[19] 中华人民共和国国家知识产权局

[51] Int. Cl.
G06F 17/14 (2006.01)
G06F 12/00 (2006.01)

[12] 发明专利说明书

专利号 ZL 03817750.1

[45] 授权公告日 2008 年 5 月 28 日

[11] 授权公告号 CN 100390781C

[22] 申请日 2003.5.28 [21] 申请号 03817750.1

[30] 优先权

[32] 2002. 5.28 [33] US [31] 60/385,253

[32] 2002. 5.28 [33] US [31] 60/385,250

[32] 2002. 5.28 [33] US [31] 60/385,251

[32] 2002. 6.21 [33] US [31] 60/390,492

[32] 2002. 6.21 [33] US [31] 60/390,345

[86] 国际申请 PCT/US2003/016908 2003.5.28

[87] 国际公布 WO2003/100655 英 2003.12.4

[85] 进入国家阶段日期 2005.1.25

[73] 专利权人 德罗普莱特科技公司

地址 美国加利福尼亚州

[72] 发明人 威廉·C·林奇

克拉西米尔·D·科拉罗夫

史蒂夫·E·桑德斯

[56] 参考文献

CN1335952A 2002.2.13

US5893145A 1999.4.6

US6141673A 2000.10.31

CN1321931A 2001.11.14

审查员 唐楹琰

[74] 专利代理机构 中科专利商标代理有限责任公司

代理人 朱进桂

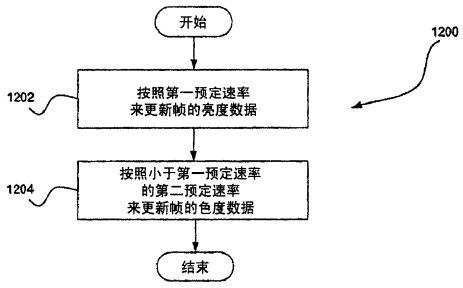
权利要求书 2 页 说明书 39 页 附图 16 页

[54] 发明名称

用于堆处理并行处理器的系统及方法

[57] 摘要

提出了一种用于处理异常的系统、方法和计算机程序产品(Fig. 12)。 提出了另一种包括可变模数的编码器和/或解码器系统和方法。 还提出了一种用于压缩数据的系统和方法，由此按照第一预定速率来更新帧的亮度数据，同时按照小于第一预定速率的第二预定速率来更新帧的色度数据。



1. 一种处理异常的方法，包括：

处理循环中的计算操作；

当处理计算操作时识别异常；

当处理计算操作时存储所述异常；和

独立于循环处理所述异常。

2.如权利要求 1 所述的方法，其特征是所述计算操作包括对多个零值进行计数。

3. 如权利要求 1 所述的方法，其特征是所述计算操作包括剪辑和饱和操作的至少之一。

4. 如权利要求 1 所述的方法，其特征是所述异常包括非零数据。

5. 如权利要求 1 所述的方法，其特征是使用变换模块处理计算操作。

6. 如权利要求 1 所述的方法，其特征是使用量化模块处理计算操作。

7. 如权利要求 1 所述的方法，其特征是使用熵编码模块处理计算操作。

8. 如权利要求 1 所述的方法，其特征是所述存储包括存储处理所述异常所需的数据。

9.如权利要求 1 所述的方法，其特征是执行所述计算操作以压缩数据。

10. 如权利要求 9 所述的方法，其特征是所述数据使用解相关变换来压缩。

11. 如权利要求 9 所述的方法，其特征是所述数据使用小波变换来压缩。

12. 如权利要求 9 所述的方法，其特征是所述数据使用离散余弦变换来压缩。

13. 一种处理异常的系统，包括：

从包括以下模块的组中选择的至少一个数据压缩模块：变换模块、量化模块和熵编码模块，所述至少一个数据压缩模块适于在循环中处理计算操作，当处理计算操作时识别异常，当处理计算操作时存储所述异常，以及独立于所述循环处理所述异常。

用于堆处理并行处理器的系统及方法

技术领域

本发明涉及数据处理。

背景技术

并行处理

当所需的算法有窄的数据宽度、串行数据依赖性、或频繁的控制指令(例如："if"、"for"、"while"指令)时，对并行处理器进行高容量编程是不容易的。有三种类型的并行机制可用来解决处理器中这样的问题。

第一种类型的并行机制由多功能单元来支持，并且它允许进行数据处理以同时在每一个功能单元内进行。超级计数处理器结构和超长指令字(VLIW)处理器结构允许指令在同一循环中被发布到多个功能单元中的每一个。一般来说，此等待时间,或完成时间，在每一种类型的功能单元中都是不同的。最简单的功能(例如：位级 AND)通常在一个循环内即可完成，而一个浮点加法则需要 3 次或更多次的循环来完成。

第二种类型的并行处理由流水线式操作的独立功能单元来支持。例如，一个浮点 ADD 可在 3 次循环内完成，并使用串连 3 个只需 1 次循环的子功能来实现。通过在子功能之间放置流水线寄存器，第二浮点加法可以在同一循环内被起动进入第一子功能中，在此循环中，前一个浮点加法可被起动进入进第二子功能中。利用这种方法，即使任一单独的浮点加法需要 3 次循环来完成，一个浮点加法也可在每个循环内被起动并完成。

第三种类型的并行处理使用一个字中不同的字段分区来实施同一计算的不同实例。例如，32 位处理器上的 32 位的字可以被分成 4 个 8 位的字段分区。如果此数据项足够小从而能放入 8 位内，则利用

同一单一指令将可以用来处理所有 4 个数值。

在每一个循环中，还可能处理的数据项数目等于字段分区的数目乘以功能单元初始化的次数。

循环展开

一般来讲，有一种对多和/或流水线功能单元进行编程的传统和一般方法：找到同一计算方法的多个实例，并根据每一个实例一起执行相应的操作。这些实例可通过已知的循环展开技术或其它同样的计算方法来产生。

虽然循环展开是一种普遍使用的技术，具体的例子对了解此方法的优点有所帮助。例如，考虑下面的程序 A。

程序 A

```
for i=0:1:255, {S(i)};
```

其中程序体 S(i)是依赖于 i 的一序列运算{S1(i); S2(i); S3(i); S4(i); S5(i);}，并且其中计算 S(i)完全独立于计算 S(j)，j≠i。不假设运算 S1(i); S2(i); S3(i); S4(i); S5(i);彼此独立。相反地，假设从一个运算到下一个运算的依赖性禁止重新排序。

同时，还假设此依赖性要求下一个运算在前一个运算结束后才能开始。如果此流水线运算需要两个循环来完成(即使此流水线的执行单元在每一循环中产生一个新的结果)，该 5 个序列的运算将需要 10 个循环来完成。另外，此循环分支需要每个循环附加的 3 个循环，除非编程工具可以重叠 S4(i); S5(i);和分支延迟。因此，如果此分支延迟是被重叠的，则程序 A 需要 640(256/4*10)次循环来完成；如果此分支延迟不被重叠，则需要 832(256/4*13)次循环来完成。

以下的程序 B 与程序 A 等同。

程序 B

```
for  n=0:4:255, {S(n); S(n+1); S(n+2); S(n+3);};
```

此循环已经被“展开” 四次。这就将昂贵的流程控制改变的数目减少了 4 倍。更重要的是，它提供了可以重新将四个 S(i)中每一个

的运算要素重新排序的机会。所以，程序 A 和程序 B 等同于程序 C。

程序 C

```
for  n=0:4:255,{S1(n); S2(n); S3(n); S4(n); S5(n);
       S1(n+1); S2(n+1); S3 (n+1); S4(n+1); S5(n+1);
       S1(n+2); S2(n+2); S3(n+2); S4(n+2); S5(n+2);
       S1(n+3); S2(n+3); S3(n+3); S4(n+3); S5(n+3);
       };
```

利用以上有关依赖性和独立性的假设集合，则可产生等同的程序 D。

程序 D

```
for n=0:4:255, {S1(n); S1(n+1); S1(n+2); S1(n+3);
       S2(n); S2(n+1); S2(n+2); S2(n+3);
       S3(n); S3(n+1); S3(n+2); S3(n+3);
       S4(n); S4(n+1); S4(n+2); S4(n+3);
       S5(n); S5(n+1); S5(n+2); S5(n+3);
       };
```

在第一循环内可以发出（issue）S1(n)；S1(n+1)；，在第二循环内可以发出 S1(n+2)；S1(n+3)；。在第三循环开始时，S1(n)；S1(n+1)；已经完成(已经经过了两次循环)，所以可以发出 S2(n)；S2(n+1)；。因此，接下来的两个运算可在每一个后续循环内发出，因此，可以在相同的 10 次循环内执行整个程序体。程序 D 在少于程序 A 运算时间的四分之一中进行运算。因此，阐明了公知的循环展开的效用。

大多数并行处理器必须具有条件分支指令，此分支指令需要在指令本身和分支实际发生的点之间的多次循环延迟。在此延迟期间，其它指令可被执行。只要足够早地知道此分支条件并且编译器或其它编程工具支持在此延迟期间指令的执行，则此分支可以花费小到一个指令发布时机的时间。这项技术甚至可以被应用到程序 A，因为分支条件(i=255)在循环的顶部就已知。

然而，过多的展开可能产生负面作用。首先，一旦利用了所有的起动时机(如在程序 D 中)，附加的展开并不增加速度。第二，每一轮

（turn）展开的循环通常需要附加的寄存器来保存该具体轮的状态。所需的寄存器的数目与展开轮的数目成线性比例。如果所需寄存器的数目的总和超过可利用的数目，一些寄存器会溢出到高速缓冲存储器中，然后，在下一个循环轮中被恢复。需要发出指令以支持溢出并且重新载入加长了程序的运行时间。所以，存在最优的次数来展开这种循环。

包含异常处理的展开循环

现在考虑程序 A’。

程序 A’

```
for i=0:1:255, {S(i); if C(i) then T(I(i))};
```

其中 C(i)是仅依赖于 S(i)的很少存在的异常条件(例如，1 / 64)；，T(I(i))是一个过长的异常处理，例如有 1024 个操作。I(i)是由异常处理所需的 S(i)计算的信息。例如，可以假定 T(I(i))平均在程序 A 的每个循环轮内增加 16 个运算，此数目超过了循环的主程序体内的 4 个运算。这种罕见但过长的异常处理是一个编程的普遍问题，不清楚应该怎么处理该问题而不会失去展开的优点。

保护指令

处理这种问题的一种方法是通过使用在许多处理器上便于使用的保护指令（guarded instruction)。保护指令指定一个布尔值作为附加的操作数，其中意味着此指令总是占用期望的功能单元,但是，如果此保护指令是假，则抑制结果的保留。

在实现“if-then-else”中，此保护被作为“if”的条件。由“if”的条件保护“then”语句的指令，由“if”的相反条件保护“else”语句的指令。在任何情况下，两条语句都被执行。“then”语句的结果只更新保护是“真”的实例。此外，“else”语句的结果只更新保护是“假”的实例。所有的实例都执行两条语句的指令，承受该损失好于在控制流程中条件变化所需的流水线延迟损失。

如果，例如在程序 A’ 中该保护很多时候是“真”并且“else”语句较大，则保护方法会遭受较大的损失。在这种情况下，所有实例都要为该较大的“else”语句付出代价，即使只有几个状态受到此语句的影响。如果具有由条件 C 保护的操作 S，则可以将其编程为 guard(C, S);

第一展开

如下所示，程序 A’ 可被展开成程序 D’：

```
for n=0:4:255,{S1(n);S1(n+1);S1(n+2);S1(n+3);
        S2(n); S2(n+1); S2(n+2); S2(n+3);
        S3(n); S3(n+1); S3(n+2); S3(n+3);
        S4(n); S4(n+1); S4(n+2); S4(n+3);
        S5(n); S5(n+1); S5(n+2); S5(n+3);
        if C(n) then T(I(n));
        if C(n+1) then (I(n+1));
        if C(n+2) then (I(n+2));
        if C(n+3) then T(I(n+3));
        };
```

假设使用上述例子中的参数，在 77%的循环轮中不会执行 T(I(n))，可能在 21%循环中执行一个 T(I(n))，并且只在 2%的循环中执行多个 T(I(n))。显然,交叉运算 T(I(n))，T(I(n+1))，T(I(n+2))，和 T(I(n+3))并没有多少好处。

因此，需要处理异常的改进技术。

编解码器

编码器是一种按照以下方式将符号的输入序列映射为另一个已编码的符号序列的处理：被称作解码器的另一个处理能够根据已编码的序列符号来重构输入的符号序列。编码器和解码器合称为“编解码器”。

作为简写，通常将符号的有限序列称为流，因此，可以将流称为

输入流和已编码流。从输入符号 I 的相关有限字母表来形成输入流的每一个符号。同样，从代码符号 C 的相关有限字母表来形成已编码流的每一个符号。

每一个字母表包含可区分符号，称为<end>符号。每一个流在与之相关的<end>符号处终止，并且<end>符号只能出现在每个流的末端。使用<end>符号的目的是能够有序地暂停编解码器的处理。

可以将能够确定输入或代码流的末端的任意方法用于合成真实或虚拟的<end>符号的效果。例如，在许多应用程序中，输入和/或代码流的长度是已知的，该信息可用于替换文字的<end>流。

可以用Φ表示编码器映射，以便如果 u 是输入流，v 是相对应的已编码流，则此关系可写成：v=Φ(u)。同样，解码器映射由Ψ表示，可以写成：u=Ψ(v)，其中需要：u=Ψ(Φ(u))。

不需要Φ(Ψ(v)) 来重构 v。如果相关的字母表 I 和 C 每个除了<end>符号之外仅包含两个符号，则将编解码器(Φ，Ψ)称为二进制编解码器。如果 a，b，和<end>是二进制字母表中的三个符号，则将有用函数~定义为：~a=b，~b=a，~<end>=<end>。

如到目前所述，当输入流的数目(和代码流的数目)无限时，编解码器没有实际的实现。如果没有对编解码器设置更多的结构和限制，其将无法灵活地在有限机内实现，实际的实现则更少。

实际上可以由众所周知的有限状态转换器来实现相当部分的编解码器的子集。有限状态转换器(FST)是一种能够从其起始符号到终止符号<end>顺序地处理流的自动装置，并且当序列化时写入代码流的符号。从输入流的符号顺序地获得信息并最终表示在代码流中。

为了桥接从输入流获得信息和在代码流中表示之间的延迟，FST 在序列化时保持并更新状态。从被称为状态空间的可能状态的有限集合中选择出所述状态。状态空间包含两个被称作<start>和<finish>的可区分状态。FST 在<start>状态开始其处理并在<finish>状态完成其处理。直到已经从输入流中读取了<end>符号并且已经将<end>符号附在代码流上，才到达<finish>状态。

由于状态空间是有限的，不可能将每一个编码器都表示为 FST。

为了实用，本说明集中于编码器和解码器都可被描述和执行为 FST 的编解码器。如果能够将编码器Φ实现为 FST，则可以通过更新函数 ø 来指定。来自输入流的第一个输入符号 a 与当前状态 s1 结合并产生下一个状态 s2。有条件地从输入流的开始去除第一个符号。有条件地将所产生的代码符号 b 附到代码流中。

如果当前状态是<finish>且 FST 终止了序列化，则函数 ø 是未定义。概括为：$(s_2, b)=(ø_s(s_1, a), ø_b(s_1, a))=ø(s_1, a)$。这里，将 $ø_s(s_1, a)$ 定义为 $ø(s_1, a)$ 的第一分量，将 $ø_b(s_1, a)$ 定义为 $ø(s_1, a)$ 的第二分量。

对于包括熵编码的许多应用程序，利用马尔可夫概率结构来配备 FST 是有用的。假设状态 s_1 和输入符号 a，则存在概率 $Prob(a/s_1)$，假设 FST 处于状态 s_1，a 将是下一个要读取的符号。根据应用程序，此概率可以被设定，可以静态地从历史数据中估计，或者可以动态地从 FST 的最近操作中被估计。在后者的情况下，可以在状态空间内编码概率估计所基于的信息。

因此，能够计算假设 FST 处于状态 s_1，FST 接下来处于状态 s_2 的概率 $Prob(s_2/s_1)$。这可以通过以下分析计算：$Prob(s_2/s_1)=(øs(s_1, a)= s_2)\ Prob(a/s_1)+(øs(s_1, \sim a)= s_2)\ Prob(\sim a/s_1)$。

可以将该马尔可夫状态转移概率的集合组合为随机矩阵 M，其中 $M_{ij}=Prob(s_j/s_i)$。可以计算渐近（asymptotic）状态概率 P(s)，作为与最大本征值 1 相对应的 M 的右特征向量（right eigenvector）的元素。

视觉方面

视频“编解码器”(压缩器/解压缩器)通过在图像质量、处理器需要(即，成本/功率消耗)和压缩比(即，结果数据率)之间进行平衡来减少数据通信流所需的数据率。目前可用的压缩方法提供了不同程度的折中，并衍生出多个编解码器的概况（profile），其中优化每一个概况来满足具体程序的需要。

有失真数字视频压缩系统对数字化的视频序列进行操作以产生更小的数字表述。重构的可视结果看上去很像原始视频，但通常不会完全匹配。对这些系统，重要的是，在此处理中丢失的信息与观众不容易注意或不容易看到的视频方面相对应。

典型的数字视频压缩系统在一系列级中进行操作，包括变换级、量化级和熵编码级。例如 MPEG 和其它基于 DCT 的编解码器算法的某些压缩系统添加了其它级，例如运动补偿搜索等。二维和三维的小波是当前基于 DCT 的编解码器算法的另一选择。由于其令人满意的图像质量和灵活的压缩比，小波已经受到高度关注，促使 JPEG 委员会在 JPEG2000 静态图像标准中采取了小波算法。

当在视频压缩器中使用小波变换作为变换级时，这种算法如同一系列滤波器对进行操作，将数据分离为高通或低通分量或波段。标准小波变换按照二维方式在单个图像的空间范围上进行运算。通过结合水平工作的滤波器和垂直工作的滤波器来处理二维情况。典型地，这些滤波器按照交替的序列进行操作，H-V-H-V，尽管不需要严格的交替。现有技术中已知在时间（temporal）方向也使用小波滤波器：对实时来自连续图像的采样进行运算。此外，小波变换可以独立地用于视频信号的亮度或辉度（luma）、颜色差异或色度(chroma)分量。

对于二维空间，可以使用 DCT 或其它非小波空间，并且在时间方向上使用小波类型的变换。这种混合三维变换用于与小波变换相同的目的。同时，对于三维 DCT 变换，可以在时间方向上使用短时 DCT。

通常，三维小波变换的时间部分因为更短而与空间部分不同。典型的空间变换大小是水平 720 个像素和垂直 480 个像素；空间变换的典型大小是二、四、八或十五帧。这些时间长度较小，因为当进行处理时，在过程中处理多个帧会导致不希望的较长延迟，此外需要存储帧，这是昂贵的。

当观众观看画面或视频序列以判断其质量时，或当观众在视觉上比较两个画面或两个视频序列时，某些缺陷或不同会比其它缺陷或不同更难察觉。这是人类的视觉系统对于其看到的某些视觉方面比其它方面具有更高敏感度的结果。例如，只有当处于高对比度时，才可以

看到非常细微的细节，但是却能看到处于非常精密对比度的中等大小的细节。对压缩来讲，这些区别是重要的。设计压缩处理，用于使这些区别和错误尽可能的不被注意。因而，压缩处理可以在亮度对比度的中等范围产生较好的逼真度，同时细微的细节中出现更多的错误。

因此，存在对于开发不同的心理物理学机会的持续需要，以改进压缩算法，而不会明显地损失所感知的质量。

前面所述的压缩系统通常用于个人视频记录器、数字视频记录器、有线机顶盒等。这些应用和其他的共同特点是用户具有暂停视频的可能性，在延长的时间内保持单个帧的显示，作为静态图像。

在现有技术中已知处理视频序列或其它图像序列，来得到比输入图像分辨率更高的单个图像。然而，这种处理在计算中是非常昂贵的，因为其必须识别或匹配场景中的运动目标、摄像机运动、光线转移和其它变化，并且单个地或结合地补偿每一个变化。然而，目前的应用程序还不支持这种用于简单暂停功能的计算浪费。

因此，存在对于开发不同的心理物理学机会的持续需要，以表示实质上比由简单重复解压缩缩视频帧产生的图像具有更高视觉质量的暂停图像。

发明内容

异常处理

提出了一种用于处理异常的系统、方法和计算机程序产品。首先，在循环中处理计算操作。此外，在处理所述计算操作的同时识别并存储异常。然后，独立于循环处理这种异常。

在一个实施例中，计算操作可以涉及非重要的数值。例如，此计算操作可以包括对多个零值进行计数。此外，此计算操作可以包括剪辑（clipping）和/或饱和（saturating）操作。

在另一个实施例中，所述异常可以包括重要的数值。例如，所述异常可以包括非零数据。

作为选择，例如，至少可以在使用数据压缩系统的变换模块、量化模块和/或熵编码模块的部分中处理该计算操作。因此，该处理可用

于压缩数据。可选地，可以利用小波变换、离散余弦变换和/或其它类型的解相关变换来压缩所述数据。

编解码器

提出了一种包括可变模数（variable modulus）的编码器和/或解码器系统和方法。

在一个实施例中，所述模数可以反映与压缩算法相关联的概率分布曲线的陡度。例如，此模数可以包括概率分布的负指数。作为选择，所述概率分布与编解码器相关联。

在另一个实施例中，所述模数可以取决于一组先前数据的内容。此外，所述模数可以避免作为运行长度(即，序列中的多个相同位)而增大。

在另一个实施例中，设计编解码器，用于使用提供了预定希望的性能水平的最小计算复杂度。

视觉方面

提出了一种用于压缩数据的系统和方法。在使用中，根据第一预定速率更新帧的亮度，同时根据小于第一速率的第二预定速率更新所述帧的色度数据。

因此，增大了压缩数目。为了实现此目的，在一个实施例中，可以省略色度数据的一个或多个频带。此外，可以使用滤波器省略一个或多个频带。这种滤波器可以包括小波滤波器。因此，当视频数据被解压缩时，可以对色度数据的省略部分进行插值。

提出了用于压缩数据的另一种系统和方法。这种系统和方法涉及压缩视频数据并在已压缩的数据中插入暂停信息。因此，当在回放期间暂停视频数据时，可以使用该暂停信息。

在一个实施例中，暂停信息可以用于提高暂停操作期间视频数据的质量。此外，暂停信息可以包括高分辨率帧。此外，暂停信息包括能够用于构造高分辨率帧的数据。

附图说明

图 1 示出了根据一个实施例的用于压缩/解压缩数据的框架。

图 2 示出了根据一个实施例的用于处理异常的方法。

图 3 示出了图 2 中方法的操作序列的例子。

图 4-9 示出了根据不同实施例的与多个操作特征相关联的多个图形和表。

图 10 是计算复杂度对性能水平图，示出了当前所述二元 / 一元（DM）编解码器框架和其它算法的关系。

图 11 是示出了根据一个实施例的编码器和解码器更新功能的转移表。

图 12 示出了根据一个实施例的具有色度（chroma）时间速率减小的压缩数据的方法。

图 12A 示出了根据一个实施例的在回放期间具有高质量暂停能力的压缩数据的方法。

图 13 示出了根据一个实施例的压缩/解压缩数据的方法。

图 14 示出了执行图 13 中方法时使用的数据结构。

图 15 示出了根据一个实施例的压缩/解压缩数据的方法。

具体实施方式

图 1 显示根据一个实施例的用于压缩/解压缩数据的框架 100。包括在该框架 100 内的是编码器部分 101 和解码器部分 103，两者一起形成“编解码器”。编码器部分 101 包括变换模块 102、量化器 104 和用于压缩数据并存储到文件 108 中的熵码编码器 106。为了执行这种文件 108 的解压，解码器部分 103 包括逆变换模块 114、解量化器 111 和用于解压缩数据以使用(即，观看视频数据的情况等)的熵解码器 110。

在使用中，为了解相关的目的，变换模块 102 执行可逆并且通常是线性的多个象素的解相关变换(即，视频数据的情况)。接下来，量化器 104 进行变换数值的量化，然后，熵编码器 106 进行响应，用于已量化的变换模数的熵编码。解码器部分 103 的不同部分实质上进行

这种处理的逆处理。

图 2 示出了根据一个实施例的用于处理异常的方法。在一个实施例中，该方法 200 可以在图 1 的框架 100 的环境中运行。然而，应该注意的是，方法 200 也可以在任何希望的环境中执行。

首先，在操作 202 中，在循环内处理计算操作。在本说明的环境下，计算操作可以涉及非重要的数值。例如，计算操作可以包括对多个零值进行计数，此操作经常在数据压缩期间被执行。另外，此计算操作还可包括剪辑和/或饱和操作。在任意情况下，此计算操作可以包括处理相对于其它数值重要性较低的任何数值。

当在循环中执行计算操作时，在操作 204-206 中识别并存储异常。可选地，存储可以包括存储处理异常所需的任何相关数据。在本说明的环境下，异常可以包括重要的数值。例如，异常可以包括非零数据。在任意情况下，异常可以包括处理相对于其它数值重要性较高的任何数值。

因此，独立于循环来处理异常。参见操作 208。在这方面，通过使能够展开循环并因此提高了在分支中出现的性能，异常的处理不会中断循环的“堆（pile）”处理。具体地，本实施例能够进行过长异常语句的并行执行。这可以通过向/从存储器写入和重新读取适当数目的数据来实现。关于与此技术和“堆”处理相关的多种选择的更多信息将在以下详细描述。

作为选择，至少可以在使用数据压缩系统的变换模块、量化模块和/或熵编码模块的部分中处理多种计算操作 202-208。例如，参见图 1 中框架 100 的多个模块。因此，可以执行操作 202-208，以压缩/解压缩数据。可选地，可以使用小波变换、离散余弦变换（DCT），和/或任何其它希望的解相关变换来压缩数据。

图 3 示出了图 2 的方法 200 的操作 300 的示例。尽管现在本发明是在图 2 的方法 200 的环境中来描述的，应该注意的是，典型操作 300 可在任何一种希望的环境中实现。

如图所示，设置了计算操作 304 的第一栈 302，用于在循环 306 中的处理。当流程通过计算操作 304 的第一栈 302 时，可以识别多种

异常 308。当被识别时，将这种异常 308 存储到分离的栈中并可以独立地处理。例如，异常 308 可以在独立的循环 310 的环境中被处理。

可选的实施例

现在将说明关于在图 2 的操作环境中可以被执行的这种“堆”处理的多种可选特征的更多信息。在本说明的环境下，“堆”是一种可以被存储在存储器内(即，随机存取存储器)的连续存储对象。堆可用于被顺序地写入并随后从起始被顺序地读取。对于堆对象定义了多种方法。

对于堆和要在并行处理环境种实现的其方法，其实现可以是几个内嵌 (即，没有返回子程序的分支)代码的指令。该内嵌代码还可以不包含分支指令。将在以下描述这种方法实现。这种实现可以使堆特别地具有优点。

表 1 示出了根据一个实施例的可实现堆处理的各种操作。

表 1

1) 由“Creat_Pile(P)”方法来建立堆。这分配了存储器并初始化内部状态变量。

2) 写入堆的基本方法是“Conditional_Append (pile,condition,record)”。如果条件是真，该方法将记录附在堆中。

3)当已经完整地写了一个堆，准备由“Rewind_Pile(P)”方法读取。这调整内部变量，以便从被写入的第一记录开始读取。

4)方法“EOF(P)”产生布尔值，指示了是否已经读取了堆的所有记录。

5)方法“Pile_Read(P,Record)”从堆 P 中读取下一个连续记录。

6)方法“Destroy_Pile(P)”通过重新分配（deallocating）所有的状态变量来破坏堆 P。

使用堆来分离条件处理

因此，可以通过堆 P 将程序 D’(见背景部分)转换为以下程序 E’。

程序 E'

```
Creat_Pile(P)
For n=0:4:255, {S1(n); S1(n+1); S1(n+2); S1(n+3);
S2(n); S2(n+1); S2(n+2); S2(n+3);
S3(n); S3(n+1); S3(n+2); S3(n+3);
S4(n); S4(n+1); S4(n+2); S4(n+3);
S5(n); S5(n+1); S5(n+2); S5(n+3);
Conditional_Append(P, C(n) , I(n));
Conditional_Append(P, C(n+1) , I(n+1));
Conditional_Append(P, C(n+2) , I(n+2));
Conditional_Append(P, C(n+3) , I(n+3));
};
Rewind_Pile(P);

While not EOF(P) {
Pile_Read(P, I);
T(I);
};
Destroy_Pile(P);
```

程序 E' 通过在堆 P 中保存异常计算 T 所需的信息 I 来操作。写入与异常条件 C(n)相对应的记录 I,以使 P 中记录 I 的数目(例如,16)少于在原始程序 A 中的循环轮的数目(例如，256)(参见背景部分)。

之后，独立的“while”循环读取执行所有异常计算 T 的堆 P。因为 P 只包含在 C(n)是真的情况下的记录 I，只处理这些情况。

尽管在此例子中数目平均是 16，因为第二循环轮的数目是不确定的，因此第二循环可能比第一循环更困难。所以，可以使用“while”循环而不是“for”循环，当文件末端(EOF)方法指示已经从堆中读取了所有记录时，终止循环。

如上和下文所述，可以内嵌且无分支地实现“Conditional_Append”方法调用。这意味着仍然以有效的方式展开

第一循环，同时具有很少的无价值发布机会。

展开第二循环

没有展开上述程序 E’ 中的第二循环，因此该程序仍是低效率。然而，可通过四个堆 P1、P2、P3、P4 将程序 E’ 转换为下面的程序 F’。其结果是程序 F’ 同时具有展开的循环和伴随的效率提高。

程序 F’

```
Creat_Pile(P1);Creat_Pile(P2);Creat_Pile(P3);Creat_Pile(P4);
for n=0:4:255, {S1(n); S1(n+1); S1(n+2); S1(n+3);
        S2(n); S2(n+1); S2(n+2); S2(n+3);
        S3(n); S3(n+1); S3(n+2); S3(n+3);
        S4(n); S4(n+1); S4(n+2); S4(n+3);
        S5(n); S5(n+1); S5(n+2); S5(n+3);
        Conditional_Append(P1, C(n) , I(n));
        Conditional_Append(P2, C(n+1) , I(n+1));
        Conditional_Append(P3, C(n+2) , I(n+2));
        Conditional_Append(P4, C(n+3) , I(n+3));
        };
        Rewind_Pile(P1); Rewind_Pile(P2); Rewind_Pile(P3);
    Rewind_Pile(P4);

        While not EOF(Pi) {
        Pile_Read(P1,I1); Pile_Read(P2,I2); Pile_Read(P3,
    I3);  Pile_Read(P4, I4) ;
        guard(not EOF(P1) , S) ; T(I1) ;
        guard(not EOF(P2) , S) ; T(I2) ;
        guard(not EOF(P3) , S) ; T(I3) ;
        guard(not EOF(P4) , S) ; T(I4) ;
        };
    Destroy_Pile(P1) ; Destroy_Pile(P2) ; Destroy_Pile(P3) ;
```

Destroy_Pile(P4) ；

程序 F’是具有第二展开循环的程序 E’。通过将程序 E’的单个堆划分成四个堆来实现此展开，能够独立于其它堆来处理每一个堆。程序 F’的第二循环的每一轮处理来自这四堆中每一个的一个记录。由于独立地处理每一个记录，每一个 T 的操作能够与其它三个 T 的操作相交错。

可以修改“while”循环的控制，使之进行循环，直到已经处理了所有堆。此外，可以保护在“while”循环体内的 T’，因为通常不必在同一循环轮中完成所有的堆。只要两个堆的记录数目彼此区别很大，效率可能会低一些，但是，从概率（即，大数定律）上讲，所述堆可以包含相似数目的记录。

当然，可以递归地应用该堆技术。如果 T 本身包含一个过长的条件语句 T’，可将 T’从具有一些附加堆的第二循环中分离出来，并展开第三循环。许多实际应用具有多个这种嵌套的异常语句。

实现堆处理

可以保持该堆对象的实现及其方法的简单，以便满足上述的实施准则。例如，除了 Creat_Pile 和 Destroy_Pile，此方法实现可能只有少量的内嵌码的指令。此外，所述实现不包含分支指令。

在其核心部分，堆可包括存储器(即，RAM)中的已分配线性数组和指针，其当前值是下一个要读或写的记录的位置索引。数组的写大小 sz 是一个指针，其数值是在堆的写入期间的索引的最大值。EOF 方法可以作为内嵌条件(sz≤index)来实现。指针的基值是指向堆中第一写入位置的数值。可由 Creat_Pile 方法来设置。

Conditional_Append 方法将记录复制到从索引的数值开始的堆数组中。然后，将索引增加零或记录的大小(sz_record)的计算量。由于参数条件具有对于真是 1，对于假是 0 的数值，能够如下计算此索引而不使用分支：index=index+condition*sz_record。

当然，此计算存在许多不同的变体，许多变体不涉及变量的给定特定数值的相乘。还可以利用以下保护来计算：

guard(condition,index=index+sz_record)。

应当注意的是，可以将记录复制到堆中而与条件无关。如果条件是假,该记录可能被下一个记录重写。如果此条件是真，在当前记录之后写下一个记录。下一记录可能会或不会被其之后的记录重写。结果，通常最优的是尽可能少的写入堆中，既使这意味着当读取和处理记录时会重新计算(即，冗余)某些数据。

Rewind 方法可以简单地通过 sz=index; index=base 来实现。该操作记录针对 EOF 方法写入的数据量，然后，将索引复位到起始。

Pile_Read 方法将堆的下一部分(长度为 sz_record)复制到 I，然后如下增加索引：index=index+sz_record。Destroy_Pile 重新分配该堆的存储。可以在几个内嵌指令且无分支地实现所有这些技术(除 Create_Pile 和 Destroy_Pile)。

利用字段分区编程

在很大但“else”很少的语句的情况下，保护处理的另一种选择是堆处理。当每一个示例开始时，“else”语句将输入数据传送到可寻址存储器(即，高速缓冲存储器或 RAM)中的堆中。在一种环境中，堆如同附在输入数据的文件进行操作。这通过写入存储器中由指针给定的地址来实现。然后，在文件处理中，将指针增加所写入数据的大小，以使下一个写入的数据被附到前一个已完成的数据上。在堆处理中，可以有保护条件地进行指针的增加。如果保护为真，则可以将下一个写入的数据附在前一个已完成数据上。如果保护为假，则指针不会被增加并且下一个写入将覆盖前一个已完成的数据。在保护很少是真的情况下，堆较短，具有“else”操作的堆的后续处理会花费与真保护的数目（即，if 条件假）不是示例的总数成比例的时间。此折衷是“else”语句的操作与增加写入和读取的额外开销的节省。

许多处理器具有能够在一个字的分离的字段分区中独立且并行地执行各种算术和逻辑运算的特定指令。当前的说明涉及在每一个字段分区中处理“一次一个位（bit-at-a-time)”的方法。作为运行的示例，考虑包括具有四个 8 位字段分区的 32 位字的示例。选择字段分

区的 8 位，使其在字中连续，因此能够在单个字段分区中执行“加法”并且“携带（carry)”传播（propagate)。普通的可用算术字段分区指令禁止从一个字段分区的最高有效位(MSB)到下一个最高有效字段分区的最低有效位(LSB)的携带。

例如，假设所有长度都等于 B，一个字长度的除数。此外，字段分区可以专用于算法的独立实例。以下是利用每一个指令同时处理一个字的所有字段分区的一些技术和代码序列。这些技术和代码序列使用表 2 的技术来避免控制的变化。

表 2

A)用逻辑/算术计算代替控制的变化。例如，

if(a<0) thyn c=b else c=d

可以被替换为

c=(a<0 ? b:d)

然后可以被替换为

c=b*(a<0)+d*(1-(a<0))

B) 使用逻辑值来有条件地抑制可变数值的替换

if(a<0) then c=b

变为

c=b*(a<0)+c*(1-(a<0))

处理器通常配备有实现该技术的保护指令。

C) 使用逻辑指令来施加条件

b*(a<0)

成为

b&(a<0 ? 0xffff:0x0000) (示例字段是 16 位，并且常数是十六进制)

D) 将逻辑值应用到存储地址和数组下标的计算。这包括聚堆

（piling）技术，该技术有条件地抑制一个被顺序写入的数组索引的增加。例如：

```
if(a<0) then{c[i]=b; i++}
```

变为

```
c[i]=b; i+=(a<0)
```

在这种情况下，两段代码并不完全等同。数组 c 可能在结尾处需要额外的保护索引。通过检查 i 的最终值,用户知道是否丢弃 c 中最后的值。

增加/移位

具有分区算术的处理器经常具有在每一个字段独立执行的 ADD 指令。这些处理器中的某些具有其它类型的逐字段指令(例如，向右移位的分区右移算术不会从一个字段移到另一个，并且不会将该字段的 MSB（符号位）复制到刚刚空出的 MSB)。

比较和字段掩蔽（mask）

一些处理器具有逐字段比较指令，产生多个条件位。如果没有，通常使用分区的减法指令来用于此功能。在这种情况下，计算 a<b，作为 a-b，其中减号表示真，加号表示假。字段的其它位是不相关的。利用具有足够长移位的算术右移，对于真是全 1 或对于假是全 0 的字段掩蔽，与表 2 的 C）中示例使用的相同。这导致两个指令中的多字段比较。

如果分区的算术右移是可用的,利用在当代所有处理器上的四个指令，能够根据符号位来创建字段掩蔽。这些如表 3 所示。

表 3

1.通过 u=u&0x8000 将不相关位设为零

2.移到字段的 LSB v=u >>15(用于 16 位字段的逻辑右移)

3.使字段掩蔽 w=(u-v)|u

4. 由 x+0x7fff 执行在正字段 x 内的分区的零检测，因此，当且

仅当 x 是零时，符号位是零。如果此字段有符号，使用 x|x+0x7fff。将符号位转换为如上所述的字段掩蔽。

当然，可以通过将字段的字的总体(没有分区的)与零进行比较来测试单一指令中所有字段是零的条件。

表述（representation）

定义一些常数是有用的。将每一个字段分区的 MSB 位置中除了“1”的零字称为 MSB。将每一个字段分区的 LSB 位置中除了“1”的零字称为 LSB。在位分区中的比特数是 B。除非另外说明，所有字都是无符号(Uint)，并且所有右移是其中在左边填零的逻辑右移。

多比特字段分区中的单一信息能够按照许多不同的方式表示。掩蔽表达使给定字段分区的所有位彼此相等并且等于信息位。当然，信息位可以从字中的字段分区变化为另一个。

另一个有用的表述是 MSB 表述。信息位被存储在对应字段分区的 MSB 位置上并且字段分区位的剩余部分是零。类似地，LSB 表述具有 LSB 位置上的信息位，且所有其它位是零。

另一个有用的表述是 ZNZ 表述，其中在字段分区中的每一个位，由零来表示零信息位，否则信息位为“1”。所有掩蔽、MSB 和 LSB 表述都是 ZNZ 表述,但反之则不必要。

转换

表述之间的转换需要一个到几个字长的指令,但是这些指令同时处理所有的字段分区。

MSB->LSB

例如，通过使用字逻辑右移指令 y=((Uint)x>>B)，能够将 MSB 表述 x 转换为 LSB 表述 y。通过使用字逻辑左移指令 y=((Unit)<<x)，能够将 LSB 表述 x 转换为 MSB 表述 y。

掩蔽->LSB

通过清除非 MSB 位能够掩蔽表述 m 转换为 MSB 表述。在大多数处理器中，在单个“andnot”指令(m∧～MSB)中，可以将一个字的所有字段分区从从掩蔽变换到 MSB。同样，通过单个“andnot”指令(m∧～MSB)，可以将掩蔽表述转换为 LSB 表述。

MSB->掩蔽

利用字长度指令，利用以下过程可以实现从 MSB 表述 x 到掩蔽表述 z 的转换。见表 4。

表 4

1．将 MSB 表述 x 转换为 LSB 表述 y。

2．从字 x 减去字 y 得到 v。这是除了作为零的 MSB 的掩蔽。

3．字 v OR 字 x 得到掩蔽结果 z。总过程是 z=(x-(x>>B))vx。

ZNZ->MSB

如下可以将字的所有字段分区从 ZNZ x 转换到 MSB y。可以使用字加法指令来在 MSB 位置将 ZNZ 字加零位，而在其它位置加“1”。加法的结果可使在 MSB 位置具有适当的位，而其它位位置具有任意位。这可以通过应用“andnot”指令来清除非-MSB 位来补救。y=(x+～msb)∧～MSB。

其它

从如上所述的 MSB 表述能够得到其它表述。

位输出

在一些应用程序中(例如，熵编解码器)，可能希望通过将给定位逐一地附在位流的末端来形成位流。当前的说明将阐述如何按照字段分区并行的方式完成此功能。字段分区和有关的位流可以独立彼此，每一个代表一个并行的实例。

此过程按照以下表 5 所述工作。

表5

1．以掩蔽表述提供输入位和有效条件。

2．有条件地(即,条件是有效真)添附信息位，直到填满字段分区。

3．当填满字段分区时，将其添附在对应字段分区流的末端。通常，该字段分区的长度全部相等并且是字长度的除数。

还没有完全填满的独立字段分组被保存在被称为累加器的单字内。存在相关的位指针字，其中该字的每一个字段分区包含一个“1”位(即，其余为零)。该“1”位位于与累加器内接收用于该字段分区的下一个添附位相对应的位位置。如果累加器的字段分区已被完全填满，将字段分区添附到对应的字段分区流并将累加器字段分区复位为零。

信息位输出

(有条件地)添附输入的信息位是灵活的。将输入位掩蔽、有效掩蔽和位指针逐字地"相加"在一起，然后逐字地与累加器"相与"。在大多数处理器上，这需要3个指令执行每字。

位指针更新

假设将位添附在位流的LSB末端,字段分区的LSB中的非更新位指针位指示填满了该字段分区。在任何情况下，此位指针字可通过将每一个有效字段分区的位指针向右旋转一个位置而被更新。下面的表6中是用于实现此操作的方法。

表6

a)将位指针分离为LSB位和非LSB位。(2字AND指令)

b)将非LSB位字逻辑右移一位。(1字SHIFT指令)

c)将非LSB位字字逻辑左移到MSB位置。(1字SHIFT指令)

d) 字 OR b) 和 c) 的结果。(1 字 OR 指令)

e) 逐位多路复用 d)的结果和原始的位指针。使用有效掩蔽来控制多路复用(在大多数的处理器上，1 个 XOR，2 个 AND，和 1 个 OR 字指令)

累加器已填满

如上所述，如果位指针 p 的相应字段分区在 LSB 分区是“ 1 ”，则字段分区被填满。由只有位指针 p 非零的 LSB 位的字来指示累加器填满的任意字段分区。f=(p∧LSB)； full=(f≠0)

通常，被填满的可能性明显小于 0.5，因此聚堆（piling）的应用程序是适当的。使用填满作为条件，将累加器 a 和 f 聚成堆 A1。堆 A1 的长度可明显小于添附操作的位数目。设计聚堆，以使除了那些整个处理循环的变化，处理不必涉及到控制流程的变化。

之后，通过循环 A1 中的项来处理堆 A1。对于 A1 中的每一项，按顺序扫描字段分区。每字的字段分区的数目较小，因此通过没有控制变化的直线编码来执行该序列。

可以希望平均来说，字内只有一个字段分区被填满。所以，应该使用(对堆 A2)另一个聚堆的应用程序。使用对应的 f 字段分区作为堆写入条件，将字段分区 a 和 a2 的每一个与对应字段分区索引 i 一起，聚堆到 A2。最后，A2 只包含那些已填满的字段分区。

之后，通过 A2 中项的循环来处理堆 A2。索引 I 用于选择应当添附对应 a2 的位流数组。文件分区的位大小，B，通常被选择为方便的二的幂(例如，8 或 16 位)。存储用于 8 或 16 位数值的指令使这些长度变得便利。在上述处理中，除基本循环以外的控制变化不是必需的。

位字段扫描

编解码器所需的共同操作是在字的字段中顺序读取位。由 bit_pointer 指定从字段 x 提取的位，bit_pointer 是除了单个“1”位其余为零的字段值(例如,0x0200)。将“1”位与被提取的位对准，以便根据所读取位的数值来使 x&bit_pointer 是零或非零。如上所述，

可以将此转换为字段掩蔽。该序列中每一个指令可以同时处理在一个字内的所有字段。

通过沿适当方向移动 bit_pointer 并重复直到适当的终止条件来完成顺序扫描。由于不是所有字段都可以在同一位位置终止,可以修改上述过程，以便被终止的字段没有产生输出同时未终止的字段产生输出。这通过产生有效字段掩蔽来完成，如果未终止字段，掩蔽全是“1”，或如果字段已被终止则全是“0”。该有效字段掩蔽用作输出条件。持续实际的扫描，直到终止了所有区间,这由全零字的有效来指示。

终止条件通常是 bit_pointer 中的位,所述 bit_pointer 到达了由 terminal_bit_pointer 的字段中“1”位指示的位置。这可以由 bit_pointer&terminal_bit_pointer 中的“1”位指示。如上所述，这些字段可被转换成有效字段掩蔽。

尽管本说明看起来依赖于顺序并且对于所扫描的每一个位位置都有控制流程变化,可以展开循环以使所需的实际计算时间最少。在位字段扫描的实际应用中,所有字段均具有相同的位数,导致对于所有字段的循环终止条件都相同。

字段分区的一致子区间

如果希望将字 w 的每个字段分区的位位置 c:d 添附到相应位流上，可以使每一个字段分区的位位置 c 中的常数 c 是只有一个“1”的零字。同样,使每一个字段分区的位位置 d 中的常数 d 是只有一个“1”的零字。此外，可执行以下操作。参见表 7。

表 7

A) 初始化 bit_pointer q 到 c q=c;

A1) 初始化 COND，使之为全真

B) 逐字 bitand q 和 w u=q∧w

u 是 ZNZ 表述

C) 将 u 从 ZNZ 表述转换到掩蔽表述 v

D) 如上所述，现在可以位流输出 v。使用全真的 COND。

E) 如果 cond(q==d) 则处理完成；否则逐字地逻辑右移 q 一位 (q>>1) 循环回到步骤 B)

对于熵编解码器应用，(d-c)的平均值经常是相当小的。操作 E) 中的测试可与操作 B)一起开始，其中延迟到操作 E)和操作 B)-D) 分支是可用的，以覆盖分支流水线的延迟。此外，由于子字段是一致的，展开多个字的处理来覆盖用于字段分区的单个字的指令中的顺序依赖性比较容易。

字段分区的非一致子区间

在字段分区 c 和 d 的区间不同的情况下,c 和 d 保持如上所述，但是上述的测试操作 E)会因区域的不同变化，而不是所有字节中的区域都一样。在这种情况下,可能希望停止已完成字段分区的扫描输出，直到所有的区间都已经完成。可能需要用表 8 中的以下方式来修改上述过程。

表 8

1) 步骤 D)需要其中对于已完成的字段分区是假，对于未完成的字段分区是真的条件。这通过添附将 cond 字与 COND “andnot” 的操作到操作 E)上来完成。COND=(COND∧～cond)

2) 需要将步骤 E)中 if 条件修改为循环返回 B)，除非 COND 是全假。

因此，操作变成：

A) 初始化 bit_pointer q 到 c q=c;

A1) 初始化 COND，使之为全真

B) 逐字地 bitand q 和 w u=q∧w

u 使用 ZNZ 表述

C) 将 u 从 ZNZ 表述变换到掩蔽表述 v

D) 如上所述，v 现在可以被位流输出。使用为全真的 COND。

E1) cond(q==d)；COND=(COND∧～cond)；

E2) 如果 COND==0 处理已完成；否则，逐字地逻辑右移 q 一位(q>>1)循环回到步骤 B)

二元到一元位字段倒计数

在熵编程中的一个共同操作是将字段从二元变换到一元，从而对于其数值是 n 的字段，产生 n 个一后面跟随一个零的流。在大多数应用中，希望数值 n 具有均值是 1 的负指数分布，从而平均来讲，希望除了输出末端的零以外，只有一个“1”。

用于首位为零的正字段的字段分区并行方法如下所述。如上所述，设 c 是除了在字 x 的每个字段的 MSB 位置上有一个“1”的全零常数。设 d 是除了在字 x 的每个字段的 LSB 位置上有一个“1”的全零常数。设 diff=c-d。初始化掩蔽到 diff。

此程序是倒计数(并行地)所讨论的字段，同时创建最初 MSB 位置为零的 c。如果在减法之后的 MSB 位置是“1”，则此字段的先前值不是零此外，应该输出“1”。如果在减法之后 MSB 位置是零，则区间的先前值是零且应该输出零。在任何情况下，MSB 位置包含要输出的位，用于字 X 的对应字段分区。

一旦字段到达零并且输出了第一个零，可以抑制其它零的输出。由于 X 的不同字段分区具有不同的值并输出不同数目的位，抑制来自具有较小数值的字段分区的输出，直到所有字段数值都到达零。利用输入到位输出过程中的掩蔽来实现该抑制，如先前所描述。一旦已经输出了字段分区的第一个零，掩蔽的对应字段分区变为零，从而抑制进一步的输出。

在对于每一个字段分区的 diff 是相同的通常情况下，不必将 diff 变到零。否则，将 diff 与掩蔽相 AND。参见表 9。

表 9

While mask≠0

X = X + diff

Y = ZNZ_2_mask(c∧X);其中 ZNZ_2_mask 是如上所述从 ZNZ 到掩蔽的转换

X = X ∧～c

如上所述输出具有掩蔽的 Y

mask = mask ∧ Y

在典型用于跳转的流水线延迟的情况下，可以根据所估计的轮数的概率分布来决定以上展开上述循环。

对于分区计算最优化循环展开

如果存在以下形式的循环：while c, {s},在第 i 次迭代运算时，c==true 的概率是 Pi,计算 c 和循环跳回的代价是 C(c)，计算 s 的代价是 C(s)。可以假设，s 的额外执行不会影响计算的输出，但是会对每一个代价 C(s)有影响。

可以将此循环展开 n 次，从而此计算变换为 s;s;s;...s; while c, {s}，其中在 while 循环之前有 n 次 s 的执行。于是，总的代价如表 10 所示。

表 10

$$nC(s) + (C(c) + P_n(C(s)+C(c)+P_{n+1}(\ldots))) =$$

$$nC(s) + C(c) + (P_n+P_nP_{n+1}+\ldots)(C(c)+C(s)) \approx$$

$$\approx (n-1)\alpha + U_n = TC(n,\alpha)$$

其中，$U_n = (P_n + P_nP_{n+1}+\ldots)$； $\alpha = \dfrac{C(s)}{C(c)+C(s)}$

作为例子，可以假设每个字有 k 个独立的字段，并且 P 是每个字段循环跳回的概率。然后，$P_n = 1-(1-P^n)^k$。

图 4 示出了根据一个实施例的图解 P_n 的图表 400。图 5 示出了根据一个实施例的图解 U_n 的图表 500。每个图表中的曲线对应于 k 的数值，而灰线对应 k=1)。

图 6 和 7 示出了图表 600 和 700，分别指示了当α=0.3 和α=0.7

时的归一化总代价 TC(n, α)。图 8 是根据一个实施例的图 800，指示了最小总代价 $\min(TC(n,\alpha)) = \overline{TC}(\alpha)$，（虚线），和此初始循环展开的最优数目 $\bar{n}(\alpha)$，

示例

在熵编码的应用程序中，输出位是一的概率是 0.5，是零的概率是 0.5。其还是独立的。利用这些假设，则可以进行以下计算。

给定字段分区要求 n 或者更少输出位（包括终止零）的概率 P(n) 是 P(n)=(1-0.5-n)。设每个字中字段分区的数目是 m。则循环中所要求的轮数是 n 或更小的概率是 $(P(n))^m = (1-0.5^{-n})^m$。图 9 示出了根据一个实施例的表 900，其中包括前述方程中的各种数值。如上所示，循环展开 2-4 次应该是正常的。

编解码器

在一个实施例中，图 1 所示的编码部分 101 和/或解码部分 103 可以包括可变模数。在本说明的情况下，此模数反映了与编解码器框架 100 所使用的压缩算法相关联的概率分布曲线的陡度。例如，此模数可以包括概率分布的负指数。

虽然此模数可以作为任何希望参数的函数而变化，在一个实施例中，此模数可以取决于先前一组数据的内容，其中这组数据指正在被编解码器框架 100 的多个模块处理的位集合。此外，模数可以避免作为运行长度（即，序列中的多个相同位）的函数而增大。

因此，设置了二元 / 一元（DM）的编解码器框架。将会在以下详细阐明有关其中模数取决于先前一组数据的内容的选择方式、模数可以避免作为运行长度的函数而增大等的更多信息。

图 10 是计算复杂度对性能水平的图形 1000，示出了本二元 / 一元的编解码器框架和其它算法的关系（即，Huffman，Rice Golomb、算术等）。如图所示，可以设计 DM 编解码器，以使用提供了预定性能水平的最小计算复杂度。

现在将说明有关在图 1 的编解码器框架环境中实现的多个可选

特征的更多信息。在一个实施例中，通过说明其状态空间和更新函数来指定 DM 编解码器（参见背景技术部分）。

每一个状态具有五个分量：位置 P、*context*、*shift*、*Aregister* 和 *Cregister*。如上所述，此模数可以根据先前一组数据而变化。根据满足前述“context”的定义的特定实施例，本 context 包括在输入字母表上长度为 k 的位流（总共2^k个状态）。每一个 *Aregister* 和 *Cregister* 可以保存小于 1 的非负倍数2^{-n}（每一个总共2^n个状态）。在<*start*>和<*finish*>状态中，*Aregister* 和 *Cregister* 的值是零。用于<start>状态的内容值，虽然是任意，但有必要对于编码器和解码器是相同的。<*start*>状态中的 P 值是开始，在<*finish*>状态中是结束。*shift* 值与<*start*>和<*finish*>状态无关。

作为更新函数ϕ的的部分说明，存在一些指定的固定函数。参见表 11。

表 11

1. 函数 *mps* 将每个 *context* 值映射到输入字母表 I 中的数值。目的在于在给定 context 的情况下，*mps* 是概率更大的 I 中的符号。

2．函数 *delta* 将2^k个 *context* 数值的每一个映射到2^{-m}，其中，$0<m\leq n$。函数 *delta* 的目的是量化地捕获下一个符号的数值概率的信息。

可以根据 DM 的限制来选择这两个函数。这些限制和概率信息一起，可以被用来选择 *mps* 和 *delta*，并且在重要的应用程序中，能够实现算法简单化和熵编码效率的有用结合。

在一个实施例中，DM 限制如表 12 中所述。

表 12

1）二元限制：

$delta(context)=2^{-m}$， $0<m\leq n$

其中 m 和 n 是整数，需要 delta 是二的负整数次幂。目的在于

delta(*context*) 近似了下一个符号不必是由 *mps*(*context*)给出的条件概率。$1-\Pr ob(a == mps(context)\,|\,context) \approx delta(context)$。

2）一元限制：

$$(1-delta(context)) \le (1-delta(2*context \bmod 2^k)+mps(context)))$$

不等式的右手边近似是假设先前读取的符号还具有最大可能性时，读取最可能符号的概率。该一元限制反映可能的情况，其中最可能的符号不会减少下一个符号是最可能符号的概率。

这两个限制提供了有效的熵编解码器。

图 11 示出了根据一个实施例的转移表 1100，示出了用于编码器和解码器的更新函数。每一行代表一组状态，其中组中的状态都满足所在判定列的每一个条件。每一行形成了可允许状态的一部分。

在识别了用于初始状态的行之后，执行该行右手边部分的动作。所使用的所有数值是处于初始状态的数值，所以，在行中的动作序列不必是发出。新状态的每一个分量可以接收唯一数值。整个空白意味着没有改变此状态的分量。

对于编码器，执行来自列的“公共”组的更新动作和来自列的“编码器”组的更新动作。对于解码器，从列的“公共”和“解码器”组中选择动作。在状态转移表 1100 的底部，设置了动作的准确定义。

DM 条件的效果在于 *Aregister* 总是上一个相加的 *detla* 的倍数（在 F13 处）。二元条件保证了 delta 的二进制的表示准确地具有一个“1”位。一元条件保证了 *delta* 不会变大，直到产生了代码符号，因此 delta 中的位只会保持在相同位置或向右移动。保持这种情况，直到产生代码符号，此时 *Aregister* 变为 0（准确说是因为只保留了上一个 delta 右边的 *Aregister* 位）。

结果，提供了表 13 的以下一组特征。

表 13

a) A 是 *delta* 的倍数

b) 在写入代码符号后，A 是零

c) 在写入代码符号后，不需要重新归一化

因为同样的原因，DM 编码器中不必使用普通算术编码中所需的 *Cregister*。紧接在产生代码符号后，此 *context* 中捕获之前状态序列的整个存储。由于 *context* 是之前的 k 个输入符号，DM 编解码器具有短期存储器，因此很快适应输入流的局部统计。

本框架的各种实际应用是存在的。例如，在考虑有效函数（significant function）的环境下；图像、视频和信号处理经常涉及到变换，变换的目的是将此信号“浓缩“，即，产生几个大的系数和多个可以忽略的系数（通过用零代替来丢弃此类系数）。非可忽略系数的识别（或位置）通常与该系数的数值一样重要。通常在“有效函数”中捕获该信息，该函数将非可忽略系数映射为“1”而将可忽略系数映射为“0”。

通过按照适当的顺序对这些系数及其有效位进行列表，可以从其前面的数值准确地预测有效位。如果该顺序列表将此系数根据希望的大小以降序排列，则可以得到以“1”显著开始并以“1”显著结束的有效位流。这种统计信息随着流的进行而改变的流被称为非稳定(*non-stationary*)。这种流的有效的熵编码需要紧接之前的存储器。该存储器可以被充分扩展，以实现良好的预测准确度，并且持续时间足够短，以允许充分快速的适应。

在编码非稳定运行并根据上述定义的环境中，有效函数中的运行包括子位流，此流中除最后一位之外的所有位具有一个数值而最后一位具有其它数值。下一次运行在前一次运行的最后一位之后立即开始。

按照一般规则，*context* 中的位越多（k 越大），则编码率越接近熵值的极限。然而，k 越大，实现成本越高。将 k 的范围充分限制在能够检查每一个 k 的数值。一旦 k 被选择，在 *mps*(*context*)和 *delta*(*context*) 函数中完全捕获用于非稳定运行的 DM 编解码器的性能。如表 14 所示，可以从经验数据中近似这些函数。

表 14

1) $mps(context)=1$ if Prob(next symbol = 1 | $context) \geq 0.5$

2) $Prob(2^{2*delta(context)} | context) = 0.5$

此过程是收集适于该环境的足够经验数据，用于每一个环境构成直方图。由此，可以近似概率函数。能够直接计算函数 $mps(context)$。此外，函数 $delta(context)$ 可以通过图 14 中 2) 的迭代法解法来计算。

因此，DM 编解码器将输入流 1:1 映射到编码流，当解码时，已编码流产生原始的输入流。不必产生所有的输出流，作为某些输入流的编码。一些输入流可能被编码为较短的已编码流，许多会被编码为较长的已编码流。关于此输入流的长度相对于输入流被编码为的已编码流，描述各种可能输入流的出现概率是有用的。如果编解码器具有有用的压缩特征，则被编码为较短流的输入流的出现概率远大于被编码为较长流的输入流出现的概率。

不必使用动态概率。对于熵编码应用，有效位流的统计信息能够并且会快速变化。自适应概率表不能跟踪这种变化，即使基于多次运行，该表也只能缓慢变化。所以，DM 编码器不一定要使用概率表；而是在最后几位或在单个运行中适应。

具有有效位数据的经验测试指示了只通过有效位流的最后几位来获得 *context* 大多数优点。作为 *context* 的有效位流的最后几位用于以下一位的概率为条件。

重要的概率量 $P_{context}$=Prob(next input bit = LSB | *context*)。注意，根据 LSB 的定义，$P_{context} < 0.5$。被添加用来最小程度表示下一位的熵如表 15 所示。

表 15

$entropy=-((1-P_{context})*\log_2(1-P_{context})+P_{context}*\log_2(P_{context}))$ 位

然后，delta(context)=entropy/2，这是因为缩放 *Aregister* 以输出此 2^{-1} 位。如果 $P_{context} \sim 0.5$，则下一位要添加的熵近似如以下表

16 所示。

表 16

$-(1-0.5)*\log_2(1-0.5)+0.5*\log_2(0.5))\sim 1$ bit

此外，*delta(context)*=1/2。如果 $P_{context}<<0.5$，下一位要添加的熵近似如以下表 17 所示。

表 17

$$(1-P_{context})*(-P_{context})-P_{context}\log_2(P_{context})\sim P_{context}*\log_2(1/P_{context})$$

此外，*delta(context)*= $P_{context}*\log_2(1/P_{context})/2$。参见以上表 14 中的方程 2)。

视觉方面

图 12 示出了根据一个实施例的一种具有色度时间速率损失的压缩数据方法 1200。在一个实施例中，在图 1 所示的变换模块 102 的环境中以执行逆变换的方式执行此方法 1200。然而，应该注意的是，方法 1200 可以在任何希望的环境中执行。

在操作 1202 中，根据第一预定速率更新帧的亮度（luma）数据。在操作 1204 中，根据小于第一预定速率的第二预定速率来更新帧的色度（chroma）。

因此，在数字视频压缩系统中，可以对于场景的不同分量，改变时间细节的有效传输速率。例如，可以设置数据流，以便比其它分量更频繁地发送已变换信号的某些分量。在这样的例子中，可以计算视频序列的三维（空间+时间）小波变换，并按照全帧（full frame）速率发送结果的 luma 系数。

此外，对于色度信息，可以从色度信号中省略一个或多个较高频带，因此降低视频的时间响应速率，即时间细节逼真度。在重构已压

缩视频用于观看期间，可以利用近似来填入或对已省略信息进行插值，而不是在没有信息发送时显示“零电平“。对于已省略的空间细节可以按照相同的方式进行。通过保持最近接收到的信息直到接收到新的信息是最简单的实现。更一般地，这通过计算逆小波滤波器，使用零或其它默认值代替已省略的信息，来产生具有正确总体电平并减少了时间细节的平滑变化可以实现。

这种色度速率压缩的具体例子如下：对于色度的分量，可以计算横跨空间变换色度值的两帧（四场）的平均值。这可以利用双 Haar 小波滤波器对并丢弃所有最低频率的分量来实现。可以只发送平均值。当重构时，可以保持色度的横跨两帧（四场）的接收数值。已发现观众不会注意到此，即使当他们在仔细检查压缩方法缺陷时。

视频压缩处理的下一阶段，即量化，通过将相似数值分组并发送代表数值来丢弃信息。该丢弃细节有关区域确切有多亮或是什么颜色。当发现已变换分量接近于零时，选择零作为所述代表数值（在具体尺度显示无变化）。可以省略发送零并使接收者将零假定为默认值。这通过减少发送的数据量有助于压缩。而且，已知人类视觉敏感度水平在亮度和色度之间是不同的。

所以，通过对于亮度和色度分量应用不同的量化水平并丢弃来自色度的更多信息，能够从以上事实中获益更多。当在时间或 3－D 变换中进行不同的量化时，效果在于减少了色度中的时间细节和空间细节。在典型情况下，对于普通的视频材料，时间变换导致了远大于高频分量的低频分量。然后，将量化应用到该已变换结果上使较小的数值和零被分为一组，导致将其从已压缩的表述中省略并降低了色度分量的时间分辨率。

图 12A 示出了根据一个实施例的一种在回放期间具有高质量暂停能力的压缩数据的方法 1250。在一个实施例中，此方法 1250 可以在图 1 的框架的环境中被执行。然而，应该注意的是，可以在任何希望的环境下执行此方法 1250。

在操作 1252 中，视频数据被压缩。在一个实施例中，数据压缩可以在图 1 的框架的编码器部分 101 的环境中执行。当然，这种压缩

可以在任何希望的环境中执行。

在操作 1254 中，暂停信息被插入到已压缩数据中。在一个实施例中，暂停信息用于提高回放视频数据的质量。此外，暂停信息可以包括高分辨率帧。暂停信息还可以包括能够用于构造高分辨率帧的数据。

所以，在操作 1256 中，在其回放期间，当暂停视频数据时，可以使用暂停信息。在本方法中，已压缩数据配备了一组特别用于视频暂停时的额外信息。该额外信息可以包括高质量帧，或者差值信息，当所述差值信息与正常压缩帧结合时，导致了高质量帧。

为了将压缩比特率保持在有用的水平，不需要对于每一帧都包括额外信息，而只用于某些帧。典型地，对于大约每 15 帧可以有一帧包括额外信息，因此在图像中，在 1 / 2 秒的时间间隔内出现一次高质量的暂停操作。这可以通过视频暂停行为的观察来完成。然而，可以包括比这更频繁的额外信息，其代价是比特率。还可以包括更少的额外信息以获得更好的压缩性能，其代价是用户的便利。可以从两帧到 60 帧或更多的范围中进行折中。

在一个实施例中，额外信息包括视频的完整帧，利用不同参数组（例如，经过量化减少信息）或使用不同的压缩方法（例如，在 MPEG 流中使用 JPEG-2000）压缩所述视频。当压缩原始视频计算这些额外帧，并且在压缩视频的发送或存储中，与正常压缩的视频帧一起进行。

在另一个实施例中，额外信息可以包括用于正常解压缩处理使用的额外信息而不是完整的额外图像。例如，在小波视频压缩器中，额外信息包括在正常压缩中被丢弃但当暂停时用于额外视觉清晰度而被保留的数据滤波带。对于另一实例，额外信息包括来自变换系数的额外信息低阶位和附加系数，来自使用为选定的可暂停帧设置的附加较小量化。

在另一个实施例中，额外信息包括用于不同于正常解压缩过程的解压缩过程所使用的数据，并且不是完整的帧。在解压缩之后，该信息可以与由正常处理解压缩的一个或多个帧结合来产生更细致的静态图像。

现在将给出关于典型基于小波变换的更多信息，可以结合图 1 和 12A 中的各种特征来使用。然而，应当高度注意，这种基于小波的变换只是为了举例的目的，而不应该被认为是任何形式的限制。例如，设想图 1 和 12A 中的多种特征可以在基于 DCT 的算法等的环境中实现。

图 13 示出了根据一个实施例压缩/解压缩数据的方法 1300。在一个实施例中，可以在图 1 的变换模块 102 中并以逆变换的方式执行方法 1300。应该注意，方法 1300 可以在任何希望的情况下执行。

在操作 1302 中，接收（即，从存储器中识别、检索等）用于压缩数据的插值公式。在本说明的环境，此数据指能够被压缩的任意数据。此外，插值公式可以包括使用插值的任意公式（即，小波滤波器等）。

在操作 1304 中，确定插值公式是否需要至少一个数据值，其中所需要的数据值是不可用的。这种数据值可以包括前述数据的任意子集。由于是不可用，所需数据值可以是不存在超出范围等。

之后，执行外推（extrapolation）操作以产生所需的不可用数据值。见操作 1306。外推公式可以包括使用外推法的任何公式。通过该方案，增强了数据的压缩。

图 14 示出了在其上执行方法 1300 的数据结构 1400。如图所示，在变换期间，通过涉及多个数据值 1402 的插值公式 1403 来实现“最优拟合”1401。注意图 13 中方法 1300 的操作 1302。如果确定数据值 1402 之一不可用（见 1404），使用外推公式来产生这种不可用的数据值。参考图 15 期间，将更详细地说明有关前述技术的实现的更多可选细节。

图 15 示出了根据一个实施例的压缩/解压缩数据的方法 1500。作为选择，方法 1500 可以在图 1 的变换模块 102 中以逆变换的方式被执行。然而，应该注意，方法 1500 可以在任何希望的情况下执行。

方法 1500 提供了用于产生小波滤波器对的边缘滤波器。首先，在操作 1502，分析小波方案以确定小波滤波器近似的局部导数。接下来，在操作 1504，根据小波滤波器特征和可用的采样数目来选择多项式的阶数以用于外推。接下来，使用所选择的多项式阶数，推导用于

每一个小波滤波器的外推公式。见操作 1506。在操作 1508，使用在每一种情况具有可用采样的外推公式来推导特定边缘小波情况。

此外，现在将更详细地说明关于典型外推公式和相关信息的更多可选信息。

JPEG-2000 标准 1）中指定的变换之一是可逆的 5-3 变换，如方程＃1.1 和 1.2 所示。

方程#1.1 和 1.2

$$Y_{2n+1} = X_{2n+1} - [\frac{X_{2n} + X_{2n+2}}{2}] \qquad eq\ 1.1$$

$$Y_{2n} = X_{2n} + [\frac{Y_{2n-1} + Y_{2n+1} + 2}{4}] \qquad eq\ 1.2$$

为了从左面近似 Y_{2n-1}，可以在左面拟合四级多项式。利用可用数值来近似 2N-1 处第二导数一半的负值，得出方程#1.1.R。

方程#1.1.R

$$Y_{2N-1} = -\frac{1}{3}(X_{2N-1} - [\frac{3X_{2N-2} - X_{2N-4} + 1}{2}]) \qquad eq\ 1.1.R$$

当点一是最右点时，方程#1.1.R 可用于代替方程#1.1。通过移动和加法来完成所示的乘 3。而除 3 比较麻烦。对于当最右的索引是 2N-1 的情况，使用方程#1.2 计算 Y_{2N-2} 将不是问题。在最右点的索引是偶数时(即 2N)，方程#1.1 不存在任何问题，但是方程#1.2 涉及丢失数值。此处的目的是只使用相关的已计算奇数索引 Y1 和 Y3，从偶数 X 中减去 Y 的估计值。能够由线性外推得到索引 2N 处的所需估计，如上所述。该近似公式在方程#1.2.R 中给出。

方程#1.2.R

$$Y_{2N} = X_{2N} + [\frac{3Y_{2N-1} - Y_{2N-3} + 2}{4}] \qquad eq\ 1.2.R$$

对应情况应用到左边界处。相似边缘滤波器从右（内部）而不是从左使用所需的外推。在这种情况下，近似滤波器由方程#1.1.L 和方程#1.2.L 表示。

方程#1.2.L和方程#1.2.L

$$Y_0 = -\frac{1}{3}(X_0 - [\frac{3X_1 - X_3 + 1}{2}]) \qquad eq\ 1.1.L$$

$$Y_0 = X_0 + [\frac{3Y_1 - Y_3 + 2}{4}] \qquad eq\ 1.2.L$$

对于这些关于原始变换的外推边界滤波器，可以获得逆变换滤波器，即，通过反向替换。在余使用正边界滤波器精确相同的环境中，可以使用逆变换滤波器来代替标准滤波器。这种滤波器由方程#2.1.Rinv、2.2.Rinv、2.1.L.inv和2.2.L.inv表示。

方程#2.1.Rinv, 2.2.Rinv, 2.1.L.inv, 2.2.L.inv

$$X_{2N-1} = -3Y_{2N-1} + [\frac{3X_{2N-2} - X_{2N-4} + 1}{2}] \quad eq\ 2.1.R\ inv$$

$$X_{2N} = Y_{2N} - [\frac{3Y_{2N-1} - Y_{2N-3} + 2}{4}] \qquad eq\ 2.2.R\ inv$$

$$X_0 = -3Y_0 + [\frac{3X_1 - X_3 + 1}{2}] \qquad eq\ 2.1.L\ inv$$

$$X_0 = Y_0 - [\frac{3Y_1 - Y_3 + 2}{4}] \qquad eq\ 2.2.L\ inv$$

因此，一个实施例可以使用重新公式化的5-3滤波器，所述滤波器避免已知技术中的附加步骤，同时保持滤波器的视觉属性。例如，见方程#3.1, 3.1R, 3.2, 3.2L。

方程#3.1, 3.1R, 3.2, 3.2L

$$Y_{2N+1} = (X_{2N} + 1/2) - [\frac{(X_{2N} + 1/2) + (X_{2N+2} + 1/2)}{2}] \quad eq\ 3.1$$

$$Y_{2N+1} = (X_{2N+1} + 1/2) - (X_{2N} + 1/2) \qquad eq\ 3.1R$$

$$(Y_{2N} + 1/2) = (X_{2N} + 1/2) + [\frac{Y_{2n-1} + Y_{2n+1}}{4}] \qquad eq\ 3.2$$

$$(Y_0 + 1/2) = (X_0 + 1/2) + [\frac{Y_1}{2}] \qquad eq\ 3.2L$$

在这些公式中，利用 1/2 的偏移量或偏差来计算特定系数，以便避免以上所述的加法。应该注意，虽然在公式中出现了很多 1/2 的增加，这些加法在实际计算中无需出现。在方程#3.1 和 3.1R 中，可以看到，1/2 的增加彼此抵消，所以无需将其应用到输入数据上。相反，括号（Y_0+1/2）中的项等可以被理解为实际计算并作为系数存储的量的名称，并被传递到此小波变换金字塔的下一层。

和正向的情况相同，JPEG-2000 逆滤波器能够在以下方程#4.2, 4.2L, 4.1, 4.1R 中被重新公式化。

方程#4.2, 4.2L, 4.1, 4.1R

$$(X_{2N}+1/2)=(Y_{2N}+1/2)-[\frac{Y_{2N-1}+Y_{2N+1}}{4}] \quad eq\ 4.2$$

$$(X_0+1/2)=(Y_0+1/2)-[\frac{Y_1}{2}] \quad eq\ 4.2L$$

$$(X_{2N+1}+1/2)=Y_{2N+1}-[\frac{(X_{2n}+1/2)+(X_{2n+2}+1/2)}{2}] \quad eq\ 4.1$$

$$(X_{2N+1}+1/2)=Y_{2N+1}+(X_{2N+1/2}) \quad eq\ 4.1R$$

从这里可以看到，逆计算的输入值与方程#3.1～3.2L 中正计算所产生项相同，无需精确计算校正的 1/2。

按照这种方式，减少了小波变换的计算期间执行的算术操作的总数。

尽管以上已经说明了本发明的各种实施例，需要理解的是，只是作为示例来进行说明而不是限制。因此，优选实施例的范围和广度不应被局限于任一上述典型实施例，而应当只根据随后的权利要求及其等同物来限定。

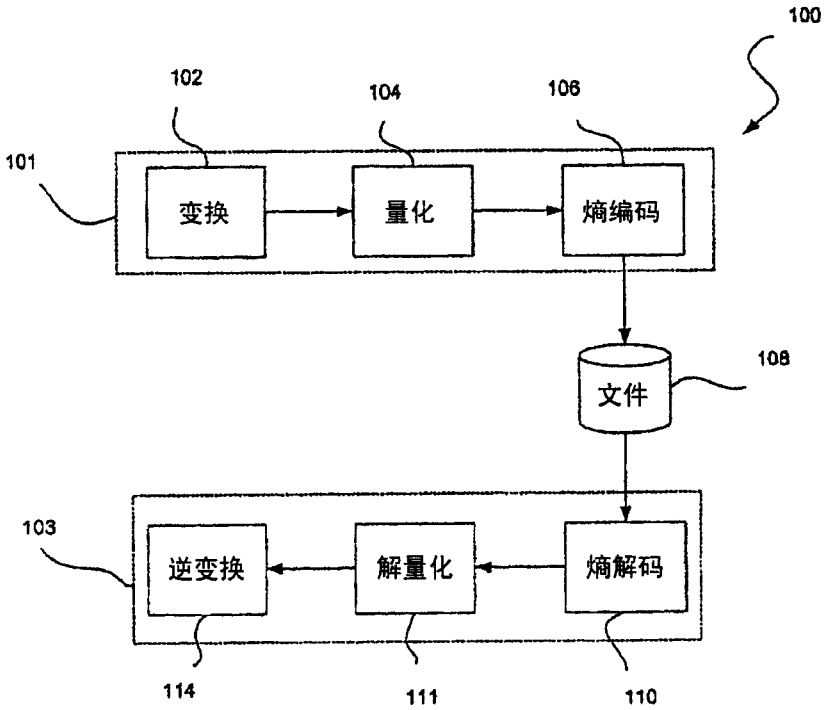


图 1

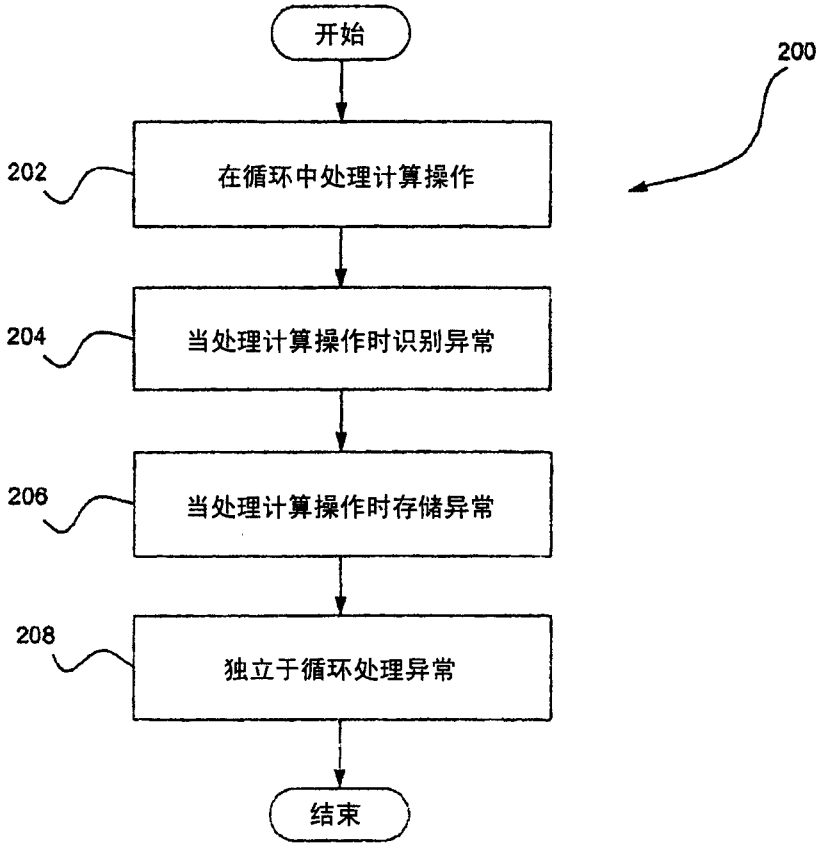

开始
200
202
在循环中处理计算操作
204
当处理计算操作时识别异常
206
当处理计算操作时存储异常
208
独立于循环处理异常
结束

图 2

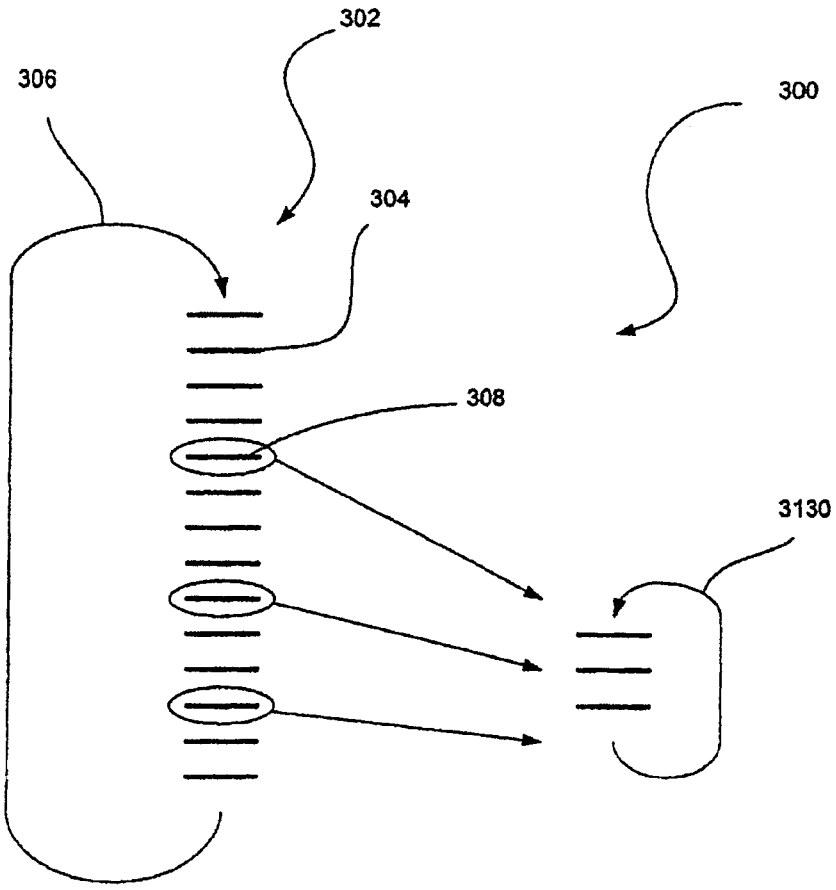


图 3

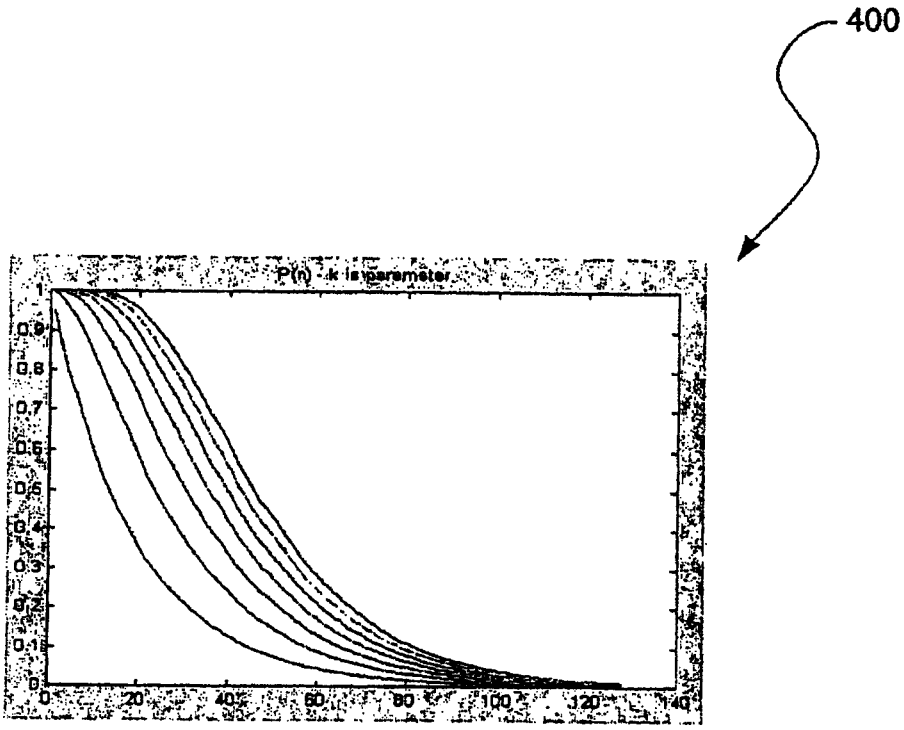

400
P(n) - k is parameter
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
20
40
60
80
100
120
140

图 4

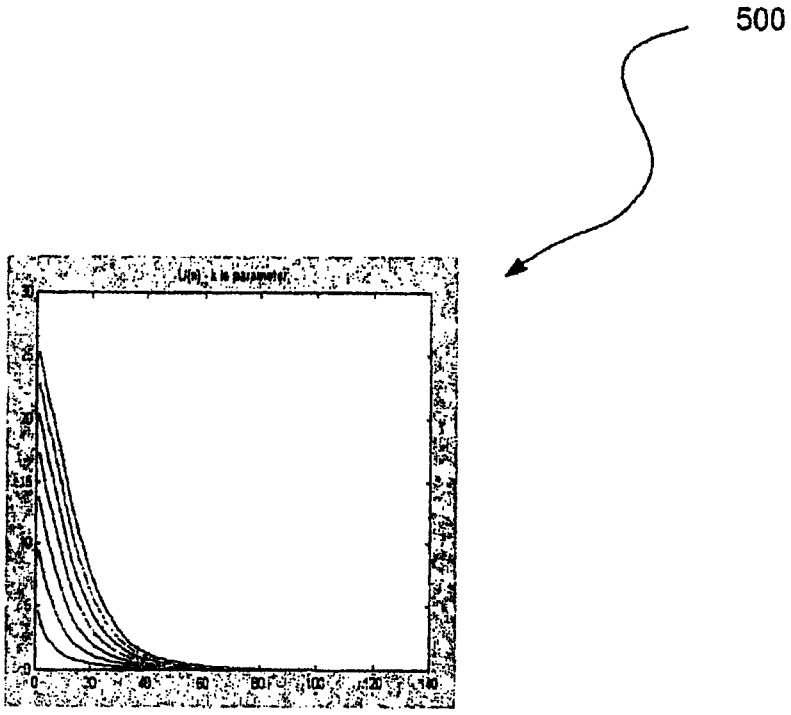


图 5

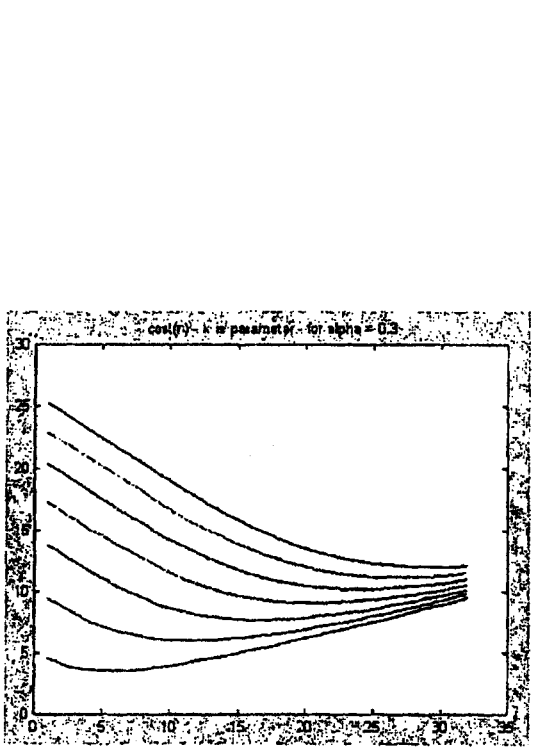


图 6

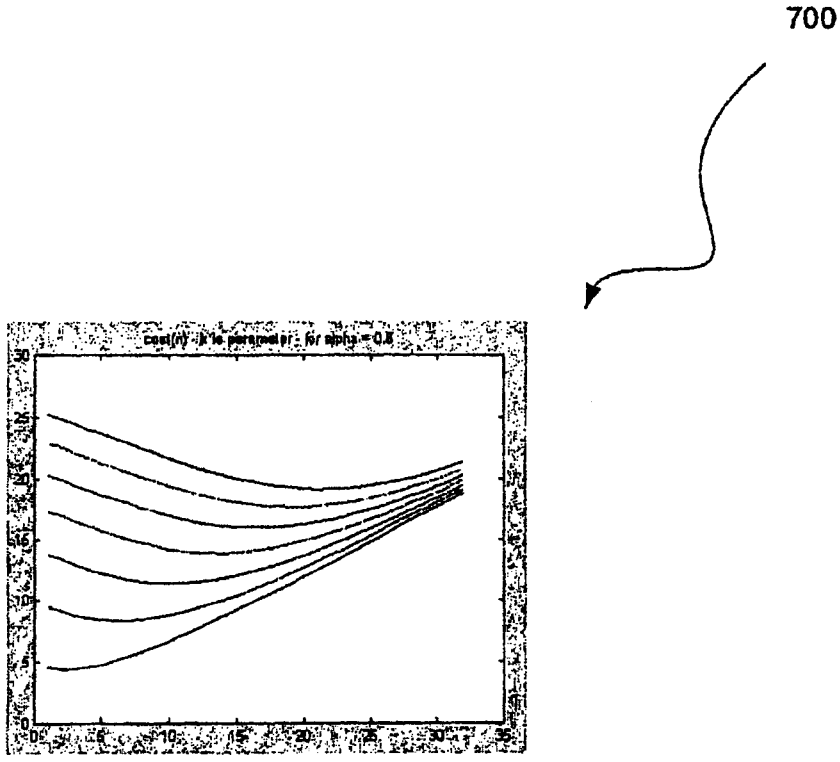


图 7

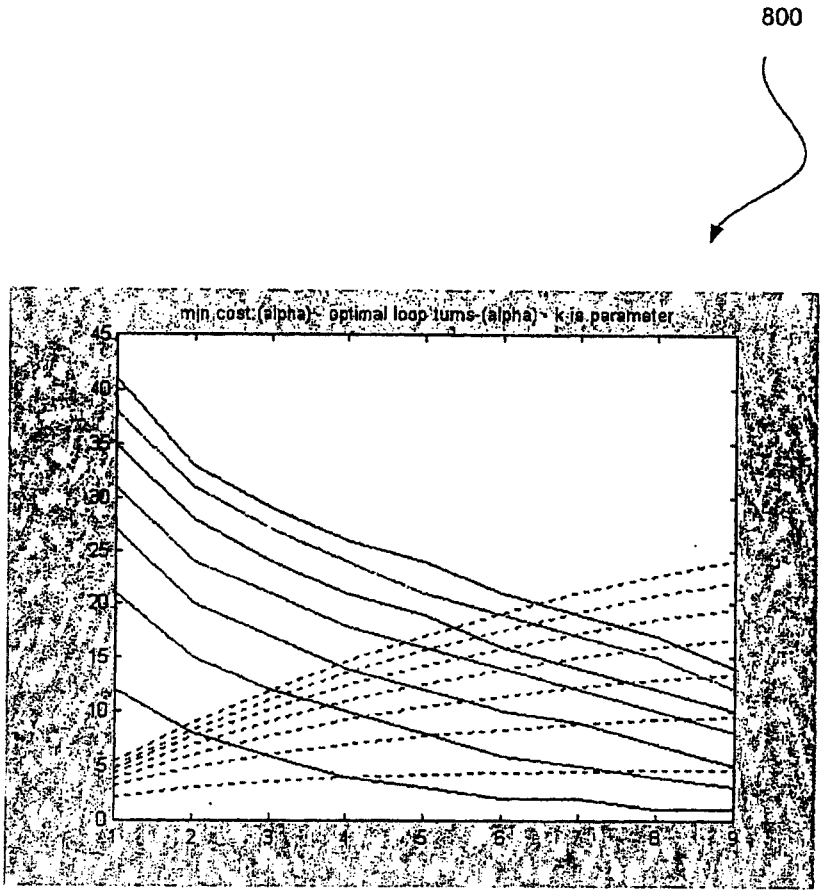

800
min cost:(alpha) - optimal loop turns-(alpha) - k-ja parameter
45
40
35
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8
9

图 8



	n=1	n=2	n=3	n=4	n=5	n=6	n=7
m=2	0.25	0.56	0.77	0.88	0.94	0.97	0.99
m=4	0.06	0.32	0.59	0.77	0.88	0.94	0.98
m=8	0.00	0.10	0.34	0.60	0.78	0.88	0.97

图 9

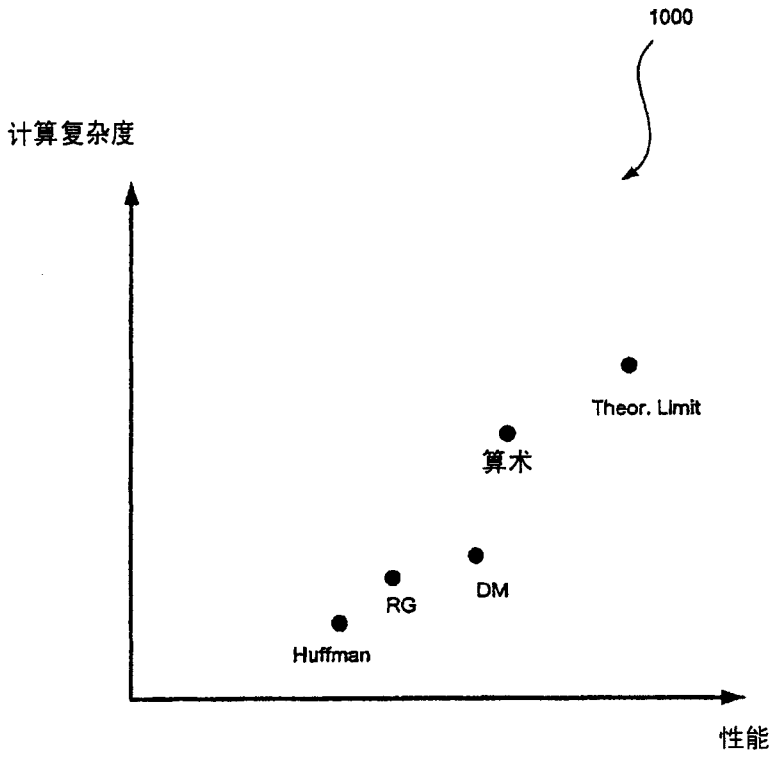


图 10

	A	B	C	D	E	F	G	H	I	J	K	L
1	**COMBINED ENCODE AND DECODE**											
2												
3	**PREDICATES**					**COMMON ACTIONS**				**ENCODE ACTIONS**	**DECODE ACTIONS**	
4			2^{-1} of									
5			*Aregister*		eof				next	encode		decode
6	P	M	== 1	*shift* == 0	*a* == <end>	*Aregister*	*shift*	*context*	P	*b*	*Cregister*	*a*
7	initialize decode											
8	start			FALSE		0	2^{-n}	initial	p2		advance	
9	p2			TRUE			advance		p2		advance	
10	p2								p3			
11												
12	mps											
13	p3	TRUE	FALSE			*Aregister* + *delta*		advance	p3			mps(context)
14	p3	TRUE	TRUE			advance			p3	2^{-1} of *Aregister*	advance	
15												
16	lps											
17	p3	FALSE					*delta*		p4			
18	p4			FALSE		advance	advance		p4	2^{-1} of *Aregister*	advance	
19	p4			TRUE	FALSE			advance	p3			(mps(context)) xor 1
20												
21	finialize											
22	p4			TRUE	TRUE				finish	<end>		<end>

M: ((*mps* (*context*)==*a*) & encode) or ((*Aregister* < *Cregister*) & decode)
M is FALSE if *a* ==<*end*>

advance: *Aregister* = (2**Aregister* mod 1)
delta is a fixed function of *context*

advance: *shift* = (2**shift* mod 1)

advance: *context* = (2**context* mod 2^k)+*a*;
Remove *a* from front of input string
a is the decode symbol

append 2^{-1} of *Aregister* to the code output string
before advancing the *Aregister*

advance:
Cregister = (2*Cregister mod 1)+b*2-n;
b is the next input code symbol
and is removed from the code input string

NOTES

An empty cell is interpreted as an unchanged value
dyadic condition: *delta* (*context*) = 2^{-m} $0 < m <= n$
monotonic condition: *delta* (*context*) >= *delta* ((2**context* mod 2^k)+*mps* (*context*))
F9 Could be "advance". The *Aregister* is zero
F14 Could be 0. The *Aregister* is 2^{-1} before the step and zero after the step
J14 Could be 1. The *Aregister* is 2^{-1} before the step and zero after the step
F19 The *Aregister* is zero during this step
F22 The *Aregister* is zero during this step

The sequence of values of P generated by decoding an encoded string is exactly the sequence generated by the original encoding.
The sequence of values of the predicates is also the exactly the same in encoding a string and in decoding that coded string.



图 11

1100

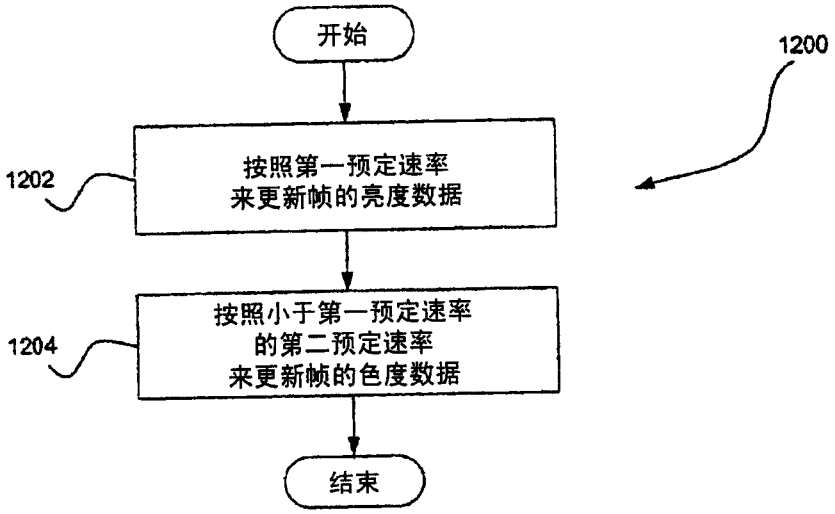

开始
1200
1202
按照第一预定速率
来更新帧的亮度数据
1204
按照小于第一预定速率
的第二预定速率
来更新帧的色度数据
结束

图 12

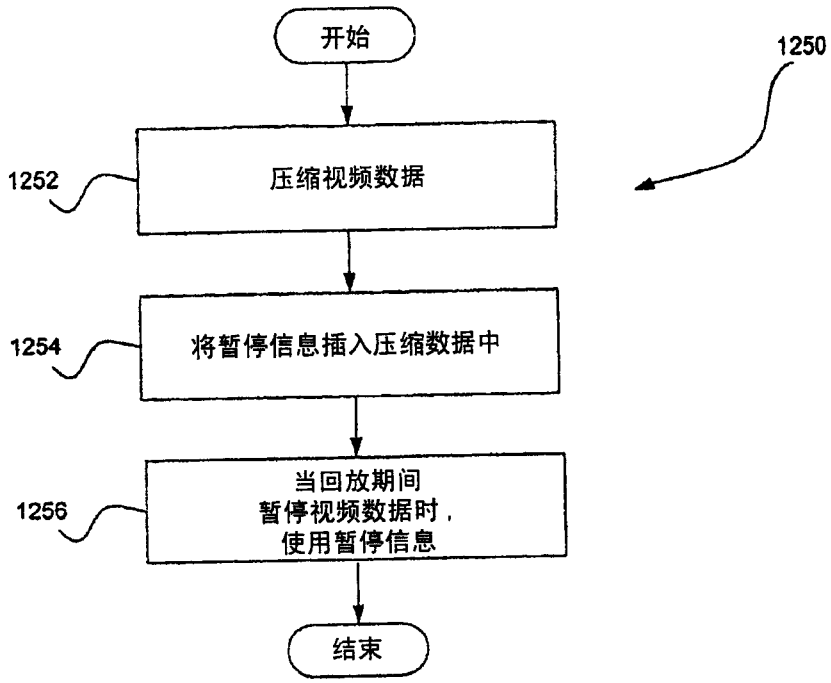

开始
1250
1252
压缩视频数据
1254
将暂停信息插入压缩数据中
1256
当回放期间
暂停视频数据时，
使用暂停信息
结束

图 12A

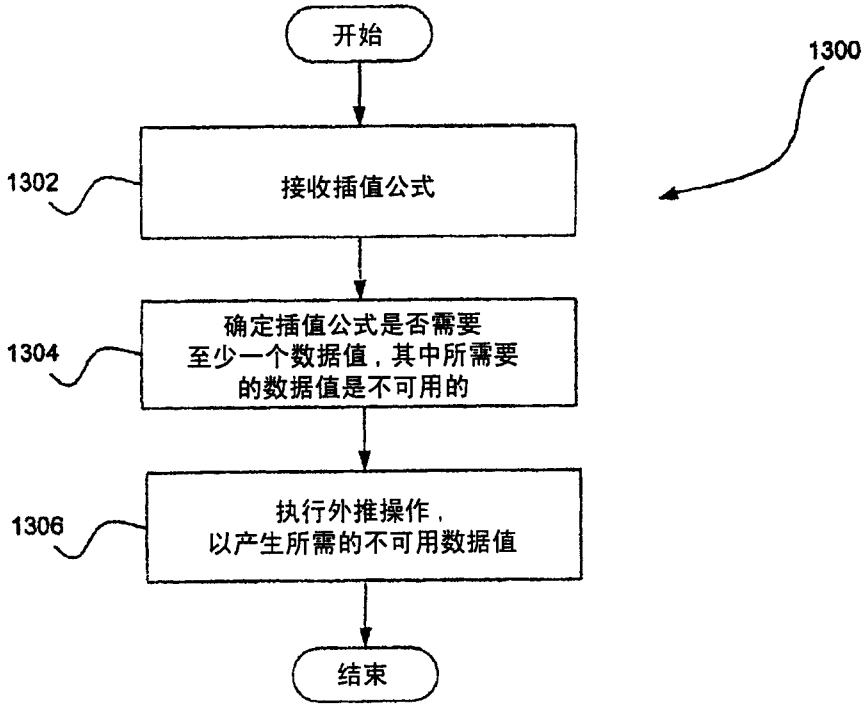

开始
1300
1302
接收插值公式
1304
确定插值公式是否需要
至少一个数据值，其中所需要
的数据值是不可用的
1306
执行外推操作，
以产生所需的不可用数据值
结束

图 13

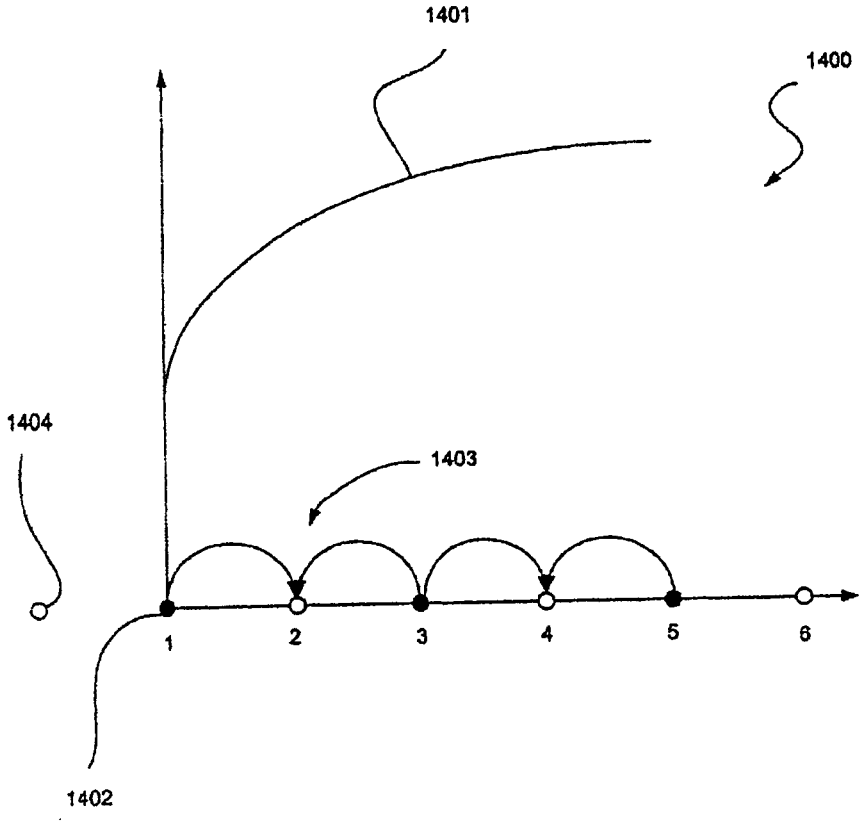


图 14

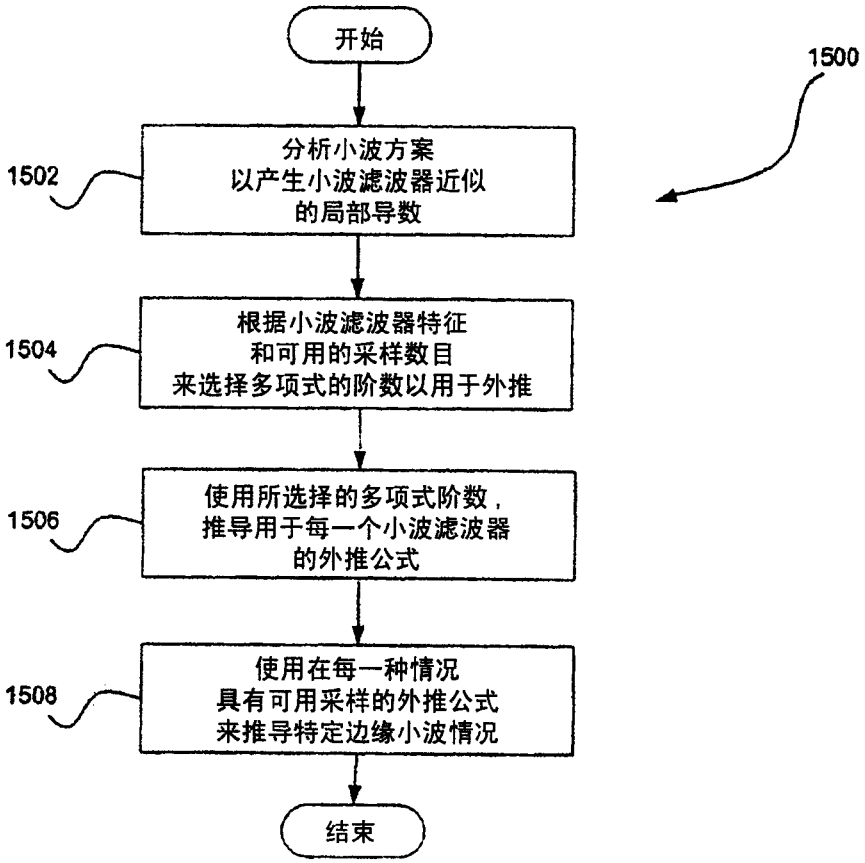

开始
1500
分析小波方案
以产生小波滤波器近似
的局部导数
1502
根据小波滤波器特征
和可用的采样数目
来选择多项式的阶数以用于外推
1504
使用所选择的多项式阶数，
推导用于每一个小波滤波器
的外推公式
1506
使用在每一种情况
具有可用采样的外推公式
来推导特定边缘小波情况
1508
结束

图 15