片上多处理器共享 Cache 的访存时间最优划分方法

李浩，谢伦国
（国防科技大学 计算机学院，湖南 长沙 410073）

摘 要：提出的访存时间最优 Cache 划分(OMTP, optimal memory time Cache partitioning)方法通过特征获取部件来获取不同应用程序的平均失效开销和Cache命中的路分布情况，以此作为划分依据来给竞争程序分配合适的Cache空间，达到优化程序整体执行性能的目的。实验结果表明，OMTP 方法相比基于利用率的 Cache 划分(UCP)方法吞吐率平均提高 3.1%，加权加速比平均提高 1.3%，整体性能更优。

关键词：片上多处理器；共享 Cache；Cache 划分；访存时间最优划分
中图分类号：TP302 **文献标识码**：B **文章编号**：1000-436X(2012)04-0136-07

Optimal memory time Cache partitioning in chip-multiprocessors

LI Hao, XIE Lun-guo
(School of Computer Science, National University of Defense Technology, Changsha 410073,China)

Abstract: Optimal memory time Cache partitioning(OMTP) was proposed. The OMTP can get the average access invalidation overheads of different application and Cache line distributation about Cache hit through the characteristic obtain unit. According to which the OMTP can allocate proper Cache capacity to competitive process, so the overall performance of the program can achieve optimizied.The experiment results showed that OMTP had higher miss rate compared with utility-based Cache partitioning(UCP), but it had better IPC throughput and weighted speedup, OMTP improved throughput by average 3.1% and weighted speedup by average 1.3% over UCP.

Key words: chip-multiprocessor; shared Cache; Cache partitioning; optimal memory time Cache partitioning

1 引言

为了保持处理器性能的持续增长，多核（multicore）处理器或片上多处理器（CMP, chip multiprocessor）已成为现代处理器发展的主流。片上处理器核数目的不断增加，以及芯片管脚数目的有限使得处理器核片外访存开销越来越大，这给本来已经很严重的存储墙问题带来了更大的挑战。人们不得不在芯片上放置更多层次更大容量的片上存储系统，以减少处理器对片外存储器的访问需求。片上存储系统性能的好坏已经成为决定 CMP 系统性能的关键因素。为了提高片上存储系统的性能，很多设计都采用大容量的末级共享 Cache。本文的研究目标旨在有效管理 CMP 末级共享 Cache，优化 CMP 上运行多个应用程序时的整体性能。

当运行在 CMP 上的多个应用程序竞争同一个共享 Cache 空间时，传统的 LRU 替换策略会显式的按照请求的频率给那些请求频率高的应用分配更多的 Cache 空间。但是，给一个请求频率高的应用分配更多的 Cache 空间不一定就能给它带来相应的性能提升。例如，当一个流应用的工作集大于 Cache 总容积时，由于它访问过的 Cache 块重用性较低，即使给它分配更多的 Cache 空间，也无法给性能带来多大的提升。为了提高多程序整体的访存性能，本文希望能够把 Cache 空间分配给那些能通过得到更多 Cache 空间获得较大性能提升的应用，

