doi:10.3772/j.issn.1002-0470.2016.05.003

基于区域协作的 Cache 压缩①

曾 露②[*,**,***] 李 鹏[*,**,***] 王焕东[****]

([*]计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)

([**]中国科学院计算技术研究所 北京 100190)

([***]中国科学院大学 北京 100049)

([****]龙芯中科技术有限公司 北京 100190)

摘 要 为提高 Cache 的有效容量,进行了 Cache 压缩研究,并提出了一种区域协作压缩(RCC)方法,以提升最后一级缓存的压缩率。与传统的 Cache 压缩算法不同,RCC 方法利用了缓存区域的压缩局部性,使用缓存区域中第一个缓存块的字典信息来协作压缩缓存区域中的其他各个缓存块,而不需要对缓存区域进行整体压缩。RCC 有效发掘了缓存区域内缓存块之间的数据冗余,实现了接近以缓存区域为压缩粒度的字典压缩的压缩率,然而压缩、解压缩延时却仍然和压缩单个缓存块时相当。实验结果表明,与单缓存块压缩算法 C-PACK 相比,RCC 方法的压缩率平均提升了12.34%,系统的性能提升了 5%。与 2 倍容量的非压缩 Cache 相比,有效容量提升了 27%,系统性能提升了8.6%,而面积却减少了63.1%。

关键词 数据压缩, 字典压缩, 区域协作压缩(RCC), 高速缓存压缩, 访存优化

0 引 言

长期以来,处理器运算性能的快速提高与访存带宽的缓慢增长之间一直存在着剪刀差,而且这种现象正愈演愈烈。计算性能与访存性能的不平衡导致了系统整体性能的下降,即使有强大的计算能力也没有办法高效利用,大量的时间被浪费在等待数据从存储系统中加载到流水线中。因此,大量的研究工作致力于访存系统的优化,提高访存的带宽,降低访存的延时。

由于半导体工艺的进步,片内可以集成更大的高速缓存(Cache)。增强动态随机存取存储器(eDRAM)技术的成熟使得动态随机存取存储器(DRAM)的高集成度技术被引用到 Cache 的设计中,进一步增加了 Cache 的容量。现代的计算机应用通常拥有超大的工作集,提供更大的 Cache 容量总是能够带来更好的程序运行性能。在体系结构层面上探索更高的 Cache 利用效率与半导体工艺追求更高的片内容量并不矛盾,反而两者相辅相成,共同为性能的提升做出贡献。然而 Cache 的设计面临着面积、功耗与延时之间的权衡取舍。通常接近处理器核访存部件的缓存需要更小的延时,从而减少流水线的空等待。为保证延时,其容量不能做到太大。而远离处理器核的缓存层次对访存延时较为不敏感,通常可以设计更大,然而更大的面积的 Cache 却面临着静态功耗和翻转功耗的问题。

数据压缩是降低数据存储空间的有效手段。而在片上 Cache 中利用数据压缩的技术理论上可以使得固定大小的 Cache 存储更多数据,即可以提升

① 国家"核高基"科技重大专项课题(2014ZX01020201, 2014ZX01030101),国家自然科学基金(61232009, 61432016)和 863 计划(2013AA014301)资助项目。

② 男,1987 年生,博士生;研究方向:计算机系统结构;联系人,E-mail: zenglu@ict.ac.cn

(收稿日期:2016-01-18)

Cache 的有效容量。已有研究表明,压缩缓存的逻辑容量可以达到其物理存储空间的一倍以上,而代价仅仅是略微增加了访问延时(主要是从压缩 Cache 中读取时解压缩的延时)和少量的 tag 位等元数据。由此可见压缩缓存的好处是显而易见的,尤其是对访问延时相对不敏感的最后一级缓存(Last Level Cache,LLC),增加几个时钟周期的访问延时对于整个访存系统的性能影响很小。因此,大量的研究[1-5]通过实现高效管理压缩数据的 Cache 结构和适配于 Cache 的数据压缩/解压缩算法以实现更高的压缩率、更低的访问延时、功耗与面积开销以及更高的 Cache 性能。因此,数据压缩在提升 Cache 尤其是最后一级缓存(LLC)的有效容量,控制设计大容量 Cache 带来的功耗,最终有效提升系统性能上具有很大的意义。本文将探索在 LLC 中实现高效数据压缩的方法。

1　Cache 压缩

1.1　Cache 压缩的挑战

传统数据压缩算法经过几十年的研究,已经发展的较为成熟。诸如 LZ77[6] 算法以及在其基础上衍生的 LZXX 算法簇,和基于统计信息编码的 Huffman 算法在某些情形下甚至能够获得接近信息熵的压缩率。这些算法及其衍生算法被广泛应用于压缩软件中对计算机数据进行压缩,如 DEFLATE。然而针对 Cache 数据的压缩,则面临与传统数据压缩不同的挑战。

首先,硬件实现压缩算法要在取得一定压缩率的同时,需要兼顾压缩算法的实现复杂度以及压缩、解压缩的速度。过于复杂的压缩算法的硬件实现成本和复杂度更高,且延时和面积开销甚至使得数据压缩得不偿失;而简单的算法虽然高效快速,但是压缩率通常较低。另外,缓存的读操作是访存的关键路径,解压缩延时对系统性能的影响大,因此压缩算法还需要具有较短的解压缩延时。

