是线程还是进程都是由task_struct这个数据结构来表示的，在任务调度的时候，task_struct所提供的信息是调度器的重要依据，在没有外部特殊指标的情况下是唯一依据。本研究的“着色”（或者分配颜色的相关信息）就是写在这个数据结构的color_masks中的。通常程序在启动的时刻可以被人为的指定“颜色”，并且这个信息将在进程被创建的时候写入该变量。动态的调整也与这个变量有直接的关系。

总的来讲，动态资源的调整通常会发生在两个不同的阶段，一是资源的融合过程，即从线程间资源的隔离状态转化到融合状态，线程间可以共享一部分资源（参见第四章的“融合原则”）；二是划分状态的形成过程，通常在工作集开始运行的初始阶段，在采集完程序的访存行为并分类之后需要根据工作集的类型分配并划分内存资源的过程。前者通过共享相同的颜色即可做到，即让两个或多个线程的task_struct中的color_masks的颜色置为相同，后者则和这个过程相反。需要注意的是，在实际情况下，由于Bank-Only划分通常不会对系统性能产生“负”影响，因此我们通常将其作为默认的初始机制，并根据具体情况做后继调整。例如，如果Bank-Only需要转化为A-VP，那么就需要调整每个线程所属的颜色，并将调整之后不再属于自身颜色的页面的内容迁移至新颜色对应的页面中。

5.7 HVR框架的实验效果与讨论

以前章节介绍的均是静态划分的性能表现。在本节，我们将介绍HVR在实验环境中带来的系统性能提升，以及在真实计算环境中根据多样化的计算需求所采用的多种策略并结合“划分”与“融合”规则指导下的动态优化效果。

5.7.1 动态策略的选择

如前文所述，HVR通过收集并分析工作集的特征并据此在决策树上检索便能够选择出最优的内存分配策略。为了较为完整的反应出HVR的功效及优势，我们的实验中将HVR与静态VP（Static-VP，SVP），以前学者提出的基于资源利用率的动态划分方式（Utility-based VP, UVP）进行比较。SVP方法对四道程序的工作集采用A-VP方法，对八道程序的工作集采用B-VP和C-VP方法。UVP会根据性能计数器监测到的缓存缺失（cache miss）来动态调整cache划分。这三种策略在50个随机生成的工作集上的性能如图5.8所示，本图按照工作

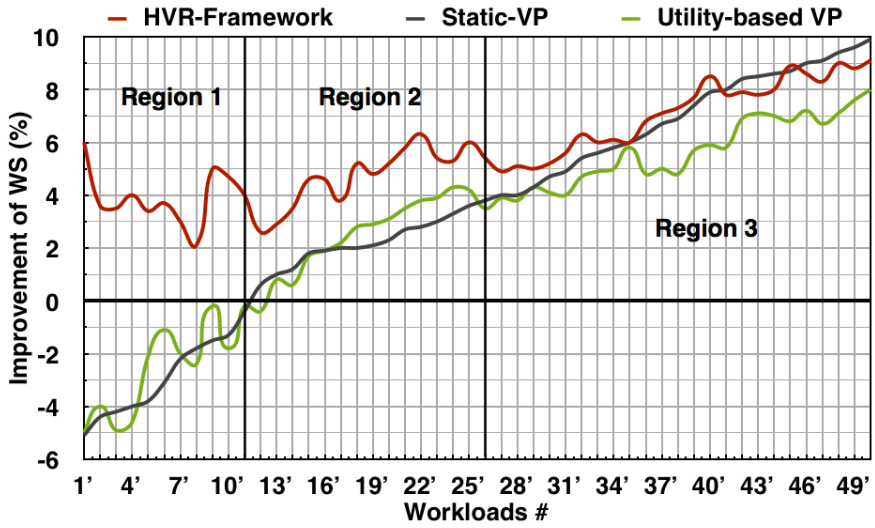


图 5.8 HVR 的运行时功效。注：这些工作集是随机产生的，与前文中所采用的均不同。

集 SVP 的性能变化的升序排序。我们将这段曲线分为三个区域。作为参考标准的基准系统是系统是未经任何修改的原始的 Linux 内核。

在区域1内，SVP和UVP的性能相对基准系统是下降的，而HVR比基准提高性能6.1%，比SVP和UVP提高性能11%。进一步分析发现，区域1内的工作集主要是｛m*LLCH, n*LLCM, k*CCF｝类型的工作集。这类工作集主要包含了LLCH或者LLCM但不含有LLCT的工作集。如前文所分析的原理，由于LLCH对Cache资源的需求很高，不适合做Cache划分。因此，区域1内的工作集只有采用Bank-Only划分是最合适的。HVR在分类程序的时候即发现了这个特点，并采用了正确的策略，因此性能会有比较好的表现。

通过动态调整资源利用率的方式能不能避免Cache划分在区域1中的“负”影响？从UVP的表现我们可以看出对于某些工作集来说可能会有好处（在区域1内出现了上下起伏的现象，有些点会明显超过SVP），但总的来说性能还处在0以下。我们不排除动态的，根据资源利用率的调整有可能会有利于这些包含LLCH/LLCM程序为主的工作集，但本文认为在Cache上的划分归根结底还是会有损这类工作集的性能，并不是最佳的方案。

在区域2内，大多是包含LLCT程序的八道程序的工作集。HVR由于资源融合而优于SVP和UVP策略。比如，对于工作集22’，其包括5个LLCT应用、2个LLCH应用和1个LLCM应用。HVR将所有的LLCT应用映射到1/8的Cache上，这样剩下的7/8的Cache由LLCM和LLCH应用所共享，因此提升了系统性能。其它两种机制对工作集中的程序是完全隔离开的。从图5.8中我们可以看到SVP的性能表现要优于UVP，这是因为后者的动态开销比较大，在工作集运行的整个生命周期不断的在调整资源的配比，优化的效果被动态的开销给抵消了。从我们大量的实验的经验来看，如果我们知道工作集的特点，静态的方案在很多时候是可以被接受的，而往往过于细粒度的任务级调度则缺乏实践意义。HVR实际上在达到稳定状态之后就不在动态调整了，从实践角度来看，我们认为恰好是二者的折衷状态。