收稿日期：2010-12-29；修回日期：2011-11-22

以提高多个竞争程序总的执行性能。

目前大部分 Cache 划分方法，无论是静态最优划分方法[1]，还是动态划分(DP, dynamic partitioning)[2]方法，还是基于利用率的 Cache 划分（UCP, utility-based Cache partitioning）[3]方法，都是以降低多个程序总的 Cache 失效率为优化目标。在很多情况下，减少 Cache 失效次数确实能够带来性能的提升，但是在某些情况下，失效次数的减少并不意味着程序执行性能的提升[4]，这是因为：1）应用程序的访存特性导致它们对 Cache 失效的敏感度各不相同，相对而言，计算密集型应用对 Cache 失效的敏感度比访存密集型应用要小很多；2）非阻塞 Cache[5]、乱序执行、以及提前执行[6]等技术的出现，使得有些 Cache 失效引发的片外访存能够重叠执行，称之为存储级并行（MLP, memory level parallelism）[7]，这种存储级并行对程序执行时间的影响是不同的。由于每个 Cache 失效带来的实际性能开销各不相同，因此不同应用程序之间，Cache 失效率的高低变化并不能直接反映程序执行性能的高低变化，划分后总的 Cache 失效率最优并不能代表总的程序执行时间最少。以往的大多数 Cache 划分方法都没有考虑 MLP 问题，Zhou 等人[8]提出了一种考虑了 MLP 问题的 Cache 划分算法，但是他们主要针对的是 Cache 的公平性。本文提出一种访存时间最优的 Cache 划分(OMTP, optimal memory time Cache partitioning)方法，通过独立的硬件获取每个划分区间内每个应用程序在占用不同大小的 Cache 空间时失效率的变化情况以及这些失效带来的平均访存时间，将两者用于指导 Cache 划分算法的执行，使多个竞争程序总的执行时间最少。模拟实验结果表明，OMTP 相比 UCP 吞吐率平均提高了 3.1%，加权加速比平均提高了 1.3%，取得了较好的性能提升。

2　访存时间最优 Cache 划分

目前许多多处理器系统都采用共享 L2 Cache 来减少片外访存，当有多个应用程序同时运行在这样一个系统上时，有可能会出现某些应用程序占用很多 Cache 空间却无法得到相应的性能提升，而某些只需要再获得很少 Cache 空间就能大幅提升性能的应用程序则不能获得所需 Cache 空间的情形。Cache 划分（partitioning）的目的就是给每一个应用程序分配适当的 L2 Cache 空间，以提高多个应用程序总的执行性能。

假设有 N 个应用程序共享同一个 Cache 空间，C_i 代表分配给应用程序 i 的 Cache 块数，C 为总的 Cache 块数，$C = C_1 + C_2 + \cdots + C_N$。则 Cache 划分可以用集合 $\{C_1, C_2, \cdots, C_N\}$ 来定义。Suh 等人[2]提出的最优 Cache 划分是使总失效次数 $M_{\text{total}} = \sum_{i=1}^{N} M_i(C_i)$（$M_i(C_i)$ 代表应用程序 i 在占用 C_i 个 Cache 块位置时的失效次数）最小的划分。把这种最优划分称作失效率最优划分。在以往的一些研究中，无论是动态划分(DP, dynamic partitioning)[1]方法，还是基于利用率的 Cache 划分（UCP）[2]方法，都是追求失效率最优的划分方法。

但是，由于每个 Cache 失效所造成的开销不同，并且各个应用程序内在的访存特征也各不相同，使得失效率的多少并不能代表 Cache 实际性能的高低，失效率最优并不能代表访存时间最优。真正的性能最优划分应该是多个应用程序总的执行时间 $T_{\text{total}} = \sum_{i=1}^{N} T_i(C_i)$（$T_i(C_i)$ 表示应用程序 i 占用 C_i 个 Cache 块位置时的执行时间）最小的集合 $\{C_1, C_2, \cdots, C_N\}$。

应用程序的执行时间可以用式(1)计算[7]：

$$T = T_{\text{CPU}} + T_{\text{Mem\_stall}} \tag{1}$$

式中 T_{CPU} 代表 CPU 运算时间，它表明了执行应用程序所有操作必须花费的时间（这个时间包括 L2 Cache 命中延迟），不会随着存储系统性能的好坏而改变，在本文的研究中，它是一个常量。$T_{\text{Mem\_stall}}$ 代表存储系统停顿时间，因为本文研究的对象是 L2 Cache，因此这个存储系统停顿时间指的是 L2 失效导致的系统停顿时间（不包括 L2 命中延迟造成的系统停顿时间重叠）。使多个应用程序总的存储停顿时间最小的集合 $\{C_1, C_2, \cdots, C_N\}$ 就是访存时间最优 Cache 划分。