其次,压缩数据的存储结构不同。通常缓存都是多路组相联结构,连续的缓存块被映射到不同的组(set),压缩算法的压缩粒度难以扩展到更大的数据范围,通常仅以缓存块为单位进行压缩。而软件压缩算法可以在一个较长的数据窗口中发掘冗余数据,甚至可以遍历数据来统计频率信息,使用占用空间最小的编码方式。

最后,缓存的数据是动态变化的,数据的修改需要将数据重新压缩。新数据与旧数据的压缩大小往往不同:如果压缩数据变小,Cache 中会空出小块的存储空间,而它们往往不足以存储新的缓存块,这就导致了碎片化;如果压缩数据变大,还需要在组内额外分配空间,如果空间不足,还会导致缓存块的替换,进一步增加了设计的复杂度。

因此,Cache 压缩需要综合考虑压缩率、解压缩延时、面积开销和实现复杂度等因素。仅侧重某一方面,而其他方面开销大,最终性能可能不好,甚至更差。

1.2　Cache 压缩算法

传统数据压缩算法可以采用更大的数据块和更复杂的编码方法,而缓存压缩算法由于需要同时满足较低的硬件实现复杂度和较好的压缩率以及压缩、解压缩延时,因此,Cache 压缩算法通常以缓存块为数据压缩单位,并对有限的常见数据模式进行编码。

零值在 Cache 中所占比例较大,在某些应用中,零值在 Cache 中甚至可以占到 40% 以上的空间。零内容增强高速缓存(ZCA)[7] 使用一个额外的缓存记录 Cache 中为 0 的缓存块的地址,而不需要在 Cache 中存储零值,从而实现基本的压缩。然而,由于仅能够压缩零缓存块,ZCA 受限于特定的应用,并不广泛适用。

然而,含有零值的缓存块仍广泛存在于 Cache 中,几乎所有的 Cache 压缩算法都会对零值进行优先压缩,使其占用最少的空间。

常见模式压缩(frequent pattern compression,FPC[8])将几种较为常见的模式进行定长编码,如连续的 0、小值数据或重复值等共分为 8 类通过 3bit 的前缀来表示。据此将缓存行以数据字(32bit)为单位划分,依次根据对应的数据模式进行匹配压缩,生成相应的前缀和压缩数据。

C-PACK[2] 算法(一种高性能微处理器高速缓

存压缩算法)首先采用和常见模式压缩(FPC)类似的常见模式匹配,直接匹配压缩 0 值和小值数据。对于不能匹配的数据,引入了字典压缩。压缩时所有未匹配的数据将插入字典,后面的数据字除匹配 0 与小值数据外,同时与字典中的项进行匹配。字典项支持部分匹配,即匹配拥有相同高字节而低字节不同的数据,仅存储低字节不同的部分,从而实现相似数据的压缩。如表 1 所示,数据压缩以 4 字节为单位,除 0、小值数据和无法压缩的情况为固定模式压缩,其他三项为字典匹配,分别表示完全匹配,匹配高两字节和匹配高三字节。由于字典的引入,大量相似的数据可以被压缩,依次可以实现不错的压缩率。C-PACK 虽然增加了压缩、解压缩延时,但对于延时相对不敏感的最后一级缓存(LLC),提高的数据压缩率相对带来的好处更大。

表 1 C-PACK 编码表

前缀	压缩模式	压缩后大小
00	0(zzzz)	2 比特
01	无法压缩(xxxx)	34 比特
10	字典匹配(mmmm)	6 比特
1100	字典匹配(mmxx)	24 比特
1101	小值数据(zzzx)	12 比特
1110	字典匹配(mmmx)	16 比特

注:压缩模式中每个字母代表一个字节,z 表示为 0,x 表示当前字节未匹配,m 表示当前字节匹配了字典项。压缩后大小为使用 16 项字典时的值。

BDI(Base-Delta-Immediate)压缩[3]认为,大量 Cache 行中的数据拥有低动态范围(low dynamic range)的数据特性,即数据字之间的差值通常比这些数据本身的值要小。BDI 为缓存块中固定大小(如 2,4,8 字节)的数据字确定公共基值,而数据字可由占用较小空间的偏移值来表示。缓存块的数据可以被压缩为一个公共基值和若干个数据字的偏移值的形式。BDI 要求缓存块中所有数据字都必须压缩成相同大小的偏移值,因此在解压缩时所有的偏移值可以直接并行读取并与公共基值进行运算完成解压缩,可以实现 2 个时钟周期的解压缩延时。BDI 的公共基值 + 偏移量的方式和 C-PACK 中字典项的部分匹配拥有一定相似性,不过由于 BDI 的基值数量固定,对于数值分散度较大的缓存块压缩效果较差,因此其压缩率不如 C-PACK。BDI 的一个改进策略是额外增加 0 作为公共基值,首先以 0 为基值计算偏移值优先过滤小值数据,再确定剩余数据字的公共基值进行压缩。

