(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
4 December 2003 (04.12.2003)

PCT

(10) International Publication Number
WO 03/100655 A1

(51) International Patent Classification[7]: G06F 17/14, 12/00

(21) International Application Number: PCT/US03/16908

(22) International Filing Date: 28 May 2003 (28.05.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/385,253	28 May 2002 (28.05.2002)	US
60/385,250	28 May 2002 (28.05.2002)	US
60/385,251	28 May 2002 (28.05.2002)	US
60/390,492	21 June 2002 (21.06.2002)	US
60/390,345	21 June 2002 (21.06.2002)	US

(71) Applicant: DROPLET TECHNOLOGY, INC. [US/US]; 20300 Stevens Creek Boulevard, Suite 250, Cupertino, CA 95014 (US).

(72) Inventors: LYNCH, William, C.; 3331 Thomas Drive, Palo Alto, CA 94303 (US). KOLAROV, Krasimir, D.; 2050 Avy Avenue, Menlo Park, CA 94025 (US). SAUNDERS, Steven, E.; 6069 Shadygrove Drive, Cupertino, CA 95014 (US).

(74) Agent: ZILKA, Kevin, J.; Silicon Valley IP Group, P.O. Box 721120, San Jose, CA 95172-1120 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NI, NO, NZ, OM, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, RO, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

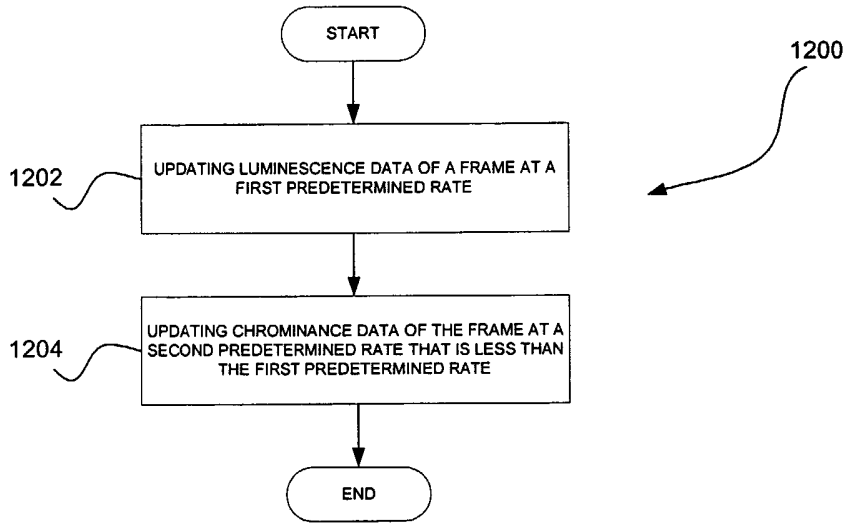
Published:

— *with international search report*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: SYSTEMS AND METHODS FOR PILE-PROCESSING PARALLEL-PROCESSORS



(57) Abstract: A system, method and computer program product are provided for processing exceptions (Fig. 12). Another coder and/or decoder system and method are provided including a variable modulas (1200). Still another system and method are provided for compressing data, whereby luminescence data of a frame is updated at a first predetermined rate (1202), while chrominance data of the frame is updated at a second predetermined rate that is less than the first predetermined rate (1204).

WO 03/100655 A1

SYSTEMS AND METHODS FOR PILE-PROCESSING PARALLEL-PROCESSORS

FIELD OF THE INVENTION

The present invention relates to data processing.

BACKGROUND OF THE INVENTION

Parallel Processing

Parallel processors are difficult to program for high throughput when the required algorithms have narrow data widths, serial data dependencies, or frequent control statements (e.g., "if", "for", "while" statements). There are three types of parallelism that may be used to overcome such problems in processors.

The first type of parallelism is supported by multiple functional units and allows processing to proceed simultaneously in each functional unit. Super-scaler processor architectures and very long instruction word (VLIW) processor architectures allow instructions to be issued to each of several functional units on the same cycle. Generally the latency, or time for completion, varies from one type of functional unit to another. The most simple functions (e.g. bitwise AND) usually complete in a single cycle while a floating add function may take 3 or more cycles.

The second type of parallel processing is supported by pipelining of individual functional units. For example, a floating ADD may take 3 cycles to

complete and be implemented in three sequential sub-functions requiring 1 cycle each. By placing pipelining registers between the sub-functions, a second floating ADD may be initiated into the first sub-function on the same cycle that the previous floating ADD is initiated into the second sub-function. By this means, a floating ADD may be initiated and completed every cycle even though any individual floating ADD requires 3 cycles to complete.

The third type of parallel processing available is that of devoting different field-partitions of a word to different instances of the same calculation. For example, a 32 bit word on a 32 bit processor may be divided into 4 field-partitions of 8 bits. If the data items are small enough to fit in 8 bits, it may be possible to process all 4 values with the same single instruction.

It may also be possible in each single cycle to process a number of data items equal to the product of the number of field-partitions times the number of functional unit initiations.

Loop Unrolling

There is a conventional and general approach to programming multiple and/or pipelined functional units: find many instances of the same computation and perform corresponding operations from each instance together. The instances can be generated by the well-known technique of loop unrolling or by some other source of identical computation.

While loop unrolling is a generally applicable technique, a specific example is helpful in learning the benefits. Consider, for example, Program A below.

Program A

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

where the body S(i) is some sequence of operations {S1(i); S2(i); S3(i); S4(i); S5(i);} dependent on i and where the computation S(i) is completely independent of the computation S(j), $j \neq i$. It is not assumed that the operations S1(i); S2(i); S3(i); S4(i); S5(i); are independent of each other. To the contrary, it assumed that dependencies from one operation to the next prohibit reordering.

It is also assumed that these same dependencies require that the next operation not begin until the previous one is complete. If each pipelined operation required two cycles to complete (even though the pipelined execution unit may produce a new result each cycle), the sequence of five operations would require 10 cycles for completion. In addition, the loop branch may typically require an additional 3 cycles per loop unless the programming tools can overlap S4(i); S5(i); with the branch delay. Program A thus requires 640 (256/4*10) cycles to complete if the branch delay is overlapped and 832 (256/4*13) cycles to complete if the branch delay is not overlapped.

Program B below is equivalent to Program A.

Program B

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

The loop has been "unrolled" four times. This reduces the number of expensive control flow changes by a factor of 4. More importantly, it provides the opportunity for reordering the constituent operations of each of the four S(i). Thus, Programs A and B are equivalent to Program C.

Program 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);
};
```

With the set of assumptions about dependencies and independencies above, one may create the equivalent Program D.

Program 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);
};
```

On the first cycle S1(n); S1(n+1); can be issued and S1(n+2); S1(n+3); can be issued on the 2nd cycle. At the beginning of the third cycle S1(n); S1(n+1); is completed (two cycles have gone by) so that S2(n); S2(n+1); can be issued. Thus, the next two operations can be issued on each subsequent cycle so that the whole body can be executed in the same 10 cycles. Program D operates in less than a quarter of time of Program A. Thus, the well-known benefit of loop unrolling is illustrated.