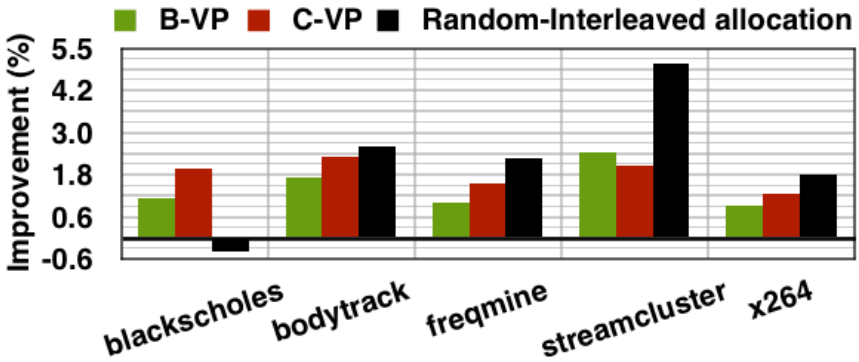


图5.9 不同分配机制对多线程工作集的性能对比。注：八线程下的测试情况

在区域3内，大多数情况下，HVR由于能选择合适的VP策略和采用合并的规则也能比另两种方法性能更优（但优势在这个区域不太明显）。但是对于一些工作集，SVP（静态VP）性能略微优势（平均比HVR高0.4%）。我们进一步分析该区域内的工作集后发现，它们基本上是包含了若干LLCM应用的适于A-VP/B-VP类型。一个LLCM应用需要［1/8，1/2］的cache容量，并通常保持稳定的Cache利用率，SVP策略没有动态开销，因为它在运行一开始就通过离线采样（profiling）决定了划分策略。另一方面，UVP方法引入了动态开销，可能抵消划分带来的性能收益，这些动态开销来自于高开销的页迁移（反复的页重新着色引发的）和性能计数器的开销，因此它的效果是最不好的。

对于多线程的工作集，采用随机交替的页分配方法是可能要优于划分的方法。如图5.9所示，我们的实验结果表明，普遍来讲，对于几个有代表性的八线程工作集（PARSEC2.1），随机交替的页分配策略都优于B-VP/C-VP策略。HVR支持随机交错的页分配策略（Sub-system B即可支持这种方式），决策树上也包括了这种机制。

5.7.2 针对实时变化的工作集的功效

本实验利用Intel处理器的IPC性能计数器采集了在Cache-Only划分策略、Bank-Only划分策略和HVR下的实时性能数据，如图5.10所示。在整个测试过程中，我们在不同时刻注入不同类型的应用。与此同时，先启动的应用一旦完成会被终止。这与真实的应用场景相符合。从图5.10看到，Cache-Only划分的性能在[-3%，6%]范围内波动。在采样的时间点5，6，9，11，26，27和28，Cache划分的性能提升低于0。这是因为在这些点启动了LLCH应用，有限的Cache资源导致了性能的下降。

HVR避免了此时的下降，因为它能识别这些LLCH应用并对它们启用了资源融合。在采样时间点16，HVR获得IPC性能提升的最大值。这是因为在这个时间点上系统同时运行了LLCT、LLCH、LLCM和CCF，HVR能够有效的隔离开产生最强干扰的LLCT和最容易收到干扰的CCF，并让LLCH和LLCM融合在一起，使用较大的资源空间，并且整体性能又得益于VP在多层次上的的累加。如第四章的图4.11所示的情况。

与HVR相比，Bank-Only划分和Cache-Only划分方法获得较少的性能提升（各自比HVR

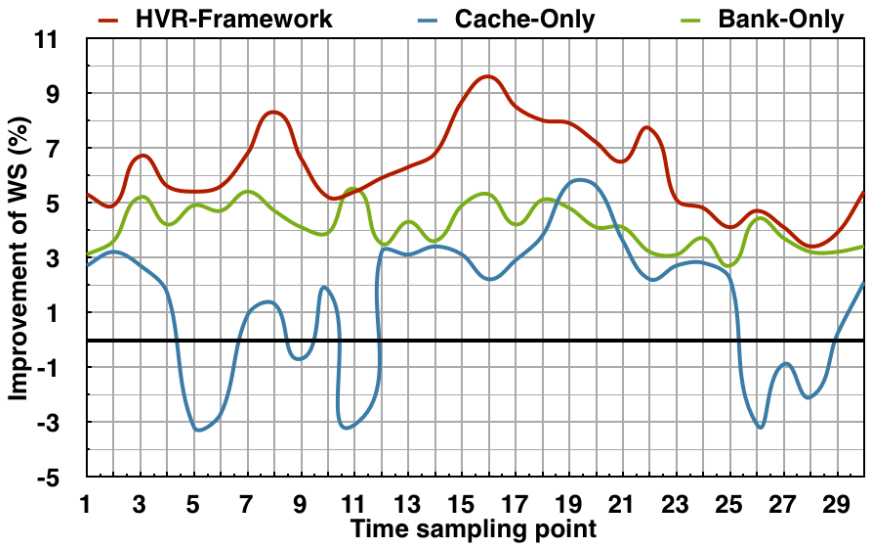


图5.10 HVR的实时效果

差5%和7%），由于它们不能动态的选择最优的资源分配机制，也不能利用资源“融合”的方法来提高性能。但是，总体来看，Bank-Only划分的表现是相对稳定的，与我们在第三、四章的结论是一致的。

5.8 HVR的开销分析及讨论

HVR的开销主要来自于以下几个方面：

（1）基于页表采用的JOB1和JOB2的开销。扫描整个页表的时间开销与程序的访存足迹有关，或者说与也表项的数目有关。在我们的实验中，这个时间开销从5μs (povray) 到4.46ms (mcf)不等。因此，从工作集的整个运行时间来看，这个时间开销不算大，不至于影响系统的性能。另外需要特别指出的是，在具体的实现过程中，JOB2的采样周期是3s，但随着系统运行的稳定，这个时间会逐步的呈梯度增大，因此开销会更低。从我们实验的结果来看，JOB2最大的开销只有0.6%。还有，针对那些访存足迹很大的程序，我们可以采用随机采样的方式代替遍历整个页表，初步的实验证明，随机抽查5%～10%的页面即可反映出大内存需求程序的访存行为，这个工作将是我们今后进一步研究的问题。

