doi: 10.3969/j.issn.1005-3026.2014.03.011

# 一种多核系统上基于页着色的内存管理方法

张 轶, 关 楠, 王 义
(东北大学 信息科学与工程学院, 辽宁 沈阳 110819)

**摘 要**: 当今多核平台多采用共享cache架构,但运行在不同核心上的任务产生的cache冲突问题使得程序最坏执行时间的计算变得十分困难.因此提出了使用页着色技术解决多核cache上访存冲突问题的方法.此方法的优势是使已有单核上的WCET分析技术可以对多核上的程序执行时间进行判断.在Linux系统上实现了支持页着色划分方法的内存管理系统,并使用通用测试集对该方法进行了测试.实验结果表明,在Linux系统中使用该内存管理策略后,在相同多核平台上程序的执行时间变得可预测.

**关 键 词**: 多核;cache;实时;页着色;操作系统
**中图分类号**: TP 302 **文献标志码**: A **文章编号**: 1005-3026(2014)03-0351-05

# A Memory Management Approach Based on Page Coloring for Multi-core Systems

*ZHANG Yi*, *GUAN Nan*, *WANG Yi*
(School of Information Science & Engineering, Northeastern University, Shenyang 110819, China. Corresponding author: ZHANG Yi, E-mail: zhangyi@ise.neu.edu.cn)

**Abstract**: Most multi-core platforms currently adopt shared cache among the processor cores. Due to the problem of cache contention, it is extremely difficult to predict the worst-case execution time of the computation tasks running on different cores. A page-coloring technique was proposed to avoid cache contention in memory access for multi-core platforms. The advantage is that the worst case execution time of tasks running on individual core can be estimated separately using the existing WCET analysis methods for uni-processor systems. A memory management system was designed based on the Linux to support page coloring mechanism, which is evaluated using the standard benchmarks. Experimental results shown that the execution time for different processor cores becomes deterministic when it is executed on the same platform running in the Linux supported with the management system.

**Key words**: multi-core; cache; real-time; page coloring; operating system

在多核系统上,各核心往往以共享的方式使用末级cache,这样的架构不仅提高了核间数据传输效率,也让cache资源可以在核间动态分配.但是由此,程序间必须以竞争的方式使用cache,使得一个程序所保持在cache上的内容不再仅仅由其自身决定,也受到来自其他核心上程序的影响,从而极大地增加了在多核系统上判断程序执行时间的难度.

在实时系统的设计与验证等环节中,能否准确判断程序执行时间对系统性能有重要影响[1].其中,针对关键任务所做的性能分析通常基于最坏情况执行时间(WCET)所得到的结果.但是因为共享cache的存在,使得传统单核上的执行时间分析技术都不能应用在多核系统上.而近来提出的多核WCET分析技术在适用性上仍存在很大问题[2],所得到的结果过于悲观,资源利用效率很低.

针对由共享cache引起的多核实时系统中程序执行时间分析问题,本文提出了一种基于页着色技术的管理方法.该方法通过隔离并行程序在

**收稿日期**: 2013-08-14
**基金项目**: 国家自然科学基金资助项目(61100023); 中央高校基本科研业务费专项资金资助项目(N120404008).
**作者简介**: 张 轶(1982-),男,辽宁沈阳人,东北大学博士研究生; 王 义(1961-),男,内蒙古赤峰人,东北大学教授,博士生导师.

共享cache上的运行空间,避免了其上的访问冲突问题. 在该方法的作用下,已有的单核系统WCET分析方法可以直接应用在多核系统上. 本文基于Linux操作系统实现了支持cache隔离的内存管理系统,并在真实系统上验证了该方法的效果. 实验结果表明,应用cache隔离策略以后,程序的执行时间变得更加稳定.

## 1 背景及意义

在多核系统上,为了提高核间通信效率与资源利用效率,内存在核间多以共享的方式被使用. 但是内存带宽远不能匹配处理器对数据访问的要求,导致以内存为媒介的数据传递效率很低. 为此,在现有的多核架构上,多使用共享的末级cache链接各个核心,以此减少核间通信的延时.

Cache上保持的数据是内存中内容的子集,多个内存地址对应同一个cache组(set),在一个cache组上包含多路(way)cache块(block,也称cache行),每一个块是cache与内存交换的基本单位,如图1所示. 替换机制会根据数据在cache上的驻留时间与访问远近为决策依据,尽量保持最有可能在未来被使用的数据在cache上. 在现有多核系统中,共享cache依然沿用传统的专注cache使用效率的工作机制设计,因此产生了由并行程序争用cache导致的执行时间波动问题.