Most parallel processors necessarily have conditional branch instructions which require several cycles of delay between the instruction itself and the point at which the branch actually takes place. During this delay period, other instructions can be executed. The branch may cost as little as one instruction issue opportunity as long as the branch condition is known sufficiently early and the compiler or other

programming tools support the execution of instructions during the delay. This technique can be applied to even Program A as the branch condition (i=255) is known at the top of the loop.

Excessive unrolling may, however, be counter productive. First, once all of the issue opportunities are utilized (as in Program D), there is no further acceleration with additional unrolling. Second, each of the unrolled loop turns, in general, requires additional registers to hold the state for that particular turn. The number of registers required is linearly proportional to the number of turns unrolled. If the total number of registers required exceeds the number available, some of the registers may be spilled to a cache and then restored on the next loop turn. The instructions required to be issued to support the spill and reload lengthen the program time. Thus, there is an optimum number of times to unroll such loops.

Unrolling Loops Containing Exception Processing

Consider now Program A'.

Program A'

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

where C(i) is some rarely true (say, 1 in 64) exception condition dependent on S(i); only, and T(I(i)) is some lengthy exception processing of, say, 1024 operations. I(i) is the information computed by S(i) that is required for the exception processing. For example, it may be assumed T(I(i)) adds, on the average, 16 operations to each loop turn in Program A, an amount which exceeds the 4 operations in the main body of the loop. Such rare but lengthy exception processing is a common programming problem in that it is not clear how to handle this without losing the benefits of unrolling.

Guarded Instructions

One approach of handling such problem is through the use of guarded instructions, a facility available on many processors. A guarded instruction specifies a Boolean value as an additional operand with the meaning that the instruction always occupies the expected functional unit, but the retention of the result is suppressed if the guard is false.

In implementing an "if-then-else," the guard is taken to be the "if" condition. The instructions of the "then" clause are guarded by the "if" condition and the instructions of the "else" clause are guarded by the negative of the "if" condition. In any case, both clauses are executed. Only instances with the guard being "true" are updated by the results of the "then" clause. Moreover, only the instances with the guard being "false" are updated by the results of the "else" clause. All instances execute the instructions of both clauses, enduring this penalty rather than the pipeline delay penalty required by a conditional change in the control flow.

The guarded approach suffers a large penalty if, as in Program A', the guards are preponderantly "true" and the "else" clause is large. In that case, all instances pay the large "else" clause penalty even though only a few are affected by it. If one has an operation S to be guarded by a condition C, it may be programmed as guard(C, S);

First Unrolling

Program A' may be unrolled to Program D' as follows:

```
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 T(I(n+1));
if C(n+2) then T(I(n+2));
if C(n+3) then T(I(n+3));
};
```

Given the above example parameters, no T(I(n)) may be executed in 77% of the loop turns, one T(I(n)) may be executed in 21% of the loop turns, and more than one T(I(n)) in only 2% of the loop turns. Clearly, there is little to be gained by interleaving the operations of T(I(n)), T(I(n+1)), T(I(n+2)) and T(I(n+3)).

There is thus a need for improved techniques for processing exceptions.

Codecs

An encoder is a process which maps an input sequence of symbols into another, coded, sequence of symbols in such a way that another process, called a decoder, is able to reconstruct the input sequence of symbols from the coded sequence of symbols. The encoder and decoder pair together are referred to as a "codec."

As shorthand, a finite sequence of symbols is often referred to as a string so one can refer to the input string and the coded string. Each symbol of an input string is drawn from an associated, finite, alphabet of input symbols I. Likewise, each symbol of a coded string is drawn from an associated, finite alphabet of code symbols C.

Each alphabet contains a distinguished symbol, called the <end> symbol. Each and every string terminates in the associated <end> symbol and the <end> symbol may only appear at the terminal end of a string. The purpose of the <end> symbols is to bring the codec processes to an orderly halt.

Any method of determining the end of an input or code string can be used to synthesize the effect of a real or virtual <end> symbol. For example, in many applications the length of the input and/or the coded string is known and that information may be used in substitution for a literal <end> string.

The encoder mapping may be denoted by Φ so that if u is an input string and v is the corresponding coded string, one can write: $v = \Phi(u)$. Likewise, the decoder mapping will be denoted by Ψ and one can write: $u = \Psi(v)$, with the requirement that: $u = \Psi(\Phi(u))$.

There is no requirement for $\Phi(\Psi(v))$ to reconstruct v. A codec (Φ,Ψ) is called a binary codec if the associated alphabets I and C each contain just two symbols in addition to the <end> symbol. If a, b, and <end> are the three symbols in a binary alphabet, the useful function $\sim$ is defined to be: $\sim a = b$, $\sim b = a$, $\sim$<end> = <end>.

Codecs, as described so far, do not have a practical implementation as the number input strings (and the number of code strings) is infinite. Without placing more structure and restrictions on a codec, it cannot be feasibly implemented in a finite machine, much less have a practical implementation.

A significant subset of codecs can be practically implemented by the well-known finite state transducer. A finite state transducer (FST) is an automaton that sequentially processes a string from its initial symbol to its terminal symbol <end>, writing the symbols of the code string as it sequences. Information is sequentially obtained from the symbols of the input string and eventually represented in the code string.

To bridge the delay between obtaining the information from the input string and representing it in the code string, the FST maintains and updates a state as it sequences. The state is chosen from a finite set of possible states called a state space. The state space contains two distinguished states called <start> and <finish>. The FST initiates its process in the <start> state and completes its process in the <finish> state. The <finish> state should not be reached until the <end> symbol has been read from the input string and an <end> symbol has been appended to the code string.

Because the state space is finite, it is not possible to represent every encoder as an FST. For reasons of practicality, the present description focuses on codecs where both the encoder and decoder can be described and implemented as FSTs. If the encoder Φ can be implemented as an FST, it can be specified by means of an update function ϕ. The first input symbol a from the input string is combined with the current state s1 and produces the next state s2. The first symbol is conditionally removed from the beginning of the input string. The produced code symbol b is conditionally appended to the code string.

The function ϕ is undefined if the current state is <finish> and the FST terminates sequencing. To summarize: $(s_2, b) = (\phi_s(s_1, a), \phi_b(s_1, a)) = \phi(s_1, a)$. Here, $\phi_s(s_1, a)$ is by definition the first component of $\phi(s_1, a)$ and $\phi_b(s_1, a)$ is by definition the second component of $\phi(s_1, a)$.

For many applications, including entropy coding, it is useful to equip the FST with a Markovian probability structure. Given a state s_1 and an input symbol a, there is a probability $\text{Prob}(a \mid s_1)$ that, given the FST is in state s_1, that a will be the next symbol read. Depending on the application, this probability may be stipulated, may be statically estimated from historical data, or may be dynamically estimated from the recent operation of the FST. In this latter case, the information on which the probability estimate is based may be encoded in the state space.

From this, one can calculate Prob(s_2 | s_1), the probability that, given the FST in state s_1, that the FST will next be in state s_2. This is calculated by case analysis as: Prob(s_2 | s_1) = (ϕs(s_1, a)== s_2) Prob(a | s_1) + (ϕs(s_1, ~a)== s_2) Prob(~a | s_1).