（2）来自于x-Buddy的开销。前文已经反复提及了这部分的开销问题。理论上，在x-Buddy中返回一个页面的时间开销是O(1)，我们的实验中对system time开销的观察发现在较大的情况下也仅有几秒的时间开销，这个开销对于长时间运行的工作集来不算大，不至于抵消优化效果（我们的工作集通常会运行几十分钟）。

（3）页面迁移带来的开销。实验证明，在这三部分的开销中，页面迁移的开销是最大的。在我们的实验平台上，通过memcpy函数迁移一个物理页面的内容需要3μs（此数据为实测，页面迁移不需要通过I/O交换区来完成）。我们研究中最为极端的一个例子需要迁移400MB左右的数据，那么需要30s左右的时间。而我们的整个工作集的运行时间超过30分钟，这个时间开销是大概1.7%。

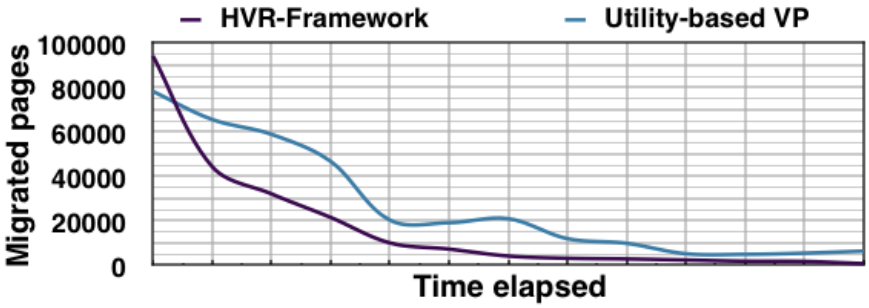


图5.11 HVR在页面迁移上的优势

我们的工作中采用的依赖稳定的分类的办法即是为了减少页面迁移的数量。如图5.11中所示的一个页面迁移的过程，HVR需要迁移的页面的数量在逐步的减少（随着分类的趋于稳定），但是UVP的方法会不断的通过页面迁移的方式来调整资源的利用率，因此我们可以看到它的迁移曲线在波动，而这个过程可能会持续在整个运行周期，因此有可能引发更大的开销。基于分类的方式一旦到达稳定即不需要再迁移了。另一个在我们研究中减少页面迁移开销的方法是lazy migration（懒惰迁移），指的是页面迁移仅仅在必要的时候发生，这样也可以减少不必要的页面迁移，实验证明这种方式在有些情况下非常有效。

5.9 本章小节

本章我们提出了AAA-Memory Management的概念模型，并基于此实现了HVR机制。该机制综合了当今最为先进（State-of-the-Art）的内存资源的管理思想，并在线的（On-the-Fly）根据工作集访存行为的多样性有针对性的通过资源的管理与分配的方式优化访存效率。实验证明，在真实环境下，我们的系统收到了良好的效果。本研究认为这种“垂直”的，工作集与计算机体系结构敏感的资源管理与优化方式是未来资源管理的发展方向。

第六章 未来工作的展望

未来工作的讨论对本文来说是至关重要的一章。在我们开展本研究的过程中，不断的有国内外的领域内相关的专家对我们的工作表示了关心和关注。他们的关注主要概括为两点：第一，未来的计算环境的变化对“页着色”技术的发展带来哪些影响？第二，“页着色”技术有哪些真实的有价值的应用场景？我们觉得这两个问题是横纵交叉的，涵盖领域之广泛，包括内容之深刻，很难让人一言以蔽之。为此，本文进行了广泛的调研，与很多活跃在工业界的学者和工程师进行交谈，并在结合自身研究和体会的基础上，将本研究的未来工作确定在以下三个主要的方向上。

方向（一）：异构的访存密集型存储架构。关于异构或新结构的存储相关的研究近些年非常火热［107,108,114］。由于 DRAM 的先天劣势（请参见第二章），业界已经在考虑由“非易失”性的新型存储介质（NVM）来代替 DRAM。例如，采用 PCM（Phase Change Memory）［21］的新型计算机已经逐渐走出实验室。但是 NVM 也同样有其劣势，因此业界考虑将两种材质的异构存储体系结合在一起使用，如图 6.1 所示，在这个架构中的主存体系同时包括了 DRAM 和 PCM，形成优势互补［17］。PCM 的优势在于“非易失”材质带来的数据存储的稳定性和可靠性（请参考第二章 DRAM 在维护数据稳定上所必须的各种额外开销），而且“读”操作的速率在理论上和 DRAM 相近，实际数据往往优于 DRAM。

