doi:10.11887/j.cn.201605006 http://journal.nudt.edu.cn

位置信息与替换概率相结合的多核共享 Cache 管理机制*

徐金波[1],庞征斌[1,2],李 琰[1]
(1.国防科技大学 计算机学院,湖南 长沙 410073;
2.国防科技大学 并行与分布式计算重点实验室,湖南 长沙 410073)

摘 要:多核系统中末级 Cache 是影响整体性能的关键。为了提出一种细粒度、低延迟、低代价的末级共享 Cache 资源管理机制,将系统性能目标转换为每个内核当前占用 Cache 资源的替换概率,以决定每个内核能够提供的被替换资源的数量;对某个需要增加 Cache 资源的内核,从可提供被替换资源的候选内核中选出距离较近且替换概率较高的一个内核,并以 Cache 块为粒度进行替换,从而实现 Cache 资源在不同内核间的动态划分。与传统以相联度为粒度的粗粒度替换机制相比,以 Cache 块为单位的替换机制具有更细的替换粒度,灵活性更高。另外,通过将位置信息和替换概率结合,保证了 Cache 资源与相应内核在物理布局上的收敛,降低了访问延迟。同时,所提出的方法只需要增加极少的硬件代价。实验结果表明,根据实验场景和对比对象的不同,所提方法与其他已有研究成果相比,可以实现从6.8%到22.7%的性能提升。

关键词:多核系统;末级 Cache;动态划分;替换策略
中图分类号:TP391.41 **文献标志码**:A **文章编号**:1001-2486(2016)05-032-07

Shared Cache management scheme with location information and eviction probability in multi-core system

XU Jinbo[1], *PANG Zhengbin*[1,2], *LI Yan*[1]
(1. College of Computer, National University of Defense Technology, Changsha 410073, China;
2. Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, China)

Abstract: LLC (last level Cache) plays an important role in multi-core systems. A shared LLC management scheme with fine granularity, low latency and simple hardware complexity was proposed. The performance goal was translated into eviction probabilities of each core. Then, a victim core, which was near the current core and had higher eviction probability was chosen to provide the victim block for replacement. In this way, LLC was dynamically partitioned among all cores at finer granularity of Cache blocks. This proposal is more flexible than traditional way-partitioning scheme. In addition, the combination of location information and eviction probability improves the locality between Cache resources and the corresponding cores, which can reduce the Cache access latency. The proposed scheme requires only little additional hardware changes to traditional Cache structure. Results on M5 simulator suggest that the performance can improve from 6.8% to 22.7% when comparing with the related works.

Key words: multi-core system; last-level Cache; dynamic partitioning; replacement policy

细粒度、低延迟、低代价是多核系统中末级共享 Cache 资源管理机制多年来的研究重点[1-9]。例如,Zhang 等基于 page coloring 进行 Cache 资源管理[1],但该方法在应用程序获得存储资源时灵活性不足,而且在进行动态 Cache 划分时需要进行大量的页拷贝(page copying),实现代价较大;刘胜等通过在 Cache 划分时采取一些可配置策略进行优化[2],但配置灵活性有待提高;El-Moursy 等提出了 V-set Cache[3],但 Cache 划分粒度仍不够精细;Chen 等通过硬件卸载的方式进行 Cache 划分[4],性能优于操作系统(Operating System, OS)层的划分策略,但该方法仍是 way-partitioning,属于粗粒度划分。

已有的 Cache 划分策略可分为粗粒度划分[4,6]和细粒度划分[7,10]两类。例如,way-partitioning 划分策略通过为每个内核各分配一部分相联度来实现划分,属粗粒度划分,虽易于实现,但其划分粒度反比于相联度,当内核数与相联

* **收稿日期**:2015-11-19
基金项目:国家自然科学基金资助项目(61202126);国家863计划资助项目(2012AA01A301,2013AA01A208);国家部委基金资助项目(2011CB309705-1)
作者简介:徐金波(1981—),男,山东高唐人,助理研究员,博士,E-mail:xujinbo@nudt.edu.cn

度较接近时,系统性能会急剧恶化。Vantage[7] 策略属于细粒度划分,但只能对大部分区域而不是全部 Cache 资源进行资源管理,且对 Cache 基础框架改动较大,并对替换算法提出了较高的要求,因此实现代价偏高。为了使用更低代价实现细粒度 Cache 划分,Manikantan 等提出了 PriSM 策略[10],通过动态计算 Cache 资源的替换概率实现以 Cache 块为单位的细粒度划分,且硬件代价较低,但该方法没有充分考虑对 Cache 访问延迟的优化,可能会出现内核距离待访问的 Cache 资源较远从而导致延迟较大的问题。