This set of Markovian state transition probabilities can be assembled into a stochastic matrix M where M_{ij} = Prob(s_j | s_i). The asymptotic state probabilities P(s) can be calculated as the elements of the right eigenvector of M corresponding to the largest eigenvalue 1.

Visual Aspects

Video "codecs" (compressor/decompressor) are used to reduce the data rate required for data communication streams by balancing between image quality, processor requirements (i.e. cost/power consumption), and compression ratio (i.e. resulting data rate). The currently available compression approaches offer a different range of trade-offs, and spawn a plurality of codec profiles, where each profile is optimized to meet the needs of a particular application.

Lossy digital video compression systems operate on digitized video sequences to produce much smaller digital representations. The reconstructed visible result looks much like the original video but may not generally be a perfect match. For these systems, it is important that the information lost in the process correspond to aspects of the video that are not easily seen or not readily noticed by viewers.

A typical digital video compression system operates in a sequence of stages, comprising a transform stage, a quantization stage, and an entropy-coding stage. Some compression systems such as MPEG and other DCT-based codec algorithms add other stages, such as a motion compensation search, etc. 2D and 3D Wavelets are current alternatives to the DCT-based codec algorithms. Wavelets have been

highly regarded due to their pleasing image quality and flexible compression ratios, prompting the JPEG committee to adopt a wavelet algorithm for its JPEG2000 still image standard.

When using a wavelet transform as the transform stage in a video compressor, such algorithm operates as a sequence of filter pairs that split the data into high-pass and low-pass components or bands. Standard wavelet transforms operate on the spatial extent of a single image, in 2-dimensional fashion. The two dimensions are handled by combining filters that work horizontally with filters that work vertically. Typically, these alternate in sequence, H-V-H-V, though strict alternation is not necessary. It is known in the art to apply wavelet filters in the temporal direction as well: operating with samples from successive images in time. In addition, wavelet transforms can be applied separately to brightness or luminance (luma) and color-difference or chrominance (chroma) components of the video signal.

One may use a DCT or other non-wavelet spatial transform for spatial 2-D together with a wavelet-type transform in the temporal direction. This mixed 3-D transform serves the same purpose as a 3-D wavelet transform. It is also possible to use a short DCT in the temporal direction for a 3-D DCT transform.

The temporal part of a 3-D wavelet transform typically differs from the spatial part in being much shorter. Typical sizes for the spatial transform are 720 pixels horizontally and 480 pixels vertically; typical sizes for the spatial transform are two, four, eight, or fifteen frames. These temporal lengths are smaller because handling many frames results in long delays in processing, which are undesirable, and requires storing frames while they are processed, which is expensive.

When one looks at a picture or a video sequence to judge its quality, or when one visually compares two pictures or two video sequences, some defects or differences are harder to detect than others. This is a consequence of the human

visual system having greater sensitivity for some aspects of what one sees than for others. For instance, one may see very fine details only when they are at high contrast, but can see medium-scale details which are very subtle in contrast. These differences are important for compression. Compression processes are designed to make the differences and errors as unnoticeable as possible. Thus, a compression process may produce good fidelity in the middle sizes of brightness contrast, while allowing more error in fine details.

There is thus a continuing need to exploit various psychophysics opportunities to improve compression algorithms, without significantly sacrificing perceived quality.

The foregoing compression systems are often used in Personal Video Recorders, Digital Video Recorders, Cable Set-Top Boxes, and the like. A common feature of these applications and others is that users have the possibility of pausing the video, keeping a single frame displayed for an extended time as a still image.

It is known in the art to process a video sequence, or other sequence of images, to derive a single image of higher resolution than the input images. This processing is very expensive in computing, however, as it must identify or match moving objects in the scene, camera motion, lighting shifts, and other changes and compensate for each change individually and in combination. Contemporary applications, however, do not presently support such computational extravagance for a simple pause function.

There is thus a continuing need to exploit various psychophysics opportunities to present a paused image that is of substantially higher visual quality than would be produced by simply repeating a frame of decompressed video.

DISCLOSURE OF THE INVENTION

Exception Processing

A system, method and computer program product are provided for processing exceptions. Initially, computational operations are processed in a loop. Moreover, exceptions are identified and stored while processing the computational operations. Such exceptions are then processed separate from the loop.

In one embodiment, the computational operations may involve non-significant values. For example, the computational operations may include counting a plurality of zeros. Still yet, the computational operations may include either clipping and/or saturating operations.

In another embodiment, the exceptions may include significant values. For example, the exceptions may include non-zero data.

As an option, the computational operations may be processed at least in part utilizing a transform module, quantize module and/or entropy code module of a data compression system, for example. Thus, the processing may be carried out to compress data. Optionally, the data may be compressed utilizing wavelet transforms, discrete cosine transforms, and/or any other type of de-correlating transform.

Codecs

A coder and/or decoder system and method are provided including a variable modulus.

In one embodiment, the modulus may reflect a steepness of a probability distribution curve associated with a compression algorithm. For example, the modulus may include a negative exponential of the probability distribution. As an option, the probability distribution is associated with a codec.

In another embodiment, the modulus may depend on a context of a previous set of data. Moreover, the modulus may avoid increasing as a function of a run length (i.e. a plurality of identical bits in a sequence).

In still another embodiment, the codec may be designed to utilize a minimal computational complexity given a predetermined, desired performance level.

Visual Aspects

A system and method are provided for compressing data. In use, luminescence data of a frame is updated at a first predetermined rate, while chrominance data of the frame is updated at a second predetermined rate that is less than the first predetermined rate.

Thus, the amount of compression is increased. To accomplish this, in one embodiment, one or more frequency bands of the chrominance data may be omitted. Moreover, the one or more frequency bands may be omitted utilizing a filter. Such filter may include a wavelet filter. Thus, upon the decompression of the video data, the omitted portions of the chrominance data may be interpolated.

Another system and method are provided for compressing data. Such system and method involves compressing video data, and inserting pause information with the compressed data. Thus, the pause information is used when the video data is paused during the playback thereof.

In one embodiment, the pause information may be used to improve a quality of the played back video data during a pause operation. Moreover, the pause information may include a high-resolution frame. Still yet, the pause information may include data capable of being used to construct a high-resolution frame.

5

BRIEF DESCRIPTION OF THE DRAWINGS

Figure **1** illustrates a framework for compressing/decompressing data, in accordance with one embodiment.

Figure **2** illustrates a method for processing exceptions, in accordance with one embodiment.

Figure **3** illustrates an exemplary operational sequence of the method of Figure **2**.

Figures **4-9** illustrate various graphs and tables associated various operational features, in accordance with different embodiments.

Figure **10** is a computational complexity v. performance level graph illustrating a relationship of the present dyadic-monotonic (DM) codec framework and other algorithms.

Figure **11** shows a transition table illustrating an update function for both an encoder and decoder, in accordance with one embodiment.

Figure **12** illustrates a method for compressing data with chrominance (chroma) temporal rate reduction, in accordance with one embodiment.

Figure **12A** illustrates a method for compressing data with a high-quality pause capability during playback, in accordance with one embodiment.

Figure **13** illustrates a method for compressing/decompressing data, in accordance with one embodiment.

Figure **14** shows a data structure on which the method of Figure **13** is carried out.