FVC (frequent value compression, 常见值压缩)[5]通过预先计算出的常见值来配置常见值表。在程序执行的过程中,该字典的内容不变,对匹配的缓存数据进行压缩。FVC 能够对程序执行过程中最常见的值进行压缩。然而,通常程序的执行随着时间的变化,存储在 Cache 中最多的值通常会发生变化,因此使用固定的值作为字典进行压缩不如动态字典的压缩效率高。

SC2(一个统计压缩缓存方案)[4]使用定期采样的统计信息进行 Huffman 编码来进行数据压缩。Huffman 编码的生成需要对数据进行采样来生成统计信息,而由于缓存中的数据在程序执行时是动态变化的,该算法需要每隔一段时间定期对缓存数据进行重新采样,更新 Huffman 编码,使用新的编码进行数据压缩。SC2 需要使用一块较大的 RAM 来存储采样的统计信息,在面积上不具有优势,但 SC2 能够实现较高的压缩率。因此,该方案在适用性上有较大限制,需要考虑到其实现的面积开销与复杂度。

常见的 Cache 设计中指令和数据在 LLC 中不加区分的存储,Cache 并不记录某个地址是指令还是数据,并且有时指令还可以被当作数据动态地进行修改。然而指令和数据拥有不同的压缩特性。指令在编码时已经考虑到存储的效率,因此指令本身的信息熵已经很大,其压缩性不足。然而,处理器运行所需的指令类型在指令集中并非均匀分布,大多数时间仅运行了少数类型的指令,而这些指令在 Cache 中冗余度很高,对于其中出现频率较高的指令进行编码压缩,可以较好地压缩指令所需的存储空间。Benini 等在文献[9]中探索了指令压缩的方法。

1.3 压缩 Cache 的结构

Cache 压缩算法决定了数据最大可以被压缩的

大小。而有效的压缩 Cache 结构为压缩算法实现数据压缩提供了基础。如果 Cache 组织不合理,要么限制了最大的压缩率,要么需要消耗大量的元数据(额外的压缩信息,用于索引压缩缓存子块的指针等)。因此,设计合理有效的压缩 Cache 结构才能配合 Cache 压缩算法实现高效快速的 Cache 压缩。

可压缩 Cache 的结构需要解决压缩缓存块的存储与索引、压缩缓存块的修改与替换和压缩策略的选择的问题,接下来将分别进行说明。

1.3.1　压缩缓存块的存储与索引

由于压缩后的缓存块大小不一,且压缩 Cache 的 tag 数量需要大于未压缩的数据块数量以存储压缩数据,传统 Cache 的 tag 与 data 一一映射的方式不适合压缩 Cache。压缩 Cache 通常使用解耦(decoupled)方式进行 tag 与 data 的映射。

VSC[8]使用 tag 中的 size 字段来表示压缩的缓存块所占用的段(segment,4 字节)数目。VSC 的缓存块的偏移地址通过累加组内该缓存块之前所有缓存块的 size 字段得到。第 k 路缓存块的偏移地址为第 1 路至第 $k-1$ 路缓存块的 size 字段的和:

$$\text{offset}(k) = \sum_{i=1}^{k-1} \text{size}(i) \tag{1}$$

缓存块的偏移和大小为 offset(k)和 size(k)。

SCC(skewed compressed cache,偏移压缩缓存)[10]根据相邻数据的压缩率相近的特性,将连续地址的压缩数据块集中存储在一个物理缓存块中,使用一个 tag 来表示超级缓存块。根据缓存块压缩率的不同,tag 可表示 1,2,4,8 个缓存块。然而,SCC 针对大缓存块中压缩率不同的缓存块需要通过偏移映射的方式单独存储。SCC 所需的元数据较少,不需要提供额外的 tag,即可支持较高的压缩率。

Pair-matching[2]限制一个物理缓存块最多保存两个压缩缓存块,在 tag 与 data 之间维持了二对一的固定映射,避免了压缩缓存块的索引开销。然而,固定的映射决定了其理想情形下 2 倍压缩率的上限,而且两个压缩缓存块存储在一个物理块内的限制决定了它放弃了大量的压缩机会。

1.3.2　压缩缓存块的修改与替换

在系统运行过程中缓存块的内容会发生变化,而对修改数据进行重新压缩后的大小也会发生变化。如果变小,那么剩余出来的空间需要通过执行紧缩(compaction)或重映射来回收以避免浪费;如果变大,则需要占用额外的空闲空间,甚至可能需要替换掉现有的一个或多个缓存块。

Cache 的紧缩需要读写组内大部分的数据,因此该操作代价巨大。即使该过程可以与读写关键路径并行,其巨大的功耗与复杂逻辑开销仍然使得代价过大。SCC 和 Pair-matching 均对缓存块的大小有限制,不允许任意大小的缓存块,因此避免了复杂的紧缩操作。

压缩 Cache 的替换策略除考虑时间局部性因素外,还需要考虑缓存块自身的压缩大小,使用最少的替换次数满足插入缓存块的空间需求,以及考虑缓存的存储布局,选择合适的替换块避免碎片的产生导致需要紧缩。ECM(effective capacity maximizer)[11]提出了大小可感知的压缩缓存替换策略,同时利用最不常用(least recently used, LRU)信息和压缩信息大小决定替换策略,保留尽可能多并且被重复利用的缓存块,从而从另一个角度提升了系统的性能。

