基于复用距离的 cache 失效率分析

付 雄，张 昱，陈意云

（中国科学技术大学 计算机科学与技术系，安徽 合肥 230027）

E-mail：yuzhang@ustc.edu.cn

摘 要：复用距离已经成为程序 cache 行为的一种重要度量标准，但高复杂度和可能的内存溢出问题使得其难以应用.本文在引入最大 cache 大小的基础上提出一种受限的复用距离分析方法.该方法有效地避免了一般复用距离分析可能导致的内存溢出问题，同时使得复用距离分析达到线性时间复杂度.文章通过对一系列整数和浮点程序的实验说明基于该复用距离分析的 cache 失效率分析的可行性和正确性.

关 键 词：复用距离；Cache 失效率；局部性

中图分类号：TP311 **文献标识码**：A **文 章 编 号**：1000-1220(2006)09-1777-05

Reuse Distance Based Cache Miss Rate Analysis

FU Xiong, ZHANG Yu, CHEN Yi-yun

(*Department of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027, *China*)

Abstract: Reuse distance has become an important metric of program cache behavior, but high complexity and possible memory overflow problem make its using difficult. Considering max cache size, this paper introduces a limited reuse distance analysis method. This method avoids possible memory overflow problem in normal reuse distance analysis, at the same time this method makes complexity of reuse distance analysis decrease to linear. Experiments from some integer and floating-point programs show it is feasible and correct that cache miss rate is analyzed by this reuse distance analysis.

Key words: reuse distance; cache miss rate; locality

1 引 言

计算机 CPU 速度的高速增长和内存速度的缓慢增长使得 CPU 和内存之间的速度差距越来越大，这导致内存系统成为性能上的瓶颈.现代计算机体系结构中广泛采用 cache 来降低这种影响，但是 cache 不能命中时形成的 cache 失效会引起较长时间的内存存取.实际中 cache 能否有效地利用取决于程序局部性（Locality）和数据的复用模式[1].随着 cache 层次数目的增加和自适应能力的增强，如何模型化程序局部性并分析 cache 失效率成为研究的热点，这对开发高性能应用程序、现代优化编译器以及高性能计算机系统都有着重要意义[1,2].

复用距离（Reuse Distance）是局部性的度量标准之一[2]，但复用距离分析具有较高的时空代价，这制约了其应用[1,3].一些研究者采用区间树（Interval Tree）[3]、伸展树（Splay Tree）[1]等数据结构来提高分析的性能.目前在精确分析程序数据的复用距离的各种算法中，最好的时间和空间复杂度分别为 O(NlogM)和 O(M)[1]，其中 N 和 M 分别为访问数据的次数和访问数据集的大小.分析的复杂度仍然很高，并且 N 和 M 都与程序及其输入相关；当 M 比较大，如达到上亿时，很容易导致系统物理内存甚至 32 位地址空间溢出.

本文通过引入最大 cache 大小（Max Cache Size），提出了一种受限的复用距离分析方法.该方法有效地弥补了其他复用距离分析的不足，其主要特点有：

（1）使分析所需的空间只与最大 cache 大小相关，不再随程序或者其输入的改变而改变，有效地避免了可能导致的内存溢出问题；

（2）将分析的时间复杂度降低到与访问数据的次数 N 成线性关系；

（3）当被访问数据的复用距离局限在某给定区间时，可以通过调整最大 cache 大小尽快完成复用距离分析.

本文的组织如下：第 2 节介绍受限的复用距离分析方法；第 3 节介绍用复用距离分析 cache 失效率；第 4 节介绍实验及结果分析；第 5 节和第 6 节分别介绍相关工作、结论及未来工作.

2 受限的复用距离分析

2.1 定 义

收稿日期：2005-06-27 基金项目：国家自然科学基金项目（60473068）资助；Intel 中国研究中心资助. 作者简介：付 雄，男，1979 年生，博士研究生，主要研究领域为程序设计语言理论和实现技术、程序分析及优化；张 昱，女，1972 年生，副教授，主要研究领域为程序设计语言理论和实现技术、XML 数据处理、软件体系结构等；陈意云，男，1946 年生，教授，主要研究方向为程序设计语言理论和实现技术、形式化描述技术、软件安全等.