Figure **15** illustrates a method for compressing/decompressing data, in accordance with one embodiment.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure **1** illustrates a framework **100** for compressing/decompressing data, in accordance with one embodiment. Included in this framework **100** are a coder portion **101** and a decoder portion **103**, which together form a "codec." The coder portion **101** includes a transform module **102**, a quantizer **104**, and an entropy encoder **106** for compressing data for storage in a file **108**. To carry out decompression of such file **108**, the decoder portion **103** includes a reverse transform module **114**, a de-quantizer **111**, and an entropy decoder **110** for decompressing data for use (i.e. viewing in the case of video data, etc).

In use, the transform module **102** carries out a reversible transform, often linear, of a plurality of pixels (i.e. in the case of video data) for the purpose of de-correlation. Next, the quantizer **104** effects the quantization of the transform values, after which the entropy encoder **106** is responsible for entropy coding of the quantized transform coefficients. The various components of the decoder portion **103** essentially reverse such process.

Figure **2** illustrates a method **200** for processing exceptions, in accordance with one embodiment. In one embodiment, the present method **200** may be carried out in the context of the framework **100** of Figure **1**. It should be noted, however, that the method **200** may be implemented in any desired context.

Initially, in operation **202**, computational operations are processed in a loop. In the context of the present description, the computational operations may involve non-significant values. For example, the computational operations may include counting a plurality of zeros, which is often carried out during the course of data compression. Still yet, the computational operations may include either clipping and/or saturating in the context of data compression. In any case, the computational

operations may include the processing of any values that are less significant than other values.

While the computational operations are being processed in the loop, exceptions are identified and stored in operations **204-206**. Optionally, the storing may include storing any related data required to process the exceptions. In the context of the present description, the exceptions may include significant values. For example, the exceptions may include non-zero data. In any case, the exceptions may include the processing of any values that are more significant than other values.

Thus, the exceptions are processed separate from the loop. See operation **208**. To this end, the processing of the exceptions does not interrupt the "pile" processing of the loop by enabling the unrolling of loops and the consequent improved performance in the presence of branches. The present embodiment particularly enables the parallel execution of lengthy exception clauses. This may be accomplished by writing and rereading a modest amount of data to/from memory. More information regarding various options associated with such technique, and "pile" processing will be set forth hereinafter in greater detail.

As an option, the various operations **202-208** may be processed at least in part utilizing a transform module, quantize module and/or entropy code module of a data compression system. See, for example, the various modules of the framework **100** of Figure **1**. Thus, the operations **202-208** may be carried out to compress/decompress data. Optionally, the data may be compressed utilizing wavelet transforms, discrete cosine transform (DCT) transforms, and/or any other desired de-correlating transforms.

Figure **3** illustrates an exemplary operation **300** of the method **200** of Figure **2**. While the present illustration is described in the context of the method **200** of

Figure **2**, it should be noted that the exemplary operation **300** may be implemented in any desired context.

As shown, a first stack **302** of operational computations **304** are provided for processing in a loop **306**. While progressing through such first stack **302** of operational computations **304**, various exceptions **308** may be identified. Upon being identified, such exceptions **308** are stored in a separate stack and may be processed separately. For example, the exceptions **308** may be processed in the context of a separate loop **310**.

Optional Embodiments

More information regarding various optional features of such "pile" processing that may be implemented in the context of the operations of Figure **2** will now be set forth. In the context of the present description, a "pile" is a sequential memory object that may be stored in memory (i.e. RAM). Piles may be intended to be written sequentially and to be subsequently read sequentially from the beginning. A number of methods are defined on pile objects.

For piles and their methods to be implemented in parallel processing environments, their implementations may be a few instructions of inline (i.e. no return branch to a subroutine) code. It is also possible that this inline code contain no branch instructions. Such method implementations will be described below. It is the possibility of such implementations that make piles particularly beneficial.

Table 1 illustrates the various operations that may be performed to carry out pile processing, in accordance with one embodiment.

Table 1

1) A pile is created by the Create_Pile(P) method. This allocates storage and initializes the internal state variables.

2) The primary method for writing to a pile is Conditional_Append(pile, condition, record). This method appends the record to the pile if and only if the condition is true.

3) When a pile has been completely written, it is prepared for reading by the Rewind_Pile(P) method. This adjusts the internal variables so that reading may begin with the first record written.

4) The method EOF(P) produces a Boolean value indicating whether or not all of the records of the pile have been read.

5) The method Pile_Read(P, record) reads the next sequential record from the pile P.

6) The method Destroy_Pile(P) destroys the pile P by deallocating all of its state variables.

Using Piles to Split Off Conditional Processing

One may thus transform Program D' (see Background section) into Program E' below by means of a pile P.

Program E'

```
Create_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(P);

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

Program E' operates by saving the required information I for the exception computation T on the pile P. I records corresponding to the exception condition C(n) are written so that the number (e.g., 16) of I records in P is less than the number of loop turns (e.g., 256) in the original Program A (see Background section).

Afterwards, a separate "while" loop reads through the pile P performing all of the exception computations T. Since P contains records I only for the cases where C(n) was true, only those cases are processed.

The second loop may be more difficult than the first loop because the number of turns of the second loop, while 16 on the average in this example, is indeterminate. Therefore, a "while" loop rather than a "for" loop may be used, terminating when the end of file (EOF) method indicates that all records have been read from the pile.

As asserted above and described below, the Conditional_Append method invocations can be implemented inline and without branches. This means that the first loop is still unrolled in an effective manner, with few unproductive issue opportunities.

Unrolling the Second Loop

The second loop in Program E' above is not unrolled, but yet is still inefficient. However, one can transform Program E' into Program F' below by means of four piles P1, P2, P3, P4. The result is that Program F' has both loops unrolled with the attendant efficiency improvements.

Program F'

```
Create_Pile (P1); Create_Pile (P2); Create_Pile (P3);
Create_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(P1); Rewind (P2); Rewind (P3); Rewind (P4);

while not all 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);

Program F' is Program E' with the second loop unrolled. The unrolling is accomplished by dividing the single pile of Program E' into four piles, each of which can be processed independently of the other. Each turn of the second loop in Program F' processes one record from each of these four piles. Since each record is processed independently, the operations of each T can be interleaved with the operations of the 3 other T's.

The control of the "while" loop may be modified to loop until all of the piles have been processed. Moreover, the T's in the "while" loop body may be guarded since, in general, all of the piles will not necessarily be completed on the same loop turn. There may be some inefficiency whenever the number of records in two piles differ greatly from each other, but the probabilities (i.e. law of large numbers) are that the piles may contain similar numbers of records.

Of course, this piling technique may be applied recursively. If T itself contains a lengthy conditional clause T', one can split T' out of the second loop with some additional piles and unroll the third loop. Many practical applications have several such nested exception clauses.

Implementing Pile Processing

The implementations of the pile object and its methods may be kept simple in order to meet the implementation criteria stated above. For example, the method implementations, except for Create_Pile and Destroy_Pile, may be but a few instructions of inline code. Moreover, the implementation may contain no branch instructions.

At its heart, a pile may include an allocated linear array in memory (i.e. RAM) and a pointer, index, whose current value is the location of the next record to read or write. The written size of the array, sz, is a pointer whose value is the maximum value of index during the writing of the pile. The EOF method can be implemented as the inline conditional ($sz \leq index$). The pointer base has a value which points to the first location to write in the pile. It may be set by the Create_Pile method.