1.3.3　压缩策略的选择。

Cache 压缩算法通常都不能普遍适用于所有情形。对于不同的应用,不同的数据类型以及不同的执行时间点,数据的压缩特性都有不同。有些应用的工作集较小,数据压缩带来的额外空间没有意义,反而产生了额外的延时,Alaa[8]提出了动态方式判断应用是否得益于压缩来决定是否压缩数据。Nitta[12]根据缓存行中超过 50% 数据的数据类型选择是否压缩、如何压缩,如针对整形、浮点和指针类型,采用不同的压缩算法。

2　区域协作压缩

2.1　Cache 压缩的粒度对压缩率的影响

尽管固定大小的缓存块可以使得 Cache 管理相对简单而且高效,使用更大粒度的缓存数据进行压缩通常可以获得更高的压缩率。应用程序在访存中大多呈现较强的空间局部,即地址相近的数据总会被相继访问。尤其是具有流式访存行为的应用(如

applu, equake, ocean, radiosity, raytrace 等),它们会连续访问大片的数据,且由于程序特性,连续访问的相邻数据拥有相似的特性,如拥有相同的高字节,而仅仅是低字节不同,或者是完全相同的数据。这类数据通常不仅仅存在于一个缓存行内部,而通常存在于多个缓存块范围,从而产生了区域压缩(region compression)的概念。

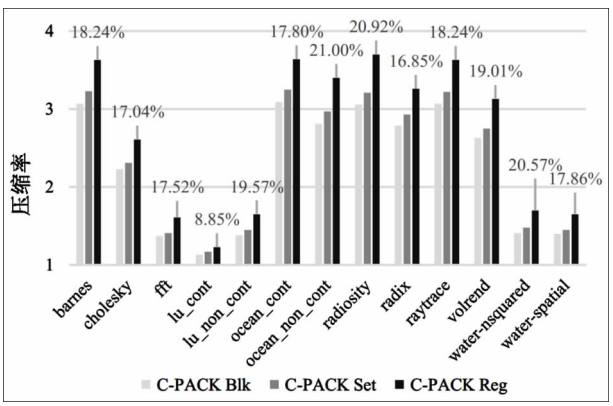

图 1 C-PACK 压缩算法在块粒度和区域粒度下压缩率对比

如图 1 所示,以 C-PACK 压缩算法为基础,分别为使用缓存块为粒度进行压缩(C-PACK Blk),使用 16 路组相联的组为粒度进行压缩(C-PACK Set)和使用 16 个连续地址的缓存区域进行压缩(C-PACK Reg)在各个应用程序下压缩率的对比。可以观察到 C-PACK Reg 方式对压缩率有较大的提升,最高可达 21%,平均也有 17.96% 的压缩率提升;而同样为 16 个缓存块为粒度的 C-PACK Set 方式则提升很小,平均在 5% 以下。

因此,对更大的数据块进行压缩,可以获得比单独压缩一个缓存块更好的压缩性,特别是使用连续地址进行压缩,比在一个组内相对随意地址的多个缓存块压缩,具有更高的压缩率,从而反映了数据压缩的局部特性。本文认为,数据压缩局部性是除时间局部性、空间局部性之外,访存系统特别是可压缩访存系统需要深入挖掘的第三种局部性。

然而,现有的 Cache 压缩算法大都使用缓存行为单位进行数据压缩,因为这样可以在不改变现有 Cache 架构的情形下实现数据压缩,而不引入额外的复杂性。在类似常见模式压缩(FPC)这种固定模式的压缩算法下,不同的压缩数据的粒度对压缩率没有影响;而在基于字典的压缩算法(如 C-PACK, BDI)中,被压缩的数据粒度越大,可以发掘的数据冗余越多。而基于单一缓存行的压缩算法不能有效压缩跨多个缓存行的冗余数据,需要在每个缓存行中单独进行存储字典项,从而造成了 Cache 空间的浪费。

利用数据压缩局部性压缩多个连续缓存块可以提升可压缩 Cache 的压缩率,然而该技术面临如下挑战:

首先,由于缓存行中替换和修改数据块是比较频繁的,且替换和修改需要对数据重新进行压缩、解压缩。如果以较大的数据块为单位进行整体压缩和解压缩,其代价显然过大,在结构设计中应当避免。Cache 的压缩和解压缩多个缓存行会带来较大功耗与延时开销,然而可以借助其他缓存块的压缩信息以协助当前缓存行的压缩和解压缩。

其次,连续缓存块通常被映射到不同的 set,通常难以通过一次访问获取多个连续的缓存块数据或者压缩信息。在设计中应综合考虑压缩性与延时的平衡:获取更多连续缓存块的信息可以实现更高的压缩率,然而需要更多的访问延时。需要在保证一定压缩率的提升下尽量减少对其他缓存块的访问。

2.2 区域协作的压缩算法

本文提出了一种区域协作压缩(region cooperative compression, RCC)算法,该算法利用了缓存的数据压缩局部性,可以以较小的代价获得比以缓存块粒度进行字典压缩更高的压缩率。