本文提出一种基于位置信息与替换概率的多核共享 Cache 管理机制,以期满足细粒度、低延迟、低代价的需求,在综合性能上优于已有研究成果。该方法首先将系统性能目标转换为每个内核当前占用 Cache 资源的替换概率,以决定每个内核能够提供的 Victim Line 数量;然后,对某个需要增加 Cache 的内核,从可提供 Victim Line 的候选内核中选出距离较近且替换概率较高的一个内核,并以 Cache 块为粒度进行替换,实现 Cache 在不同内核间的动态划分。与传统 way-partitioning 等粗粒度替换机制相比,以 Cache 块为单位的替换机制具有更细的替换粒度,灵活性更高。另外,通过将位置信息和替换概率相结合,保证了 Cache 资源与相应内核在物理布局上的收敛,降低了访问延迟。同时,所提出的方法只需增加极少的硬件代价。模拟器实验结果表明,所提出的方法与已有研究成果相比,根据实验场景、性能衡量标准和对比对象的不同,可实现从 6.8% 到 22.7% 的性能提升。本文工作与最近最少使用算法(Least Recently Used, LRU)相比可以在 32 核片上多处理器(Chip Multi-Processor, CMP)上实现约 21% 的性能提升,与 PriSM 相比在整体性能上可提高约 6.8%。

1　动机

粗粒度划分策略对 Cache 资源的调整幅度过大,理论上的最优划分边界很难与粗粒度划分的划分边界重合,因此有必要研究使用更细粒度的 Cache 划分策略来提升性能的可能性。way-partitioning 方法对所有 Cache 组统一以一个相联度为粒度进行调整,如 16 路组相联 Cache 每次调整会将 6.25% 的 Cache 从一个内核重新分配给另一个内核。类似地,64 路和 256 路组相联 Cache 的调整幅度分别是 1.6% 和 0.39% 的 Cache 资源。已有研究工作发现更细粒度的 Cache 划分策略有利于提高系统性能[11],但是通过提高相联度所实现的细粒度划分策略是以非常复杂的控制逻辑和过多的硬件开销为代价的,因此需要考虑实现更简单、粒度更精细的 Cache 划分。以 Cache 块为单位的 Cache 划分可以实现比高相联度方法更精细的调整粒度。当 Cache 块总数为 N 时,每个内核可以以 $1/N$ 的调整幅度对 Cache 资源进行增减。

以 Cache 块为粒度进行划分需要考虑的关键问题是资源分配和划分机制的设计,只有能够动态及时地对 Cache 资源进行科学分配和划分,才能使得理论上的最优划分边界尽可能地与实际划分边界重合,从而充分发挥所有 Cache 资源的作用。Cache 分配机制要保证每个 Cache 块都得到管理并被分配给其中一个内核,尽量避免出现 Vantage[7] 方法中“unmanaged”区域内 Cache 块不属于任何一个内核的情况。基于这种分配原则,以 Cache 块为粒度的 Cache 划分机制本质上是将 Cache 块在不同的内核之间进行“转移”的问题。根据每个内核的负载情况以及系统希望达到的性能目标,如果能够算出每个内核除满足自身需求外能够贡献给别的内核的 Cache 资源份额,那么对于某个需要增加 Cache 的内核,就可以从具有空闲资源的候选内核中选择一个内核为其提供 Cache 块。因此,每个内核能够贡献给别的内核的 Cache 资源份额可以反映该内核中的 Cache 被替换的可能性,即替换概率。不同内核替换概率的差异可以作为 Victim Line 的选取依据。另一方面,Victim Line 的选取还需要考虑内核到 Cache 块的访问延迟问题,当内核到不同候选 Victim Line 的访问延迟不同时,显然应该选择较近的 Cache 块。因此,位置信息也可作为 Victim Line 的选取依据。

基于以上动机,所提出的多核共享 Cache 管理机制既考虑了替换概率,又考虑了内核与 Cache 之间的位置信息,希望在实现细粒度划分的同时,也能够保证内核对 Cache 的较低的访问延迟。

2　基于位置信息与替换概率的多核共享 Cache 管理机制

为便于说明,使用 N 表示所有 Cache 块的数量。W 表示对 Cache 替换概率进行重新计算的时间间隔,通常采用发生特定次数的 Cache 失效的时间长度来衡量。当某次重新计算的时间点到来时,每个内核所期望的 Cache 资源百分比将被重

