# Cache Compression

qishuo hua

2025.1

**摘要**

随着现代计算机体系结构的快速发展，缓存系统的性能已经成为影响处理器整体性能的关键因素之一。为了提高缓存的有效性并降低存储需求，缓存压缩技术逐渐成为优化缓存管理的有效手段。本文探讨了三种常见的缓存压缩方法：频繁值压缩（FVC）、频率优先压缩（FPC）和基准增量压缩（BDI），并详细分析了它们在提高缓存存储密度、减少访问延迟以及提升系统性能方面的优缺点。

频繁值压缩方法主要通过对缓存中出现频率较高的数据进行压缩，利用这些频繁出现的值来减少冗余数据的存储需求，该方法在处理重复数据密集型应用时表现尤为突出。频率优先压缩方法则采用了一种更加细粒度的策略，通过对缓存中每个数据的访问频率进行排序，并优先压缩那些频繁访问的数据。FPC 方法特别适用于访问模式高度动态变化的应用场景。基准增量压缩方法则通过将缓存中的数据分解为基准值和增量值的形式来压缩数据。此方法适用于数据之间有较强的连续性或相似性的情况。此外，本文还探讨了这些压缩方法在实现中的挑战，如计算开销、硬件复杂度以及在不同缓存体系中的适应性等问题，对 FVC、FPC 和 BDI 三种压缩方法进行了尽可能全面的评估。

# 1 Frequent Value Compression（频繁值压缩）

## 1.1 背景与动机

FVC（Frequency Variability Coding）的背景动机源于优化数据存储和传输的效率，尤其是在包含大量频繁值的数据中。在许多信号处理和数据压缩应用中，数据可能包含大量重复或相似的值，通过对这些频繁出现的值进行压缩，可以显著减少存储空间和传输带宽的需求。此外，FVC 方法可以根据数据的频率分布动态调整编码策略，从而提高压缩效率，降低冗余数据的占用，提升系统整体性能。这种方法适应性强，广泛应用于数据流、图像处理和音频信号等领域。

芯片多处理器（CMP）的普及导致了大型片上高速缓存的集成。出于可扩展性的原因，大型片上缓存通常被划分为较小的 bank，这些 bank 通过基于数据包的片上网

络（NoC）互连。随着单个芯片上集成的内核和高速缓存库数量的增加，片上网络会带来显著的通信延迟和功耗。频繁值压缩可以优化 NoC 的功率和性能。

在读到的论文中的数据显示，这个方案使路由器功率降低了 16.7%，CPI 降低了 23.5%。与零模式压缩方案相比，频繁值方案可节省高达 11.0% 的路由器功率，并将 CPI 降低多达 14.5%。(引自论文)

## 1.2 基本思想

通过识别数据中频繁出现的值，将这些值替换成更小的标识符（如整数编号或指针），以减少存储所需的空间。

对于不常见或不频繁出现的值，通常保留原始数据或使用某种较为高效的压缩策略。

## 1.3 常见实现方式

### 1.3.1 基于频率排序的映射（频繁值替换）

首先统计数据中各个值的出现频率，然后将频繁出现的值替换为一个小的编号或更紧凑的表示。较不常见的值则保留原样或使用更高效的压缩算法。

优点：对频繁值进行压缩，节省空间。通过编号替换，降低存储负担。

缺点：对不频繁值的压缩效果较差，可能会占用更多存储空间。

### 1.3.2 哈夫曼编码（Huffman Encoding）

根据各个值的出现频率分配变长编码。频繁出现的值被分配较短的编码，较少出现的值分配较长的编码。

优点：对频繁出现的值进行高效压缩，减少存储空间。编码和解码过程通常较为高效。

缺点：对于小数据集或者非常不均匀的数据分布，哈夫曼编码的效果可能较差。

### 1.3.3 字典压缩（Dictionary Compression）

通过建立词典，将数据中的重复模式替换为短的标识符。在频繁值压缩中，字典压缩通过将常见的值或模式映射到较小的标识符来实现压缩。

优点：对重复模式和频繁值进行有效压缩，节省空间。支持动态和静态字典，适用于不同类型的数据。

缺点：如果频繁值变化较快，字典更新会带来额外的开销。字典存储本身也需要一定的空间。

### 1.3.4 Delta 编码

Delta 编码是一种通过记录值之间的差异（增量）来压缩数据的方法。如果数据中的频繁值变化较小，Delta 编码可以通过存储连续值的差异而不是原始值来减少存储空间。

