doi:10.11887/j.cn.201502016 http://journal.nudt.edu.cn

面向访问模式的多核末级 Cache 优化方法*

刘 胜,陈海燕,葛磊磊,刘 仲
(国防科技大学 计算机学院,湖南 长沙 410073)

摘 要:多核处理器架构已经成为当前处理器的主流趋势,应用程序中访问模式的多样性给多核处理器的末级 Cache 带来了许多挑战。提出了访问模式的多核末级 Cache 优化方法,它包含"可配置的共享私有 Cache 划分"、"可配置的旁路 Cache 策略"和"优先权替换策略"三个协同递进的层次。通过使用该方法,程序员能够灵活地改变末级 Cache 执行行为,从而高效地适应应用程序访问模式的变化。实验结果表明,提出的方法能够显著降低末级 Cache 的缺失率,进而提高系统的整体性能。

关键词:多核处理器;末级 Cache;访问模式;共享私有 Cache 划分;旁路 Cache;优先权替换
中图分类号:TP302 **文献标志码**:A **文章编号**:1001-2486(2015)02-079-07

Optimization method for multi-core last level Cache considering the memory access modes

LIU Sheng, *CHEN Haiyan*, *GE Leilei*, *LIU Zhong*
(College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: Multi-core architectures have been broadly utilized in current processors. Meanwhile, the diversity of memory access modes in applications brings challenges to the last level Cache in multi-core processors. An optimization method for the last level Cache based on the access modes is proposed. This method includes three coordinated and progressive levels: the configurable share/private Cache partition, the configurable bypass Cache policy, and the priority replace policy. Using this method, programmers can neatly alter the behavior of the last level Cache to effectively adapt the variety of memory access modes in applications. Experiment results show that the proposed method can observably decrease the miss rate of the last level Cache and increase the system performance of the processor.

Key words: multi-core; last level Cache; access mode; share/private Cache partition; bypass Cache; priority replacement

当前处理器架构已经由"单核"时代进入"多核"时代[1]。现代多核处理器设计中,普遍采用多级 Cache 来缓解"存储墙"问题。片上末级 Cache(Last Level Cache,LLC)一般支持多个核共享地访问,是多核处理器存储层次的关键组成部分。如何有效地管理和利用 LLC 将会对于整个系统的性能产生重要影响。

随着应用需求的不断扩展和变化,应用程序中访问模式的多样性给多核处理器的 LLC 带来了许多挑战。文献[2]将应用程序中的访存模式分为四种,即友好访问模式、流式访问模式、颠簸访问模式及混合访问模式。其中友好访问模式是指短时间内会重复访问某一段数据,这种模式具有良好的时空局部性。流式访问模式是指某段数据块只会被访问一次,这种模式将会引起强制性缺失并且命中率较低。颠簸访问模式是指周期性地访问某段长度的数据块,但是其长度超过 Cache 的容纳范围,导致数据块未被访问就被替换出 Cache。混合访问模式是上述三种模式综合混合的结果。在多核环境下由于不同的核对共享/私有数据的同时访问,上述访问模式变得愈发复杂多样,从而导致了 LLC 的性能得不到有效发挥,迫切需要更高效 LLC 的管理和控制策略。

刘胜等以一款自主研发的高性能多核数字信号处理器(Digital Signal Processor,DSP)Matrix-M 为背景,提出了访问模式的多核 LLC 优化方法。

1 访问模式的 LLC 优化方法

跟经典的 Cache 机制一样,LLC 的性能依然由命中率(或缺失率)、命中开销和缺失开销所决定。该方法主要通过降低缺失率来提高性能,所有的减少命中/缺失开销的优化方法和该方法是

* **收稿日期**:2014-10-22
基金项目:国家自然科学基金资助项目(61133007,61402500)
作者简介:刘胜(1984—),男,河南南阳人,助理研究员,博士,E-mail:liusheng83@nudt.edu.cn