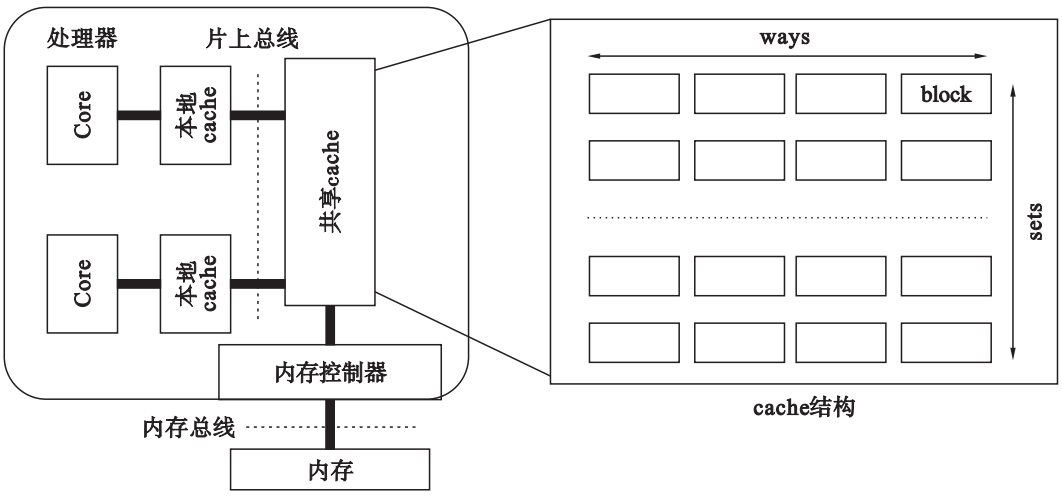


图1 多核系统上的缓存架构

Fig. 1 The memory hierarchy on multi-core system

在实时系统尤其是含有硬实时任务的系统中,必须保证任务满足最严格的时间约束. 否则,如果硬实时任务未能在限定时间内完成将给系统带来灾难性后果. 因此判定硬实时任务的执行时间需要考虑系统最坏条件下的执行情况.

为了给出实时任务的执行时间,需要精确的计算程序对cache的访问延时. 在共享cache的情形下,计算一个程序的访存请求是否命中cache就扩展成了计算所有同时运行程序的访存请求的cache访问情况. 再进一步,如果考虑到每一个处理核心上都可能运行多个程序,而不同核心上的程序并不一定保持严格的同步运行关系,这样,计算一个程序的执行时间就可能跟所有其他核上运行的程序发生关联. 如果以如上条件计算最坏情况下程序的运行时间,则问题空间和计算难度都变得非常巨大,还可能由于得到的结果过于悲观而造成系统资源的巨大浪费. 由此,本文提出在实时系统上,以隔离程序间cache运行区间的方式使用共享cache,并由此提出了通过页着色技术实现这一策略的方法.

## 2 页着色技术及相关工作

在当今的主流系统上,普遍使用虚拟地址运行程序,而在cache、内存与I/O设备上使用物理地址表示数据. 页着色是在程序的虚拟地址和物理地址的映射环节,通过控制程序所分配到的页面来影响程序在存储器上所占有的物理区间的一种方法[3]. 因为虚拟地址以内存的页式管理为基础,所以页着色以页面大小为单位控制映射区间.

图2中示例了页着色如何控制程序到cache位置上的映射. 通常,末级cache使用物理地址映射,又由于物理地址范围远大于cache容量,所以一段cache区域对应多个不同的物理页面,且所对应的页面是固定的. 这样,如果需要程序使用某一区间的cache,在该程序申请物理页面的过程

中，分配该区间所对应的物理页面就实现了对映射区间的控制.

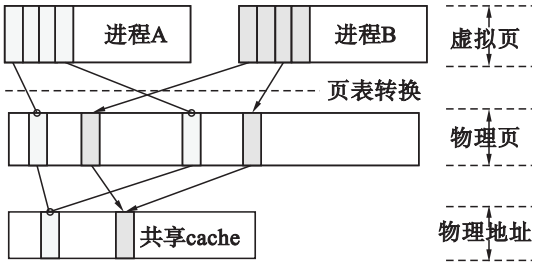


图 2　页着色及 cache 映射

Fig. 2　Page coloring and cache mapping

近年来，已经有很多研究工作关注了利用页着色技术解决多核上共享资源使用的问题[4-9]. 文献[4]中利用页着色改进了非一致性 cache 上的执行性能；文献[5]研究了不同策略在使用页着色时对系统性能产生的影响；文献[6]利用页着色改进 LRU 替换策略下的 cache 性能；文献[7]提出了基于访问热点动态页着色的方法；文献[8]中研究了使用页着色改善内存读写效率的方法. 以上的研究工作多数是针对页着色在非实时领域的应用. 在文献[9]中作者给出了基于 cache 划分的实时任务调度方法的研究. 但是到目前，还没有工作给出在多核实时系统的应用背景下，应用页着色策略的系统运行结果.

## 3　机制设计与实现

### 3.1　Cache 隔离的页着色机制

针对多核系统上共享 cache 给实时系统开发所带来的问题，本文提出在多核实时系统上采用隔离 cache 使用区间的方式解决共享 cache 争用问题. 具体地，使用页着色技术控制每一个程序只固定使用共享 cache 上的一个子集，然后通过任务的核间划分跟调度的方式保证同时运行的程序不共同映射相同 cache 区间[9]. 因为每一个任务所使用到的 cache 都是独占的，使得任务的运行条件等效成单核架构下的运行条件. 这样，可以运用已有的针对单核的 WCET 分析技术确定多核系统上程序的执行时间.

使用页着色机制解决 cache 冲突的另一理由是良好的通用性. 页着色机制的核心内容是按照规定的条件实现虚拟页面到物理页面的映射. 因为虚拟地址的页面映射信息是由软件写入的，可以不受硬件结构的限制，所以基于页着色的策略能够应用在多数现有平台上.

### 3.2　页颜色判定

本文页颜色判定主要解决的是如何计算系统中可用颜色的问题. 在现有的多核系统上，普遍采用了多级 cache 的架构，使用页着色隔离共享 cache 使用区间也可能同时影响到其他层级上的 cache 使用. 这里使用 $C_i$ ( $C_i = 2^{c_i}$ ) 表示第 $i$ 级 cache 的容量，$A_i$ ( $A_i = 2^{a_i}$ ) 表示其相连度，则寻址第 $i$ 级 cache 需要的地址位数可知为 $c_i - a_i$. 则当页面大小为 $P$ ( $P = 2^p$ ) 时，该级上可使用颜色数为 $2^{c_i - a_i - p}$. $C_i$ 与 $A_i$ 的值可以通过系统命令查看. 当系统命令无效时，可以通过文献[10]中提供的方法获得.

确定页面颜色也就是确定页面在 cache 上的映射位置，这是由 cache 的地址解析逻辑决定的，对于不同层级上的 cache 其地址解析的工作原理是不完全相同的. 在与处理器核心相连的第一级 cache 上，通常 $2^{c_1 - a_1} = 2^p$. 因为虚拟页内的地址与物理页上的地址是直接对应的，这样的设计在该级上就可以使用虚拟地址直接读取 cache，避免了 cache 命中时的页表转换延时. 因为在第一级 cache 上超过页面大小的范围里没有 cache 寻址位，所以页着色不会影响第一级 cache 的使用.

在第一级 cache 之后的层级上都使用物理地址进行寻址，且 $2^{c_i - a_i}$ 都要大于页的大小，此时页着色就可能限制该级上的可使用 cache. 由于各级 cache 的容量有限，而且还要同时运行用户程序和系统代码，使得限制末级 cache 以外的其他层级的本地 cache 会对程序的延时产生较大影响. 因此，在本文的划分策略上保证页着色不会缩小本地 cache 的使用范围. 则得到：

当系统有 $k$ 级 cache 时，系统可用的颜色数 $N = 2^n$，其中 $n, k$ 为自然数，

$$n = (c_k - a_k) - \mathrm{Max}\{(c_i - a_i) : i \in (1, k-1)\}.$$

图 3 所示为采用上述颜色判定方法后在共享 cache 上的着色关系示意图.

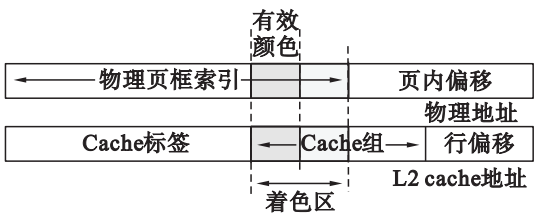


图 3　共享 cache 上的页着色

Fig. 3　Page coloring on shared cache

### 3.3　着色机制的实现

本文选择在操作系统的内存管理模块上验证页着色机制. 一方面，当今多数系统上都是由操作

系统负责直接维护虚拟地址到物理地址的映射;另一方面也比较容易对系统性能参数进行收集与分析. 本文选择了 Linux 系统为原型系统,但是该实现方法也可以适用于其他的主流操作系统.

在现有的 Linux 系统上,页面分配采用先来先服务的管理策略,进程请页所得到的页面物理地址是随机的[11]. 为了实现受控的页面分配,本文修改了现有的页面分配机制,实现方法如下.

在系统中,完成请页过程需要涉及三部分,分别是用户接口、请页逻辑与页面管理部分. 其中请页逻辑负责根据请页条件判定如何分配页面给内存申请者,再在空闲内存池中找到合适的页面赋给申请者. 为了减少由策略的实现方法所引入的执行时间变化,本文所实现的着色逻辑依然沿用 Linux 系统中的原有结构. 在修改后的系统中,同时支持着色策略与 Linux 默认策略. 本文所做的主要工作是改写请页流程上的请页条件判断与空闲页面查找函数,在其中添加页着色请求的判断与执行逻辑.

在图 4 中给出了修改后的请页机制框架,color 是着色请页所使用的域,在普通的请页过程中,该域为空,执行系统自带页面申请,反之执行按颜色请页. 在本文的实现方法中,选择着色页的过程都已在请页逻辑中完成,页面管理部分无需再进行改动. 另外,还需要修改进程控制块 task_struct 以及用户接口部分等相关部分,如程序的加载函数 fork 跟 exec,使系统能够识别并加载进程颜色信息.

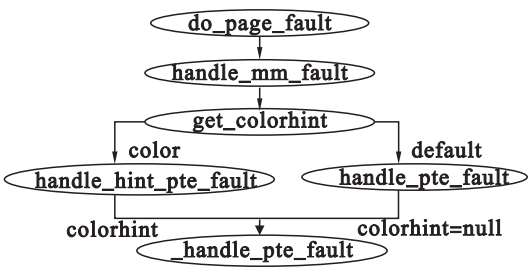


图 4 带着色策略的请页机制框图

Fig. 4 Page allocation with coloring mechanism

# 4 实验评估

## 4.1 实验环境

本文中使用了以 Intel Core 2 Duo 为处理器的实验平台. 该处理器为双核,两级 cache 结构,L1 cache 分别含有 32KB 8 路组相联的指令与数据 cache,L2 cache 为 2MB 8 路组相联的共享 cache. 系统使用 512MB DDR2 内存,操作系统版本为 Linux 2.6.32. 测试程序选择了两类程序:一种是嵌入式应用领域的测试集 MiBench[12],并从中选择了能够反映 cache 使用性能的测试程序 sha,susan edges(suse),susan smoothing(suss);另一种是仿照 STREAM[13] 测试集构造的矩阵有序读写程序 mcol 与矩阵随机读写程序 cnt.

## 4.2 实验设计

为了检验在使用着色隔离策略以后,程序在多核系统上的执行性能,设计了如下实验. 在一处理器上固定运行一个程序,在另一处理器上分别运行其他程序. 每一对程序的组合在一次运行中各自循环 300 次,因为每一程序运行的时间长短不相同,这样不同次的循环里可以产生不同的代码段冲突. 在不同的循环中,程序被重新创建,这样请页逻辑需要从内存池中重新为程序分配页面,能为程序段产生不同的页面映射位置.

实验分别在默认系统条件下以及着色策略下运行. 通过 3.2 节中的方法可知,在本系统上可以使用的颜色数为 64,在着色实验中,每一个任务分配 32 个颜色. 针对实时系统的需要,在执行中记录程序的最长执行时间. 实验结果如图 5 所示,其中纵坐标代表执行时间,横坐标表示测试集的组合情况. 图 5a 为程序运行在 Linux 默认环境下的结果,图 5b 为程序在着色隔离策略下的运行结果. 实验的基准程序为 mcol,对应每对组合中的左侧数值.

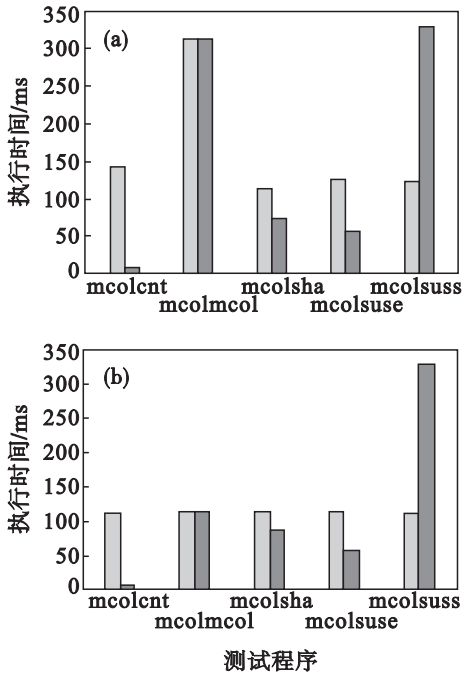


图 5 程序在有无着色策略下的执行时间

Fig. 5 Execution time without & with coloring policy

(a)—默认系统配置;(b)—着色隔离策略.