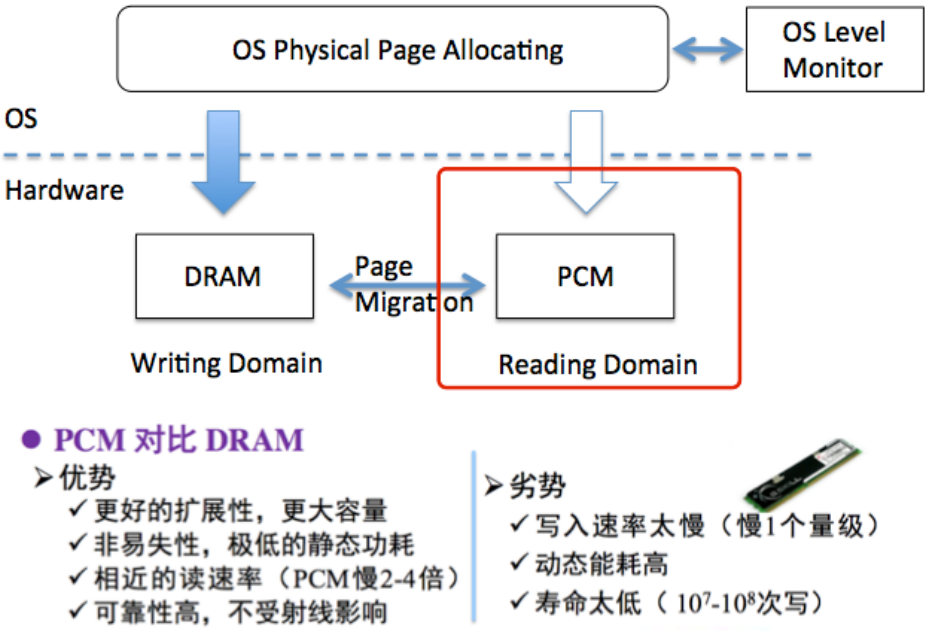


图 6.1 异构材质的访存密集型架构（设想）。注：图中的“对比”参考清华舒继武老师提供的学术交流材料。

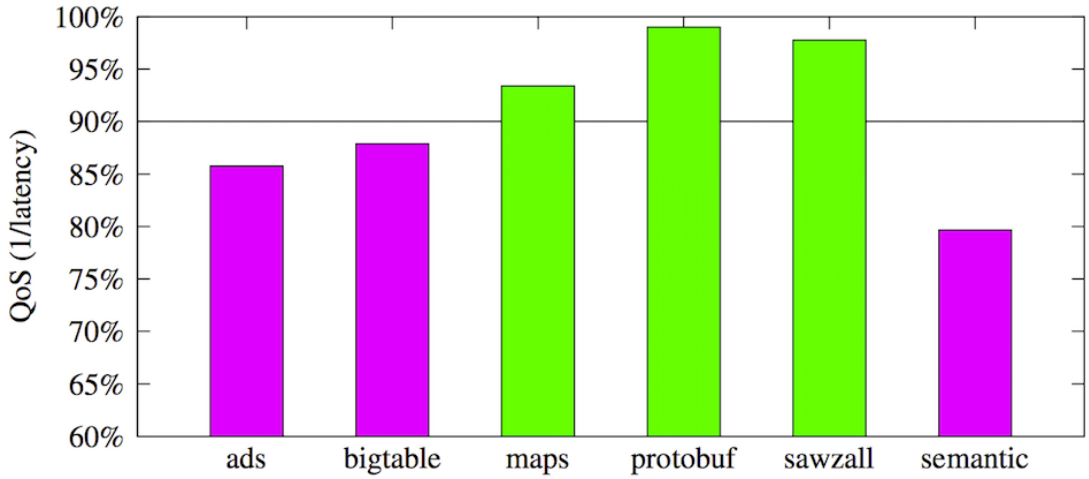


图 6.2 谷歌数据中心中的关键应用的服务质量［104，105］。服务质量（QoS）在 90%的情况是可以被接受的。

在 6.1 所描述的计算机体系中，通过 OS 或者其它的监控方式将工作集所使用的物理页面区分为频繁写“写”和频发“读”的两类，并通过“页着色”的方法将这两种页面分布在不同的存储介质上。因此，PCM 可以发挥其存储稳定，更适应“读”操作的特点，而避免了在“写”操作上寿命有限且性能不佳的劣势。

计算单元异构和存储体系的异构是未来高性能计算机体系结构的发展方向。图 6.1 中展示的是我们关于这方面研究的一个设想，不足为训，但异构计算环境中合理高效的内存资源的分配与管理依然是体系结构研究者所要面临的严峻挑战。挑战往往与机遇并存，挑战越大则机遇越大。我们认为这种机遇恰恰是“页着色”技术在未来计算机科学与技术发展中能够有所发展的地方。

方向（二）：“云计算”环境中，面向数据中心的优化。我们已经进入了“Datacenter as a Computer”的“大数据”时代，那么访存干扰的问题也自然延伸到了数据中心，并且被由成千上万台协同工作的计算机组成的计算系统放大了。在谷歌的数据中心里即存在这由于对共享内存资源竞争所导致的服务质量下降的情况。来自谷歌内部的数据显示，如图 6.2 所示，在他们的关键应用中，ads、bigtable、semantic 等关键的应用的服务质量经常不能达到在 90%以上的业务需求。另一种情况是，数据中心所提供的服务通常要经历 N 台计算机串行和并行的处理之后才能返回。我们可以假设，一台计算机上由于访存延迟而产生的性能损失是 m，那么被放大 N 倍之后，N*m 的性能损失就很有可能就会影响到用户的产品体验。

根据有关资料介绍（涉及到商业问题不便引用此文献），为了保证关键程序的服务质量，很多服务商只能采用“多核变单核”的简单的处理办法，即让高性能多核服务器仅运行一道程序，如果有 N 个需要保证服务质量的应用，那么就需要 N 台这样的服务器，如图 6.3 所示。相关资料显示，在这样的情况下资源有效利用率实际还不到 10%，对于数据中心这样的大规模计算环境（动辄数以十万计的计算机规模），造成的资源浪费与引发的各种开销难以估计，

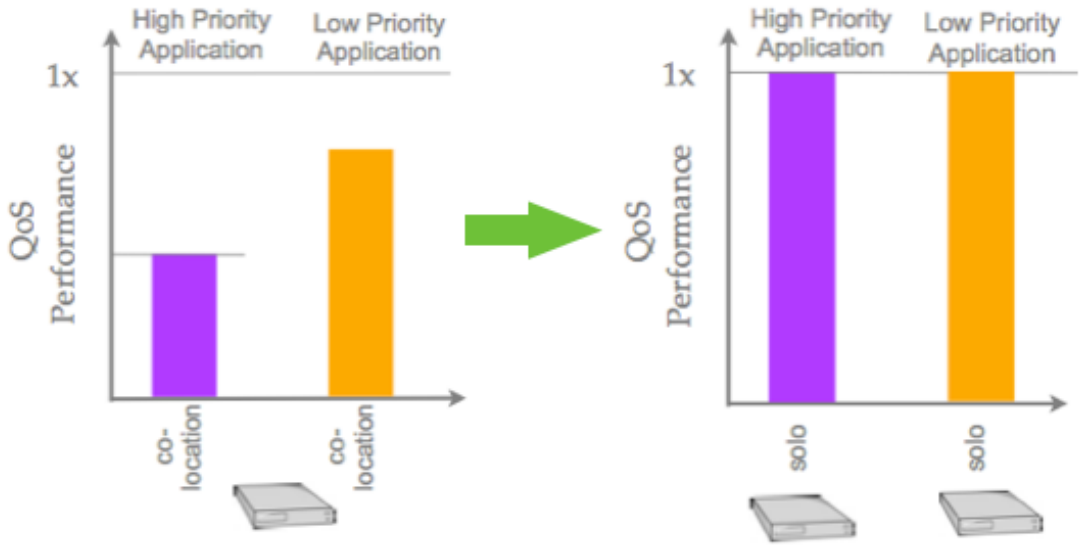


图 6.3 “多核变单核”的“优化”过程［104，105］。注：在共享资源的情况下，高优先级的程序有可能会遭受道更大的性能损失，服务质量难于保证。但如果单纯的以“横向”的扩展机器的方式来解决，则又会引发更多的问题。