应用程序总的存储停顿时间 $T_{\text{Mem\_stall}.total} = \sum_{i=1}^{N} T_{\text{Mem\_stall}.i}(C_i)$，式中 $T_{\text{Mem\_stall}.i}(C_i)$ 表示应用程序 i 占用 C_i 个 Cache 块时的存储停顿时间，它可以用式（2）计算：

$$T_{\text{Mem\_stall}} = C_{\text{average}} \times N_{\text{Misses}} \tag{2}$$

式中 N_{Misses} 代表应用程序失效次数，C_{average} 代表每个失效的平均失效开销。因此，只要能够知道每个应用程序的失效次数和对应的每个失效的平均失效开销，就可以预测出总的存储停顿时间。

因为对组相联 Cache 中的每一个 Cache 块进行划分硬件开销太大，本文只对组相联 Cache 的路（way）进行划分。

3 划分算法与实现

3.1 平均失效开销计算

根据式(2)，每个应用程序的平均失效开销可以根据该应用程序在划分区间内总的存储停顿时间和总的失效次数来计算。

通过监测每个处理器核流水线的状态，可以统计出处理器核上运行应用程序的流水线停顿周期数，但是流水线停顿周期数里包含了与 L2 命中延迟所造成的停顿时间重叠的部分，从所有的停顿周期数中减去这一重叠部分就可以得到由 L2 Cache 失效引起的总停顿周期数。为了判断一段时间是否属于重叠时间，给每一个 L1 指令和数据 Cache 都增加一个计时器，当发生一次 L1Cache 失效，向 L2 发送一次请求时，该计时器被赋予一个值 *L2_Latency*，*L2_Latency* 的大小为 L2 Cache 的命中延迟（一般是一个固定值），每过一个时钟周期，计时器减 1。当计时器时间大于 0 时，说明这段时间还属于与 L2 命中延迟时间重叠的部分，不能记入总的存储停顿时间。

每个划分区间内各处理器核在 L2 Cache 上的失效次数通过监测 L2 失效状态保持寄存器（MSHR, miss status holding register）的状态获得，为了判断 MSHR 中的失效属于哪个处理器核，在 MSHR 的每一项中添加 lb$N$ 位用于标识该失效属于哪个处理器核。平均失效开销计算的算法如算法 1 所示。

算法 1 计算平均失效开销 C_{average}

/*在每个划分区间开始时执行*/

$N_{\text{Misses}} = 0$;

$T_{\text{Mem\_stall}} = 0$;

/*每一个失效进入 L2 MSHR，且该失效属于本处理器时执行*/

N_{Misses} ++;

/*每个时钟周期执行，对于本 Profiler 模块对应的 L1 指令和数据*/

if 本时钟周期内发生 L1 失效

then *timer_of_L*1=*L2_Latency*;

else *timer_of_L*1−−;

/*每个时钟周期执行*/

if 流水线停顿了

then pipelinestall=true;

else pipelinestall=false;

if L2 MSHR 中存在属于本处理器的失效

then hasL2Request=true;

else hasL2Request=false;

if L1 指令和数据计时器不为 0

then isNotL2hitLatency=false;

else isNotL2hitLatency=true;

if (pipelinestall and hasL2Request and isNotL2hitLatency)

then $T_{\text{Mem\_stall}}$ ++;

/*每个划分区间结束时执行*/

$$C_{\text{average}} = \frac{T_{\text{Mem\_stall}}}{N_{\text{Misses}}}$$

3.2 Cache 失效率预测

为了能够知道应用程序在占用不同数目 Cache 路时的失效率变化情况，给每个处理器核增加一个额外 tag 目录（ATD, auxiliary tag directory）[3]，就像给每个处理器核分配了一个虚拟的私有 L2 Cache。ATD 跟目标 L2 Cache（待划分的共享 L2 Cache）组相联结构相同，采用 LRU 替换策略。每个 ATD 仅仅包含目录项，不包含数据块。当 L1 发生失效时，请求会同时发向实体 L2 Cache 和对应处理器核的 ATD。ATD 和真正的私有 L2 Cache 一样运转，但它不会返回数据，而是用来统计处理器在不同路上的命中信息。由于基本的 LRU 替换策略遵循一种堆栈机制，即如果一个数据在 Cache 路数为 N 时命中，那么它在 Cache 路数大于 N 时也必然命中，因此根据应用程序在不同 Cache 路上的命中次数分布情况，可以得到应用程序占用 Cache 路数目发生变化时失效率的变化情况。