The Conditional_Append method copies the record to the pile array beginning at the value of index. Then index is incremented by a computed quantity that is either 0 or the size of the record (sz_record). Since the parameter condition has a value of 1 for true and 0 for false, the index can be computed without a branch as: index = index + condition* sz_record.

Of course, many variations of this computation exist, many of which do not involve multiplying given special values of the variables. It may also be computed using a guard as: guard(condition, index = index + sz_record).

It should be noted that the record may be copied to the pile without regard to condition. If the condition is false, this record may be overwritten by the very next record. If the condition is true, the very next record may be written following the current record. This next record may or may not be itself overwritten by the record thereafter. As a result, it is generally optimal to write as little as possible to the pile even if that means re-computing some (i.e. redundant) data when the record is read and processed.

The Rewind method is implemented simply by sz = index; index = base. This operation records the amount of data written for the EOF method and then resets index to the beginning.

The Pile_Read method copies the next portion of the pile (of length sz_record) to I and increments the index as follows: index = index + sz_record. Destroy_Pile deallocates the storage for the pile. All of these techniques (except Create_Pile and Destroy_Pile) may be implemented in a few inline instructions and without branches.

Programming with Field-Partitions

In the case of the large but rare "else" clause, an alternative to guarded processing is pile processing. As each instance begins, the "else" clause transfers the input data to a pile in addressable memory (i.e. cache or RAM). In one context, the pile acts like a file being appended with the input data. This is accomplished by writing to memory at the address given by a pointer. In file processing, the pointer may then be incremented by the size of the data written so that the next write would be appended to the one just completed. In pile processing, the incrementing of the pointer may be made conditional on the guard. If the guard is true, the next write may be appended to the one just completed. If the guard is false, the pointer is not incremented and the next write overlays the one just completed. In the case where the guard is rarely true, the pile may be short and the subsequent processing of the pile with the "else" operations may take a time proportional to just the number of true guards (i.e. false if conditions) rather than to the total number of instances. The trade-off is the savings in "else" operations vs. the extra overhead of writing and reading the pile.

Many processors have special instructions which enable various arithmetic and logical operations to be performed independently and in parallel on disjoint field-partitions of a word. The current description involves methods for processing "bit-at-a-time" in each field-partition. As a running example, consider an example including a 32-bit word with four 8-bit field-partitions. The 8 bits of a field-partition are chosen to be contiguous within the word so the "adds" can be performed and "carry's" propagate within a single field-partition. The commonly available

arithmetic field-partition instructions inhibit the carry-up from the most significant bit (MSB) of one field-partition into the least significant bit (LSB) of the next most significant field-partition.

For example, it may be assumed all equal lengths B, a divisor of the word length. Moreover, a field-partition may be devoted to independent instances of an algorithm. Following are some techniques and code sequences that process all of the fields of a word simultaneously with each instruction. These techniques and code sequences use the techniques of Table 2 to avoid changes of control.

Table 2

A) replacement of changes of control with logical/arithmetic calculations. For example,
if (a<0) *then* c=b *else* c=d
can be replaced by
c = (a<0 ? b : d)
which can in turn be replaced by
c = b*(a<0) + d*(1-(a<0))

B) use logical values to conditionally suppress the replacement of variable values
if (a<0) *then* c=b
becomes
c = b*(a<0) + c*(1-(a<0))
Processors often come equipped with *guarded* instructions that implement this technique.

C) use logic instructions to impose conditionals
b*(a<0)
becomes

b&(a<0 ? 0xffff : 0x0000) (example fields are 16 bits and constants are in hex)

D) apply logical values to the calculation of storage addresses and array subscripts. This includes the technique of piling which conditionally suppresses the advancement of an array index which is being sequentially written. For example:

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

becomes

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

In this case, the two pieces of code are not exactly equivalent. The array c may need an extra guard index at the end. The user knows whether or not to discard the last value in c by inspecting the final value of i.

Add/Shift

Processors that have partitioned arithmetic often have ADD instructions that act on each field independently. Some of these processors have other kinds of field-by-field instructions (e.g., partitioned arithmetic right shift which shifts right, does not shift one field into another, and does copy the MSB of the field, the sign bit, into the just vacated MSB).

Comparisons and Field Masks

Some of these processors have field-by-field comparison instructions, generating multiple condition bits. If not, the partitioned subtract instruction is often pressed into service for this function. In this case, a<b is computed as a-b with a minus sign indicating true and a plus sign indicating false. The other bits of the field are not relevant. Such a result can be converted into a field mask of all 1's for true or all 0's for false, as used in the example in C) of Table 2, by means of a partitioned

arithmetic right shift with a sufficiently long shift. This results in a multi-field comparison in two instructions.

If a partitioned arithmetic right shift is not available, a field mask can be constructed from the sign bit by means of four instructions found on all contemporary processors. These are set forth in Table 3.

Table 3

1. Set the irrelevant bits to zero by u = u & 0x8000
2. Shift to LSB of the field v = u >> 15 (logical shift right for 16 bit fields)
3. Make field mask w = (u-v) | u
4. A partitioned zero test on a positive field x can be performed by x + 0x7fff so that the sign bit is zero if and only if x is zero. If the field is signed, one may use x | x + 0x7fff. The sign bit can be converted to a field mask as described above.

Of course, the condition that all fields are zero can be tested in a single instruction by comparing the total (un-partitioned) word of fields to zero.

Representations

It is useful to define some constants. A zero word except for a "1" in the MSB position of each field-partition is called MSB. A zero word except for a "1" in the LSB position of each field-partition is called LSB. The number of bits in a bit-partition is B. Unless otherwise stated, all words are unsigned (Uint) and all right shifts are logical with zero fill on the left.

A single information bit in a multi-bit field-partition can be represented in many different ways. The mask representation has all of the bits of a given field-

partition equal to each other and equal to the information bit. Of course, the information bits may vary from one field-partition to another within a word.

Another useful representation is the MSB representation. The information bit is stored in the MSB position of the corresponding field-partition and the remainder of the field-partition bits are zero. Analogously, the LSB representation has the information bit in the LSB position and all others zero.

Another useful representation is the ZNZ representation where a zero information bit is represented by zeros in every bit of a field-partition and a "1" information bit otherwise. All of the mask, MSB, and LSB representations are ZNZ representations, but not necessarily vice versa.

Conversions

Conversions between representations may require one to a few word length instructions, but those instructions process all field-partitions simultaneously.

MSB -> LSB

As an example, an MSB representation x can be converted to an LSB representation y by a word logical right shift instruction, y = (((Uint) x) >> B). An LSB representation x is converted to an MSB representation y by a word logical left shift instruction, y = (((Uint) x) << B).

Mask -> LSB

The mask representation m can be converted to the MSB representation by clearing the non-MSB bits. On most processors, all field-partitions of a word can be converted from mask to MSB in a single "andnot" instruction (m $\wedge\sim$ MSB).

Likewise, the mask representation can be converted to the LSB representation by a single "andnot" instruction (m $\wedge\sim$ LSB).