无论与“低碳”环保的全球共赢的社会理念，还是与利益至上的商业目标，都是严重冲突的。

本文所进行的研究工作可以为数据中心相关的优化提供有价值的参考。我们最初的目的即在于在多核体系结构中，通过消除访存干扰的方式来优化共享内存资源的访问，并且，内存资源的“垂直”管理的概念模型也能为数据中心的性能优化提供理论上和实践上的参考，具有指导意义。如果我们提供的机制再与优先级的相关的调度策略相结合，那么很有可能会更适用于数据中心的优化［22］。

我们这里引用一段关于数据中心性能与商业价值之间的关系的评论，“At the datacenter scale, a performance improvement of 1% for key applications, such as web-search, can result in millions of dollars saved.”［52,104,105］。对数据中心的优化是目前业内广泛关注的问题，方法多种多样，但都逃不开对共享内存资源访存干扰相关的问题。因此，我们的研究具有较强的应用和产业价值，我们希望在未来的工作中，如果有机会可以和企业界合作，就共同关注的问题进行更为深入的研讨［22］。

方向（三）：与虚拟机相关性能优化与安全。在多核服务器上运行多个虚拟机并分别向外部提供服务是多核服务器的一种流行的使用方式。在实质上，这种方式和本文所研究的应用背景相似，每个运行中的虚拟机即可被看作是一道程序（线程）。如果我们方式对多道程序的工作集有这样那样的好处，那么就有可能也会给虚拟机环境带来好处。如图 6.4 所示，从本文提出的资源垂直管理的角度来讲，每个虚拟机资源如果能够通过着色的方式仅使用属于它自己的一部分存储资源，那么无论是系统整体的服务质量还是吞吐量都会得到优化。

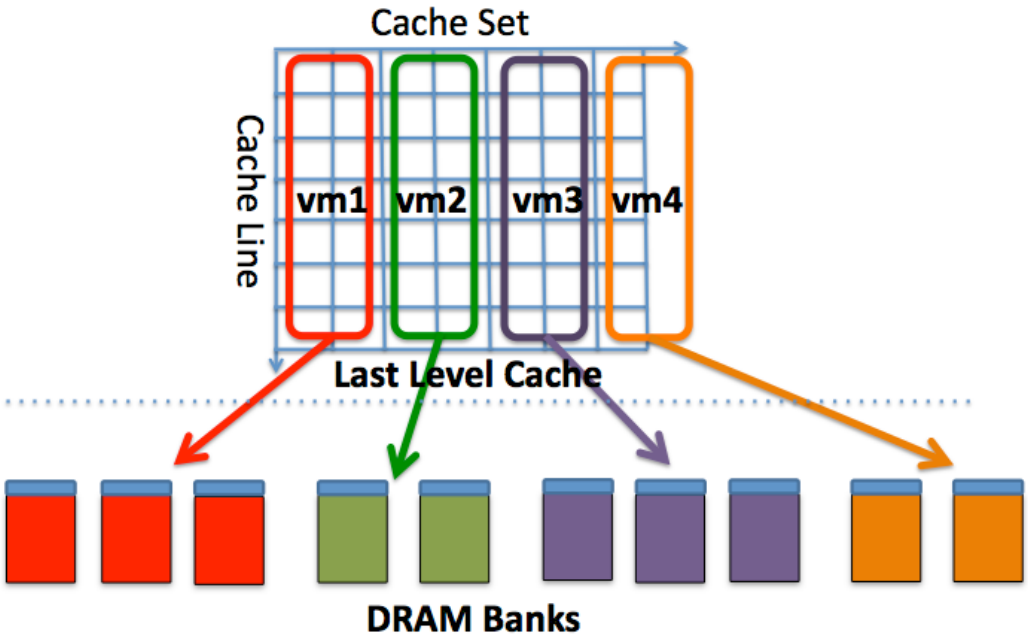


图 6.4 虚拟机与内存资源的垂直管理

除性能之外，虚拟机环境还存在安全性问题。虚拟机的用户虽然有可能都运行在同一台机器上，但他们的信息还必须保证是私有化的。目前很多恶意的攻击就是利用系统在共享内存体系上的漏洞进行的。比如，在共享 Cache 上，由于地址映射的原因，一个用户的信息就有可能被另一个用户读取；在 DRAM 层次，如果某个恶意用户开启了“吃”内存的程序，那么内存资源将会迅速被恶意的消耗，并且带宽也会被恶意占用，有可能会造成 Deny of Service（DoS）的情况。针对此问题，学术界的相关研究可见于文稿，基本被概括为通过辅助硬件或者系统软件控制的方式进行安全维护［2，54］。

本文的工作也可被用在解决虚拟机安全的问题上。通过水平划分的方式即可彻底消除虚拟机之间的安全隐患，通过垂直划分可以消除 DoS 的情况。本文非安全性问题研究的论文，很多问题这里也不再展开。进行这方面的研究对本工作来说也是个挑战，我们希望在条件允许的情况下能够与相关研究者合作。

我们目前基于开放源代码的虚拟机 XEN 进行了分析。XEN 的关于资源分配的核心代码部分与 Linux 类似（这部分代码在 Hypervisor 的内存管理部分），物理页面都是基于“伙伴”的组织方式。在虚拟机 Dom0 启动的时候，即可为其它虚拟机资源 DomU 分配资源。从理论上来讲，我们恰好可以在这个时候对页面“着色”，给每个 DomU 分配的资源在此时即被划分开来。我们关于这部分的工作还在继续过程中。XEN 的整体结构如图 6.5 所示，本研究认为，XEN 和本工作能够很好的结合在一起。KVM 是另一个流行的高效的虚拟机，后继的研究也可能会考虑在多种虚拟机机制下进行研究和开发。［2，19，86］