因为每一个处理器核都要有一个对应的 ATD，每个 ATD 的大小都跟真正的共享 L2Cache 目录一样大，当处理器核数目增加时，ATD 的硬件开销将会变得不可接受。采用组抽样技术[3]来减少硬件开销。ATD 以某一个固定间隔选取组（这个策略叫做简单静态策略[4]）来进行统计，最后得到的数据近似全局的统计数据。Qureshi[3]认为，总共

选取 32 组作为抽样进行统计可以很好地近似全局统计数据。

3.3　OMTP 划分算法

每个划分区间结束时，通过监测每一个竞争程序在 ATD 中各路的命中情况以及计算出来的 C_{average}，选择使总访存时间最少的划分策略。假设 m_a 和 m_b 分别代表一个应用占用 a 路 Cache 和 b 路 Cache 时的失效率（$a<b$），则该应用程序占用的 Cache 路数从 a 增加到 b 时所节省的访存时间 U_a^b 可以如下定义：

$$U_a^b = (m_a - m_b)C_{\text{average}} \tag{3}$$

假设 2 个应用程序 A 和 B 共用一个 16 路的 Cache。A 和 B 分别代表 2 个应用程序节省的访存时间，则应用 A 和 B 总共节省的访存时间 U_{tot} 可以如下计算：

$$U_{\text{tot}} = A_1^i + B_1^{16-i}, i = 1 \sim 15 \tag{4}$$

使 2 个程序总共节省的访存时间值最大的划分（i,16−i）就是访存时间最优的划分。

划分算法的目的是从所有的划分组合中寻找一个使总的访存时间最优的一种划分。当应用程序只有 2 个时，N 路 Cache 的划分组合只有 N+1 种。但是当应用程序数量增加时，划分组合的数量会呈指数值增长，这样要找出所有组合中访存时间最优的划分不太现实。例如当有 4 个应用程序共享一个 32 路组相连 Cache 时，划分组合有 6 545 种，当应用程序数目增加到 8 个时，划分组合猛增至 15 380 937 种。这个寻找最优划分的问题已经被证明是一个 NP 难（NP-hard）问题[9]。采用 Qureshi 等人提出的超前（lookahead）算法[3]来寻找符合条件的划分，算法伪码如算法 2 所示。

算法 2　超前算法

Balance = N　/*待分配的块*/

allocation[i] = 0　/*给每个竞争应用程序分配的块数*/

while(balance)do:

foreach application i, do:/*获得最大 T_{mu} 值*/

alloc = allocation[i]

max_mu[i]=get_max_mu(i, alloc, balance)

blocks_req[i]=应用程序 i 获得最大 T_{mu} [i] 时所用最小块数

winner = 拥有最大 T_{mu} 值的应用程序

allocation[winner] += blocks_req[winner]

balance -= blocks_req[winner]

return allocation

get_max_mu(p, alloc, balance):

max_mu = 0

for(ii=1; ii<=balance; ii++) do:

mu = get_mu_value(p, alloc, alloc+ii)

if(mu > max_mu) max_mu = mu

return max_mu

get_mu_value(p, a, b):

U = 当分配的块从 a 增加到 b 时，应用 p 所节省的访存时间

return U/(b−a)

