Software Engineering and Applications 软件工程与应用, 2016, 5(3), 181-189
Published Online June 2016 in Hans. http://www.hanspub.org/journal/sea
http://dx.doi.org/10.12677/sea.2016.53020

Hans汉斯

The IBP Replacement Algorithm Based on Process Binding

Guan Wang[1], Hanyu Zhao[1,2], Liang Sun[3]

[1]College of Computer Science, Beijing University of Technology, Beijing
[2]Key Laboratory of Trustworthy Computing in Beijing, Beijing
[3]ZD Technologies (Beijing), Limited, Beijing
Email: wangguan@bjut.edu.cn

Received: May 19th, 2016; accepted: Jun. 4th, 2016; published: Jun. 7th, 2016

Abstract

LRU as the last level Cache replacement algorithm will cause the Cache "jitter" phenomenon which influences the Cache efficiency. This paper, based on the binding process of IBP replacement algorithm, binds the process of data to the Cache, chooses the different replacement algorithm according to different situations, and achieves the division of the Cache. And compared with other Cache partitioning schemes, the hardware dependence is obviously weakened. The running speed of the work load increased by 7% in the same environment; the loss rate of Cache decreased by 14%, and the efficiency of the Cache increased significantly with the increase of the number of cores.

Keywords

IBP Replacement Algorithm, Process Binding, Cache Jitter

基于进程绑定的IBP缓存替换算法

王　冠[1]，赵涵宇[1,2]，孙　亮[3]

[1]北京工业大学计算机学院，北京
[2]可信计算北京市重点实验室，北京
[3]中电科技(北京)有限公司，北京
Email: wangguan@bjut.edu.cn

收稿日期：2016年5月19日；录用日期：2016年6月4日；发布日期：2016年6月7日

文章引用：王冠，赵涵宇，孙亮．基于进程绑定的 IBP 缓存替换算法[J]．软件工程与应用, 2016, 5(3): 181-189.
http://dx.doi.org/10.12677/sea.2016.53020

摘　要

LRU作为末级缓存的替换算法造成的缓存“抖动”现象严重的影响了缓存的效率，为此本文提出了基于进程绑定的IBP替换算法，通过将进程数据绑定到缓存中，并根据数据源的不同选取不同的替换策略，从替换策略的角度粗粒度的实现了缓存的划分，并与其他Cache划分方案相比硬件依赖明显减弱。在相同环境下提高了工作负载7%的运行速度，Cache缺失率下降了14%，并且随着核数的增多，对缓存效率的提升也更加明显。

关键词

IBP替换算法，进程绑定，缓存抖动

1. 引言

在过去的 20 多年中，处理器的性能以每年大约 55%速度快速提升，而内存性能的提升速度则只有每年 10%左右[1]。不均衡的发展速度造成了当前内存的存取速度严重滞后于处理器的计算速度，内存瓶颈对日益增长的高性能计算形成了极大的制约。而缓存的出现极大的缓解了这一现象，但末级缓存中大量的有害替换造成的缓存抖动极大的影响了缓存的效率。

1.1. 末级缓存的抖动现象

在系统运行时，有些缓存中的数据块会当缓存空间不足时，根据替换策略替换出去，但是，在被替换不久后可能又会因为使用到再次被放置到缓存中，这就是所谓的缓存抖动现象，如果被替换出去的数据在很短时间内又被使用到，那么这就是一种有害替换[2] [3]，而这种有害替换有很大几率出现在后台监控等大型应用中，因为这些应用是在系统中持续执行的，所以，当这些应用的数据被替换出去时，会很快再被替换回来放置到缓存中，这样的有害替换将会大大的影响缓存的效率。而在末级缓存上会发生缓存抖动的原因是常用的 LRU 替换策略并不适用于末级缓存中，因为末级缓存通常是共享的，这会使得末级缓存的访问局部性很差，并且多线程之间的干扰也是造成缓存抖动的原因之一，为了解决这一点，我们提出的基于进程绑定的 IBP 替换算法加入了对线程(即数据源)的判断，并进行保护，避免了多线程之间缓存的干扰以及竞争，在一定程度上也改善了缓存抖动的现象，提高了缓存的命中率。