正交的。

基于访问模式的 LLC 优化方法的三个子策略“可配置的共享私有 Cache 划分”、“可配置的旁路 Cache 策略”和“优先权替换策略”是协同互补的关系。首先,可配置的共享私有 Cache 划分策略从整体上将 LLC 空间划分为共享空间与私有空间,提高整个 Cache 的空间利用率;其次,对流式访问模式以及其他较低重用性的访存行为,采用“可配置的旁路 Cache 策略”可以使其请求不缓存在 Cache 中,减少与正常的进 Cache 请求间的干扰;最后,在传统替换算法的基础上融入优先权,通过“优先权替换策略”可将高重用性数据块设置优先权使其驻留在 Cache 中,从而有效地减少颠簸访存模式带来的失效开销。

1.1 可配置的共享私有 Cache 划分

私有方式和共享方式是多核 LLC 常见的两种设计方式。私有方式中,LLC 的每个子体只能被就近的核访问,访问 LLC 的网络延时很小,但在访存负载不均衡时可能会造成某一个子体繁忙而另一个子体空闲的情况,从而造成 LLC 的利用率较低。而共享 Cache 方式中,每个核都可以访问所有的 LLC 子体,可以充分地利用整个 Cache 空间,但是互连线延迟的增加也将导致访问延迟加大。因此,高效的 LLC 控制策略要充分地结合两者的优点,合理地划分共享私有空间,既要减少访问 Cache 网络延迟,又要充分地利用共享资源。

在共享 Cache 的基础上,通过提供给程序员控制寄存器改变双倍速率同步动态随机存储器(Double Data Rate,DDR)空间的高低位地址交叉映射方式,从而实现可配置的共享私有 Cache 划分。如图 1 所示,外存空间被分界线划分成两个部分,上半部分采用高位地址交叉编址,下半部分采用低位地址交叉编址。其中,分界线是通过编址模式寄存器的配置来确定的,能够上下移动。考虑到实际应用需求以及实现复杂度,整个 DDR 空间以 2 的幂分成 8 种高低位地址交叉区间比例。假设整个外存空间大小是 S,那么对应的高位地址交叉编址区间的大小分别为 S,$S/2$,$S/4$,$S/8$,$S/16$,$S/32$,$S/64$,$S/128$,剩余区间采用低位地址交叉编址。

程序员可以通过设定外存空间的编址方式来实现共享 Cache 与私有 Cache 空间的划分。其中,私有 Cache 空间可以通过高位地址交叉区间映射到 LLC 来实现,即程序员将不同核的私有数据分别放置在不同的 DDR 上,DSP 核就可以高效地通过本地 LLC 就近访问这一区间,不仅可以减少互连网络的负担,而且可以减少访问延迟。共享 Cache 空间则通过低位地址交叉编址方式将共享数据段分散放置在不同的 DDR,这样的设置可以使 DDR 的访问比较均匀,同时能充分发挥多个 LLC 子体和多个 DDR 的带宽。