区域协作压缩(RCC)的原理是利用缓存区域内第一个缓存块(first block in region, FBR)压缩时所生成的字典项来协助压缩区域内后续缓存块(successive block in region, SBR)的数据,由于缓存的压缩局部性,该方法可以近似达到对整个缓存区域共同使用一个字典进行压缩的压缩率。

RCC 同时兼顾了延时、功耗与压缩率。首先,缓存块的压缩、解压缩仅针对当前命中的缓存块,不会修改组内的其他缓存块,而需要额外读取的 FBR 字典项可以通过多路命中的方式覆盖延时,因此延时并没有增加;其次,缓存块在压缩、解压缩时仅额

外读取了 FBR 的字典项,带来少量的功耗开销;最后,由于 FBR 的协作压缩,SBR 能够利用 FBR 字典项进行压缩,其压缩率可以更高。

2.2.1　压缩

RCC 过程以 C-PACK 算法为基础,所不同的是在压缩 FBR 和 SBR 时对字典的不同处理过程。

FBR 的压缩是一个独立的过程,和 C-PACK 一样,根据读取到的数据字生成字典进行后续数据字的压缩。FBR 作为协作者,如果插入 Cache 时已有该缓存区域中的其他块存在,则 SBR 的压缩有所不同,在压缩前如果在 Cache 中存在 FBR,则需要将 FBR 的字典项读取到压缩器的字典中。如图 2 所示,在执行压缩时,若存在 FBR,其字典项会被读取到字典中。如果命中,则直接使用该字典项进行压缩;如果没有命中,处理方式与 C-PACK 相同。

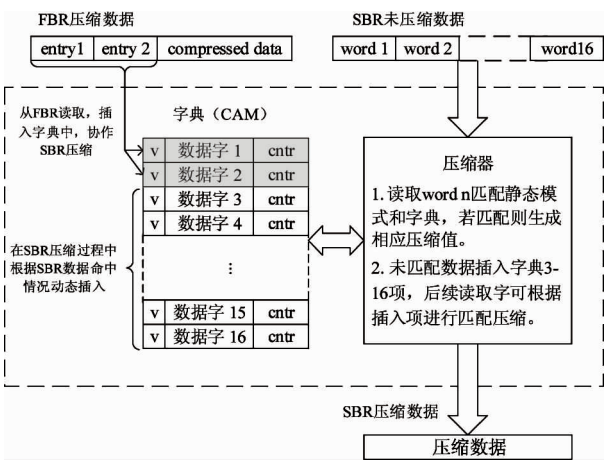

图 2　RCC 器的字典

由于 FBR 的字典项需要在 SBR 压缩时使用,FBR 的压缩方式与 SBR 有所不同。FBR 的压缩仅针对自身,不需要从其他缓存块读取字典项。然而,由于 FBR 需要在 SBR 进行压缩时被快速访问,因此,FBR 在压缩时,选择字典中计数器值最大的两个字典项放置在数据块的头部,在读取 SBR 字典时,仅读取数据头的 8 个字节作为 SBR 字典项。

2.2.2　解压缩

解压缩的过程与 C-PACK 相同,只是在为 SBR 进行解压缩前需要读取 FBR 的字典项。然而,如果该过程顺序执行,将增加额外的延时。解压缩过程是访存的关键路径,解压缩延时将直接影响 Cache 的访问延时,从而影响性能。为此,当读取 SBR 时,FBR 字典项的读取需要与 SBR 同步进行以避免增加额外的延时。

为实现 SBR 与 FBR 的压缩信息同步读取,消除额外的等待延时,需要 Cache 支持多路命中机制。多路命中的意思是在 Cache 访问 SBR 时,同时命中 FBR(若存在),使得两者的读取可以同步进行。由于 cache 的每一路都是独立的 RAM,在访问 Cache 时,可以使得部分路访问 SBR 所映射组的索引,另一部分路访问 FBR 所映射组的索引。通过在分配时通过地址哈希函数确定 FBR 和 SBR 各自允许存放的路数,使得不同缓存区域的 FBR 和 SBR 的路映射不同,从而在整个 Cache 范围内消除了路限制带来的潜在冲突。映射为 FBR 的路的 tag 与 FBR 的 tag 比较,映射为 SBR 的路与 SBR 的 tag 比较,最终根据各自的命中情况读取 FBR 的压缩信息和 SBR 的数据。

定义 $W = \{w_1, w_2, \cdots, w_n\}$ 为所有路的集合,W_{FBR} 和 W_{SBR} 分别表示 FBR 和 SBR 的候选可分配的 way,且定义 $w_k = w_{k-1} + 1$, $w_1 = w_n + 1$。那么有:

$$W_{FBR} = \{w_1 + h, w_2 + h, \cdots, w_{n/2} + h\} \quad (2)$$

其中 $h = hash(addr[39:10])$ 且 $0 < h \leqslant n$

$$W_{SBR} = W - W_{FBR} \quad (3)$$

W_{FBR} 和 W_{SBR} 的值根据地址 10-40 位的哈希运算得到,因此不同缓存区域,FBR 和 SBR 所能分配的候选路不同。而在命中查找时,根据 hash 运算得到相应的 W_{FBR} 和 W_{SBR},然后再根据 FBR 和 SBR 的索引分别对 W_{FBR} 和 W_{SBR} 的路进行访问,即可实现一次访问,同时命中 FBR 和 SBR。