之前提到的缓存抖动现象出现的实质原因是因为多核处理器的出现造成的末级缓存时间局部性差所引发的，虽然每个处理器的私有缓存通常可以表现出很强的时间以及空间的局部性，在 LRU 替换策略下往往效率也不低，但是由于末级缓存是共享的缘故，使得多个处理器对有限的末级缓存资源进行竞争，从而才引起的线程间即数据间的干扰，继而产生缓存的抖动，尤其在末级缓存中这种抖动尤其明显，时间局部性低的数据被替换进缓存之后就很少在会被使用到，但是由于上一级缓存的过滤作用，使得这些很少被使用到的缓存数据块并不能第一时间被找到并替换出缓存，这就造成了缓存中的数据污染，相反，这些经常被使用到的地址上的数据反而会被经常替换出去，如图 1 所示，这样交替的数据替换大大的降低了缓存的效率，处理器只能一次又一次的从内存中读取相应数据，所以处理好末级缓存中由于时间局部性及共享性引起的数据污染问题是非常关键的，通常情况下会有两种方案，一个是对 LRU 替换算法进行改进，在末级缓存替换算法中用更优质的替换方案来找到被替换的数据块，二就是对缓存本身做划分，以避免进程间的干扰，本文用的是第二种方案，通过对绑定进程数据的保护达到缓存划分的效果，从而

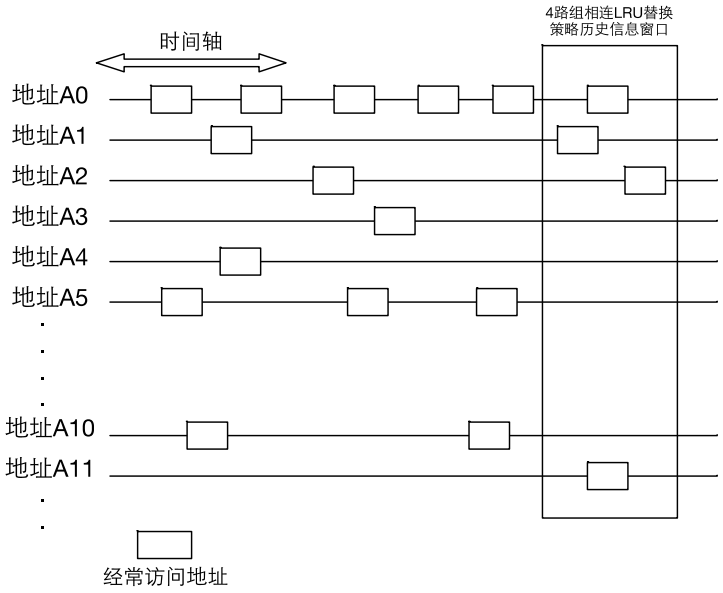

Figure 1. Example of final cache jitter
图 1. 末级缓存抖动示例

保护关键进程的数据，在进行数据替换的时候尽量不对关键进程的数据进行替换，缓解了数据交替替换引起的系统效率降低以及末级缓存数据时间局部性差引起的数据污染现象，从而提高了缓存的效率。

1.2. 末级缓存的“抖动”现象

缓存的抖动现象在末级缓存中非常明显，浙江大学的陈天舟等在 LRU 替换算法，16 路组相联，大小为 512kB 的 L2Cache 的环境下运行 SPEC CPU2000 测试集，结果发现平均 60%的替换数据多次被取回 Cache 重复使用，重复使用次数超过 2 次的约为 20% [4]，而这些有害替换会造成 CPU 直接从内存中读取数据，而其速度要远慢于直接从缓存读取的速度。现代最主流的解决方案是通过缓存划分的方式来解决缓存抖动的问题，Cho and Jin [5]提出一种操作系统级页分配方案来解决 Cache 局部性、容量以及隔离性方面的问题。基于共享 Cache 组织，该方案通过将物理页映射到 Cache 体来开发局部性，并且使用“虚拟多核”来提供对多道程序各线程之间的隔离。然而，这一方案需要页映射硬件的支持，同时需要对操作系统进行大量修改，如位置感知页分配，局部性和容量感知调度等。