在该算法中，T_{mu} 代表应用程序在分配的块数从 a 增加到 b 时的平均每一路节省的访存时间，它的定义如下：

$$T_{\text{mu}} = (m_a - m_b)C_{\text{average}}/(b-a) \tag{5}$$

超前算法的时间复杂度为 $O(N^2)$，对于一个 32 路组相联 Cache，采用超前算法只需要 512 次操作，相对于 500 万时钟周期的划分区间来说几乎可以忽略不计。在本文的研究中，每一个应用都至少要分配 1 路，并且在每个划分区间结束开始一个新的划分区间时，将 ATD 中所有命中计数器的值减半，使得 ATD 得到的命中信息具有局部时间相关特征。

3.4　OMTP 方法硬件实现

OMTP 方法的硬件结构如图 1 所示，每个处理器核都拥有各自的 L1 指令和数据 Cache，每个处理器核对应一个 Profiler 模块，该模块的作用主要有 2 个：1）获取每一个应用程序的平均 MLP 失效开销；2）获取每个应用程序在占用不同大小 Cache 路数时的失效率变化情况。Profiler 模块获得信息提供给划分算法硬件单元。划分算法硬件单元负责每一个划分时间区间内划分算法的实现。

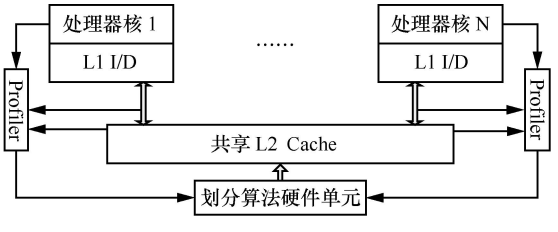

图 1　Cache 划分机制硬件结构

对 LRU 算法（或者是伪 LRU 算法）进行扩展。给共享 L2Cache 中每一个 Cache 块的 tag 中增加 lb$N$（N 为处理器核数目）位用于标识该块被哪个处理

器核占用。每个划分区间结束时，通过 3.3 节的算法，可以得到每个处理器核所分配的最高占用 Cache 块数值 $A(i)$，用于指导下一个划分区间内的替换策略。在每一个划分区间内，当一个 Cache 失效，替换引擎统计引发该失效的应用程序 i 在 Cache 中占用的路数，如果不超过 $A(i)$值，则在本应用占用的 Cache 块之外选择一个 LRU 块驱逐，如果达到或者是超过了 $A(i)$值，则在本应用占用的 Cache 块中选择一个 LRU 块驱逐。每执行 5M 指令运行一次划分算法，进行重新划分。

4 测试与分析

使用全系统模拟器，从吞吐率、失效率以及加速比 3 个方面评估 LRU 替换机制、UCP 方法以及本文提出的 OMTP 方法的性能。

4.1 测试平台

GEMS 模拟器[10]是威斯康辛大学在 Virtutech 公司开发的通用并行模拟器 simics[11]的基础上扩展实现的。simics 是一个全系统虚拟机，可以进行功能模拟和时序模拟，它支持 3 种模拟模式：1）正常模式；2）微体系结构模式；3）暂停(stall)模式。GEMS 模拟器在暂停模式下增加了对 Cache 系统和互联网络的模拟。采用 GEMS 模拟器对 OMTP 划分方法进行模拟测试。

本文模拟实验平台为一个双核 CMP 系统，每个处理器核为乱序执行的超标量处理器，有自己的 L1 指令和数据 Cache，2 个处理器核共享一个统一编制的 L2 Cache 和主存。具体参数如下：

1) 处理器核：2 个，4 流出，乱序执行指令窗口大小为 64，保留站大小为 128；

2) L1Cache：ICache DCache 为 32KB，数据块大小为 64，B4 路组相联；

3) 共享 L2Cache：1MB，数据块大小为 64byte，16 路组相联，命中延迟为 15 个时钟周期，MSHR 大小为 32；

4) 主存：400 个时钟周期。

4.2 度量指标

多道程序并行执行时的度量指标有很多种，本文选取其中的 3 种：吞吐率、失效率以及加权加速比。设 I_i 为应用程序 i 在并行执行时的 IPC，S_i 为应用程序 i 在单独执行（独享 L2 Cache）时的 IPC，M_{Ri} 为应用程序 i 在并行执行时的 L2 Cache 失效率。则这 3 种度量指标定义如下：