2.3　Cache 组织结构

RCC 采用基于 VSC[8] 的压缩 Cache 结构,tag 的数量为 data 的 4 倍,以最高支持 4 倍压缩率。

图 3 所示为 RCC 的 tag 字段。addr 标识缓存块地址,可据此判断该块是 FBR 还是 SBR;state 记录缓存块的状态和一致性信息;LRU保存访问历史以

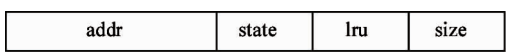

图 3　RCC 的 tag 字段

实现 LRU 替换算法；size 是该缓存块被压缩的大小，以 segment 为单位，所在块的偏移通过将其之前所有块的 size 相加得到。

对于拥有 8 个 64 字节缓存块的组，由于提供了 4 倍的 tag，每个组有 32 个 tag 用来保存压缩块的信息。由于 RCC 使用了多路命中的机制，FBR 仅会在其中 16 个 tag 中进行查找，SBR 则会在另外 16 个 tag 中进行查找。多路命中技术对两个地址分别进行 16 路的全相联比较，而不是各自进行 32 项的全相联比较，从而避免了额外的延时和功耗开销。

由于 SBR 的压缩利用了 FBR 的字典项，如果 FBR 发生替换，所有使用 FBR 字典项压缩的 SBR 都需要进行压缩数据的恢复，而恢复的代价很大。RCC 通过以下 3 种途径解决该问题：

（1）采用修改的 LRU 算法，发生替换时如果 LRU 队尾是 FBR，则向 LRU 队列前面查找，直到找到非 FBR 为止。如果不存在非 FBR 的缓存块，在 FBR 的 tag 中记录了被 SBR 使用的次数（usage count，UC，如图 3），优先选择 UC 为 0 的 FBR 进行替换。

（2）FBR 被充分哈希到不同的组，采用如下映射方式：index = *hash*(addr[39:10]) + addr[9:6]，使得不同缓存区域的 FBR 均等的映射到所有的组中，且相同缓存区域内的块被映射到不同的组。

（3）将被替换的 FBR 插入 victim buffer，直到 SBR 依次被替换，UC 降为 0 后 FBR 才被替换。

2.4 存储开销

表 2 列出了 RCC，C-PACK，2X Base 与 Base 存储空间需求比较结果。RCC 在比 Base 多 22.6% 的面积开销，对于 4MB 的 Cache，RCC 需要多消耗 992kB 的存储空间，以实现大于 2 倍的压缩率。不过，相比 2X Base，RCC 的面积开销大大降低，减少了63.1%。

表 2 RCC，C-PACK，2X Base 与 Base 存储空间需求比较
（Base 为 4MB 存储，8 路组相联，64B 缓存块）

	C-PACK	RCC	2X Base	Base
Tag (bit per block)	(30+2+4+4)×4=160		30+2+4=36	30+2+4=36
Data (bit per block)	512		512	512
物理缓存块数	64k		128k	64k
总计	5376kB		8768kB	4384kB
较 Base 增加存储空间	22.6%		100%	0

3 实验数据

3.1 实验平台

本文采用 MIT 大学发布的 Graphite 分布式多核模拟器[13]，Graphite 可以直接在主机上运行被模拟的程序，基于动态插桩的方式获取执行过程中的信息，模拟目标结构的运行机制，从而获取性能等参数。该模拟器的特性是系统开销小，可并行模拟多核结构，速度较快，但是由于它不是时钟精确类型的模拟器，对于模拟要求具有精确时钟信息的结构如流水线结构模拟准确性较差。然而该模拟器提供了完整的多处理器多路组相联 Cache 的行为模型和一致性模型，支持对压缩 Cache 的行为进行建模，因此该模拟器能够满足本文的实验要求。

本文主要研究数据压缩在 LLC 中的运用，以提升缓存的有效大小，从而降低其失效率最终实现性能的提升。而处理器核的微结构以及缓存一致性协议则不在考察的范围内，因此相关参数仅采用比较常用的设计参数。具体的配置如表 3 所示。

表 3 目标系统的参数配置

结构名称	参数配置
处理器核	按序发射，1GHz，4 核
L1 缓存	私有，64kB 数据，64kB 指令，4 路组相联，2 时钟周期，64 字节缓存块，未压缩
L2(LLC)缓存	共享，4MB，8 路组相联，4 倍 tag，20 时钟周期，64 字节缓存块，RCC 压缩
缓存一致性	MSI 协议，位向量目录
互连网络	Crossbar 结构，128bit 位宽，链路传输延时 2 时钟周期
内存	200 时钟周期