Qureshi [6]提出的基于效用的缓存分区方法，监控所有正在运行的进程的对于共享资源的利用率，从而通过将缓存按路划分的方式分配给不同的进程，从而提高共享资源的利用率。McKee [4]综合考虑了缓存带宽，功耗的情况下来划分共享缓存，通过实时的监控应用程序在不同的划分方式下反馈的效果，按照作者提出的算法对划分模型进行计算，最后以此为依据对划分空间进行探讨，并且为了进一步探究缓存空间划分的有效性，提出通过 5 种指标进行加权计算得出分配共享缓存空间的方式[7]，并提出了两种空间划分的算法。

然而，按路划分的缓存空间分配方式的缺点也是显而易见的。首先，为了实现性能的隔离，组相连高速缓存的路数要比并发进程的数量更多[8]，随着当今系统上并发进程数的增加，这类实现方式对硬件的需求也变的越来越高，并伴随着对操作系统大量的修改[9]，而 IBP 替换算法是从替换策略的角度粗粒度的实现了缓存的划分，通过在数据替换时对数据源(即进程)的判断来对关键数据进行保护，从而避免了关键数据的交替替换，缓解了缓存的抖动现象。

2. 基于进程绑定的 IBP 替换算法

针对 Cache 的抖动现象以及 LRU 本身作为末级 Cache 效率低下的情况，IBP 替换算法用进程绑定的思想建立了缓存中数据与进程之间的联系，对缓存中关键线程的数据进行了保护，在 Cache 替换算法中加入了数据对进程的感知，并根据不同的情况来选择不同方案进行数据替换，避免了重要的数据块受到进程间对共享缓存的竞争所造成的干扰，增加了末级缓存中数据的局部性，从而缓解了 Cache 抖动的现象。

2.1. 基于进程绑定的 IBP 替换算法

IBP 替换算法的思想实质是通过识别关键进程来对缓存中该进程的数据块进行保护，在根据替换算法进行数据块替换的时候保护该数据块不会被其他数据块替换，从进程感知的角度实现了缓存的划分。IBP 替换算法具体流程如图 2 所示。

系统首先根据 tag 标示通过地址映射的方式从 Cache 中匹配相应的数据块，如果匹配成功，便直接从 Cache 中将该数据块的内容部分传给 CPU，如果发生 Miss，则需要从内存中读取相关数据并进行数据回写，根据 IBP 替换算法进行数据替换，在替换的时候，首先会根据 LRU 替换策略找到相应的数据块，然后对此数据块进行进程匹配，即通过比对该数据块的数据源(进程)的 pid 与守护进程数组中的关键进程 pid 进行匹配(check 方法)，如果匹配成功，则说明该数据的数据源属于关键进程，然后则需要通过基于寻找最近最少使用的替换策略的 LRU 替换算法找到下一个进行数据替换的位置，并判断是否超过阈值，在实验中，阈值为两个实验负载容量大小相对整体容量的百分比，若超过阈值，再重复之前的流程直到找到符合要求且不受保护的数据块，并进行数据替换。

LRU 替换算法通常会有一个 tag 专门来记录数据块被使用的次数，被访问的数据块会前移并且 tag 数值会增加，当缓存容量不足或者发生冲突时会根据数据在缓存中的位置以及标识被引用次数的 tag 来综合选出最符合条件的数据块进行替换，在本文用来实验的 GEM 全系统模拟框架中 LRU 替换算法以及本文提出的 IBP 替换算法的原理对比如图 3 所示。

假设缓存只能存储四个数据块 a1，a2，a3，a4，并且这 4 个数据块的数据源(即进程)分别是 t1，t2，t3，t4，并且用户绑定了 t3 进程，其具体访问流程如下：