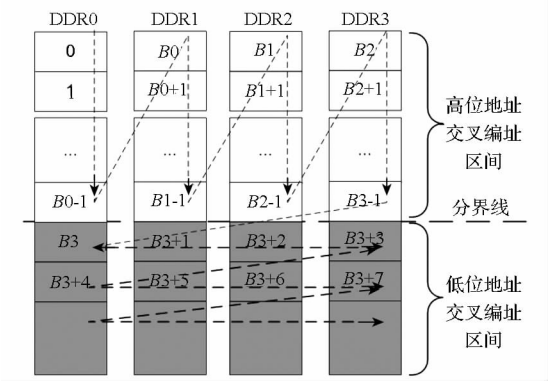


图 1 DDR 空间的高低地址交叉编址方式

Fig. 1 High-low interleaved mode of DDR space

1.2 可配置的旁路 Cache 策略

上述可配置的共享私有 Cache 的划分策略提高了 LLC 的整体空间利用率。然而应用中普遍存在的零重用数据块访问(如只会对数据块访问一次的流式访问模式),将会严重破坏高重用数据块的访存行为,造成整个 LLC 性能的极大损失。为了尽可能地削弱这些低重用数据块对整个 LLC 的影响,通过可配置的旁路 Cache 策略使之不缓存在 Cache 中,并且在实际处理中,驻留 Cache 请求与旁路 Cache 请求隔离,前台的可 Cache 请求不会被后台旁路 Cache 请求干扰。

具体实现如图 2 所示:每一个子 LLC 提供一组旁路寄存器,包括旁路使能寄存器、旁路起始地址寄存器与旁路字偏移寄存器,程序员根据需求通过设置这三个寄存器来配置其旁路地址区间。每个请求源根据请求地址和旁路寄存器的内容判断请求是否旁路 LLC,对于旁路 Cache 的请求直

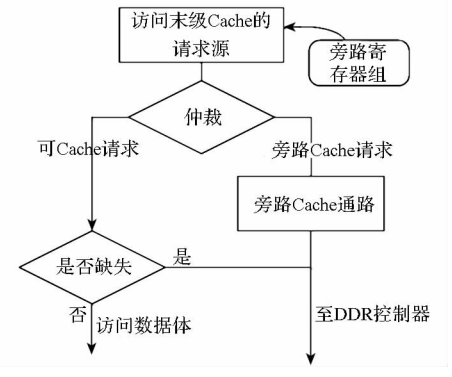


图 2 可配置的旁路 Cache 的实现流程

Fig. 2 Flow of the configurable bypass Cache

接访问 DDR,可 Cache 的请求则走正常的 Cache 处理流水线判断是否缺失,如果缺失则访问 DDR 控制器,否则直接访问数据体。由于外存 DDR 的访问速度比较慢,而旁路 Cache 请求直接访问 DDR,其访问速度也较慢,如果不进行请求隔离处理,则可 Cache 请求会被旁路 Cache 请求阻塞;本设计通过从源端将请求分离成两条通路,让可 Cache 请求与旁路 Cache 请求可以并行运行,互不干扰。

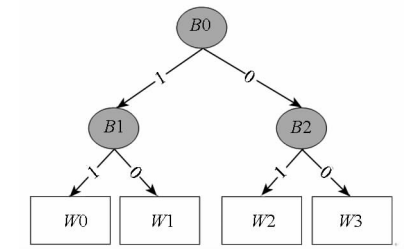
1.3　优先权替换策略

除了零重用数据块外,应用中多个进程之间相互干扰引起 LLC 访问颠簸所造成的低重用数据块,也会影响 LLC 的执行效率。考虑到尽可能地将高重用数据块缓存在 LLC 中,提出了一种融入优先权的替换策略,通过将高重用数据块的替换优先级设定为高,使其不能被低驻留优先权的 Cache 行替换,从而在一段时间驻留在 LLC 中。

实现优先权替换策略包含提供一组由用户配置的控制寄存器组和硬件支持的融入优先权的树形最近最少访问(Least Recently Used,LRU)机制两个方面。控制寄存器组包含优先权区间的设置寄存器组和优先权清除寄存器。优先权区间的设置可以通过配置优先权使能寄存器、起始地址寄存器与字偏移寄存器来实现。在一段程序执行完成之后,又可以通过优先权清除寄存器来清除优先权。

接下来以四路组相连为例描述融入优先权的树形 LRU 机制的硬件实现。更多路数的融入优先权的树形 LRU 机制与之类似,不再详述。树形 LRU 替换算法的工作原理是利用一个二叉树结构存储历史访问信息(*B*0,*B*1 和 *B*2),保护最近的访问块不会被替换出 Cache,如图 3 所示。其中四路相连的组分别用 *W*0 ~ *W*3 表示,用 *B*0 ~ *B*2 来记录历史访问信息。该算法包含更新规则和替换规则两个方面。当 Cache 行命中或被分配之后利用更新规则(如图 3(b)所示)对其历史访问信息进行更新,例如,当访问第 0 路时,更新 *B*0,*B*1 的值为 00,使之成为最不可能替换出去的行。当需要 Cache 替换时则利用替换规则(如图 3(c)所示)选出一路被替换出 Cache。如当 *B*0,*B*1 的值为 11 时,根据替换规则,第 0 路被替换出 Cache。

如图 4 所示,融入优先权的树形 LRU 替换算法需要在一组 Cache 中的每一路都添加一个优先级位,用 *P*0,*P*1,*P*2,*P*3 表示。当 Cache 行命中或分配时,融入优先权的树形 LRU 算法按照表 1 到



(a) 二叉树结构

(a) Binary tree structure

访问的路	*B*0	*B*1	*B*2
*W*0	0	0	不变
*W*1	0	1	不变
*W*2	1	不变	0
*W*3	1	不变	1

(b) 更新规则

(b) Update rule

替换的路	*B*0	*B*1	*B*2
*W*0	1	1	0/1
*W*1	1	0	0/1
*W*2	0	0/1	1
*W*3	0	0/1	0

(c) 替换规则

(c) Replacement rule

图 3　树形 LRU 替换算法

Fig. 3　Tree style LRU algorithm

表 3 的规则更改 *B*0,*B*1,*B*2 的值。其中 $L = P0\&P1$,*L* 为 1 代表左侧一组优先级全部为高,*L* 为 0 代表其优先级不全为高;同理 $R = P2\&P3$,*R* 为 1 代表右侧一组优先级全部为高,*R* 为 0 代表其优先级不全为高;如假设当前访问的路数为 *W*0 ~ *W*1 中的一个且 $L = 1, R = 0$,则 *B*0 被更新为 0,代表下次被替换时右侧一组被替换出 Cache;而如果 $L = 0, R = 1$,则 *B*0 被更新为 1,代表下次被替换选择左侧一组。

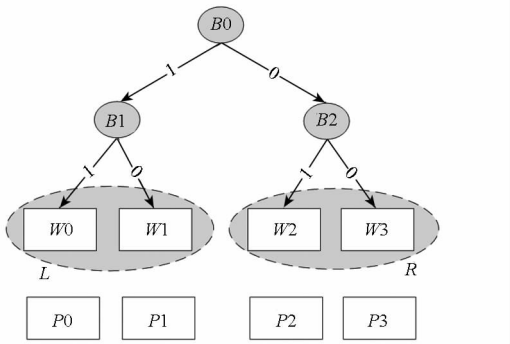


图 4　融入优先权的树形 LRU 替换算法

Fig. 4　Tree style LRU algorithm with the priority

融入优先权的替换机制遵循以下规则:(1)优先级低的一路先被替换;(2)同种优先级时,按照树形 LRU 算法进行替换,其中优先级为 1 的 Cache 行只能被其他优先级为 1 的 Cache 行替换。融入优先权的树形 LRU 机制是在树形 LRU 算法的基础上实现的,两者替换规则相同。前者的更新规则有所更改。

表 1 *B*0 的更新规则

Tab. 1 Update rule of *B*0

当前访问路				更新 *B*0
*W*0 ~ *W*1		*W*2 ~ *W*3		
L	*R*	*L*	*R*	
0	1	00,01,11		1
00, 10, 11		1	0	0

表 2 *B*1 的更新规则

Tab. 2 Update rule of *B*1

当前访问的路						更新 *B*1
*W*0		*W*1		*W*2, *W*3		
*P*0	*P*1	*P*0	*P*1	*P*0	*P*1	
0	1	00, 01, 11		0	1	1
00, 10, 11		1	0	1	0	0
—	—	—	—	00,11		不变

表 3 *B*2 的更新规则

Tab. 3 Update rule of *B*2

当前访问的路						更新 *B*2
*W*2		*W*3		*W*0, *W*1		
*P*2	*P*3	*P*2	*P*3	*P*2	*P*3	
0	1	00, 01, 11		0	1	1
00,10,11		1	0	1	0	0
—	—	—	—	00,11		不变

2 开销与性能评测

该技术已经在本单位在研的一款多核 DSP Matrix-X 中的 LLC 中实现。Matrix-X 的结构如图 5 所示,由八个 DSP 超节点组成,超节点通过用片上互连网络(Network on Chip,NoC)进行互连和通信,LLC 分为八个子体,采用分布式共享结构。每一个超节点可以通过节点访问控制器

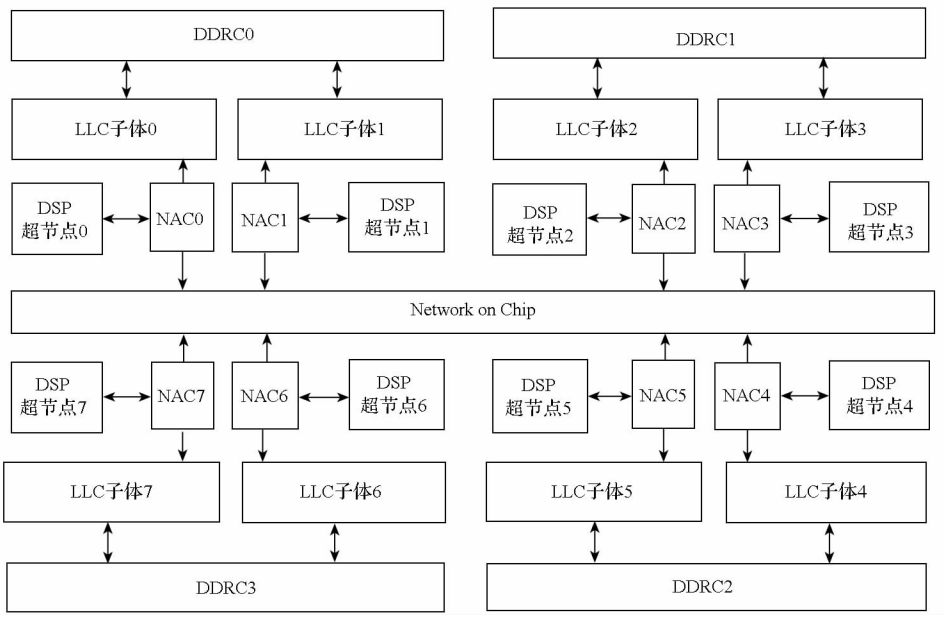


图 5 Matrix-X 的整体结构图

Fig. 5 Block diagram of Matrix-X

(Node Access Controller,NAC)和 NoC 访问任一个末级 Cache 子体。每两个 LLC 子体和一个 DDR3 控制器(DDR3 Controller,DDRC)相连。LLC 的总容量为 4MB,采用八路组相连机制,Cache 行的宽度为 1024bits,采用读/写分配机制和写回法,支持八深度的 Miss-on-Miss 处理。

2.1 提出方法的硬件实现开销

该访问模式的 LLC 优化方法的三个层次中,可配置的共享私有 Cache 划分所带来的硬件开销非常小,仅仅增加了两个配置寄存器以及简单的选择逻辑;可配置的旁路 Cache 策略除了引入相关旁路配置寄存器外,在请求的源端需要增加旁路仲裁逻辑,还需要额外的旁路 Cache 缓冲。相比于传统策略,优先权替换策略需要为每一路组相联增加一位优先权位记录分配数据块的优先级,对于 N 组 M 路的组相联 Cache 结构,需要额外的 $M \times N$ bits 的触发器。此外还包括优先权配置寄存器和分配 Cache 行时判断优先级的逻辑等。

在某厂家 45nm 工艺库下,使用 Cadence™ 公司的 RTL Complier 工具在典型情况按照频率

1GHz 对 LLC 子体进行了综合,结果见表 4。其中可配置的旁路 Cache 策略引入的面积占了控制逻辑面积的 12.68%,主要原因为 Matrix-X 采用无缓冲 NoC 机制,LLC 需要专门为不可 Cache 请求设置专门的输入缓冲,如果 NoC 结构不采用无缓冲结构,该面积可以减少;优先权替换策略引入的面积占了控制逻辑面积的 4.57%,主要由记录每一个 Cache 行的优先权的触发器导致的。可配置共享私有 Cache 划分引入的面积较小。

表 4　提出方法的面积开销
Tab. 4　Area cost of the proposed policy

	面积 (μm^2)	占控制逻辑的比
LLC 子体	5 267 899	—
数据体 + Tag 体	2 508 136	—
控制逻辑	2 759 763	100.00%
可配置共享私有 Cache 划分	1634	0.06%
可配置的旁路 Cache 策略	349 914	12.68%
优先权替换策略	126 142	4.57%

2.2　性能评测

选取下三角矩阵与矩阵的乘积(TRiangular Matrix Multiplication, TRMM)、通用矩阵乘积(Generalized Matrix Multiplication, GEMM)、转置下三角阵与矩阵的乘积(Transpose TRiangular Matrix Multiplicaion, TTRMM)作为评测激励。这三种 Kernel 均是 LINPACK 算法中完成矩阵更新操作的算法,占据了 LINPACK 运算量的 80%。在评测时,这三种 Kernel 的子块规模均设置为 384 的整数倍,8 个超节点一起参与运算,采用双缓冲的策略同时进行运算和数据搬移。此外,还选取了快速傅里叶变换(Fast Fourier Transformation, FFT)算法作为另外一类评测激励,同时计算 8 个不相关的 1M 点 FFT 运算(每一个超节点分别对应一个),采用两维转置算法将其分成多个 1K 点 FFT 运算,采用双缓冲策略同时进行运算和数据搬移。

由于三种子优化策略是相互协同的关系,测试程序不仅要进行单个优化策略的性能分析,而且要分析结合三种优化策略所达到的最优性能提升。因此每一个程序分成五组进行测试:A 组,未做任何优化的原始程序;B 组,添加合理划分共享私有 Cache 空间的优化;C 组,在合理划分共享私有 Cache 空间的优化下对低重用数据设置旁路 Cache 的优化;D 组,在合理划分共享私有 Cache 空间的优化下,对高重用数据设置高优先权的优化;E 组,结合三种策略进行全面优化。从缺失率、程序运行时间两个方面对提出的方法进行评估。

首先,统计了可配置共享私有 Cache 划分对程序性能的影响。从图 6 中可以得出,合适的共享私有 Cache 划分设置对程序的执行效率影响非常大,如对于 TRMM, GEMM 和 TTRMM 算法,如果将共享私有划分设置为高位地址交叉,应用程序在同一段时间内多个核将会集中访问一个 LLC 子体和一个 DDR,即 LLC 的带宽和 DDR 的带宽分别只有峰值的 1/8 和 1/4,因而和采用合适的共享私有 Cache 划分设置相比,程序的节拍数将会变为 5.87 ~ 8.36倍,缺失率会变为 2.03 ~ 2.42 倍。对于 FFT 程序,如果将共享私有划分设置为低位地址交叉,FFT 本来只需要访问本地 LLC 子体的请求大部分将会变成网络请求访问远程 LLC 子体,等价于访问 LLC 的延时增加了,因而和采用合适的共享私有 Cache 划分设置相比,程序的节拍数将会变为 4.20 倍,缺失率会变为 1.77 倍。

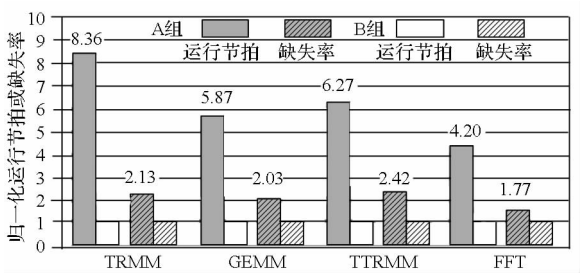


图 6　可配置的共享私有 Cache 划分对性能的提升
Fig. 6　Improvement to the system of the configurable shared/private partition policy

其次,在设置合适的共享/私有 Cache 划分的基础上评测了可配置的旁路 Cache 策略和优先权替换策略对程序性能的影响。如图 7 和图 8 所示,相比仅设置合适的共享/私有 Cache 划分,采用可配置的旁路 Cache 策略能够使应用 TRMM, GEMM 和 TTRMM 的运行节拍和缺失率得到显著

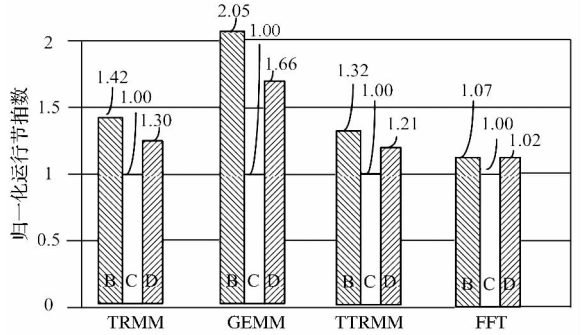


图 7　B/C/D 组的归一化执行节拍数
Fig. 7　Normalized cycles of B/C/D groups

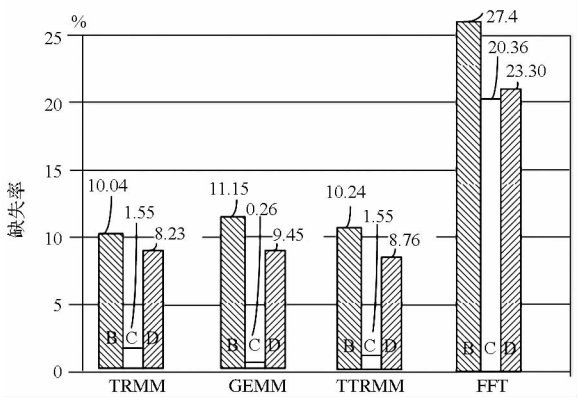


图8 B/C/D组的缺失率

Fig. 8 Miss rates of B/C/D groups

降低;对于FFT,运行节拍数和缺失率虽然改善相对小一些,但也有明显改善。相比仅设置合适的共享/私有Cache划分,采用高优先权的优化策略能够使应用程序的运行节拍数和缺失率得到一定程度的降低。

再次,当三种子策略结合使用时(见表5),对于选取的应用程序,运行节拍数能够降低到未优化版本的1/12.1 ~ 1/4.7。对于TRMM,GEMM和TTRMM,采用提出的三种策略,可以基本上保证其所有的请求在LLC中命中,此时运行效率是最高的;对于FFT算法,能够将缺失率从48.50%降低到18.87%。

表5 三种子策略结合使用对性能的提升

Tab. 5 Increase to the system of three sub policies

	运行节拍数(万拍)		缺失率(%)	
	A组	E组	A组	E组
TRMM	571.5	48.0	24.75	1.55
GEMM	2 822.7	234.1	22.62	0.26
TTRMM	647.3	78.0	24.75	1.55
FFT	774.7	162.2	48.50	18.87