优点：对于具有较小变化的频繁值特别有效，压缩率高。适用于时间序列数据和差异较小的数值型数据。

缺点：如果数据变化较大，Delta 编码的效果不好。需要额外的逻辑来处理边界值和解码。

## 1.4 步骤

统计频率：分析数据集中的每个元素出现的频率。

识别频繁值：根据频率阈值来决定哪些值是频繁值。频繁值是那些出现次数大于某个预设阈值的元素。

替换频繁值：将频繁值替换为短标识符或编码。

存储字典：将频繁值与其替代符号（如 ID、编码）映射存储到字典中。

处理非频繁值：对于出现次数较少的元素，可以使用其他压缩算法。

## 1.5 一些介绍

**压缩比：**频繁值压缩通常可以大幅提高存储效率，尤其是在数据集中频繁值占有较大比例时。

**性能优化：**在实际应用中，可以结合多种压缩技术进行优化。例如，使用频繁值压缩和哈夫曼编码组合，针对不同数据特性选择最佳压缩策略。

一旦为应用程序确定了频繁值及其各自的码字，就需要将它们合并到集成到存储器体系结构中的压缩引擎中。为了实现这一点，我们使用一组 CAM 来存储频繁值和它们各自的码字。为了便于实现这一点，我们使用 N 个最常见的值构造 Huffman 编码树，其中 N 是 CAM 的大小。压缩只需使用一个简单的 CAM 将输入的写入数据与保存的频繁值进行匹配，并用它们的码字替换匹配值。解压也同样简单，使用 TCAM 进行相同的过程，代码字中未使用的位设置为不关心值。TCAM 非常适合这种应用，因为 Huffman 代码是前缀自由的，这意味着没有任何码字是任何其他码字的前缀。

频繁值压缩是有益的，因为它在硬件中实现简单，压缩和解压缩速度快。一个 CAM 查找操作的电路级模拟显示，在高达 128 个值的 CAM 大小上，访问延迟仅为一个周期。此外，使用 Huffman 编码作为我们的编码技术应该会进一步减少位写入，代价是在执行之前需要在应用程序上执行 profiling。

### 1.6 不足

**频率统计开销：**计算频率和构建映射可能会增加一定的计算和内存开销，尤其在数据量非常大的时候。

**动态变化的数据集：**如果数据集中的值频繁变化，动态更新频繁值的映射或字典可能会比较复杂，可能影响压缩效果。

**解码性能：**对于解码过程，需要额外的存储（如字典或哈夫曼树），如果存储和解码过程不够优化，可能会影响性能。

## 2 Frequent Pattern Compression（频繁模式压缩）

**Frequent Pattern Compression (FPC)** 是一种数据压缩技术，旨在通过识别和利用数据中频繁出现的模式来实现高效的压缩。频繁模式是指在数据集中频繁出现的项或项集。FPC 的目标是通过找到这些频繁出现的模式，然后利用它们来有效地压缩数据。