1) 访问 a4 数据块

根据标识进行匹配，从缓存中找到相应数据块 a4，从缓存中提取该数据并对缓存内数据历史信息进行修改，以便 LRU 替换策略可以更准确找到最优替换数据块。

2) 访问 a5 数据块

根据标识进行匹配 a5 数据块，没有找到相应数据块。根据 LRU 替换算法找到替换数据块 a3，IBP 替换算法会根据数据对应的进程 PID 与守护进程模块中的 PID 进行匹配，匹配成功即表示该数据的数据源与缓存绑定，不进行替换，而 LRU 替换算法则直接对数据块 a3 进行替换操作，并对数据块的历史信息进行更新。

3) 再次访问 a3 数据块

IBP 替换算法由于之前没有将关键进程的数据 a3 替换出去，当再次访问时，会根据标示找到相匹配的 a3 数据块，但是由于 LRU 替换策略之前并没有对 a3 数据块进行特殊处理，导致再次访问时从缓存中无法匹配到相应数据，需要从内存中读取相应数据，并进行数据替换，造成了缓存的数据抖动，降低了缓存的运行效率。但是 IBP 替换算法因为“保护”了 a3 数据块，避免了其受到其他线程数据的干扰，并没有进行二次替换，从效率上比传统的 LRU 替换算法要提高了很多。

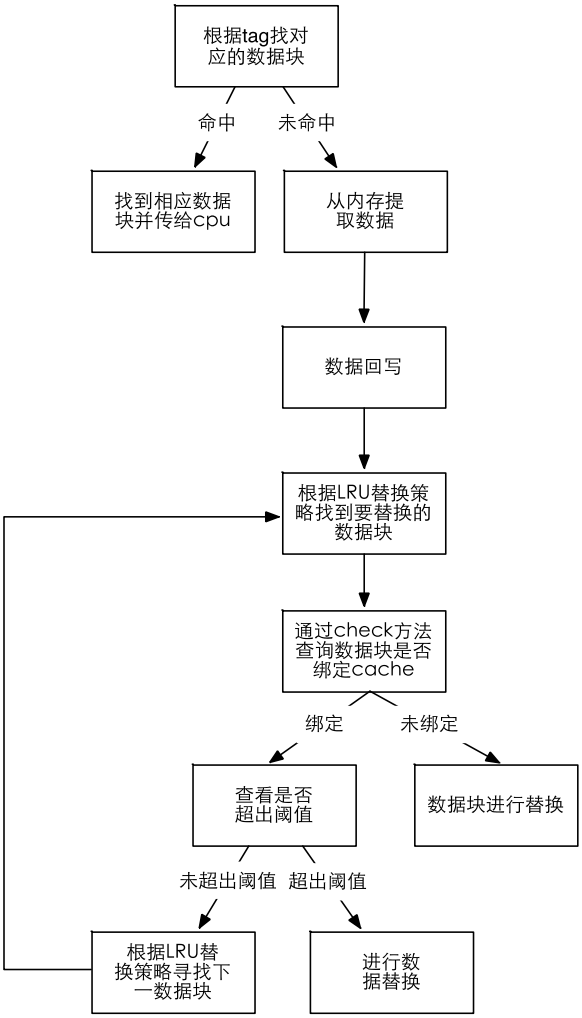

Figure 2. Flow chart of IBP replacement algorithm
图 2. IBP 替换算法流程图

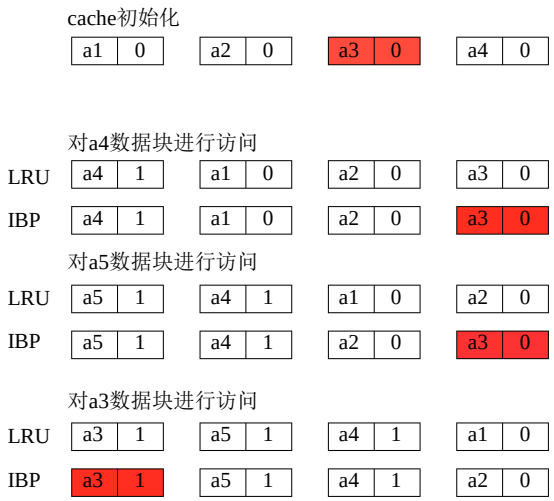

Figure 3. Comparison of operation mechanism of LRU and IBP replacement algorithm
图 3. LRU 与 IBP 替换算法运行机制对比