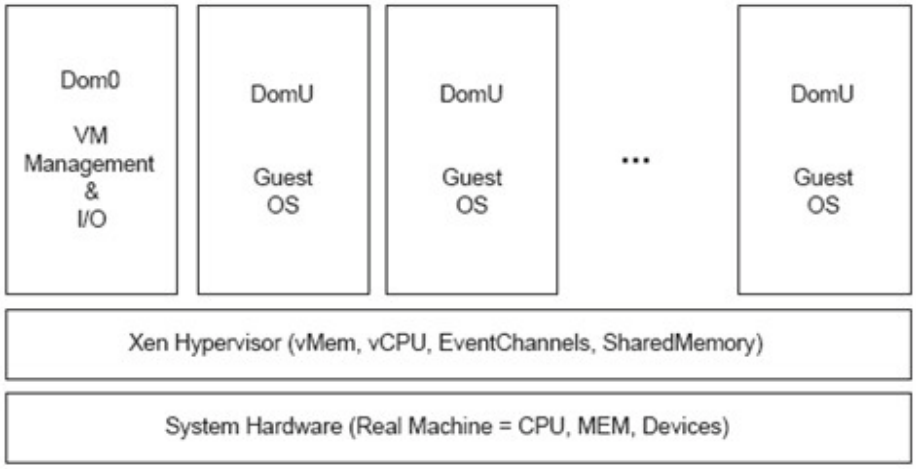


图 6.5 XEN 的结构

综上所述，本研究可能会在异构内存的新体系结构、数据中心的优化以及虚拟机相关的工作中有应用前景。另外，从资源管理的角度来讲，本文认为我们提出的“Going Vertical”的内存储器资源“垂直”管理的概念与模型是未来多核、众核体系结构中存储器资源管理的发展方向。本文希望能够给从事体系结构、操作系统、内存结构与性能优化等相关研究工作的研究者提供可靠的参考。

参考文献

[1] N. Aggarwal et al. Power Efficient DRAM Speculation. In HPCA-14, 2008.

[2] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. In ACM SIGOPS Operating Systems Review 43(2): 56-65, 2009.

[3] J. H. Ahn et al. Multicore DIMM: An energy efficient memory module with independently controlled DRAMs. IEEE CAL, Jan. 2009.

[4] J. H. Ahn et al. Improving system energy efficiency with memory rank subsetting. ACM TACO, Mar. 2012.

[5] M. Awasthi, D. W. Nellans, R. Balasubramonian, and A. Davis. Prediction based dram row-buffer management in the many-core era. In PACT-2011.

[6] Y. Bao et al. HMTT: A Platform Independent Full-System Memory Trace Monitoring System. In SIGMETRICS-08, 2008

[7] S. Beamer et al. Re-Architecting DRAM Memory Systems with Monolithically Integrated Silicon Photonics. In ISCA-37, 2010.

[8] Basu, Mark D. Hill and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In ISCA-2012.

[9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In OSDI-2010.

[10] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton Univ, Jan. 2008.

[11] M. N. Bojnordi and E. Ipek. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. In ISCA-2012.

[12] S. Cho, and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level page Allocation. In MICRO-39, 2006.

[13] Rohit Chandra, Scott Devine and Ben Verghese. Scheduling and Page Migration for Multiprocessor Compute Servers. In ASPLOS-1994.

[14] Z. Cui et al. A Fine-grained Component-level power measurement method. In PMP-11.

[15] H. Cook, M. Moreto, S. Bird, K. Dao, D.A. Patterson, and K. Asanovic, A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. in ISCA. 2013

[16] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. in HPCA. 2005.

[17] N. Chatterjee et al. Staged reads: Mitigating the impact of DRAM writes on DRAM reads. In HPCA, 2012.

[18] Y. Chou et al. Microarchitecture optimizations for exploiting memory-level parallelism. In ISCA, 2004.

[19] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: a System for Energy Efficient Computing in Virtualized Environments. In Proceedings of International Symposium on Low Power Electronics and Design. In ISLPED-2009.

[20] J. Demme et al. Rapid Identification of Architectural Bottlenecks via Precise Event Counting. In ISCA-2011.

[21] Yu Du, Miao Zhou, Bruce R. Childers, Daniei Mosse, Rami Melhem. Bit mapping for balanced PCM cell programming. In ISCA-2013.

[22] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In ASPLOS-2014.

[23] R. Das, R. Ausavarungnirun, O. Mutlu et al, Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems. In HPCA-2013.

[24] P. J. Denning, The Working Set Model for Program Behaviour. Commun. ACM, 1968. 11(5).

[25] X. Ding, K. Wang, and X. Zhang. SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores. in EuroSys. 2011.

[26] X. Ding, K. Wang, and X. Zhang. ULCC: a user-level facility for optimizing shared cache performance on multicores in PPoPP. 2011.

[27] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero and Alexander V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. In MICRO-2012.

[28] E. Ebrahimi et al. Parallel application memory scheduling. In MICRO, 2011.

[29] E. Ebrahimi et al. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In ASPLOS-2010.

[30] E. Ebrahimi et al. Prefetch-Aware Shared-Resource Management for Multi-Core Systems. In ISCA-2011.

[31] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In HPCA-8, 2002.

[32] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache memory. In Journal of Supercomputing, 28(1), 2004.

[33] H. Fredriksson and C. Svensson. Improvement potential and equalization example for multidrop DRAM memory buses. IEEE Transactions on Advanced Packaging, 2009.

[34] Saugata Ghose, Hyodong Lee and José F. Martínez. Improving Memory Scheduling via Processor-Side Load Criticality Information. In ISCA-2013.

[35] Mark Gebhart, et al. A Compile-Time Managed Multi-Level Register File Hierarchy. In MICRO-2011.

[36] I. Hur and C. Lin. Memory Scheduling for Modern Microprocessors. ACM Transactions on Computer Systems, 25(4), December 2007.

[37] H. Hidaka et al. The cache DRAM architecture: A DRAM with an on-chip cache memory. IEEE Micro, Mar. 1990.

[38] John L. Hennessy and David A. Patterson. Computer Architecture-A Quantitative Approach. Fifth Edition.

[39] K. Itoh. VLSI Memory Chip Design. Springer, 2001

[40] R. Iyer et al. QoS policy and Architecture for Cache/Memory in CMP platforms. In SIGMETRICS-07, 2007.

[41] Hiroshi Inoue, Hideaki Komatsu and Toshio Nakatani. A Study of Memory Management for Web-based Applications on Multicore Processors. In PLDI-2009.

[42] E. Ipek, O. Mutlu, J. F. Mart´ınez, and R. Caruana. Self- optimizing memory controllers: A reinforcement learning approach. In ISCA-2008.