### 2.1 步骤

**频繁模式发现**：首先需要从原始数据中发现频繁的模式或项集。常见的算法包括 Apriori、FP-growth 等。

**模式压缩**：一旦识别出频繁模式，就将这些模式替换成更紧凑的表示形式。例如，使用一个小的符号或数字代替长的字符串。

**编码和压缩**：通过使用这些频繁的模式作为编码符号，将原始数据进行压缩。频繁模式可以通过哈夫曼编码、算术编码或其他方法进一步压缩。

### 2.2 特点

此方案中，频繁模式压缩（FPC）通过以压缩格式存储常见的字模式并附带适当的前缀，逐字压缩单个缓存线。

对于 64 字节缓存线，假设每个周期 12 FO4 门延迟，压缩可在三个周期内完成，解压缩可在五个周期内完成。

我们提出一种压缩缓存设计，其中数据以压缩形式存储在二级缓存中，但在一级缓存中不压缩。二级缓存线被压缩到预定的大小，永远不会超过其原始大小，以减少反编译开销。

这个简单的方案提供了与具有更高缓存命中延迟的更复杂方案相当的压缩比。

## 2.3 压缩方案

### 2.3.1 压缩前缀编码特点

缓存线 = Cache line，频繁模式压缩（FPC）基于缓存线压缩/解压缩。每个高速缓存线被划分为 32 位字（例如，64 字节线为 16 个字）。每个 32 位的字被编码为 3 位 pre-fix（前缀）加上数据。

### 2.3.2 压缩层次特点

提出了一种压缩缓存设计，其中数据未压缩存储在一级缓存中，压缩存储在二级缓存中。

**优点**：有助于减少许多阻碍性能的代价高昂的二级缓存未命中，同时又不会影响一级命中的常见情况。

**缺点**：增加了在两个级别之间移动时压缩或解压缩缓存线的开销。

### 2.3.3 压缩

当数据从一级缓存写入二级缓存时，会发生缓存线压缩。使用一个简单的电路（并行地）检查每个字的模式匹配，可以很容易地压缩缓存线。如果一个字与七种可压缩模式中的任何一种匹配，则使用一个简单的编码器电路将该字编码为其最紧凑的形式。如果未找到匹配项，则整个字将与 pre fix “111” 一起存储。

对于 zero-run 模式，我们需要检测连续的 0 的个数，并增加数据值来表示它们的计数。由于在我们的设计中，零运行被限制为 8 个零（多了的前缀就不是 000 了），因此可以使用简单的多路复用/加法器电路在一个周期内实现。

缓存线压缩可以在内存管道中实现，方法是在 L1 到 L2 写入路径上分配三个管道阶段（一个用于模式匹配，一个用于零运行编码，一个用于收集压缩线）。一个包含压缩和未压缩形式的几个条目的小型缓存可用于隐藏一级写回的压缩延迟。

### 2.3.4 解压缩

当数据从二级缓存读取到一级缓存时，会发生缓存线解压缩。对于工作集不在一级缓存中的大多数基准测试来说，这是一个经常发生的事件。

解压缩延迟非常关键，因为它直接添加到二级命中延迟中。

解压是一个比压缩慢的过程，因为行中所有单词的前缀都必须串联访问，因为每个 pre-fix 用于确定其对应的编码单词的长度，从而确定所有后续压缩单词的起始位置。

举个例子，现在 L2CACHE 里面装的数据是 001010001000011111：

第一条：读 001，前缀，则后面 4bit 是数据位，所以 L1CACHE 中的数据是：

$$0000, 0000, 0000, 0000$$

第二条：继续读 010，前缀，则后面 8 位是数据位，所以 L1CACHE 中的数据是：

$$0000, 0000, 0001, 1111$$

### 2.4 一些注意点

**频繁模式的挖掘效率**：例如：Apriori 算法：适合小型数据集，但在大规模数据集上可能效率较低，因为它需要多次扫描数据集。FP-growth 算法：适用于大规模数据集，通过构建频繁模式树来减少重复计算，效率较高。

**最小支持度（Min-Support）设置**：即定义一个项集在数据中出现的最小频次比例。最小支持度的选择直接影响挖掘到的频繁模式的数量：如果最小支持度设置得过高，可能会导致识别出的频繁模式过少，压缩效果不显著。如果最小支持度设置得过低，可能会导致挖掘出过多的频繁模式，增加了存储模式的开销，压缩效果变差。