3. 实验及结果分析

3.1. IBP 替换算法实验设计

实验将 SPLASH2 测试集以及 PARSEC 测试集进行两两分组，并对每一组内其中的一个测试负载进行进程绑定，在相同的运行环境下，横向比较不同的替换算法在运行相同负载时的运行时间，以此比较 IBP 替换算法对于传统的 LRU 以及 FIFO 替换算法对系统效率带来的提升，并且会在不同核数下运行实验负载，横向比较缓存的缺失率，以此来探究不同的核数对 IBP 替换算法的影响。

本实验模拟的实验环境为 Gem5 下的 x86 CMP 体系结构，FS 全系统模拟，存储模型为 Classic，而实验模拟的系统的 CPU 个数是 4 和 8，缓存一共分为两级，一级缓存是私有的，末级缓存是共享的，如表 1 所示。

本实验中所使用到的负载程序是 PARSEC 测试集与 SPLASH2 测试集，这两种测试集在各大论文中也都被广泛使用，是公认的用于测试 Cache 效率的测试集。PARSEC 的当前版本包含各方各面的 13 个应用程序，例如视频编码技术、金融分析和图像处理等。而本实验选用的是 PARSEC 测试集中的 Canneal，Facesim，Fluidanimate，Streamcluster，如表 2 所示。

SPLASH2 是本实验用到的第二个负载测试集，SPLASH-2 使用 C 语言，由 12 个程序组成，使用 PThread 并行方式，SPLASH-2 包含 4 个核心程序：Cholesky，FFT，Radix 以及 LU。这 4 个核心程序也将和 Parsec 中的 4 个负载一起作为验证 IBP 替换算法效率的实验负载，这些测试集在实验的时候会分别进行归类，充当被绑定的进程，以及未绑定的进程，具体方案在实验结果中会做相应说明。

3.2. 实验结果及分析

如表 3 所示，本实验共有 4 组负载测试，这四组测试的测试集分别来自 SPLASH2 测试集以及 PARSEC 测试集，在这 4 组负载测试中，每一组实验都会绑定一个特定的负载，而另一个负载作为辅助进程，最后根据总运行时间进行对比分析，得出我们最后的实验结论。

实验一共分为 4 组，第一组实验的负载程序是 Facesim 和 Fluidanimate，Facesim 为绑定的进程，Fluidanimate 是未进行绑定的进程，而替换算法的对比是用最普遍的两种末级缓存替换算法 LRU 替换算法以及 FIFO 替换算法以及计时替换策略 ATR，而纵轴的单位是以 LRU 替换算法运行负载的时间归一化后的结果，这样方便进行横向比较，实验结果如图 4 所示。

对于绑定的进程负载，从总运行时间上看，对于末级缓存来讲，LRU 从效果上比 FIFO 替换算法要好很多，而 IBP 替换算法会对绑定的进程负载会有 10%左右的效率提升，但是同样的对于未绑定的负载就会有一定程度的下降，但是从总时间上看，对于 LRU 替换算法来说提升了 6%左右，而相较于 FIFO 替换算法更是提高了 12%的效率。并且要比计时替换策略 ATR 要提高了 2%左右的效率。

第二组的实验的负载分别是 Streamcluster 和 Canneal，Streamcluster 为绑定后的负载，而 Canneal 是未绑定的负载。实验结果如图 5 所示。

在第二组负载测试中，LRU 替换算法明显优于了不稳定的 FIFO 替换算法，而本文提出的 IBP 替换算法对于绑定的进程的效果比第一组负载实验要更加明显，从总时间上对比，比 LRU 替换算法提高了接近 10%的效率，比 ATR 的缓存效率基本持平。

第三组实验的负载是 Radix 和 FFT，Radix 是绑定的进程，FFT 是未绑定的进程，实验结果如图 6 所示。

与第三组负载测试实验结果相似，FIFO 替换算法的效果依然很不稳定，而 IBP 替换算法对于绑定的进程提升效果非常明显，虽然在未绑定的负载测试结果中总时间上比 LRU 替换算法多了 3%左右，但是

从总时间上对比还是比 LRU 替换策略提升了 7%左右的效率，而相较于 ATR 计时替换策略来说也有相应的提高。

最后一组实验的负载分别是 Cholesky 和 LU，Cholesky 作为实验中被绑定的进程，而 LU 是未绑定的进程，实验结果如图 7 所示。

第四组实验中，对于绑定和未绑定的进程 IBP 替换算法的效果都不是很明显，从效率上分析，绑定的 Cholesky 负载来说只比 LRU 替换算法提升了不到 5%的收益，而从总时间上看，IBP 替换算法比传统的 LRU 替换算法也只提升了 2%左右的效率，并且总时间也与 ATR 替换策略基本持平。

由上述四个实验可以看出，对于不同大小以及不同风格的负载，IBP 替换算法平均减少了负载 7%的运行时间，比传统的 LRU 替换算法减少了 14%的缺失率，并且效率也略优于基于计时替换策略的 ATR 替换算法。

Table 1. x86 architecture parameters
表 1. x86 体系架构参数

参数	取值
Number of cores	4,8
Core type	Out-of-order
Base frequency	2Ghz
L1 缓存(I & D)	32 KB, 2-way set associative, 64-byte line
L2 缓存	4 MB, 64-way set associative, 64-byte line
Coherence protocol	MOESI snooping protocol
Memory	268 cycles access time
OS	Linux 2.6.27

Table 2. PARSEC workloads
表 2. PARSEC 测试集

Programe	Suite	Application domain	Size
Canneal	PARSEC	Engineering	400.000 elements
Facesim	PARSEC	Animation	1 frame, 372126 tetrahedra
Fluidanimate	PARSEC	Animation	5 frames, 300,000 particles
Streamcluster	PARSEC	Data mining	16384 point per block

Table 3. The set of binding workloads
表 3. 绑定负载集合

负载	绑定的负载	未绑定的负载
W1	Facesim	Fluidanimate
W2	Streamcluster	Canneal
W3	Radix	FFT
W4	Cholesky	LU