新计算。C_i 是每段时间间隔开始时 $Core_i$ 所占用的 Cache 资源百分比。M_i 则是每段时间间隔内 $Core_i$ 所发生的失效次数占所有失效次数的百分比。T_i 为 $Core_i$ 为了达到一定的性能目标所期望占有的 Cache 资源百分比。τ_i 为重新划分之后 $Core_i$ 实际所能够占有的 Cache 资源百分比。E_i 为 $Core_i$ 所占有的 Cache 块被替换出去的概率。本文假定程序数量与内核数量相同,且不存在程序迁移。

2.1 替换概率的计算

本文工作的基础之一是对每个内核所占用的 Cache 资源计算出一个替换概率。替换概率每隔一定时间间隔重新计算一次,计算过程参考文献[10]。在以失效次数 W 衡量的时间间隔内,$Core_i$ 所导致的失效次数百分比为 M_i,因此失效次数为 $M_i \times W$。假设在时间间隔内 $Core_i$ 的 Cache 资源没有被替换出去,那么 $Core_i$ 所占用的 Cache 资源将会从 C_i 增加到$[C_i + (M_i \times W/N)]$。如果将替换概率 E_i 考虑进去,该时间间隔内 $E_i \times W$ 个 Cache 块将会被替换出去,因此调整之后 $Core_i$ 实际占有的 Cache 资源百分比将会变为$\tau_i = C_i + [(M_i - E_i) \times W/N]$。如果要达到期望的性能目标,就需要使得 τ_i 尽快地接近期望值 T_i。当在单个时间间隔内 τ_i 无法达到 T_i 时,如果$C_i < T_i$,则将 E_i 设为 0;如果 $C_i > T_i$,则将 E_i 设为 1。反之,如果在单个时间间隔内 τ_i 可以达到 T_i,E_i 可以通过公式 $T_i = \tau_i = C_i + [(M_i - E_i) \times W/N]$ 计算得到。综上,E_i 的计算公式如式(1)所示。

$$E_i = \begin{cases} 0, \text{if}[(C_i - T_i) \times N/W + M_i] < 0 \\ 1, \text{if}[(C_i - T_i) \times N/W + M_i] > 0 \\ (C_i - T_i) \times N/W + M_i, \text{otherwise} \end{cases} \quad (1)$$

从式(1)中可以看出,计算 E_i 首先需要确定 C_i,M_i 和 T_i。C_i 和 M_i 可以通过为每个内核设置相应的计数器分别对所占用的 Cache 块数量和 W 时间间隔内的失效次数进行统计来得到。T_i 的计算则需要与系统所期望达到的性能目标进行关联,本文主要研究命中率最大化的性能目标。命中率最大化的性能目标试图为更容易获得较高命中率的内核分配更多的 Cache 资源。通常,通过假设将所有 Cache 资源都分配给某个内核 $Core_i$ 时,该内核所能够得到的命中率 $StandAloneHits[i]$ 与所有内核共享 Cache 时 $Core_i$所得到的 Cache 命中率 $SharedHits[i]$ 之间的差异来衡量该内核对 Cache 资源的依赖程度。在每个时间间隔内,$StandAloneHits[i]$ 与 $SharedHits[i]$ 采用不同的实现机制并行获得:在每个内核中,使用计数器对 $SharedHits[i]$ 进行记录;而对于 $StandAloneHits[i]$,则使用文献[6]中的方法进行估计,具体处理过程是使用辅助标签目录(Auxiliary Tag Directory, ATD)和命中率计数器对 Cache 资源的利用情况进行分析,并通过将命中率进行累加的方法得到 $StandAloneHits[i]$。对 Cache 资源依赖程度高的内核将期望得到更多的 Cache 资源,基于这种相关性,可以计算出每个内核 i 所期望得到的 Cache 资源百分比 T_i,进而计算出每个内核的替换概率 E_i。算法描述如图 1 所示。

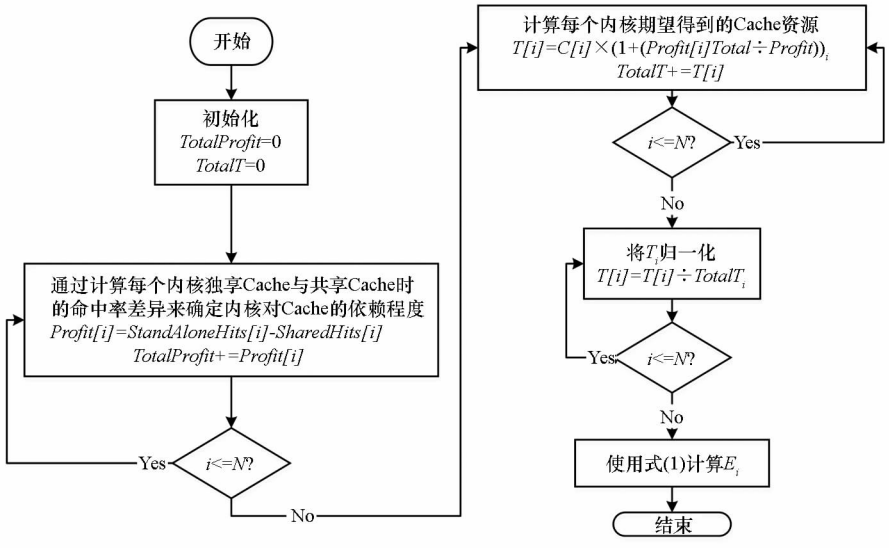


图 1 T_i 和 E_i 的计算

Fig. 1 Calculation of T_i and E_i

对于某个需要增加 Cache 资源的内核,就可以根据其他内核的替换概率确定一个 Victim

Core,该 Victim Core 从它所拥有的 Cache 资源中选取部分 Cache 块,将数据进行替换,贡献给需要增加 Cache 资源的内核。

Victim Core 的选择以及 Victim Core 内 Cache 块的选择是接下来需要考虑的另一个关键问题。对于 Victim Core 的选择,可以选择替换概率最高的内核或者从替换概率分布中随机选取内核。对于已选定 Victim Core 内 Cache 块的选择,可以采用各种已有研究中所提出的 Cache 替换策略进行选择,如 LRU,动态插入策略(Dynamic Insertion Policy, DIP)[12]等。但是上述方法没有考虑对访问延迟的优化,当被选中的 Cache 块与内核之间距离较远时,较大的访问延迟导致 Cache 效率降低。本文在 Victim Core 以及 Cache 块的选择过程中将内核和 Cache 的相对位置信息与替换概率融合起来,目的是实现访问延迟的优化。

2.2　位置信息的融合

多核处理器中内核与 Cache 体的布局方式是多种多样的,如 Cache 居中方式、内核居中方式、分组布局方式、瓦片结构布局方式等。限于篇幅,本文仅以应用较广泛的瓦片结构为基础进行阐述。瓦片结构多核处理器基本结构示意图如图 2 所示。每个瓦片由一个(或多个)内核、私有 L1 指令和数据 Cache、共享 L2 Cache、用于维护 Cache 一致性的目录结构、瓦片间互联的路由部件组成。L2 Cache 虽然分布布局在每个瓦片中,但是这些 Cache 由所有瓦片中的内核共享,某个瓦片内的内核可能会对其他瓦片内的 L2 Cache 进行访问。显然,内核到 Cache 的网络距离越远,访问延迟越大。因此,为了尽量减少 Cache 访问延迟,在进行 Cache 划分时需要尽量使得内核与相应 Cache 资源的相对位置较近。

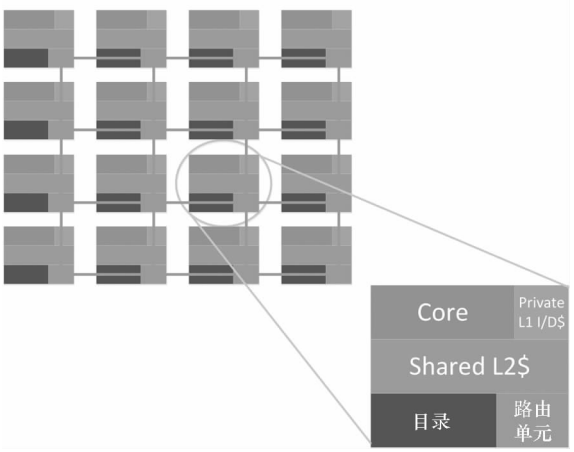


图 2　采用 2D Mesh 网络的瓦片结构多核处理器

Fig. 2　Tile-structured multi-core processor with 2D mesh network

基于替换概率的细粒度 Cache 划分策略基本思想是从其他替换概率非零的内核拥有的 Cache 资源中选择合适的 Cache 块,将其替换为当前内核需要的数据,实现 Cache 块从其他内核向当前内核的转移。当内核访问 Cache 命中时,工作模式与传统 Cache 一样;当内核访问 Cache 失效时,通过两步操作实现替换,第一步是选择合适的 Victim Core,第二步是从 Victim Core 拥有的 Cache 资源中选择合适的 Victim Line。

将内核和 Cache 的相对位置信息加入到基于替换概率的细粒度 Cache 划分策略设计思想中,以优化内核对共享 Cache 的访问延迟。为了实现这种方法,在每个瓦片内设置一个查找表 E-Table,查找表可采用寄存器队列方式进行组织,队列中每个寄存器项设两个字段,分别保存当前瓦片内共享 Cache 所属的所有内核的 ID 以及这些内核的替换概率。对于位于第 i 行第 j 列瓦片 $Tile(i,j)$ 中的内核 $Core(i,j)$,如果该内核需要增加 Cache,首先根据当前瓦片的 E-Table 内的信息选出其中一个替换概率非零的内核作为 Victim Core,选择时可采用随机原则或者选择替换概率最大的内核。然后,对于被选中的 Victim Core,从该 Victim Core 所占有的当前瓦片内的 Cache 资源中选择一个 Cache 块进行替换,替换策略可以使用传统 Cache 中普遍使用的任何一种替换策略。如果从当前瓦片 E-Table 中可以查到的内核中至少有一个可以提供可替换的 Victim Line,就可以将该 Cache 块通过替换操作从该 Victim Core 转移给当前瓦片的内核;否则,如果在当前瓦片内无法找到可替换的 Cache 块,就考虑扩大范围,从该瓦片 $Tile(i,j)$ 周围的其他瓦片 $Tile(i\pm1,j\pm1)$ 的 E-Table 中进行查找,当找到某个内核的替换概率非零,且存在该内核的一个 Cache 块可以替换为 $Core(i,j)$ 的 Cache 数据时,就可以实现该 Victim Core 向 $Core(i,j)$ 的 Cache 块转移。图 3 给出了基于位置信息和替换概率的多核共享 Cache 划分策略示例,当 $Tile(i,j)$ 中的 $Core(i,j)$ 发生访问 Cache 失效并需要增加 Cache 时,通过查找 E-Table 发现当前瓦片内还有 $Core(i-1,j)$,$Core(i-1,j+1)$,$Core(i,j+1)$ 的 Cache 资源,且替换概率均非零,通过随机方式从中选择一个 Victim Core(假定为 $Core(i,j+1)$),然后使用替换策略(如 DIP[12])从 $Core(i,j+1)$ 在 $Tile(i,j)$ 中的 Cache 中选择 Victim Line;如果无法为失效的 Cache 组找到可替换的 Cache 块,则重新选择 Victim Core 和

Victim Line;如果 $Core(i-1,j)$,$Core(i-1,j+1)$,$Core(i,j+1)$在 $Tile(i,j)$中的 Cache 都无法满足需求,则从 $Tile(i,j)$周围的 8 个 Tile 中选择,直到找到可替换的 Cache 块。这种方法能够尽量保证 Cache 资源与相应内核之间在物理布局上的收敛,使每个内核所访问的 Cache 资源都位于当前瓦片及其周围相邻的瓦片中,实验数据也验证了这种局部性。相应地,正是由于这种局部性,也保证了 $Core(i,j)$在 $Tile(i\pm1,j\pm1)$所组成的瓦片组中一般都可找到可替换的 Cache,在较小的 Cache 划分探索空间内实现了较低的访问延迟。

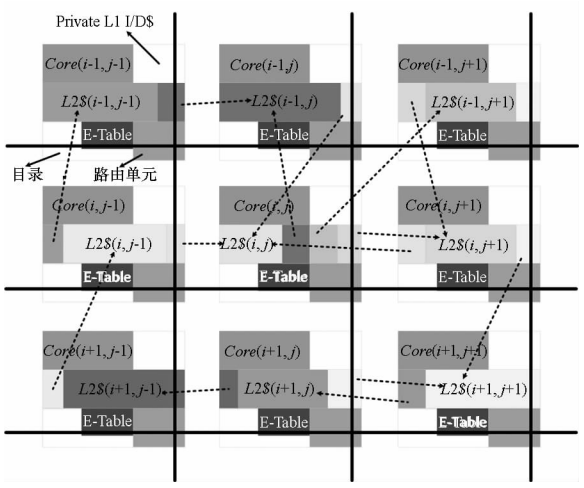


图 3 本文工作的硬件结构设计

Fig. 3 Hardware structure design of the proposal

本文提出的 Cache 划分策略仅需要增加少量硬件代价。所有 Cache 访问都需要在 Tag 信息中记录进行此次访问的内核 ID,以对 Cache 资源占用情况进行跟踪。由于其他相关工作中的 Cache 划分策略也都需要这一信息,因此这一硬件代价不属于本文工作的额外代价。替换概率的计算过程仅需要几个计数器对 C_i 和 M_i 进行统计,且这些计数器在传统 Cache 设计中通常是已有的[13]。另外,只要 N 和 W 是 2 的幂,并将图 1 中的 *TotalProfit* 舍入为 2 的幂,T_i 和 E_i 的计算就可通过简单的整数加法和移位操作实现。当核数为 4 时,图 1 中的算术操作个数为 20,当核数为 32 时,算术操作个数为 160。另外需要增加一部分存储空间用来保存 E-Table,每个 E-Table 表项保存某个内核的 ID 以及对应的替换概率。为降低硬件开销,本文使用整数来记录替换概率的近似值。实验表明,使用 6 位或 8 位整数记录替换概率已经可以达到与使用浮点数进行记录时几乎相同的性能。

3 实验结果

与 PriSM 相比,本文对局部性能进行了优化。为了验证这种优化对性能的影响,本文使用 M5 模拟器[14]对所提出的 Cache 管理策略进行模拟,并与 PriSM[10],Vantage[7],基于利用率的 Cache 划分(Utility-based Cache Partitioning, UCP)[6],提升/插入伪划分方法(Promotion/Insertion Pseudo Partitioning, PIPP)[9]等方法进行比较。模拟器的参数配置如表 1 所示,其中 L2 Cache 为末级 Cache,采用 LRU 替换策略,为 4 核和 8 核系统配置 16 路组相联 4MB 的 L2 Cache,为 16 核系统配置 32 路组相联 8MB 的 L2 Cache,为 32 核系统配置 64 路组相联 16MB 的 L2 Cache。为了更好地与已有相关工作进行比较,本文的实验方法尽量与这些相关工作保持一致[6-7,9-10],共采用了 71 个测试程序进行测试,包含 21 个 4 核程序、16 个 8 核程序、20 个 16 核程序、14 个 32 核程序。这些测试程序取自 SPEC CPU2000 和 SPEC CPU2006,首先将 SPEC CPU2000 和 SPEC CPU2006 中的程序按照类似于文献[15]中的方法分为 4 类:insensitive, cache-friendly, cache-fitting, thrashing/streaming;然后从每一类程序组中同时随机选取 1/2/4/8 个测试程序进行混合,形成前述的 71 个 4/8/16/32 核测试程序。每个测试程序执行 200M 个时钟周期。本文使用平均归一化周转时间(Average Normalized Turnaround Time, ANTT)[11]作为性能测试指标:

$$ANTT = \sum (IPC_i^{SP}/IPC_i^{MP})/n \qquad (2)$$

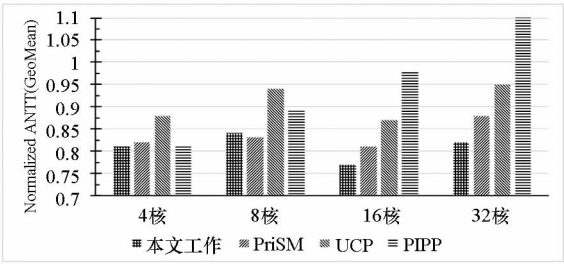
其中,IPC_i^{SP} 为 $Core_i$ 独占 Cache 资源时的 IPC,IPC_i^{MP} 为 $Core_i$ 与所有其他内核共享 Cache 资源时的 IPC。ANTT 数值越小,整体性能越高。

表 1 M5 模拟器参数配置

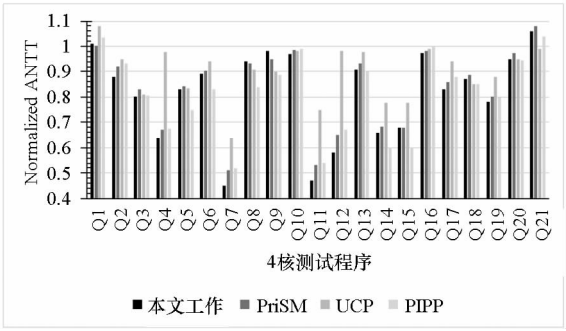
Tab. 1 Configuration of M5 simulator parameters