吞吐率：$IPC_{\text{sum}} = \sum IPC_i$

失效率：$M_{\text{sum}} = \sum M_{Ri}$

加权加速比：$W = \sum (I_i / S_i)$

这三者中，吞吐率反映了处理器单位时间里的处理能力，失效率反映了系统的整体性能，而加权加速比直接反映了执行时间的变化。

4.3 测试程序

本文从 SPEC CPU2000 测试程序里面选择了一些程序，并将它们两两组合起来，构成本文的基本测试用例（如表 1 所示），表 1 中第 3、4 列数据分别是 2 个应用程序独享 L2 Cache 时 C_{average} 平均值，第 5 列给出了 2 个竞争程序平均失效开销差异。

表 1 基本测试用例及它们的平均失效开销特征

序号	应用程序组合	App1 平均失效开销(cycle)	App2 平均失效开销(cycle)	平均失效开销差异(cycle)
1	ammp+art	259	208	51
2	mcf+art	243	208	35
3	applu+mcf	311	243	68
4	applu+ammp	311	259	52
5	equake+mcf	323	243	80
6	apsi+art	359	208	151
7	vpr+equake	305	323	18
8	vpr+gzip	305	307	2
9	apsi+swim	359	351	8
10	swim+gzip	351	307	44
11	apsi+mcf	359	243	116
12	ammp+vpr	259	305	46

采用 simics 提供的 magic 指令控制程序的执行，在每个程序完成初始化 Cache 的过程后，每个程序最少执行 250M 条指令，每 5M 条指令为一个划分区间。

4.4 实验结果与分析

通过比较LRU替换机制、UCP划分方法和OMTP划分方法在吞吐率、失效率以及加权加速比 3 个性能度量指标上的变化来评估 OMTP 方法的性能。

4.4.1 吞吐率

图 2 比较了 LRU 和 2 种划分方法吞吐率的差异，标号 1 到标号 12 分别对应测试用例中的 1 到 12 组测试程序，标号 13 为所有测试用例吞吐率平均值。在第 7、8、9 组中，由于 2 个竞争应用程序的平均失效开销相差很小，OMTP 方法在 UCP 方法基础上性能提升不明显。在第 3 组中，因为程序

本身性能提升空间有限，UCP 方法和 OMTP 方法均无明显作用。在其他各组测试中，OMTP 方法都取得比较好性能提升。跟 UCP 方法相比，OMTP 方法的吞吐率平均提高了 3.1%。

4.4.2　失效率

图 3 给出了 LRU 和 UCP 方法及 OMTP 方法在失效率上测试结果。标号 1 到标号 12 分别对应测试用例中的 12 组测试程序，标号 13 为所有 12 组测试程序的平均失效率。从图 3 可以看出，OMTP 方法的失效率比 UCP 方法要略高。

4.4.3　加权加速比

图 4 给出了 LRU 和 2 种划分方法在加权加速比上的差异。标号 1 到标号 12 分别对应测试用例中的 12 组测试程序，标号 13 为所有 12 组测试程序的平均加权加速比。在第 7、8、9 组中，因为同组 2 个应用程序的平均 MLP 失效开销相差不大，因此 OMTP 方法和 UCP 方法性能几乎一样，而在其他各组中，OMTP 方法的性能都要好于 UCP 方法。跟 UCP 方法相比，OMTP 方法的加权加速比平均提高了 1.3%。加权加速比的对比充分说明了 OMTP 方法能够提高程序的实际执行性能。