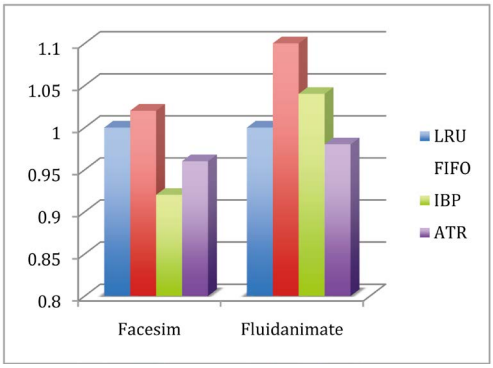

Figure 4. The first experiment result
图 4. 第一组负载实验结果

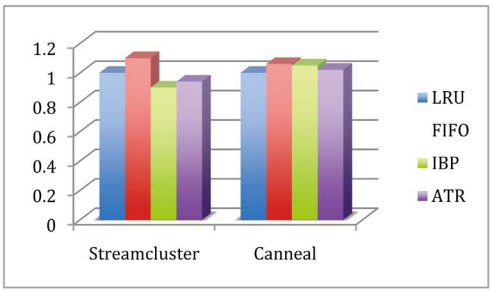

Figure 5. The second experiment result
图 5. 第二组负载测试结果

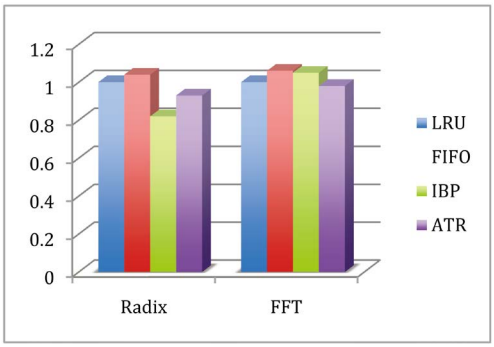

Figure 6. The third experiment result
图 6. 第三组负载测试结果

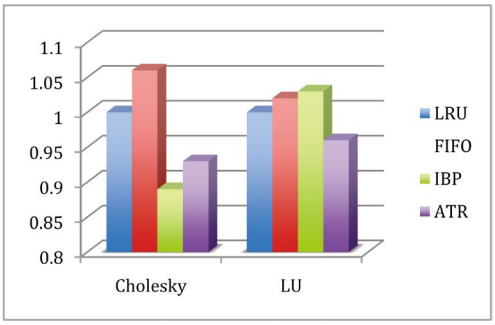

Figure 7. The fourth experiment result
图 7. 第四组负载测试结果

4. 结束语

针对有害替换造成的缓存抖动现象，IBP 替换算法通过对不同进程的数据在替换时采取不同的替换策略，对缓存中关键进程的数据进行保护，缓解了缓存中抖动的现象。并通过在相同环境下选取不同的替换算法来运行相同的实验负载进行实验，实验结果显示 IBP 替换算法平均减少了工作负载 7%的运行时间，并且降低了 14%的缓存缺失率，并且通过实验发现随着核数的增加，对末级缓存效率的提升也更加明显。IBP 替换算法从数据替换的角度粗粒度的实现了 Cache 的划分，相较于通过硬件来实现 Cache 划分以解决缓存抖动的解决方案，IBP 替换算法对硬件的需求也明显要低很多，从工程化的角度来看更易实现。

参考文献 (References)

[1] Xiang, L.-X., Chen, T.-Z., Shi, Q.-S., *et al*. (2009) Less Reused Filter: Improving L2 Cache Performance via Filtering Less Reused Lines. *Proceedings of the* 23*rd International Conference on Supercomputing*, New York, 68-79. http://dx.doi.org/10.1145/1542275.1542290

[2] Patterson, D.A. and Hennessy, J.L. (2005) Computer Architecture: A Quantitative Approach. 3rd Edition, In: Kaufmann, M., Ed., China Machine Press, Beijing.

[3] Hennessy, J.L. and Patterson, D.A. (2009) Computer Organization & Design. The Hardware/Software Interface. In: Kaufmann, M., Ed., China Machine Press, Beijing.

[4] Xu, X. and Peng, M. (2013) Critical Thread Guided Fine-Grained Adaptive Capacity Management for Shared CMP Caches. *Information Technology Journal*, **12**, 1366-1372. http://dx.doi.org/10.3923/itj.2013.1366.1372

[5] Yeh, T.Y. and Reinman, G. (2005) Fast and Fair: Data-Stream Quality of Service. *Proceedings of the* 2005 *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 237-248. http://dx.doi.org/10.1145/1086297.1086328

[6] Bitirgen, R., Ipek, E. and Martinez, J.F. (2008) Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. *Proceedings of the* 41*st Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 318-329. http://dx.doi.org/10.1109/micro.2008.4771801

[7] Qureshi, M.K. and Patt, Y.N. (2006) Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *Proceedings of the* 39*th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 423-432. http://dx.doi.org/10.1109/micro.2006.49

[8] van der Aalst, W. and van Hee, K. 工作流管理——模型方法和系统[M]. 王建民, 译. 北京: 清华大学出版社, 2010.

[9] Sejong, O.H. and Seog, P. (2013) Task-Role-Based Access Control Model. *Information System*, **28**, 533-562.