### 4.3　结果分析

通过实验可以看到，在没有着色策略下，当 mcol 与不同程序运行，其执行时间波动较大. 但是在有着色策略的环境下，mcol 的执行时间是相对稳定的，说明着色策略可以有效地避免程序执行时间的波动问题.

实验中也存在程序在两种系统上运行时间差别不大的情形. 通过分析发现，当任务集使用较少的 cache 或者使用所需 cache 远大于所分配的数量时这样的结果出现较多. 前一种情况的产生是因为系统上发生冲突的概率相对少；第二种情况的产生是因为程序本身的 cache 缺失率也很高，发生冲突所占比例不足以反映到整体运行时间上. 而且，Intel 处理器有较强的预取功能，也掩盖了部分 cache 缺失延时. 在很多嵌入式处理器上，并没有足够的资源实现完善的预取策略，那样的环境下 cache 冲突的问题将会表现地更为明显.

## 5　结　　论

本文主要研究了如何通过软件方法解决多核 cache 冲突引起的执行时间预测困难问题. 提出了通过页着色隔离策略解决程序间共享 cache 访问冲突的方法. 该方法的优势是能够使用已有单核系统上 WCET 分析技术分析多核系统程序的运行时间，并且该方法还具有使用简单、适用性好的特点. 本文设计并实现了支持该策略的原型系统，并在实际系统上选择有代表性的实时任务集对其进行了测评. 实验结果表明，在使用该策略后，可以对程序的执行时间进行预测.

**参考文献：**


[1] Wilhelm R, Engblom J, Ermedahl A, *et al*. The worst-case execution-time problem-overview of methods and survey of tools[J]. *ACM Transactions on Embedded Computing Systems*, 2008, 7(3): 1 – 53.

[2] Li Y, Suhendra V, Liang Y, *et al*. Timing analysis of concurrent programs running on shared cache multi-cores [C]//Real-Time Systems Symposium. Washington D C: IEEE, 2009: 57 – 67.

[3] Kessler R E, Hill M D. Page placement algorithms for large real-indexed caches [J]. *ACM Transactions on Computer Systems*, 1992, 10(4): 338 – 359.

[4] Cho S, Jin L. Managing distributed, shared L2 caches through OS-level page allocation [C]//IEEE/ACM International Symposium on Microarchitecture. Washington D C: IEEE, 2006: 455 – 468.

[5] Lin J, Lu Q, Ding X, *et al*. Gaining insights into multicore cache partitioning: bridging the gap between simulation and real systems[C]//High Performance Computer Architecture. Salt Lake City: IEEE, 2008: 367 – 378.

[6] Soares L, Tam D, Stumm M. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer[C]//International Symposium on Microarchitecture. Washington D C: IEEE Computer Society, 2008: 258 – 269.

[7] Zhang X, Dwarkadas S, Shen K. Towards practical page coloring-based multicore cache management[C]//Proceedings of the 4th ACM European Conference on Computer Systems. New York: ACM, 2009: 89 – 102.

[8] Jeong M K, Yoon D H, Sunwoo D, *et al*. Balancing DRAM locality and parallelism in shared memory CMP systems[C]// High Performance Computer Architecture. New Orleans: IEEE, 2012: 1 – 12.

[9] Guan N, Stigge M, Yi W, *et al*. Cache-aware scheduling and analysis for multicores[C]//Proceedings of the seventh ACM International Conference on Embedded Software. New York: ACM, 2009: 245 – 254.

[10] Yotov K, Pingali K, Stodghill P. Automatic measurement of memory hierarchy parameters [J]. *ACM SIGMETRICS Performance Evaluation Review*, 2005, 33(1): 181 – 192.

[11] Wolfgang M. Professional Linux kernel architecture [M]. Indianapolis: Wiley Publishing, 2008: 1 – 1337.

[12] Guthaus M R, Ringenberg J S, Ernst D, *et al*. MiBench: a free, commercially representative embedded benchmark suite [C]//IEEE International Workshop on Workload Characterization. Washington D C: IEEE, 2001: 3 – 14.

[13] McCalpin J D. STREAM: sustainable memory bandwidth in high performance computers[EB/OL]. [2013 – 09 – 01]. http://www.cs.virginia.edu/stream/.