此外,对比图8和表5,能够发现TRMM,GEMM和TTRMM的缺失率在C组和E组中相同。这是由于TRMM,GEMM和TTRMM中计算访存比较大,在采用子策略1和子策略2优化之后已经能够保证绝大部分访问在LLC中命中(即只存在强制性缺失、不存在容量缺失和冲突缺失),因而在此基础上子策略3的优化效果不明显。这并不意味着子策略3不重要,对比图7和图8可以得出,对于TRMM,GEMM和TTRMM,在子策略1的基础上使用子策略3能够使程序的执行节拍数降低(相对值分别为从1.42到1.30、从2.05到1.66和从1.32到1.21)和缺失率得到降低(分别从10.04%到8.23%、从11.15%到9.45%和从10.24%到8.76%)。而对于FFT程序来说存储带宽的瓶颈效应更加显著,即在采用子策略1和子策略2后,LLC依然存在容量缺失和冲突缺失现象,因而再采用子策略3依然能够对程序产生较明显的优化效果。

3 相关研究

文献[7]提出了一种自适应的共享/私有Cache划分方法,可以根据负载特征动态调整私有和共享部分的比例,但其硬件开销比较大。文献[8,9]提出了一种合作式的Cache,在LLC私有Cache的基础上允许L1D间直接传递数据,以实现核间通信。刘胜等提出的通过设置高低位交叉区间来实现“可配置的共享私有Cache划分”的方法与之前的方案均不相同,并且软硬件开销均更加合理。

传统的LLC优化策略,如采用调度策略优化[3],页着色划分[4]等减少LLC中并发进程间的冲突,过分依赖操作系统。文献[5]将替换策略融合允许程序在不同的替换策略切换,以获得较大的性能。该方法硬件开销较大。刘胜等提出的“优先权替换策略”与已有的替换机制均不相同,在树形伪LRU机制上进行了扩展,实际上也可以扩展到其他替换机制上去。

文献[6]提出两级重用预测器用于指导旁路判定数据块是否不会再被访问,若是则采用旁路技术,避免该数据块访问高速缓存。可配置的旁路Cache策略借鉴了该方法,并将设置权限交给了用户以减少预测器的软硬件开销。

4 结论

本文提出了一种基于访问模式的多核LLC优化方法,该方法在一款自主研发的高性能多核DSP Matrix-M中进行了实现和模拟评估。评估结果显示该方法能够灵活地改变LLC执行行为和显著提升应用程序的性能。下一步将采用更多的应用对提出的机制进行评测和改进。

参考文献(References)

[1] Blake G, Dreslinski R G, Mudge T. A survey of multicore processors[J]. IEEE Signal Processing Magazine, 2009, 26(6): 26 - 37.

[2] Jaleel A, Theobald K B, Steely Jr S C, et al. High performance cache replacement using re-reference interval prediction[C]//Proceedings of Computer Architecture News, 2010, 38(3): 60 - 71.

[3] Fedorova A. Operating system scheduling for chip

multithreaded processors[D]. USA: University of Harvard, 2006.

[4] Soares L, Tam D, Stumm M. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer[C]//Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, 2008: 258 - 269.

[5] Subramanian R, Smaragdakis Y, Loh G H. Adaptive caches: effective shaping of cache behavior to workloads[C]// Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006: 385 - 396.

[6] Xiang L X, Chen T Z, Shi Q, et al. Less reused filter: improving L2 cache performance via filtering less reused lines[C]//Proceedings of the 23rd International Conference on Supercomputing, 2009: 68 - 79.

[7] Dybdahl H, Stenstrom P. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors[C]// Proceedings of IEEE 13th International Symposium on High Performance Computer Architecture, 2007: 2 - 12.

[8] Chang J, Sohi G S. Cooperative caching for chip multiprocessors[C]//Proceedings of the 33nd Annual International Symposium on Computer Architecture, 2006: 264 - 276.

[9] Herrero E, González J, Canal R. Distributed cooperative caching[C]//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008: 134 - 143.