MSB -> Mask

Conversion from MSB representation x to mask representation z can be done with the following procedure using word length instructions. See Table 4.

Table 4

1. Convert the MSB representation x to an LSB representation y.
2. Word subtract y from x giving v. This is the mask except for the MSB bits which are zero.
3. Word OR v with x to give the mask result z. The total procedure is z = $(x - (x >> B)) \vee x$.

ZNZ -> MSB

All of the field partitions of a word can be converted from ZNZ x to MSB y as follows. One may use the word add instruction to add to the ZNZ a word with zero bits in the MSB positions and "1" bits elsewhere. The result of this add may have the proper bit in the MSB position, but the other bit positions may have anything. This is remedied by applying an "andnot" instruction to clear the non-MSB bits. $y = (x + \sim msb) \wedge \sim MSB$.

Other

Other representations can be reached from the MSB representation as above.

Bit Output

In some applications (e.g., entropy codecs), one may want to form a bit string by appending given bits, one-by-one, to the end of the bit string. The current description will now indicate how to do this in a field-partition parallel way. The field partitions and associated bit strings may be independent of each other, each representing a parallel instance.

The process is to work the following way set forth in Table 5.

Table 5

1. Both the input bits and a valid condition are supplied in mask representation.
2. The information bits are conditionally (i.e. conditioned on valid true) appended until a field-partition is filled.
3. When a field-partition is filled, it is appended to the end of a corresponding field-partition string. Usually, the lengths of the field-partitions are all equal and a divisor of the word-length.

The not-yet-completely-filled independent field-partitions are held in a single word, called the accumulator. There is an associated bit-pointer word in which every field-partition of that word contains a single 1 bit (i.e. the rest zeros). That single 1 bit is in a bit position that corresponds to the bit position in the accumulator to receive the next appended bit for that field-partition. If the field-partition of the accumulator fills completely, the field-partition is appended to the corresponding field-partition string and the accumulator field-partition is reset to zero.

Information Bit Output

Appending (conditionally) the incoming information bit may be feasible. The input bit mask, the valid mask, and the bit-pointer are wordwise "ANDed" together

and then wordwise "ORed" with the accumulator. This takes 3 instruction executions per word on most processors.

Bit-Pointer Update

Assuming that the bits are being appended at the LSB end of the bit string, a non-updated bit-pointer bit in the LSB of a field-partition indicates that that field-partition is filled. In any case, the bit-pointer word may be updated by rotating each valid field-partition of the bit-pointer right one position. The method for doing this is as follows in Table 6.

Table 6

a) Separate the bit-pointer into LSB bits and non-LSB bits. (2 word AND instructions)
b) Word logical shift the non-LSB bits word right one. (1 word SHIFT instruction)
c) Word logical shift the non-LSB bits word left to the MSB positions (1 word SHIFT instruction)
d) Word OR the results of b) and c) together (1 word OR instruction)
e) Mux together bitwise the results of d) and the original bit-pointer. Use the valid mask to control the mux (1 XOR, 2 AND, and 1 OR word instructions on most processors)

Accumulator is Full

As stated above, a field-partition is full if the corresponding field-partition of the bit-pointer p has its 1 in the LSB partition. Any field-partition of the accumulator full is indicated by the word of LSB bits only of the bit-pointer p not zero. $f = (p \wedge LSB)$; $full = (f \neq 0)$

The probability of full is usually significantly less than 0.5 so that an application of piling is in order. Both the accumulator a and f are piled to pile A1, using full as the condition. The length of pile A1 may be significantly less than the number of bit append operations. Piling is designed so that processing does not necessarily involve control flow changes other than those involved in the overall processing loop.

At a later time, pile A1 is processed by looping through the items in A1. For each item in A1 the field-partitions are scanned in sequence. The number of field-partitions per word is small, so this sequence can be performed by straight-line code with no control changes.

One may expect that, on the average, only one field-partition in a word may be full. Therefore, another application of piling (to pile A2) is in order. Each of the field-partitions of a, a2, along with the corresponding field partition index i, are piled to A2 using the corresponding field-partition of f as the pile write condition. In the end, A2 may contain only those field-partitions that are full.

At a later time, pile A2 is processed by looping through the items of A2. The index I is used to select the bit-string array to which the corresponding a2 should be appended. The file-partition size in bits, B, is usually chosen to be a convenient power of two (e.g., 8 or 16 bits). Store instructions for 8 bit or 16 bit values make those lengths convenient. Control changes other than the basic loops are not necessarily required throughout the above processes.

Bit Field Scanning

A common operation required for codecs is the serial readout of bits in a field of a word. The bit to be extracted from a field x is designated by a bit_pointer, a field value of 0s except for a single "1" bit (e.g., 0x0200). The "1" bit is aligned with the bit to be extracted so that x & bit_pointer is zero or non-zero according to

the value of the read out bit. This can be converted to a field mask as described above. Each instruction in this sequence may simultaneously process all of the fields in a word.

The serial scanning is accomplished by shifting the bit_pointer in the proper direction and repeating until the proper terminating condition. Since not all fields may terminate at the same bit position, the above procedure may be modified so that terminated fields do not produce an output while unterminated fields do produce an output. This is accomplished by producing a valid field mask that is all "1"s if the field is unterminated or all "0"s if the field is terminated. This valid field mask is used as an output conditional. The actual scanning is continued until all fields are terminated, indicated by valid being a word of all zeros.

The terminal condition is often the bit in the bit_pointer reaching a position indicated by a "1" bit in a field of terminal_bit_pointer. This may be indicated by a "1" bit in bit_pointer& terminal_bit_pointer. These fields may be converted to the valid field mask as described above.

While it may appear that the present description has many sequential dependencies and a control flow change for each bit position scanned, this loop can be unrolled to minimize the actual compute time required. In the usual application of bit field scanning, the fields all have the same number of bits leading to a loop termination condition common to all of the fields.

Congruent Sub-Fields of Field-Partitions

If one wishes to append bit positions c:d of each field-partition of word w onto the corresponding bit-strings, one may let the constant c be a zero word except for a "1" in bit position c of each field-partition. Likewise, one may let the constant d be a zero word except for a "1" in bit position d of each field-partition. Moreover, the following operations may be performed. See Table 7.

Table 7

A) initialize the bit-pointer q to c q = c;

A1) initialize COND to all true

B) wordwise bitand q with w $u = q \wedge w$
u is in ZNZ representation

C) convert u from ZNZ representation to mask representation v

D) v can now be bit-string output as described above. Use a COND of all true.

E) if cond = (q == d) processing is done; otherwise wordwise logical
shift q right one (q >> 1) loop back to step B)

The average value of (d-c) is often quite small for entropy codec applications. The test in operation E) can be initiated as early as operation B) with the branch delayed to operation E) and operations B)-D) available to cover the branch pipeline delay. Also, since the sub-fields are congruent it is relatively easy to unroll the processing of several words to cover the sequential dependencies within the instructions for a single word of field-partitions.

Non-Congruent Sub-Fields of Field-Partitions

In the case that c and d vary by field-partition, c and d remain as above but the test in operation E) above varies by field-partition rather than being the same for all field-partitions of the word. In this case, one may want the scan-out for the completed field partitions to idle until all field-partitions have completed. One may need to modify the above procedure in the following ways in Table 8.