**压缩比与计算开销的平衡**：频繁模式压缩的目标是通过发现频繁模式来减少冗余数据。然而，频繁模式挖掘本身也需要消耗一定的计算资源。因此，需要平衡压缩比和计算开销。

**数据稀疏性与压缩效果**：频繁模式压缩对于稀疏数据的压缩效果较差。在稀疏数据集（即大部分项集都不常出现的情况下），频繁模式的发现可能会很少，从而导致压缩效果不明显。这是因为稀疏数据集的项集很少重复，频繁模式难以识别，进而影响压缩效率。对于稀疏数据，可以考虑结合其他压缩方法（如差分压缩、字典压缩等）来提高压缩效果。

### 2.5 不足

**大数据集的处理**：随着数据集的增大，频繁模式的发现和压缩变得更加困难。

**在线和增量压缩**：对于流数据，需要开发更高效的增量式算法，以便实时更新频繁模式并执行压缩。

**多模态数据的压缩**：如何同时压缩多种类型的数据（如文本、图像和视频等）中的频繁模式，是一个重要的研究方向。

## 3 Base-Delta-Immediate Compression（BDI 压缩）

### 3.1 基本思想

BDI 压缩的动机主要源于数据间变化较小时的高效压缩需求。它通过增量编码方法，减少冗余信息，降低存储和传输成本。特别适用于传感器数据、物联网等场景，

能够有效压缩大规模数据，节省存储空间和带宽。BDI 方法实现简单、效率高，适合实时应用，特别是在存储和计算资源有限的环境中。

BDI 压缩方法的核心是将数据序列中的每个数字分解为以下几个部分：Base（基础值）：选择一个基础值，通常是数据序列中的最小值或某个公共参考值。所有的数值都基于这个基础值进行计算。Delta（增量值）：计算每个数据项与基础值之间的差异，这个差值称为增量。由于增量值通常较小，它们可以使用更少的位来表示，从而实现压缩。Immediate（直接表示值）：在某些情况下，某些值可能直接采用原始的二进制表示，而不需要计算增量或基础值。

关键思想是，对于许多缓存线，缓存线中的值具有较低的动态范围，即缓存线中存储的值之间的差异很小。因此，可以使用一个基值和一个差异数组来表示缓存线，这些差异数组的组合大小远小于原始缓存线（我们称之为基 + 增量编码）。

对于我们研究的工作负载，最好的选择是有两个基，其中一个基总是零。使用这两个基值（零和其他值），我们的方案可以有效地压缩包含两个分离率动态范围的混合缓存线：一个以从缓存线的实际内容中选择的任意值（例如指针值）为中心，另一个接近于零（例如小整数值）。

以往的方法都是在一个缓存线里面进行压缩的工作，粒度是单个字是否压缩，而 BDI 只有两种情况：1. 整个缓存线都压缩。2. 整个缓存线都不压缩。

根据论文，BDI 压缩可提高单核（8.1% 的提升）和多核工作负载（双核/四核 9.5%/11.2% 的提升）的性能。对于许多应用程序，BDI 提供了将基线系统缓存大小加倍的性能优势，有效地将平均缓存容量提高了 1.53 倍。

## 3.2 BDI 压缩方案

### 3.2.1 一些约定

每个缓存线是 C 字节，每个被压缩的集合都是 k 字节。

对于 64B 的缓存线而言，有 88B，164B，32×2B 三种规格。

目标是确定 B*(BASE 值) 和 k 的值，$\{k, B^*, \delta = (\delta_1, \delta_2, \dots)\}$。

要使缓存线可压缩，表示差异所需的字节数必须严格小于表示其自身值所需的字节数。

B 值的确定和缓存线的关系：B 的最佳值应介于 min(S) 和 max(S) 之间。事实上，只有在最小值、最大值或介于两者之间时才能达到最佳值。

### 3.2.2 参数的确定

- **确定 k 的值**：为所有缓存线选择一个 k 值将显著减少压缩的机会。举个例子，考虑两条缓存线，一条表示指向某个内存区域的 4 字节指针表（类似于图 4），另一条表示存储为 2 字节整数的窄值数组。对于第一个缓存线，k 的可能最佳