[43] A. Jaleel, H.H. Najaf-Abadi, S. Subramaniam, S.C. Steely, and J. Emer, CRUISE: cache replacement and utility-aware scheduling, in ASPLOS.2012.

[44] B. J. Lee et al. Improving Memory Bank-level Parallelism in the Presence of Prefetching. In MICRO-42, 2009.

[45] C. Kroft. Lockup-free instruction fetch/prefetch cache organization. In ISCA, 1981.

[46] R. Kho et al. 75nm 7Gb/s/pin 1Gb GDDR5 graphics memory device with bandwidth-improvement techniques. In ISSCC, 2009.

[47] Y. Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu and Onur Mutlu. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In ISCA-2012.

[48] Y. Kim, M. Papamicheal and O. Mutlu. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In MICRO-43, 2010.

[49] Y. Kim et al. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple MCs. In HPCA-16, 2010.

[50] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. in PACT. 2004.

[51] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the many-core Era. In MICRO-44, 2011.

[52] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In ASPLOS-2014.

[53] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. In Micro-41, 2008.

[54] D. Kaseridis, J. Stuecheli, J. Chen, and L.K. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. in HPCA. 2010.

[55] K. C. Knowlton, A Fast storage allocator. Communications of the ACM, 1996.

[56] G. L. Yuan et al. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In MICRO-42, 2009.

[57] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In HPCA-14, 2008.

[58] Jiang Lin, Hongzhong Zheng, Zhichun Zhu, HowardDavid, and Zhao Zhang. Thermal Modeling and Management of DRAM Memory Systems. In ISCA-2007.

[59] J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In RTAS-3, 1997.

[60] L. Liu et al. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In PACT-21, 2012.

[61] L. Liu et al. BPM/BPM+: Software-based Dynamic Memory Partitioning Mechanisms for Mitigating DRAM Bank-/Channel-level Interferences in Multicore Systems. In TACO-2014. Jan.

[62] L. Liu et al. Going Vertical in Memory Management: Handling Multiplicity by Multi-policy. In ISCA-2014.

[63] B. C. Lee et al. Architecting phase change memory as a scalable DRAM alternative. In ISCA-2009.

[64] J. Liu, B.Jaiyen, R.Veras and O.Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In ISCA-2012.

[65] Yong Li, Rami Melhem, and Alex K. Jones. Practically Private: Enabling High Performance CMPs Through Compiler-assisted Data Classification. In PACT-2012.

[66] B. C. Lee et al. Architecting phase change memory as a scalable DRAM alternative. In ISCA, 2009.

[67] J. Lee and H. Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. in HPCA. 2012.

[68] O. Mutlu and T. Moscibroda. Parallelism-aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In ISCA-35, 2008.

[69] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of Memory Service in Multi-core Systems. In USENIX Security, 2007.

[70] O. Mutlu and T. Moscibroda. Stall-time Fair Memory Access Scheduling for Chip Multiprocessors. In MICRO-40, 2007.

[71] Krishna T. Malladi, et al. Towards Energy-Proportional Datacenter Memory with Mobile DRAM. In ISCA-2012.

[72] Micron. DDR3 SDRAM System-Power Calculator, 2010.

[73] Micron. 2Gb: x16, x32 Mobile LPDDR2 SDRAM, 2012.

[74] Micron. 2Gb: x4, x8, x16, DDR3 SDRAM, 2012.

[75] Micron. DDR3 SDRAM Verilog Model, 2012.

[76] M. J. Miller. Bandwidth engine serial memory chip breaks 2 billion accesses/sec. In HotChips, 2011.

[77] Y. Moon et al. 1.2V 1.6Gb/s 56nm 6F2 4Gb DDR3 SDRAM with hybrid-I/O sense amplifier and segmented sub-array architecture. In ISSCC, 2009.

[78] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli and José F. Martínez. Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems. In ISCA-2013.

[79] W. Mi et al. Software-Hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors. In Proc. the 2010 IFIP Int'l Conf. NPC, Sep. 2010.

[80] C. Natarajan, B. Christenson, and F. Briggs. A Study of Performance Impact of MC Features in Multi-Processor Environment. In Proceedings of WMPI, 2004.

[81] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, James E. Smith: Fair Queuing Memory Systems. In MICRO-39, 2006

[82] S. P. Muralidhara et al. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In Micro-44, 2011.

[83] H. Park et al. Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-Core, Multi-Bank Systems. In ASPLOS-2013.

[84] M. K. Qureshi and Y. N. Patt. Utility-based Cache Partitioning: A low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches. In MICRO-39, 2006.

[85] M. K. Jeong, D. H. Yoon et al. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems. In HPCA-18, 2012.

[86] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha and Jaehyuk Huh. Architectural Support for Secure Virtualization under a Vulnerable Hypervisor. In MICRO-2011.

[87] S. Rixner et al. Memory Access Scheduling. In ISCA-27, 2000.

[88] B. Rogers et al. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In ISCA-42, 2009.

[89] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of DRAM bandwidth in multicore processors. In PACT-16, 2007.

[90] K. Sudan, et al. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware. In ASPLOS-2010.

[91] H. S. Stone, J. Turek, and J. L. Wolf. Optimal Partitioning of Cache Memory. In IEEE Transactions on Computers, 41(9), 1992.

[92] L. Subramanian et al. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In HPCA-2013.

[93] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. in MICRO. 2008.

[94] J. Sim, et al. Resilient Die-stacked DRAM Caches. In ISCA-2013.

[95] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. in ICS. 1999.

[96] K. Sudan et al. Micro-pages: Increasing DRAM efficiency with locality-aware data placement. In ASPLOS, 2010.

[97] S. Srikantaiah, M. Kandemir, and Q. Wang. Sharp control: controlled shared cache management in chip multiprocessors. in MICRO. 2009.

[98] Jerome H. Saltzer and M. Frans Kaashoek. Principles of Computer System Design –An Introduction.

[99] Karthik T. Sundararajan, Vasileios Porpodas, Timothy M. Jones, Nigel P. Topham, Bj¨orn Franke. Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs. In HPCA-2012.

[100] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki and Babak Falsafi. Spatio-Temporal Memory Streaming. In ISCA-2009.

[101] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In ASPLOS-2000.

[102] Y. H. Son, et al. Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations. In ISCA-2013.