Table 8

1) Step D) may need a condition where the field-partition value is false for completed field-partitions and true for not-yet-completed field-partitions. This is accomplished by appending to operation E) an operation which "andnot" the cond word onto COND. COND = (COND $\wedge \sim$ cond)

2) The if condition in step E) needs to be modified to loop back to B) unless COND is all FALSE.

Thus, the operations become:

A) initialize the bit-pointer q to c q = c;

A1) initialize COND to all true

B) wordwise bitand q with w $u = q \wedge w$

u is in ZNZ representation

C) convert u from ZNZ representation to mask representation v

D) v can now be bit-string output as described above. Use a COND of all true.

E1) cond = (q == d); COND = (COND $\wedge \sim$ cond);

E2) if COND==0 processing is done; otherwise wordwise logical shift q

right one (q >> 1) loop back to operation B)

Binary to Unary - Bit Field Countdown

A common operation in entropy coding is that of converting a field from binary to unary – that is producing a string of n ones followed by a zero for a field whose value is n. In most applications, the values of n are expected to have a negative exponential distribution with a mean of one so that, on the average, one may expect to have just one "1" in addition to the terminal zero in the output.

A field-partition parallel method for positive fields with leading zeros is as follows. As above, let c be a constant all zeros except for a "1" in the MSB position

of each field of the word X. Let d be a constant all zeros except for a "1" in the LSB position of each field. Let diff = c – d. Initialize mask to diff.

The procedure is to count down (in parallel) the fields in question and at the same time carry up into the initially zero MSB position c. If the MSB position is a "1" after the subtraction, the previous value of the field was not zero and a "1" should be output. If the MSB position is a zero after the subtraction, the previous value of the field was zero and a zero should be output. In any case, the MSB position contains the bit to be output for the corresponding field-partition of the word X.

Once the field has reached zero and the first zero is output, further outputs of zero may be suppressed. Since different field-partitions of X may have different values and output different numbers of bits, output from the field-partitions having smaller values may be suppressed until all field values have reached zero. This suppression is implemented by means of the mask input to the bit output procedure, as described earlier. Once the first zero for a field-partition has been output, the corresponding field-partition of the mask is turned zero, suppressing further output.

In the usual case where diff is the same for each field-partition, it is not necessary to change diff to zero. Otherwise, diff may be ANDed with the mask. See Table 9.

Table 9

While mask $\neq 0$

$X = X + diff$

$Y = ZNZ_2_mask(c \wedge X)$ where ZNZ_2_mask is the ZNZ to mask conversion above

$X = X \wedge \sim c$

Output Y with mask as described above

mask = mask ∧ Y

In the case of typical pipeline latencies for jumps, it may make sense to unroll the above loop according to the estimated probability distribution of the number of its turns.

Optimizing Loop Unrolling for Partitioned Computations

If one has a loop of the form: while c, {s}, the probability of c==true on the ith iteration is P_i, the cost of computing c and looping back is $C(\mathbf{c})$, and the cost of computing s is $C(\mathbf{s})$. One may assume that extra executions of s do not affect the output of the computation but do each incur the cost $C(\mathbf{s})$.

One may unroll the loop n times so that the computation becomes s; s; s; . . . s; while c, {s} where there are n executions of s preceding the while loop. The total cost is then that set forth in Table 10.

Table 10

$$nC(\mathbf{s})+\left(C(\mathbf{c})+P_n\left(C(\mathbf{s})+C(\mathbf{c})+P_{n+1}(...)\right)\right)=$$
$$nC(\mathbf{s})+C(\mathbf{c})+\left(P_n+P_nP_{n+1}+...\right)\left(C(\mathbf{c})+C(\mathbf{s})\right)\approx$$
$$\approx(n-1)\alpha+U_n=TC(n,\alpha)$$
$$\textit{where}\quad U_n=\left(P_n+P_nP_{n+1}+...\right)\quad \textit{and}\quad \alpha=\frac{C(\mathbf{s})}{C(\mathbf{c})+C(\mathbf{s})}$$

As an example, one may suppose that he or she has k independent fields per word and that p is the probability of looping back for each individual field. Then, $P_n=1-\left(1-p^n\right)^k$.

Figure **4** shows a graph **400** illustrating P_n, in accordance with one embodiment. Figure **5** shows a graph **500** illustrating the corresponding U_n, in accordance with one embodiment. The curves in each figure correspond to the values of k with blue corresponding to k=1).

Figures **6** and **7** illustrate graphs **600** and **700** indicating the normalized total cost $TC(n,\alpha)$ for $\alpha = 0.3$ and $\alpha = 0.7$, respectively. Figure **8** is a graph **800** illustrating the minimal total cost $\min_n(TC(n,\alpha)) = \overline{TC}(\alpha)$ (dotted lines) and the optimal number of initial loop unrolls $\bar{n}(\alpha)$, in accordance with one embodiment.

Example

In entropy coding applications, output bits may have a 0.5 probability of being one and a 0.5 probability of being zero. They may also be independent. With these assumptions, one can make the following calculations.

The probability P(n) that a given field-partition may require n or less output bits (including the terminating zero) is P(n) = (1 - 0.5-n). Let the number of field-partitions per word be m. Then the probability that the required number of turns around the loop is n or less is $(P(n))^m = (1 - 0.5^{-n})^m$. Figure **9** illustrates a table **900** including various values of the foregoing equation, in accordance with one embodiment. As shown, unrolling of the loop above 2 - 4 times seems to be in order.

Codecs

In one embodiment, the coder portion **101** and/or decoder portion **103** of Figure **1** may include a variable modulus. In the context of the present description, the modulus may reflect a steepness of a probability distribution curve associated

with a compression algorithm utilized by the codec framework **100**. For example, the modulus may include a negative exponential of the probability distribution.

While the modulus may vary as a function of any desired parameter, the modulus may, in one embodiment, depend on a context of a previous set of data, where such set of data may refer to a set of bits being processed by the various modules of the codec framework **100**. Moreover, the modulus may avoid increasing as a function of a run length (i.e. a plurality of identical bits in a sequence).

A dyadic-monotonic (DM) codec framework may thus be provided. More information regarding optional ways in which the modulus may depend on a context of a previous set of data, the modulus may avoid increasing as a function of a run length, etc. will be set forth hereinafter in greater detail.

Figure **10** is a computational complexity v. performance level graph **1000** illustrating a relationship of the present dyadic-monotonic (DM) codec framework and other algorithms (i.e. Huffman, Rice Golomb, arithmetic, etc.). As shown, the DM codec framework may be designed to utilize a minimal computational complexity given a predetermined performance level.

More information regarding various optional features that may be implemented in the context of the codec framework of Figure **1** will now be set forth. In one embodiment, the DM codec may be specified by describing the state space and update function thereof (see Background section).