3.2 压缩率

如图 4 所示，与 C-PACK 相比，RCC 压缩率有较大提升。如 radiosity，lu_non_contiguous，water-

nsquared 的压缩率提升超过了 13%,对于所有测试程序,压缩率平均提升了12.34%,压缩率从2.2倍提升到2.54倍。实验结果说明了 RCC 对压缩率的提升是拥有普遍好处的,这是由 Cache 的空间局部性与压缩局部性决定的。虽然 Cache 仍然使用缓存块为粒度进行存储和压缩,但是其压缩率已经接近直接对缓存区域进行压缩,而代价仅仅是每次访问需要同时多读一个 tag 和 FBR 的字典项。

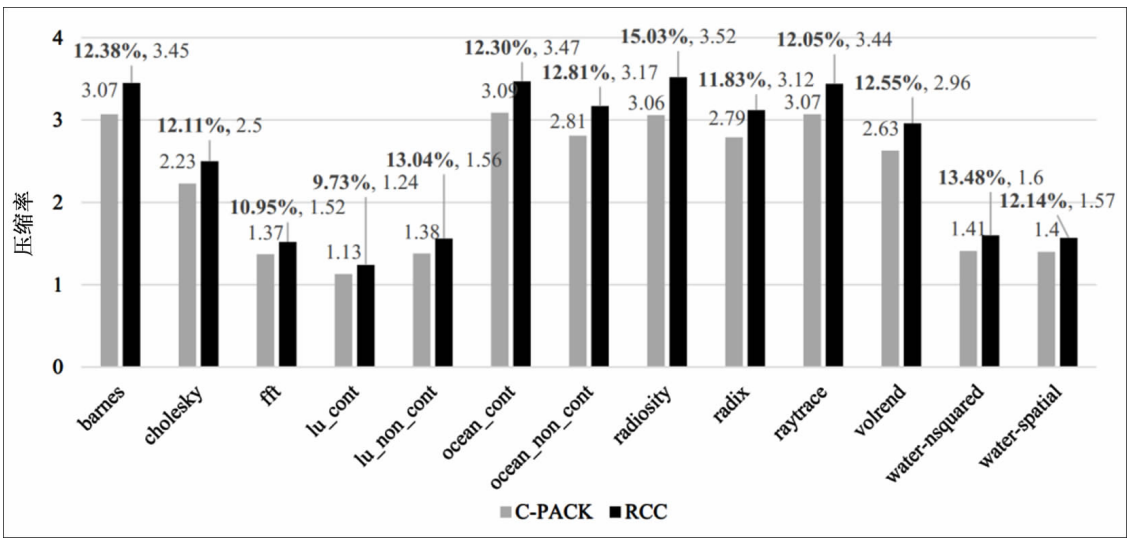

图 4　RCC 与 C-PACK 压缩率实验结果对比

3.3　性能分析

与 C-PACK 相比,RCC 需要在压缩 SBR 时同时读取 FBR 的字典项来协作压缩,如果该过程串行,至少需要增加 2 拍延时(SBR 需要等待 FBR 命中和字典项的读取),而多路命中很好地并行化了该过程。任意 SBR 可以和 FBR 同时被命中,FBR 字典项的读取延时刚好被 SBR 的读取延时覆盖,SBR 在压缩、解压缩时,FBR 的字典项已经在字典中就绪。因此,RCC 的延时与 C-PACK 相同。而且,实验结果表明,RCC 的缺失率相比 C-PACK 更低,这是由于 Cache 中存储了更多数据,有用数据被替换的概率降低。如图5所示均一化的IPC数据,RCC相比