4.5　硬件开销

OMTP 方法的硬件开销主要集中在 2 个方面：

1）为了划分算法的执行，每个 Cache 块都要增加 lb$N$（N 为处理器核数目）位用于标识所属处理器核；2）Profiler 模块所用硬件开销。假设目标系统为双核、共享 1MB L2 Cache、16 路组相联、块大小为 64byte、L2 MSHR 有 32 项的系统。则第一项开销如下计算：

$$((1\text{MB}/64)\times \text{lb}2)/8 = 2\text{KB}$$

Profiler 模块中 ATD 的硬件开销统计为：ATD 目录项（1 有效比特+24bit tag+4bit LRU）为 29bit，

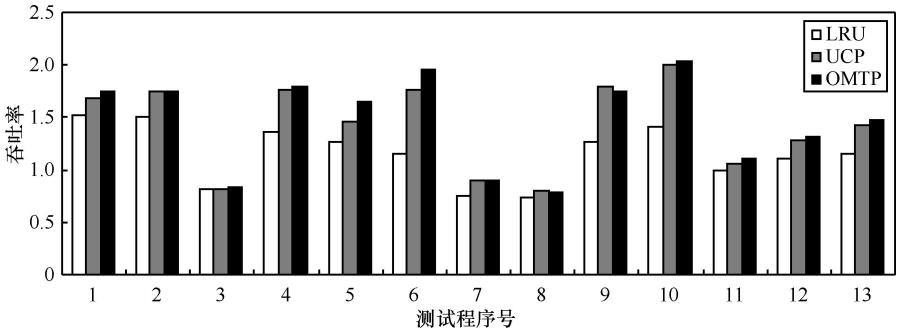

图 2　吞吐率测试结果

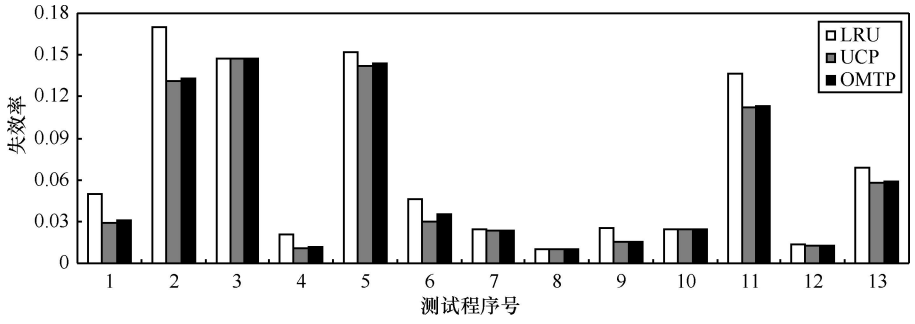

图 3　失效率测试结果

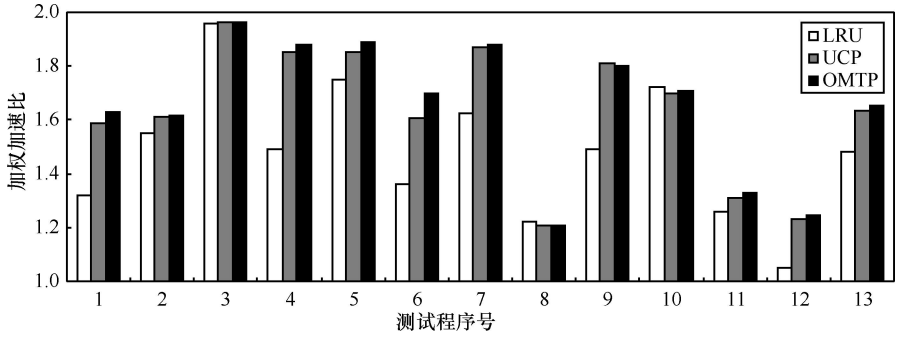

图 4　加权加速比测试结果

每一组 ATD 项数为 16 项，抽样组数为 32 组，ATD 目录开销（29bit×16 路×32 组）为 1 856byte。Profiler 模块中还有计数器 18 个（16 个用于统计命中信息，1 个用于统计 L2 失效次数，1 个用于统计存储停顿周期数）、计时器 1 个，所用硬件开销合计 1 856+4×19=1 932byte。