值是 4，因为将缓存线划分为一组具有不同 k 的值可能会导致动态范围的增加并降低压缩的可能性。类似地，对于第二个缓存线，k 的最佳值可能是 2。

因此，为了通过迎合多种模式来增加压缩的机会，我们的压缩算法尝试同时使用三个不同的 k 值：2、4 和 8 来压缩缓存线。然后使用提供最大压缩率或根本不压缩的值压缩缓存线。

- **确定 B***：对于 $k \in \{2, 4, 8\}$ 的每个可能值，缓存线被分割成大小为 k 的值，并且对于基的最佳值 B 可以使用观察 2 来确定。然而，以这种方式计算 B* 需要计算值集的最大值或最小值，这增加了逻辑复杂性并显著增加了压缩的延迟。

  为了避免压缩延迟增加和降低硬件复杂性，我们决定使用值集的第一个值作为 B 的近似值。

### 3.2.3 解压过程

$v_i$ 简单地由 $v_i = B^* + \Delta_i$ 给出。因此，可以使用 SIMD 式矢量加法器并行计算缓存线中的值。因此，使用一组简单的加法器，可以在进行整数向量加法所需的时间内对整个缓存线进行解压缩。

### 3.2.4 使用多个基的原因

很明显，并不是每个缓存线都可以用这种形式表示，因此，一些基准没有高压缩比，例如 MCF。发生这种情况的一个常见原因是，其中一些应用程序可以在同一缓存线中混合不同类型的数据，例如指针和 1 字节整数的结构。

经过实验，发现两个基压缩是最好的。不幸的是，有两个基的 B+ 有一个严重的缺点：必须找到第二个基。搜索第二个任意基值（甚至是次优值）可以为压缩硬件增加重要的复杂度。这就引出了如何有效地找到两个基本值的问题。

## 3.3 一些注意点

选择合适的基准数据（Base）：它决定了压缩过程中的增量（Delta）和即时变化（Immediate）的计算。如果基准选择不当，可能会导致增量部分不具有足够的规律性或数据相似性，进而影响压缩效率和效果。理想的基准数据应该能够代表数据集的大部分内容，且变化较小，使得增量和即时变化能够有效地减少冗余。

增量的计算与压缩效率：增量的大小和压缩效率是密切相关的。如果增量较大，压缩算法可能会出现较高的开销，因为需要更多的计算资源来处理这些差异数据。而如果增量较小，则能更好地压缩数据。因此，增量的选择需要权衡数据特性与压缩效果。

即时变化（Immediate）的影响：即时变化部分通常是系统中频繁变化的部分，例如缓存数据或者实时更新的数据。需要优化处理即时变化数据的压缩算法。

压缩与解压缩的速度平衡：需要根据实际需求权衡压缩比和解压缩速度。可以通过调整压缩算法的复杂度或选择合适的硬件支持来优化这一平衡。

硬件支持和资源消耗：虽然 BDI 压缩算法能显著提升存储效率，但它也可能增加处理器的计算负担。在部署 BDI 压缩时可以尽可能利用硬件加速（如专用压缩引擎）来优化性能。

数据的稳定性与重复性：BDI 压缩依赖于数据的稳定性和重复性。如果数据集变化频繁且没有明显的模式或规律，压缩的效果可能不如预期。特别是在面对动态且高度随机的数据时，压缩效果可能会较差。

### 3.4 不足

**基准值的选择**：基准值的选择可能影响压缩效果。如果基准值选择不当，增量值可能会变得较大，从而降低压缩率。

**计算开销**：对于某些数据，尤其是非连续变化的数据，BDI 压缩可能并不适用，增量计算可能带来额外的计算开销。

**数据不连续性**：BDI 压缩的效果最佳于连续数据，对于差异较大的数据，增量的编码效率较低，可能导致压缩率不理想。

## 4 参考文献


1. csdn、维基百科、chatgpt
2. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches
3. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches