一个基于图着色的CACHE优化方法*

邓　宇,王　蕾,张　明,龚　锐,郭御风,窦　强
(国防科技大学 计算机学院,湖南 长沙　410073)

摘　要:提出了一个编译时的Cache管理优化方法。该方法根据访存行为将程序中的数据划分成若干数据对象,根据数据对象的大小将Cache划分为一个带有别名的伪寄存器文件,每个伪寄存器由若干Cache行组成,可以容纳一个数据对象;使用一个经过改进的图着色寄存器分配算法来决定这些对象在Cache中的位置以及发生冲突时的替换关系。数据对象的划分将Cache的管理分为两个层次,一个是编译时编译器对粗粒度的数据对象的管理,另一个是运行时硬件对细粒度的Cache行的管理,这样编译器和硬件的优势都得到发挥。基于GCC进行了方法实现,并通过simplescalar构造了支持Cache Coloring的硬件模拟平台。实验结果表明Cache Coloring能较好地开发程序的局部性,降低Cache失效率。

关键词: Cache优化;图着色;编译时优化
中图分类号:TP302　　**文献标志码:**A　　**文章编号:**1001－2486(2012)06－0020－06

A cache optimizing method based on graph coloring

DENG Yu, *WANG Lei*, *ZHANG Ming*, *GONG Rui*, *GUO Yufeng*, *DOU Qiang*
(College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: A graph coloring based management optimizing algorithm for cache, namely Cache Coloring, has been proposed. This algorithm first partitions the data into several data objects according to their memory accessing behaviors. Then it partitions the cache into a pseudo register file with alias according to the size of the data objects. Each pseudo register in this register file can hold one of the data objects. Finally, it uses an extended graph coloring register allocation algorithm to determine the position of each data object in the cache and their replacement relationship. The data object partitioning divides the management of cache into two levels, one for the coarse-granularity management of the data objects in the compile-time and the other for the fine-granularity management of the cache lines in the run-time. So the advantages of both compiler and hardware are exploited. Cache Coloring is implemented in GCC. A hardware simulation platform which supports Cache Coloring is built based on the Simplescalar processor simulator. The primary experimental results show that Cache Coloring can exploit the locality well and reduce the cache miss rate.

Key words: cache optimizing; graph coloring; compile-time optimizing

作为处理器到主存之间的高速缓存,Cache对于现代微处理器十分重要。目前大多数处理器都采用硬件管理Cache的方式,包括Cache的映射方式、查找方式以及替换策略等。硬件管理Cache的优点是速度快,对软件透明,不需要修改程序和编译就能够获得性能提高。而随着片上Cache容量的不断增大,其缺点也越来越突出,主要是硬件开销越来越大,功耗问题越来越严重[1-2]。为了缓解这些问题,有效利用Cache,研究人员提出了让软件参与Cache管理的思想。软件参与Cache管理的方式包括通过硬件提供的管理接口部分控制Cache的行为,例如修改标识和显式地作废某一行等[3-6],或者完全由软件来控制Cache,例如将Cell的大容量片上存储器作为软件控制的Cache来使用[7-8]。文献[15]从地址映射、替换策略、取数策略以及编程和编译等方面对纯硬件管理方法与纯软件管理方法之间的差异进行了比较,其结论是,硬件管理机制适合管理细粒度数据项(例如字,Cache行),而软件管理机制则对粗粒度的数据项(例如数组)十分有效,二者相结合才能实现对当前大容量片上Cache的有效管理。

基于这样的观察,本文提出了一个称为Cache Coloring的编译时优化方法,基本思想是在编译时将程序中的数据划分成若干数据集合,称为数据对象(简称对象),然后使用图着色算法来

*　**收稿日期:**2012－09－06
基金项目:国家“核高基”重大专项资助项目(2009ZX01028－002－002);国家自然科学基金资助项目(61170045,61103011);信息保障技术重点实验室基金项目(KJ－11－04)
作者简介:邓宇(1977—),男,江苏如皋人,助理研究员,博士,E-mail: yudeng@nudt.edu.cn

决定这些对象在Cache中的位置和替换关系,而对象内的数据替换则由硬件在运行时完成。这样,Cache的管理被分为两个层次,第一个层次是粗粒度的数据对象的管理,由编译器在编译时静态完成;第二个层次是细粒度的Cache行的管理,由硬件在运行时动态完成。

在后面的论述中,我们首先通过矩阵乘的例子说明传统LRU(Least Recent Used,最近最少使用)算法存在的弊端;然后对Cache Coloring方法进行详细的描述,并对比其在矩阵乘示例中相对LRU的优势;最后通过一组测试程序评测了该方法在一般应用程序的Cache命中率以及适用范围。

1 动机

图1给出了两个4×4矩阵相乘的代码,每个内层循环迭代将**A**矩阵的一行与**B**矩阵的一列对应元素相乘并求和,构成**C**矩阵的一个元素。假设矩阵一行有4个元素,Cache的一行刚好容纳矩阵的一行,总共有6个Cache行。另外还假设矩阵在存储器中按行优先存放。

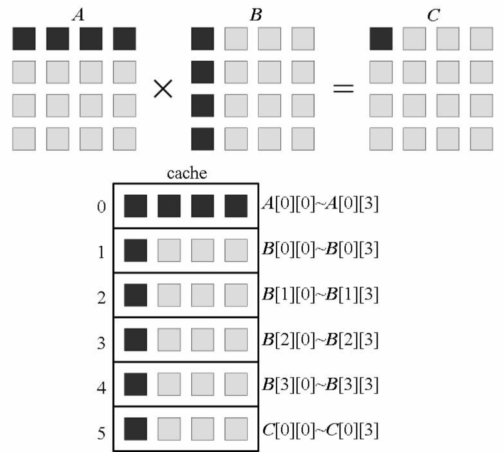
```
#define N 4
for ( i =0; i <N; i + + )
    for ( j =0; j <N; j + + )
      for ( k =0; k <N; k + + )
        C[i][j] + = A[i][k] * B[k][j];
```

图1 4×4矩阵乘代码

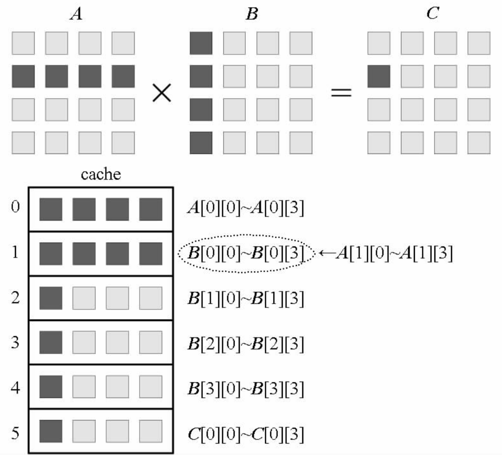
Fig. 1 4x4 Matrix multiplication code

图2给出了在LRU替换算法下Cache内数据的分布和替换情况。如图2(a)所示,当最内层循环完成第一次迭代时,**A**矩阵的第0行占据了0号Cache行的位置,**B**矩阵依次占据1号Cache行到4号Cache行,**C**矩阵的第0行占据了5号Cache行的位置,Cache刚好被占满。当内两层循环结束,需要将**A**矩阵的第1行装入Cache,必须将其中一行替换出Cache。LRU替换算法是将最近最长时间没有被使用的行替换出去,因此应该被替换的是1号Cache行的数据,即**B**矩阵的第0行数据。于是得到如图2(b)所示结果。然而,后续操作又要访问**B**矩阵的第0行,使得第0行要重新装入Cache,并将2号Cache行的数据,即**B**矩阵的第1行替换出Cache。以此类推,引起多米诺骨牌效应,最终失效次数达到24次。

造成这种结果的原因是,硬件LRU算法只能依据固定且有限的信息推测数据将来的使用情况。一个简单的观察结果是,**B**矩阵完全可以保



(a) 最内层循环完成第一个迭代时Cache的数据分布



(b) 外层循环进入第二个迭代时Cache的替换情况

图2 LRU算法下4×4矩阵乘

Fig. 2 4×4 Matrix multiplication under LRU

留在Cache中而不被替换出去,因为**A**矩阵和**C**矩阵的一行在完成相应运算之后就不再使用,如果让**A**矩阵和**C**矩阵的新行去替换各自相应的旧行,失效次数可以大大减少。问题的关键在于如何在编译时感知**A**矩阵的行不会被交替使用这一特点,从而将**A**矩阵的所有行都分配到同一个Cache行上。通过后面的论述我们将看到,Cache Coloring能够做到这一点。

2 Cache Coloring方法实现

如图3所示,Cache Coloring由对象划分、Cache划分和Cache分配三个步骤组成。对象划分主要是从程序所使用的数据集合中提取出数据对象;Cache划分主要是根据数据对象的大小,将Cache划分成若干区域,形成一个带有别名的伪寄存器文件;Cache分配是使用改进的图着色寄存器分配算法,将数据对象分配到划分好的Cache中。

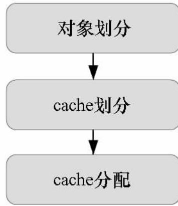


图3 Cache Coloring 的基本流程

Fig. 3 Base flow of Cache Coloring

2.1 对象划分

数据对象的划分实际上是确定软硬件管理的界面,对发挥 Cache Coloring 的有效性有很大的影响。一个直观的方法是以数组作为数据对象,但这对于拥有大数组的科学计算程序而言并不是一个高效的做法,因为这些数组内部的数据使用情况并不相同,而且过大的划分粒度将使软件管理失去意义。数据对象划分的关键在于发现时间和空间上关系密切的数据,并能够控制其大小。理想的数据对象有合适的大小,并且经常有规律的重复出现。我们通过一个名为 SEQUITUR[11] 的压缩算法来寻找这样的数据对象。

SEQUITUR 算法的原理如图4所示,它不断使用产生式来替换符号序列中重复出现的子序列,最后得到一个比原序列短的符号序列。输入序列为字符串"abcbcabcabc",首先将重复出现子串"*bc*"替换为字符 *A*,并得到产生式 $A \rightarrow bc$;然后将子串"*aA*"替换为字符 *B*,得到产生式 $B \rightarrow aA$;最后生成一个非终结符 *S* 以及产生式 $S \rightarrow BABB$。

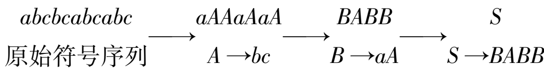


图4 SEQUITUR 的压缩原理

Fig. 4 The compress principle of SEQUITUR

使用 SEQUITUR 算法对数据引用的地址序列进行分析,就可以得到我们想要的数据对象。可以看出,每次得到产生式时新生成的符号,就可以作为一个数据对象的候选。

对象划分过程分为预处理、调用 SEQUITUR 和确定数据对象3个步骤,如图5所示。

预处理的目的是将访存地址序列中的字地址转换为数据块地址,同时将地址序列符号化,以便于 SEQUITUR 算法进行处理。

SEQUITUR 算法对符号化的块地址序列进行压缩处理,得到一个层次化的符号序列结构,作为潜在的数据对象划分候选。

最后根据 SEQUITUR 算法的结果,将符号序列中的每个符号作为一个数据对象,并记录该对象的相关信息,例如所属数组名、符号名、起始数

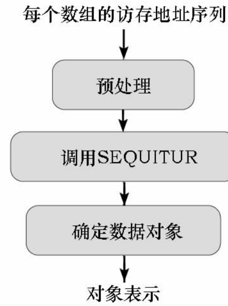


图5 数据对象的划分过程

Fig. 5 The partition process of data object

据块号、对象大小,等等。

2.2 Cache 划分

算法1 构造寄存器文件
输入:数据对象的集合 *QS*;
输出:寄存器类的集合 *RS*;

```
1  对 QS 中的每个对象 O,循环执行以下语句:
2  定义一个寄存器类 R,并将其加入集合 RS;
3  R.size ← O.size;
4  i ← 0;
5  当 i 小于 L-O.size+1 时,循环执行以下语句:
6    定义一个寄存器 r,并将其加入 R;
7    r.id ← i;
8    r.addr ← i;
9    r.size ← O.size;
10   i ← i+1;
```

图6 寄存器文件的构造算法

Fig. 6 The construct algorithm of register file

Cache 划分的目的是将数据对象在 Cache 中的分配问题转换为寄存器分配问题,从而可以使用已有的图着色寄存器分配算法来解决这个问题。为此要将 Cache 划分为一个带有别名的伪寄存器文件。带有别名的目的是充分利用 Cache 的空间。

寄存器文件的构造算法如图6所示。对每一个对象,Cache 空间都按照对象的大小 *O.size* 划分为 *L-O.size*+1 个块(第5行到第10行),每个块的大小都是 *O.size*(第9行),其中 *L* 是 Cache 行总数。这样的一个块就被称为一个(伪)寄存器,可以容纳一个对象。寄存器按照划分的顺序进行编号(第7行),并记录起始地址(起始 Cache 行号)(第8行)。一个对象划分一次就构成一个寄存器类(第2行),多次划分得到的所有寄存器类构成的集合 *RS* 称为寄存器文件。

图7给出了矩阵乘示例的划分结果,矩阵 ***A*** 和 ***C*** 各有6个寄存器可以使用,矩阵 ***B*** 有3个寄存器可以使用,并且这3个寄存器互为别名。

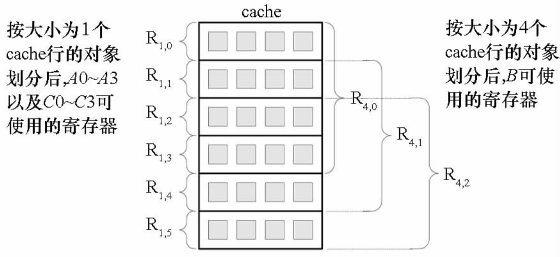


图7　划分 Cache 得到的伪寄存器文件

Fig. 7　Pseudo register file after Cache partition

本文所提出的寄存器文件构造算法是对文献[12]提出的便笺存储器划分算法的一个扩展。主要区别在于文献[12]将同一类寄存器放在不重叠的空间,因此同一类寄存器不存在别名。这在一定程度上可以简化寄存器分配的过程,但是可能降低空间利用率。

2.3　Cache 分配

对象划分和 Cache 划分的最终目的是将数据对象在 Cache 中的放置问题转换为寄存器分配问题,从而可以使用已有的图着色寄存器分配算法来解决。经典的图着色寄存器分配算法[13]由 Smith 于1998年提出,随后于2004年提出了一般化的图着色寄存器分配算法[10],增加了对寄存器别名的支持。由于 Cache 与真正的寄存器文件存在较大差异,以上算法必须经过一定的修改才能用于 Cache 的分配。

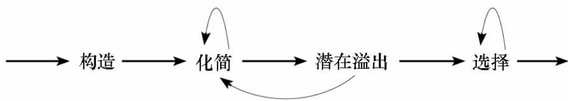


图8　Cache 分配的流程

Fig. 8　The flow of Cache allocation

图8给出了 Cache 分配的流程。与经典算法相比,主要区别如下。首先,生命周期相干图的构造不同。传统的构造方法只能定性给出两个对象是否相干,而 Cache 分配需要定量的刻画对象的相干程度。其次,在化简步骤之后一般还有合并和冻结步骤,用于消除冗余的 move 语句。但是在 Cache 分配中,要识别出数据对象之间的赋值比较复杂,而且不是很有意义。为了提高效率并降低复杂度,去掉了合并和冻结步骤。最后,去掉了实际溢出时的处理,例如分割对象生命周期,重新执行分配流程等。

图9给出了矩阵乘示例的最终分配结果,对象 **B** 被分配到寄存器 $R_{4,0}$,也就是0号到3号 Cache 行;$A0 \sim A3$ 被分配在4号 Cache 行(寄存器 $R_{1,4}$),$C0 \sim C3$ 被分配在5号 Cache 行(寄存器 $R_{1,5}$)。各个对象的位置确定之后,它们之间的替换关系也随之确定。如 **A** 矩阵和 **C** 矩阵的新行只会替换它们的旧行,而不会去替换 **B** 矩阵的行,使得 **B** 矩阵能够完全保留在 Cache 中,总的失效次数只有8次,比前面 LRU 算法减少了16次。

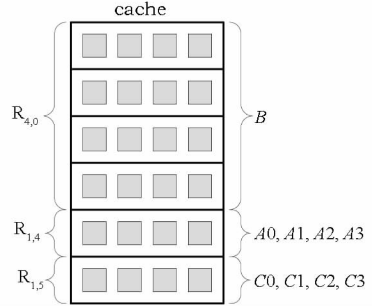


图9　矩阵乘中对象在 Cache 中的分配结果

Fig. 9　The Cache allocation result of the objects in matrix multiplication

3　评测

我们以 GCC 2.7 为平台实现了 Cache Coloring 方法,并以 simplescalar 为模拟和测试平台。为了给 Cache Coloring 方法提供必要的硬件支持,我们对 simplescalar 进行了修改。主要需要解决两个问题,一个是将 Cache 划分的各种信息传入模拟器,另一个是模拟出 Cache 上具有多个分段使用的空间效果。为了减少对指令集和编译器的修改,我们使用程序中对模拟器特定存储空间写数据的方法解决第一个问题。通过在模拟器中创建多个 Cache 来解决第二个问题。在与其他替换策略比较时,均使用传统的全相联 Cache。

参与评测的程序如表1所示。EP、IS、FT、CG 来自 NPB(NAS Parallel Benchmark),其中 EP 计算 Gauss 伪随机数,主要测试数学函数浮点运算性能;IS 完成整数桶型排序;FT 求解基于 FFT(快速傅立叶变换)谱分析法的三维偏微分方程;CG 采用共轭梯度法求解稀疏矩阵的特征值。GEMM 完成二维矩阵乘。Swim 来自 SPEC2000,对浅水方程组进行有限差分模拟。这些程序主要反映科学计算领域应用的特点。

表1　评测 Cache Coloring 方法的测试程序

Tab. 1　The benchmarks to evaluate Cache Coloring

程序	来源	数组	问题规模(单精度)
EP	NPB	1	131072
IS	NPB	5	65536
FT	NPB	9	32×32×32
CG	NPB	8	500×500
GEMM	BLAS	3	64×64
Swim	SPEC2000	14	64×64

首先,我们考察第 2 节提出的数据对象划分算法对以上程序的划分情况。我们将 Cache 的大小设置为 16KB,行大小设置为 16B,结果列于表 2 中。FT、CG、GEMM 和 Swim 等程序的数组引用具有明显的重用特征,对于这样的程序,我们的算法具有较好的划分效果,大数组被划分为多个对象。EP 和 IS 的数组只是被顺序引用,局部性较差。对于这样的程序,我们的算法几乎不能再进一步从数组中划分出更小粒度的对象。这些对象的大小可能超过 Cache 的大小,造成无法对 Cache 进行划分。因此我们对这些程序进行了修改,人为地将数组分割为多个小数组,再重新划分对象,得到结果列于表 2 中的"修正"一栏,短横线表示该程序不需要手工划分。

表 2 数据对象划分结果

Tab. 2 The partition results of data objects

程序		EP	IS	FT	CG	GEMM	Swim
对象数目	初始	1	8	128	64	144	512
	修正	32	64	–	–	–	–

标量数据对象的大小对 Cache 失效率有一定的影响。图 10 以 FT 为例,给出了设置不同标量数据对象大小对 Cache 失效率的影响。纵坐标是 Cache 总的失效率,不同的颜色分别代表数组和标量失效次数所占比例。FT 的标量数据主要是循环索引、临时变量和全局变量,总和不超过 256 字节(16 个 Cache 行)。从图 10 可以看出,当标量数据对象占用的 Cache 行较少时,失效率较高,其中标量部分的失效次数较多。当标量数据对象大小超过 16 个 Cache 行时,标量部分的失效次数很少并保持不变,数组部分的失效次数增加,并导致失效率增加。可见标量数据对象大小在 12 个 Cache 行左右时是一个较好的平衡点。在后面的实验中,各程序均选取类似的平衡点附近的值作为标量数据对象大小。

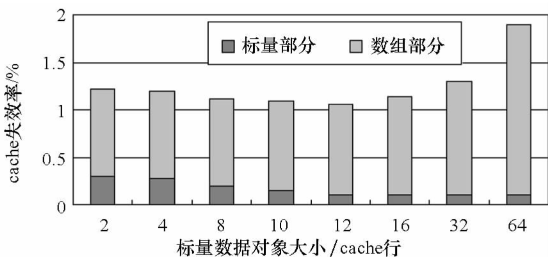


图 10 在不同标量数据对象大小下 FT 程序的 Cache 失效率变化

Fig. 10 The Cache miss rate of FT under various scalar data object size

我们固定 Cache 大小为 16KB,考察 Cache Coloring、LRU 以及 FFU 替换策略下产生的 Cache 失效率。FFU 每次替换将在未来最远使用的 Cache 行,它是理论上的最优替换策略[14],LRU 实际上是在一个局部时间范围内对 FFU 的逼近。FFU 策略下的结果并不是实测得到,而是获得访存序列后通过软件分析得到。图 11 给出了各个程序在三种算法下的 Cache 失效率。可见,对于 FT、GEMM 和 Swim 等数据局部性较好的程序,Cache Coloring 得到的失效率低于 LRU,并与 FFU 接近。说明 Cache Coloring 能够更好地开发这些程序的数据局部性。对于 EP 和 IS,三种算法下的失效率差不多,因为这些程序中能够开发的局部性非常有限。CG 的不规则访存给局部性开发带来一定困难,使得 Cache Coloring 相对 LRU 的提高很小。

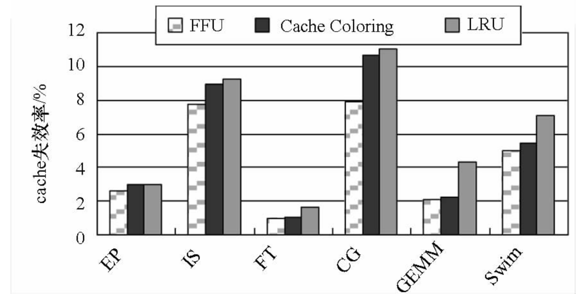


图 11 Cache 大小为 16KB 时,Cache Coloring、LRU 和 FFU 的失效率比较

Fig. 11 The miss rate comparison of Cache Coloring, LRU and FFU with a Cache size of 16KB

图 12 给出了 EP、CG、GEMM 和 Swim 等程序在不同 Cache 大小下,三种算法的失效率比较。除了 EP 以外,其他程序在三种算法下的失效率都随 Cache 增大而降低,Cache Coloring 的失效率始终保持接近 FFU 的水平。EP 产生的失效大多是强制失效,与 Cache 的大小无关。图 12(b)显示对于 CG,Cache Coloring 与 FFU 还存在一定的差距,原因如下:CG 主要的运算是稀疏矩阵乘,访问次数最多的数组是表示稀疏矩阵的数组,由于稀疏矩阵的特殊表示方式,对这些数组的访问大多是间接索引访问,跨越度往往比较大且没有规律,因此导致数组访问频繁失效,使其与理想情况下的 FFU 存在较大差距。

以上实验结果说明,Cache Coloring 能较好地开发程序的局部性,降低 Cache 失效率。

4 结论

本文提出了一个基于图着色的 Cache 优化方法——Cache Coloring,它的基本思想是将程序中

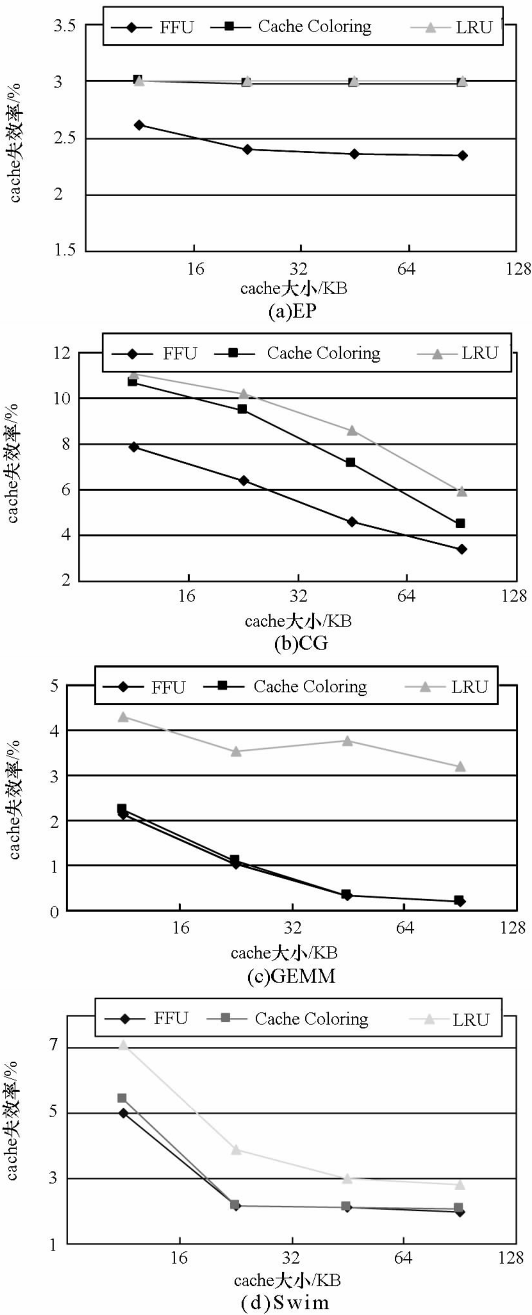


图 12　EP、CG、GEMM 和 Swim 在不同 Cache 大小下的失效率比较

Fig. 12　The miss rate of EP, CG, GEMM and Swim under various Cache sizes

的数据划分成若干数据对象,然后根据对象的大小将 Cache 划分为一个伪寄存器文件,最后使用一个经过改进的图着色寄存器分配算法来决定这些对象在 Cache 中的位置。在划分数据对象时,我们使用一个压缩算法来对访存地址序列进行分析,将经常重复出现的数据块作为一个对象。Cache 经过多次划分,被视为一个存在别名的伪寄存器文件。根据 Cache 访问的特点,我们对一个已有的支持别名寄存器的图着色寄存器分配算法进行改进,提高 Cache 分配的效率。为了进行评测,我们在 GCC 中实现了 Cache Coloring 方法,并通过 simplescalar 构造了支持 Cache Coloring 的 Cache 硬件模拟平台。经过实验测试,Cache Coloring 得到的 Cache 命中率高于传统的 LRU 算法,接近理论上的最优替换策略。这说明 Cache Coloring 能较好地开发程序的局部性,降低 Cache 失效率。

参考文献(References)

[1] Unsal O S, Ashok R, Koren I, et al, Cool-cache for hot multimedia[C]//MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society, 2001, 274 – 283.

[2] Yang C L, Tseng H W, Ho C C, et al. Software-controlled cache architecture for energy efficiency[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2005, 15(5): 634 – 644.

[3] May C, Silha E, Simpson R, et al. The powerPC architecture: a specification for a new family of RISC processors[M]. Morgan Kaufmann Publishers, Inc., 1994.

[4] Microsystems S, UltraSparc User's Manual[R]. 1997.

[5] Kessler R, The alpha 21264 microprocessor: Out-Of-Order execution at 600 Mhz[C]//Proceedings of the 10th Hot Chips, 1998.

[6] Intel, IA – 64 application developer's architecture guide [R]. 1999.

[7] Eichenberger A E, O'Brien K, et al. Optimizing compiler for the CELL processor[C]//PACT 05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, 2005, 161 – 172.

[8] Eichenberger A E, O'Brien K, et al. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture[J]. IBM Systems Journal, 2006, 45(1): 59 – 84.

[9] Patterson D A, Hennessy J L. Computer architecture: a quantitative approach [M]. 2nd ed. China Machine Press, 1999.

[10] Smith M D, Ramsey N, Holloway G. A generalized algorithm for graph-coloring register allocation [C]//PLDI'04: Proceedings of the ACM SIGPLAN 2004 conference on programming language design and implementation, ACM Press, 2004, 277 – 288.

[11] Nevill-Manning C G, Witten I H. Linear-time, incremental hierarchy inference for compression[C]//Proceedings of the Data Compression Conference, 1997.

[12] Lian L, Gao L, Xue J. Memory coloring: a compiler approach for scratchpad memory management [C]//PACT ´05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, 2005, 329 – 338.

[13] Appel A W, Modern compiler implementation in C[M]. Cambridge University Press, 1998.

[14] Belady L A, A study of replacement algorithms for a virtual-storage computer[J]. IBM Systems Journal, 1966, 5(2): 78 – 101.

[15] 邓宇. 基于图着色的存储层次优化技术研究[D]. 长沙:国防科技大学,2008.
DENG Yu. Research on graph coloring based optimization technologies for memory hierarchies[D]. Changsha: National University of Defense Technology, 2008.