除了 1）和 2）中的开销以外，L2_MSHR 每一项中增加了 1bit，共 32bit（4byte）。因此实现 OMTP 方法额外硬件总开销为 2 000byte+1 932byte×2+ 4byte =5 868byte。不采用任何划分方法的 L2 Cache 硬件开销为 1MB+(24+4+1)×(1MB/64byte)=1 488KB。因此 OMTP 硬件开销在不采用任何划分策略的原始硬件开销上共增加了 5.868KB/1 488KB ≈0.4% 。

5 结束语

当多个竞争应用程序共用一个共享 Cache 时，以往的 Cache 划分算法都是以减少竞争程序总的失效率为第一目标。但是由于每个应用程序的 Cache 失效开销各不相同，减少总的 Cache 失效率并不一定能够提高程序整体执行性能，Cache 失效率最优并不能代表程序实际执行时间最优。本文提出一种访存时间最优的 Cache 划分（OMTP）方法，该方法通过统计计算出每一个应用程序的 MLP 失效开销，通过 ATD 辅助硬件得到应用程序在各个 Cache 路上的命中分布情况，并将这 2 项用于指导划分算法的执行，优化了多个竞争程序总的执行时间。

模拟实验结果表明，虽然 OMTP 方法的 Cache 失效率比 UCP 方法略有增加，但是具有更高的吞吐率和加权加速比，降低了程序实际执行时间。但是本文的实验结果均来自一个双核系统，当处理器核数目增加时，算法的可扩展性问题还有待进一步研究。

参考文献：

[1] STONE H S. Optimal partitioning of Cache memory[J]. IEEE Transactions on Computers, 1992, 41(9): 1054-1068.

[2] SUH G E, RUDOLPH L, DEVADAS S. Dynamic partitioning of shared Cache memory[J]. Journal of Supercomputing, 2004, 28(1): 7-26.

[3] QURESHI M K, PATT Y N. Utility-based Cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared Caches[A]. MICRO-39[C]. 2006.423-432.

[4] QURESHI M K, LYNCH D N, MUTLU O, *et al*. A case for MLP-aware Cache replacement[A]. ISCA-33[C]. 2006. 167–178.

[5] KROFT D. Lockup-free instruction fetch/prefetch Cache organization[A]. Proceedings of the 8th Annual International Symposium on Computer Architecture[C]. 1981.81-88.

[6] MUTLU O, STARK J,WILKERSON C. *et al*. Runahead execution: an alternative to very large instruction windows for out-of-order processors[A]. Proceedings of the 9th International Symposium on High Performance Computer Architecture[C]. 2003.129-140.

[7] CHOU Y, FAHS B, ABRAHAM S G. Microarchitecture optimizations for exploiting memory-level parallelism[A]. ISCA[C]. 2004. 76-89.

[8] ZHOU X, CHEN W G, ZHENG W M. Cache sharing management for performance fairness in chip multiprocessors[A]. Proc 18th International Conference on Parallel Architectures and Compilation Techniques (PACT 2009)[C]. Raleigh, North Carolina, USA, 2009. 384-393.

[9] RAJKUMAR R. A resource allocation model for QoS management[A]. The 18th IEEE Real-time Systems Symposium[C]. 1997.298-307.

[10] MARTIN M M K, SORIN D J, BECKMANN B M, *et al.* Multifacets general execution-driven multiprocessor simulator (gems) toolset[J]. SIGARCH Comput Archit News, 2005,33(4):92-99.

[11] MAGNUSSON P S, CHRISTENSSON M, ESKILSON J, *et al*. Simics: a full system simulation platform[J]. Computer, 2002, 35(2):50-58.

[12] PATTERSON D A, HENNESSY J L. Computer Architechture: a Quantitative Approach[M]. San Francisco: Morgan Kaufmann Publish, 1996.

作者简介：

李浩（1980-），男，湖南岳阳人，国防科技大学博士生，主要研究方向为高性能计算机体系机构。

谢伦国（1947-），男，湖南隆回人，国防科学技术大学教授、博士生导师，主要研究方向为高性能计算机和微处理器体系结构。