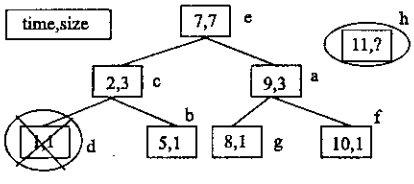
复用距离的概念可以追溯到上个世纪 70 年代，Mattson 等人提出了栈距离(Stack Distance)的概念[4]，用栈来评估虚拟内存中的页面置换算法.为了体现程序分析的思想，用体系结构无关的特征描述局部性，Ding 等人首次提出了和栈距离接近的复用距离概念[2].

定义 1.数据元素(Data Element)是指程序运行中被访问数据的标识，可以是内存地址，也可以是内存区域.在本文中，用符号 a,b,c 来表示数据元素.

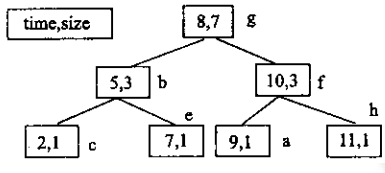
定义 2.复用距离(Reuse Distance)指串行程序运行中连续两次访问同一数据元素之间所访问的不同数据元素的数目.

序 号	1	2	3	4	5	6	7	8	9	10	11	12
数据元素	d	c	a	b	b	f	e	g	a	f	h	e
上次序号	∞	∞	∞	∞	4	∞	∞	∞	3	6	∞	7
复用距离	∞	∞	∞	∞	0	∞	∞	∞	4	3	∞	4

(a) 上次序号和复用距离示例



(b) 数据元素 h(序号 11)的复用距离，访问记录节点按上次序号组织为树，最大 cache 大小为 7.



(c) 数据元素 e(序号 12)的复用距离，访问记录节点按上次序号组织为树，最大 cache 大小为 7.

图 1 受限的复用距离计算示例

图 1 (a)示例了数据元素序列及其复用距离.从图 1 中看出，复用距离很好地量化了数据元素自身的复用关系，反映了程序时间局部性(Temporal Locality).当采用 cache 块作为数据元素时，对同一 cache 块中不同内存地址的多次访问也被视为复用，这反映了程序的空间局部性(Spatial Locality).在实际的复用距离分析中，常采用 cache 块作为数据元素，本文也采用这种形式.

程序的复用距离由该程序所有数据元素的复用距离组成，通常用如图 2 所示的柱状图来表示.其中横坐标 i 为∞时，表示首次访问该数据元素不存在复用关系；为其他整数时，表示复用距离在区间$[2^{i-1}, 2^i)$.纵坐标表示复用距离在该区间的访问次数占全部访问次数的百分比.

2.2 受限的复用距离分析

在进行复用距离分析及 cache 失效率分析时，我们发现了两个重要特征：

一是所关注的往往是 cache 大小在某一区间时 cache 失效率的变化情况，这时通过分析复用距离在该区间的访问就可以获得相关的 cache 失效率；

二是因为程序的局部性，复用距离往往集中在比较小的一个区域，绝大部分数据元素的复用距离都不会很大.

这给我们很大的启示：在分析 cache 失效率时，一般只需分析复用距离在某一区间的数据元素的访问，不用分析其他数据元素的访问，也即对每次访问进行分析时只需搜索一定区间的访问记录，判断在这个区间是否存在复用关系.这在很大程度上加快了复用距离的分析速度；同时分析过程中只需保存一定数目的访问记录，所需空间也会降低.我们将这种复用距离分析称为受限的复用距离分析.

我们采用最大 cache 大小作为搜索的限制长度.当搜索距离达到最大 cache 大小时停止搜索，不再分析复用距离的具体值，而认为复用距离为∞.作为参数的最大 cache 大小是可以调整的，如图 2 所示，最大 cache 大小(图中 Max Cache Size 线所示)只需要不小于所分析的 cache 大小(图中 Cache Size 线所示)，即可以准确分析该 cache 的 cache 失效率.显然，当最大 cache 大小为该给定区间的上界时，将能更快地分析该区间的 cache 失效率.

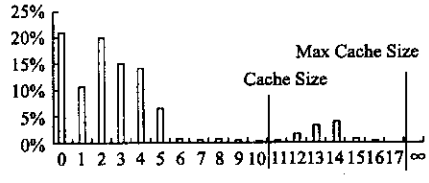


图 2 复用距离分布柱状图

为实现受限的复用距离分析，本文采用一种自调整的二叉查找树(即伸展树)[5]管理数据元素的访问记录.树中每个节点对应一个数据元素；节点包含两个域，time 域表示该数据元素的上次访问序号，size 域表示该节点为根的子树包含的节点数目，即不同的数据元素数目；树按节点 time 域排序.图 1 为受限的复用距离分析示例，其中最大 cache 大小为 7，(a)图为相关的数据元素序列等信息，(b)显示了计算第 11 个数据元素 h 的复用距离之前的伸展树的状态，此时 h 的上次序号为∞，则其复用距离也为∞.接着更新伸展树的状态来保存访问记录，删除最早的 d 节点并将 h 节点插入，然后按 time 域排序，此刻树为(c)中的状态.(c)显示了计算第 12 个 e 的复用距离，上次序号为 7，在树上节点 time 域大于 e 的共有 4 个节点(g、a、f 和 h)，表示上次访问和本次访问之间所访问的不同的数据元素为 4，所以 e 的复用距离为 4.图 3 给出了受限的复用距离计算算法.其中算法通过 Last_access_time_computation( )完成上次序号计算，对该问题的求解可参考 Bennett 等提出的算法[9]，这里不详细讨论.

```
//输入:数据元素序列 addr,长度为 N;
//      最大 cache 大小 max_cache_size;
//输出:复用距离序列 rd,长度为 N;
Limited_Reuse_Dsitance(int addr[], int N, int max_cache_size) {
  //计算上次访问序列 last
  Last_access_time_computation(addr, last);
  root = Tree_initialization();//初始化伸展树
  for (i = 0;i<N;i++) {
    if (last[i] != ∞) {
      //查找 time 域为 last[i]的节点
      node = Tree_find_node(root, last[i]);
      if (node != NULL) {
        //求 time 域大于 node 的 time 域的节点数目
        num= Max_node_num(root, node);
        rd[i] = num;node.time = i;
      } else {
        rd[i] = ∞;
        if (root.size >= max_cache_size)
          //删除树上最早的节点
          Tree_delete_oldest_node(root);
        //插入 time 域为 i 的新节点 node
        Tree_add_node(root, i);
      }
    } else {
      rd[i] = ∞;
      if (root.size >= max_cache_size)
        Tree_delete_oldest_node (root);
      Tree_add_node (root, i);
    }
    //调整使树按 time 域排序
    Tree_adjusting(root);
  }
  return rd;//返回复用距离序列
}
```

图 3　受限的复用距离分析算法

2.3　性能分析

下面对 2.2 节中的算法进行性能分析.假设最大 cache 大小为 S.算法执行的过程中,需要保存的访问记录的数目不会超过最大 cache 大小 S,故所需的空间大小最多为 S,算法的空间复杂度为 O(S).实际中 S 为常数,则空间复杂度为常量阶.

一次数据元素访问的复用距离分析的时间由在伸展树上的查找(Tree_find_node)、删除(Tree_delete_oldest_node)、插入(Tree_add_node)操作以及伸展树的调整(Tree_adjusting)操作来决定.当伸展树中的节点数最多为 S 时,根据伸展树的性质,这些操作的时间复杂度均为 O(logS)[5].从而每次数据元素访问的复用距离分析的时间复杂度为 O(logS),这样长度为 N 的数据元素序列的复用距离分析的时间复杂度为 O(NlogS).当最大 cache 大小设定 S 为常量时,整个程序复用距离分析时间复杂度与 N 成线性关系.

3　cache 失效率分析

若复用距离分析中的数据元素为 cache 块,则复用距离就为连续两次对同一 cache 块访问之间所访问的不同的 cache 块数目.当 cache 采用在全关联(FA,Fully Associative)最近最少使用(LRU,Least Recently Used)的页面替换策略时,cache 块的复用距离也就决定了第二次对该 cache 块的访问是否能够命中.利用复用距离来分析 cache 失效率就是基于这个原理.Cache 失效率的计算公式如下:

$$p=\begin{cases}1 & if\ \text{cache size} <= \text{rd} \quad //cache\ miss\\ 0 & if\ \text{rd} < \text{cache size} \quad //cache\ hit\end{cases} \tag{1}$$

$$P_{total}=\sum_{i\in G} p_i \times perc_i \tag{2}$$

其中公式(1)计算一次访问的 cache 失效率 p,如果复用距离 rd 不小于 cache 大小,则表示 cache 失效,失效率为 1.公式(2)用于计算程序的 cache 失效率 P_{total},将复用距离分成多个组 G,程序的 cache 失效率 P_{total} 为每个组的 cache 失效率 p_i 乘以该组的访问次数所占比例 $perc_i$ 的和.

图 2 也体现了 cache 失效率的分布情况,当复用距离大于 cache 大小(即 Cache Size 线右部)时,纵坐标表示失效的访问所占的比例,也即此时的 cache 失效率.

4　实验及结果分析

在实验中,我们采用工具 Pin[7] 对被测试的程序进行插桩,收集程序中所有的访问地址,这些地址包含程序所有的读指令和写指令操作的地址.

表 1　测试程序复用距离信息

测试程序	访问次数	复用距离				
		0		∞		其它平均
164.gzip	1,257,322,879	366,655,159	29.1616%	18,091	0.0014%	88.26
176.gcc	731,217,636	252,501,729	34.5317%	117,462	0.0161%	50.32
181.mcf	59,222,122	12,152,319	20.5199%	43,782	0.0739%	1192.41
197.parser	1,689,710,350	654,521,072	38.7357%	203,891	0.0121%	223.90
256.bzip2	3,781,432,077	2,583,014,793	68.3078%	3,567,656	0.0943%	350.55
179.art	1,390,561,635	326,897,504	23.5083%	64,431	0.0046%	4494.51
183.equake	538,863,120	146,180,412	27.1276%	9,982,792	1.8526%	327.28
188.ammp	2,743,136,395	577,383,657	21.0483%	2,014,455	0.0734%	1007.09

实验的测试环境为 RedHat Linux 9.0 平台,测试集由表 1 显示的 SPEC CPU2000 中的 8 个测试程序组成,其中包含 5 个整数程序和 3 个浮点数程序,所有的测试程序都采用 test 输入集,并且所有的测试程序用 gcc3.4 作为编译器,O3 优化

选项.分析中 cache 块为 32byte,最大 cache 大小为 2^{17}.

为了验证 cache 失效率分析结果的正确性,本文采用 SimpleScalar 3.0 工具集包含的 cache 失效率分析程序 Cheetah[8]来验证我们的 cache 的失效率分析,所验证的 cache 配置包含 cache 块为 32byte 的全关联、1 路、2 路、4 路和 8 路组关联.

表 1 给出了各测试用例的访问次数和复用距离信息,复用距离分为 0、∞和其他三种,其中 0 和∞分别给出了访问次数和占总访问次数的百分比,其他给出复用距离的加权平均距离.从表 1 中可以看出,复用距离为 0 的访问次数占到总的访问次数的 20%到 70%之间,平均约为 33%;而复用距离为∞的访问次数占总访问次数在 0.0014%到 1.8256%,平均不

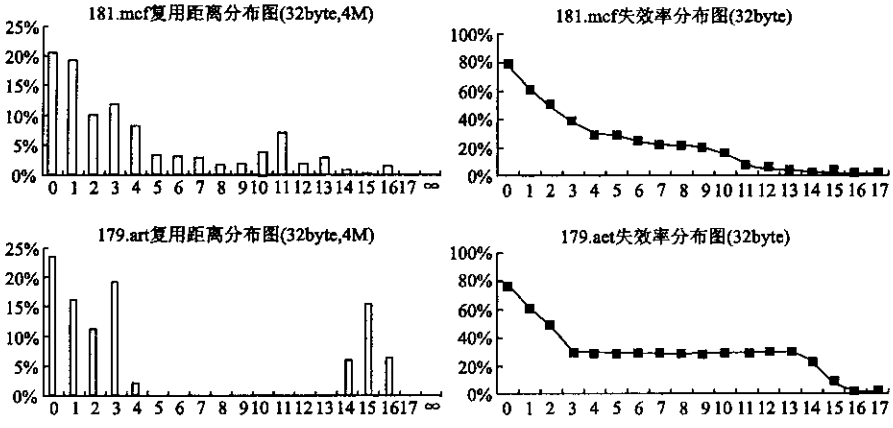


图 4 复用距离分布图和 cache 失效率分布图

到 0.3%,可见当最大 cache 大小足够大时,能够准确分析绝大部分访问的复用距离.可以看出,在 8 个测试程序中 176.gcc 复用距离比较小,所以局部性也相对比较好.

图 4 给出了部分测试程序的复用距离分布柱状图和 cache 失效率分布图.复用距离分布图中横、纵坐标含义同图 2;失效率分布图中横坐标为 cache 块数目取以 2 为底的对数值,纵坐标为 cache 失效率.在图中可看出 179.art 的访问在横坐标为 5～13 的区间相对较少,而在横坐标为 14～17 的区间有一定的分布,从而失效率在横坐标为 5～13 的区间基本保持不变,大约维持在 30%左右.因此对 179.art,在这一区间增加 cache 大小并不能明显减少 cache 失效率,这对为程序选择特定的 cache 配置有重要的指导意义.

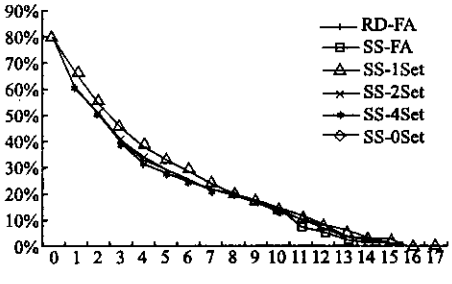


图 5 181.mcf cache 失效率比较图

图 5 给出了本文分析 cache 失效率方法与 Cheetah 程序的一个比较;这些测试都是基于 LRU 替代策略的 cache 的.对全关联 cache,实际上两种方法的分析具有完全相同的结果;在组关联 cache 中,当关联度数比较大的时候,其 cache 失效率与全关联的 cache 更加接近.这显示了复用距离不但能够准确分析全关联 cache 的失效率,而且还能近似分析组关联 cache 的失效率.由此可见,利用受限的复用距离分析来分析的 cache 失效率是准确的.

5 相关工作

在复用距离分析方面,文献[4]提出了和复用距离具有同样意义的栈距离,并且用于 cache 设计;文献[3]给出了几种更快速的栈距离计算方法,其时间复杂度为 O(NlogM);文献[1]提出了一种近似复用距离计算方法,通过该方法能够将时间复杂度降低到近似线性关系 O(loglogM),但是这种计算针对以 cache 块为复用单元的时候会有比较大的误差.相比本文提出的受限的复用距离分析,这些分析所需空间随程序或者输入而变化,可能会导致内存溢出;另外时间复杂度也比较高.

在 cache 失效率分析分析方面,文献[9]给出了一种模拟分析全关联 cache 的栈算法,通过这种算法能够只用一次模拟就可以得到各种 cache 大小下的 cache 失效率;文献[2]讨论了利用复用距离分析计算 cache 失效率的方法;文献[6]给出了一种 cache 失效率预测的方法,通过一些输入集来预测其他输入集的 cache 失效率.

本文采用的 cache 失效率分析方法和这些基本原理一样,不同之处在于基于的复用距离分析不同.文献[8]提出了一种完全用树来模拟 cache 失效率的方法.

6　结论及未来的工作

本文在引入最大 cache 大小的基础上提出受限的复用距离分析方法,该方法有效地避免了可能的内存溢出问题,同时有效地降低了复用距离分析的时间复杂度,通过一系列的实验证明了该方法的可行性和正确性.

文章相关的复用距离分析都是在二进制代码上进行的,不能将 cache 行为和源代码关联起来.如何将复用距离和源代码两者关联起来,帮助在源代码级别分析程序的局部性,为程序员或编译器进行程序优化提供指导,这是我们未来工作的方向.

References:

[1] Ding C, Zhong Y. Predicting whole-program locality with reuse distance analysis[C]. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA, June, 2003: 245-257.

[2] Beyls, Kristof. Software methods to improve data locality and cache behavior[D]. Ghent University, 2004.

[3] Almasi G, Cascaval C, Padua D. Calculating stack distances efficiently[C]. In: Proceedings of the First ACM SIGPLAN Workshop on Memory System Performance, Berlin, Germany, June 2002: 37-43.

[4] Mattson R L, Gecsei J, Slutz D, et al. Evaluation techniques for storage hierarchies[J]. IBM System Journal, 1970, 9(2): 78-117.

[5] Sleator D D, Tarjan R E. Self adjusting binary search trees[J]. Journal of the ACM, 1985, 32(3): 652-686.

[6] Zhong Y, Dropsho S G, Ding C. Miss rate preidiction across all program inputs[C]. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, New Orleans, Louisiana, September 2003.

[7] Chi-Keung Luk, Robert Cohn, et al. Pin: building customized program analysis tools with dynamic instrumentation[C]. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, Illinois, USA, 2005: 190-200.

[8] Sugumar R A, Abraham S G. Efficient simulation of multiple cache configurations using binomial trees[R]. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.

[9] Bennett B T, Kruskal V J. LRU stack processing[J]. IBM Journal of Research and Development, 1975, 19(4): 353-357.