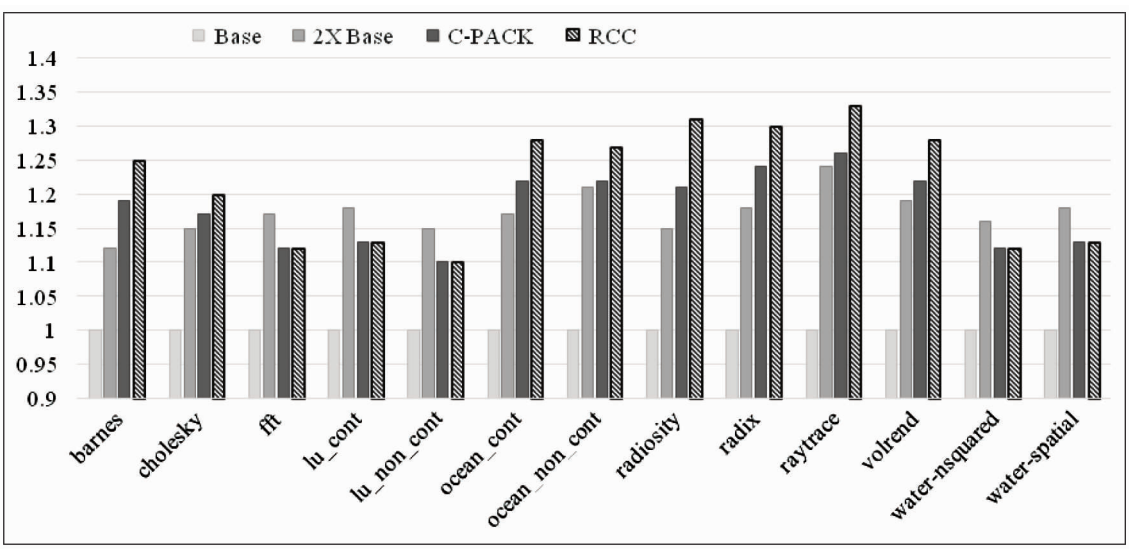

图 5　RCC, C-PACK, Base 和 2X Base 的均一化 IPC 数据

Base 均有 10% ~30% 不等的性能提升,而与 2X Base 相比,barnes, cholesky, ocean, radiosity, radix, raytrace, volrend 等程序均有 5% ~13% 不等的性能提升,而 fft, lu, water 等程序则有 3% 左右的性能下降,这是由于程序压缩率的不同导致。压缩率高的程序能带来更低的缺失率,从而从整体上增加 IPC;而压缩率较低的程序对数据压缩不敏感,反而由于解压缩等带来的开销降低了系统性能。这种问题可以通过动态关闭数据压缩来解决,且与本文的方案相容,可以共同提升系统的整体性能。

4 结 论

本文总结了目前压缩缓存的结构与算法研究进展,并基于数据在缓存中呈现的压缩局部特性,提出了区域协作的 Cache 压缩。相比 C-PACK 压缩,RCC 有明显的压缩率提升,同时没有带来额外的延时,因而带来了近乎“免费”的性能提升。另外,RCC 对于所有基于字典的 Cache 压缩算法和结构都具有适应性,而不仅限于特定的结构和算法。

未来的工作包括进一步的发掘数据压缩的粒度,与虚拟内存的特性结合,在页内寻找压缩的可能性;对缓存数据进行分类,对指令、浮点数据这类数据使用专用的压缩算法;以及在保证压缩率的同时,探索进一步降低解压缩延时的方法。

参考文献

[1] Alaa R A, David A W. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. In: Technical Report 1500, Computer Sciences Department, UW-Madison, 2004

[2] Chen X, Yang L, Dick R P, et al. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2010, 18(8): 1196-1208

[3] Pekhimenko G, Seshadri V, Mutlu O, et al. Base-delta-immediate compression: practical data compression for on-chip caches. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, Minneapolis, USA, 2012. 377-388

[4] Arelakis A, Stenstrom P. SC2: A statistical compression cache scheme. In: Proceedings of the 41st Annual International Symposium on Computer Architecuture, Minneapolis, USA, 2014. 145-156

[5] Yang J, Zhang Y, Gupta R. Frequent value compression in data caches. In: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture, Monterey, USA, 2000. 258-265

[6] Ziv J, Lempel A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 1977, 23(3): 337-343

[7] Dusser J, Piquet T, Seznec A. Zero-content augmented caches. In: Proceedings of the 23rd International Conference on Supercomputing, Yorktown, Heights, USA, 2009. 46-55

[8] Alaa R A, David A W. Adaptive cache compression for high-performance processors. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, Munich, Germany, 2004. 212-223

[9] Benini L, Macii A, Macii E, et al. Selective instruction compression for memory energy reduction in embedded systems. In: Proceedings of the 1999 International Symposium on Low Power Electronics and Design, San Diego, USA, 1999. 206-211

[10] Sardashti S, Seznec A, Wood D. Skewed compressed caches. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 2014. 331-342

[11] Baek S, Lee H G, Nicopoulos C, et al. ECM: Effective capacity maximizer for high-performance compressed caching. In: Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture, Shenzhen, China, 2013. 131-142

[12] Nitta C, Farrens M. Techniques for increasing effective data bandwidth. In: Proceedings of the IEEE International Conference on Computer Design, Lake Tahoe, USA, 2008. 514-519

[13] Miller J E, Kasture H, Kurian G, et al. Graphite: A distributed parallel simulator for multicores. In: Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture, Bangarlore, India, 2010. 1-12

The Cache compression based on region cooperation

Zeng Lu[*,**,***], Li Peng[*,**,***], Wang Huandong[****]

([*] State Key Laboratory of Computer Architecture (Institute of Computing Technology, Chinese Academy of Science), Beijing 100190)

([**] Institute of Computing Technology, Chinese Academy of Science, Beijing 100190)

([***] University of Chinese Academy of Science, Beijing 100049)

([****] Loongson Technology Corporation Limited, Beijing 100190)

Abstract

The Cache compression was studied to increase Cache's effective capacity, and a region cooperative compression (RCC) algorithm was proposed to improve the compression ratio of the last level Cache. Different to traditional Cache compression algorithms, the RCC algorithm exploits the compression locality to compress Cache blocks in a Cache region by the cooperation of the first block in the region, instead of compressing the whole Cache region. RCC effectively explores the duplications across the Cache blocks in a Cache region and shows a comparable compression ratio with dictionary compression approaches with the whole Cache region as the compression granularity, whereas the (de)compression latency is not increased. The experimental results show that RCC provides the better average compression ratio than the compression algorithm of C-PACK by 12.34%, which causes the performance improvement of 5%. Compared to the non-compressive Cache with double size, the effective capacity increases by 27%, the performance increases by 8.6% and the area decreases by 63.1%.

Key words: data compression, dictionary compression, region cooperative compression (RCC), Cache compression, memory access optimization