[103] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. in WIOSCA. 2007.

[104] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt and Mary Lou Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In ISCA-2011.

[105] Lingjia Tang, Jason Mars, WeiWang, Tanima Dey and Mary Lou Soffa. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In ASPLOS-2013.

[106] A. S. Tanenbaum, Modern Operating Systems. 3rd ed. 2008: Pearson-Prentice Hall.

[107] A. Udipi et al. Rethinking DRAM Design and Organization for Energy-constrained Multi-cores. In ISCA-2010.

[108] A. Udipi et al. Combining Memory and a Controller with Photonics through 3D-Stacking to Enable Scalable and Energy-Efficient Systems. In ISCA-2011.

[109] A. Wolfe. Software-based cache partitioning for real-time applications. in RCS. 1993.

[110] Y. Xie and G. H. Loh. Dynamic classification of program memory behaviors in CMPs. in the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects. 2008.

[111] Di Xu, Chenggang Wu and Pen-Chung Yew. On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling. In PACT-2010.

[112] Doe Hyun Yoon, et al. The Dynamic Granularity Memory System. In ISCA-2012.

[113] Doe Hyun Yoon, Min Kyu Jeong and Mattan Erez. Adaptive Granularity Memory Systems: A Tradeoff between Storage Efficiency and Throughput. In ISCA-2011.

[114] Wing-kei S. Yu, Ruirui Huang, Sarah Q. Xu, Sung-EnWang, Edwin Kan, and G. Edward Suh. SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading. In ISCA-2011.

[115] X. Zhang, S. Dwarkadas, K. Shen. Hardware Execution Throttling for Multi-core Resource Management. In USENIX ATC-2009.

[116] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. in EuroSys. 2009.

[117] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu. Memory access scheduling schemes for systems with multi-core processors. In ICPP, 2008.

[118] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In MICRO-33, 2000.

[119] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In ASPLOS-2010.

[120] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. in ASPLOS. 2004.

[121] 米伟，软硬协同的 DRAM 访问优化研究，博士学位论文，2009。

[122] 贾耀仓，程序行为敏感的多核共享 cache 软件划分方法研究，博士学位论文，2010。

[123] Standard Performance Evaluation Corporation. Available from: http://www.spec.org/cpu2006/CINT2006/.

[124] Hewlett-Packed Development Company Perfmon project. http://www.hpl.hp.com/researh/linux/perfmon.

致　谢

值此论文完成之际，我在计算所的博士研究工作也接近尾声。总的来讲，在这个阶段，我受到太多的人的来自方方面面的关心与帮助，这些养份是我成长与发展的动力。

首先我想说，非常幸运的来到计算机系统结构国家重点实验室开展我的研究工作。在这里，浓厚的学术氛围，尖端前沿的研究课题，学力深厚学识渊博的老师与学者，无时无刻的不感染着我们，启发着我们，引领着我们。如何发现一个问题，如何开展一项研究，如何控制科研进程，如何发表学术论文，等等，凡是与科研有关的问题都能在国重这个集体中找到答案。有些时候我也在想像，如果当初不在国重，那么我们博士研究阶段会是怎么样的？但我却无法给出答案。

感谢引领我进入研究领域的编译组的老师们。我在吴承勇老师的直接领导与指导下开展了接近 4 年的研究工作，我受益匪浅。在我的研究过程中，困难重重，我自己很迷茫，吴老师非常支持和理解，在必要的时候给我必要的协助和指导。没有吴老师的理解和支持即不会有我的研究成果。感谢冯晓兵老师。我作为科学研究上的后来人，从你们身上得到很多宝贵的精神财富。还要特别感谢冯老师对我工作生活上的关心、指导和协助。编译组是个团结友善的大家庭，武成岗老师、陈莉老师、崔慧敏老师都不同程度的给予过我帮助，感谢你们。特别要向张兆庆老师和乔如良老师致敬，二老是我们青年学者的楷模！

感谢我可爱的师弟师妹们。特别是彭亮、房双德、侯敬一、齐霞光、何文婷、杜子东、李晶、宋育庚等，每当想到你们，我都非常高兴，希望你们能越来越好。感谢我的朋友、同学和同事们，邢明杰、李勇（pitt）、陈洋、黄元杰、（大）刘雷、小卢（卢兴敬）、老唐（唐生林）、韩冬妮、李胜梅、吕方、王蕾、黄磊、小杜等。不能枚举，请见谅。

感谢我的父母和家人。你们为我付出的太多太多，我非常感动。你们在我生命中是至关重要的，谨以此文，献给你们。我平常对你们表达的太少，但你们一直在我心里，一直在我的世界里，我的优点全都来自于你们，你们懂的！

作者简介

姓名：刘磊　　性别：男　　出生年月：1981/9/11　　籍贯：辽宁沈阳

教育背景：

大连理工大学　计算机科学与工程　学士

中国科学技术大学　计算机软件　硕士

发表文章：

1. Going Vertical in Memory Management: Handling Multiplicity by Multi-Policy.

第一作者，ISCA-2014

2. BPM/BPM+: Software-based Dynamic Memory Partitioning Mechanisms for Mitigating DRAM Bank-/Channel-level Interferences in Multicore Systems.

第一作者，TACO-2014

3. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems.（截止本博士论文提交之时，该文被已被引用***20***余次）

第一作者，PACT-2012

4. Dynamic I/O-Aware Scheduling for Batch-Mode Applications on Chip Multiprocessor Systems of Cluster Platforms.

Fang Lu, Huimin Cui, Lei Wang, **Lei Liu**, Cheng-Gang Wu, Xiao-Bing Feng, and Pen-Chung Yew. JCST-2014

5. WiseThrottling: A New Asynchronous Task Scheduler for I/O Bottleneck in Large-Scale Servers of Datacenter.

Fang Lu, **Lei Liu**, Huimin Cui, Lei Wang, Ying Liu, Xiaobing Feng, Pen-chung Yew. In submitting to JPDC-2014

申请专利：

一种页着色协同划分消除多应用缓存内存干扰的方法；已受理；第一发明人

读博期间主要参与课题：

（973）面向亿级并发负载的编程模型与支撑环境

所获奖励：

国家奖学金 博士生奖 2012

中科院三好学生 2012

北纬通信自主创新奖学金二等奖 2013

系统结构国家重点实验室优秀研究生 2012