参数	配置
发射宽度	4
时钟频率	4 GHz
重排序缓存 ROB/发射队列/Load 队列/Store 队列	96/32/32/32
一级指令/数据 Cache	64KB,64B Cache Line,2 路组相联
二级共享 Cache(LLC)	4MB/8MB/16MB,64B Cache Line,16/32/64 路组相联
内核数	4/8/16/32

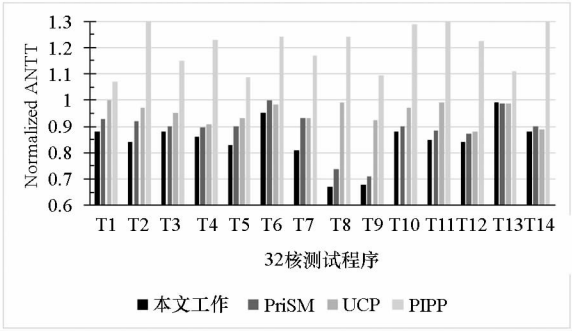
图 4(a)给出了本文与 PriSM,UCP,PIPP 的性能比较结果,采用归一化到 LRU 的 ANTT 作为性能指标。实验结果表明,对于 4/8/16/32 核 CPU,本文与 LRU 相比可以分别得到 19%,16%,21%,15%的性能提升;与 UCP,PIPP 相比在核数较多的情况下也得到了较明显的性能提升,与已有工作中性能较好的 PriSM 相比,本文在 16 核和 32 核的情况下也得到了 4.9%和 6.8%的性能提升。



(a) 平均结果
(a) Mean results



(b) 4 核详细结果
(b) Results of 4-core CMP

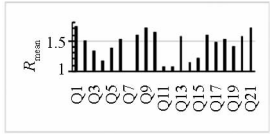


(c) 32 核详细结果
(c) Results of 32-core CMP

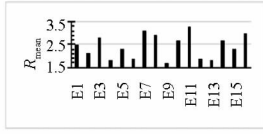
图 4　本文与 PriSM,UCP,PIPP 的性能比较
Fig. 4　Performance comparison among the proposed work and PriSM, UCP, PIPP

本文能够实现性能提升的主要原因是通过更细粒度的 Cache 资源划分使得不同内核之间的实际划分边界可以与理论上的最优划分边界尽可能地重合,从而充分发挥所有 Cache 资源的作用;同时,由于在 Cache 划分策略中融合了内核与 Cache 资源的相对位置信息,优化了内核对所拥有 Cache 资源的访问延迟,提高了 IPC,从而提高了系统性能。这种方法在内核数较多的情况下对系统性能提升的效果更为明显,这是由于本文中的 Cache 划分探索空间在内核越多的系统中所占的比重越小,Cache 资源距离相应内核的物理位置收敛性越好,延迟优化的效果也就越好。图 4(b)和图 4(c)分别给出了本文与 PriSM,UCP,PIPP 使用 4 核和 32 核测试程序进行测试时的详细性能比较结果。结果表明,本文可以在多数程序中获得优于其他相关工作的性能,有些测试程序性能提升明显(如 Q7,Q11,T5,T7,T8 等)。

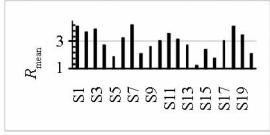
除了性能测试,本文也对 Victim Line 定位的迭代次数进行了分析。对 Victim Line 进行选择和定位的迭代次数会直接影响 Cache 划分的效果和效率。如果 Victim Line 可以在当前瓦片内定位成功,就可以较快地完成 Cache 块转移;反之,如果 Victim Line 在当前瓦片内的候选 Victim Core 中无法找到可替换的 Cache 块,则继续到其他瓦片中进行选择,这将影响 Cache 划分的效率,并导致较高的 Cache 访问延迟。本文通过为每个瓦片的 E-Table 增加计数器来统计每个 E-Table 表项被访问的次数,然后与相应的失效次数进行比较,使用 E-Table 访问次数与失效次数的比值 R 来衡量 Victim Line 定位迭代次数的多少。图 5 给出了所提方法在 4 核、8 核、16 核、32 核处理器上的迭代次数衡量参数 R 的结果,图中纵坐标为所有内核上多个时间间隔内 R 值的平均值。结果表明,随着核数的增加,Victim Line 定位迭代次数略有增加,虽然增加了少量复杂性,但是由于本文方法能够将 Cache 访问约束在距离内核较近的范围内,优化了访问延迟,因此可以获得优于其他



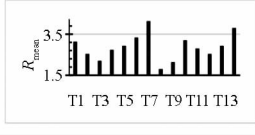
(a) 4 核
(a) 4-core



(b) 8 核
(b) 8-core



(c) 16 核
(c) 16-core



(d) 32 核
(d) 32-core

图 5　Victim Line 定位迭代次数分析
Fig. 5　Iteration analysis of locating victim lines

相关工作的整体性能,具有较大的实用性。

4 结论

本文提出了一种基于位置信息与替换概率的多核共享 Cache 管理机制,实现了细粒度、低延迟、低代价的末级共享 Cache 划分,在综合性能上优于已有研究成果。通过将系统的性能目标转换为每个内核当前所占用的 Cache 资源的替换概率,并基于替换概率分布以 Cache 块为粒度进行实时替换,实现了 Cache 资源在不同内核之间的细粒度划分。另外,通过将位置信息和替换概率相结合,保证了 Cache 资源与相应内核之间的物理布局上的收敛,有效降低了访问延迟。同时,所提方法只需增加极少的硬件代价。通过实验验证了所提出的方法与其他已有研究成果相比具有更高的整体性能。

参考文献(References)

[1] Zhang L D, Liu Y, Wang R, et al. Light-weight dynamic partitioning for last-level Cache of multicore processor on real system [J]. Journal of Supercomputing, 2014, 69(2): 547 – 560.

[2] 刘胜, 陈海燕, 葛磊磊, 等. 面向访问模式的多核末级 Cache 优化方法 [J]. 国防科技大学学报, 2015, 37(2): 79 – 85.
LIU Sheng, CHEN Haiyan, GE Leilei, et al. Optimization method for multi-core last level Cache considering the memory access modes [J]. Journal of National University of Defense Technology, 2015, 37(2): 79 – 85. (in Chinese)

[3] El-Moursy A A, Sibai F N. V-set Cache: an efficient adaptive shared Cache for multi-core processors [J]. Journal of Circuits, Systems and Computers, 2014, 23(7): 815 – 822.

[4] Chen G, Hu B, Huang K, et al. Shared L2 Cache management in multicore real-time system [C]//Proceedings of IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines(FCCM), 2014: 170.

[5] Tang Y X, Wu J M, Chen G L, et al. A utility based Cache optimization mechanism for multi-thread workloads [J]. Journal of Computer Research and Development, 2013, 50(1): 170 – 180.

[6] Qureshi M K, Patt Y N. Utility-based Cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared Caches [C]//Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro-architecture, 2006: 423 – 432.

[7] Sanchez D, Kozyrakis C. Vantage: scalable and efficient fine-grain Cache partitioning [C]//Proceedings of the 38th Annual International Symposium on Computer Architecture, 2011: 57 – 68.

[8] Zhu D, Chen L Z, Yue S Y, et al. Balancing on-chip network latency in multi-application mapping for chip-multiprocessors [C]// Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium, 2014: 872 – 881.

[9] Xie Y J, Loh G H. PIPP: promotion/insertion pseudo partitioning of multi-core shared Caches [C]//Proceedings of the 36th Annual International Symposium on Computer Architecture, 2009: 174 – 183.

[10] Manikantan R, Rajan K, Govindarajan R. Probabilistic shared cache management (PriSM) [C]//Proceedings of the 39th Annual International Symposium on Computer Architecture, 2012: 428 – 439.

[11] Eyerman S, Eeckhout L. System-level performance metrics for multi-program workloads [J]. IEEE Micro, 2008, 28(3): 42 – 53.

[12] Qureshi M K, Jaleel A, Patt Y N, et al. Adaptive insertion policies for high performance caching [C]// Proceedings of the 34th Annual International Symposium on Computer Architecture, 2007: 381 – 391.

[13] Eyerman S, Eeckhout L, Karkhanis T, et al. A performance counter architecture for computing accurate CPI components [C]// Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, 2006: 175 – 184.

[14] Binkert N L, Dreslinski R G, Hsu L R, et al. The M5 simulator: modeling networked systems [J]. IEEE Micro, 2006, 26(4): 52 – 60.

[15] Jaleel A, Hasenplaugh W, Qureshi M, et al. Adaptive insertion policies for managing shared Caches [C]// Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008: 208 – 219.