Each state has five components, the position P, the *context,* the *shift*, the *Aregister*, and the *Cregister*. As mentioned earlier, the modulus may vary based on a context of a previous set of data. In accordance with a specific embodiment meeting the aforementioned definition of a "context," the present *context* may include a bit string of length k over the input alphabet (total 2^k states). Each of the *Aregister* and *Cregister* may hold a non-negative multiple of 2^{-n} that is less than one

(total 2^n states each). In both the <*start*> state and the <*finish*> state, both the *Aregister* and the *Cregister* have the value zero. The context value for the <*start*> state, *initial*, though arbitrary, may be necessarily the same for both the encoder and decoder. The P value in the <*start*> state is start and in the <*finish*> state is finish. The *shift* value is irrelevant in the <*start*> and <*finish*> states.

As a part of the specification of the update function ϕ there are some specified fixed functions. See Table 11.

Table 11

1. The function *mps* maps each *context* value to a value in the input alphabet *I*. The intention is that *mps* is the symbol in *I* that is the more probable symbol given the *context*.

2. The function *delta* maps each of the 2^k *context* values to 2^{-m} where $0 < m \leq n$. The intention of the function *delta* is that it quantitatively captures the information about the probability of the value of the next symbol.

These two functions may be chosen subject to the DM constraints. These constraints, together with probability information, may be used in choosing *mps* and *delta* and enable a useful combination of algorithm simplicity and entropy coding efficiency in important applications.

In one embodiment, the DM constraints are those set forth in Table 12.

Table 12

1) The Dyadic constraint:

$$delta(context) = 2^{-m} \quad 0 < m \leq n$$

where m and n are integers, requires that *delta* be a negative integral power of two. It is intended that *delta*(*context*) approximate the conditional probability that the next symbol will not necessarily be the one given by *mps*(*context*). 1-Prob(a == *mps*(*context*)| *context*) ≈ *delta*(*context*)

2) The Monotonic constraint:

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

The right hand side of such inequality is approximately the probability of reading the most probable symbol given that the previously read symbol was also most probable. The monotonic constraint reflects the plausible situation where a more probable symbol does not decrease the probability that the next symbol will be most probable.

These two constraints provide an effective entropy codec.

Figure **11** shows a transition table **1100** illustrating an update function for both an encoder and decoder, in accordance with one embodiment. Each line represents a set of states, where the states of the set are those that satisfy each of the conditions in the predicate columns. Each row forms a partition of the allowable states.

After the proper row is identified for the initial state, the actions in the right hand part of the row are executed. All values utilized are values at the initial state, so action sequence within a row is not necessarily an issue. Each component of the new state may receive a unique value. Blank entries mean that that component of the state is unchanged.

For the encoder, the update actions from the "common" group of columns and the update actions from the "encoder" group of columns are carried out. For the decoder, the actions are chosen from the "common" and "decoder" groups of columns. At the bottom of the state transition table **1100,** precise definitions of the action are provided.

The effect of the DM conditions is that the *Aregister* is always a multiple of the last *delta* added (at F13). The dyadic condition ensures that the binary representation of delta has exactly one "1" bit. The monotonic condition ensures that delta not become larger until a code symbol is produced, so that the bit in delta remains only in the same position or moves to the right. This situation remains until a code symbol is produced, at which point the *Aregister* becomes zero (precisely because only the *Aregister* bits to the right of the last delta are preserved).

As a result, the following set of properties of Table 13 is provided.

Table 13

a) A is a multiple of delta
b) A is zero after writing a code symbol
c) Renormalization after writing a code symbol is unnecessary

For this same reason, the *Cregister*, required in general arithmetic coding, is not necessarily used in the DM encoder. Immediately after producing a code symbol, the entire memory of the preceding state sequence is captured in the *context*. Since the *context* is the previous k input symbols, the DM codec has a short-term memory and consequently adapts quickly to the local statistics of the input string.

Various applications of the present framework exist. For example, in the context of significant functions; image, video, and signal processing often involves a

transform whose purpose is to "concentrate" the signal, i.e., produce a few large coefficients and many negligible coefficients (which are discarded by replacing them with zero). The identity (or location) of the non-negligible coefficients is usually as important as their values. This information is often captured in a "significance function" which maps non-negligible coefficients to "1" and negligible ones to "0."

By listing the coefficients and their significance bits in an appropriate order, a significance bit can be predicted with good accuracy from its immediate predecessors. If that order lists the coefficients in descending order by their expected magnitude, one may obtain a significance bit string that begins with predominantly 1's and ends with predominantly 0's. Such a string, whose statistics change as the string goes on, is called *non-stationary*. Effective entropy coding of such a string may require a memory of the immediately preceding context. This memory may be extensive enough to achieve good prediction accuracy and short-lived enough to allow sufficiently rapid adaptation.

In the context of coding non-stationary runs and in accordance with the aforementioned definition, a *run* within the significance function may include a substring of bits where all but the last bit have one value and the last bit has the other value. The next run begins immediately after the last bit of the preceding run.

As a general rule, the more bits in *context* (the larger k), the more closely the coding rate approximates the limiting entropy. However, the larger k, the more expensive the implementation. The range of k is sufficiently limited that each of the values of k can be examined. Once k is selected, the performance of the DM codec for non-stationary runs is captured entirely in the *mps(context)* and *delta(context)* functions. These functions may be approximated from empirical data as follows in Table 14.

Table 14

1) $mps(context) = 1$ if Prob(next symbol = 1 | *context*) ≥ 0.5

2) $\text{Prob}(2^{2*delta(context)} \mid context) = 0.5$

The procedure is to collect sufficient empirical data, qualified by context, and for each context form a histogram. From this, the probability functions can be approximated. The function *mps(context)* can be calculated directly. Moreover, the function *delta(context)* can be calculated by an iterative solution of 2) above in Table 14.

The DM codec thus maps input strings 1:1 into coded strings and a coded string, when decoded, yields the original input string. Not all output strings can necessarily be generated as the encode of some input string. Some input strings may encode to shorter coded strings – many may encode to longer coded strings. Regarding the length of an input string vis-à-vis the coded string to which it encodes, it may be helpful to describe the probabilities of occurrence of various possible input strings. If the codec has useful compression properties, it may be that the probability of occurrence of an input string which encodes to a short string is much larger than the probability of occurrence of an input string which encodes to a long string.

Dynamic probabilities need not necessarily apply. For entropy coding applications, the statistics of the significance bitstream can and does change often and precipitously. Such changes cannot necessarily be tracked by adaptive probability tables, which change only slowly even over many runs. The DM coder therefore does not necessarily use probability tables; but rather adapts within either the last few bits or within a single run.

Empirical tests with significance bits data indicate that most of the benefit of the *context* is obtained with only the last few bits of the significance bit string. These last few bits of the significance string, as *context*, are used to condition the probability of the next bit.

The important probability quantity $p_{context}$ = Prob(next input bit = LSB | *context*). It may be noted that, by the definition of LSB, $p_{context} < 0.5$. The entropy that may be added to minimally represent that next bit is as follows in Table 15.

Table 15

$$entropy = -((1- p_{context})*\log_2(1- p_{context}) + p_{context} * \log_2(p_{context})) \text{ bits}$$

Then *delta(context)* = *entropy*/2 because the *Aregister* is scaled to output the 2^{-1} bit. If $p_{context}$ ~0.5, then the entropy that may be added for that next bit is approximately as follows in Table 16.

Table 16

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

Moreover, *delta(context)* = ½. If $p_{context} << 0.5$, the entropy that may be added for that next bit is approximately that set forth in Table 17.

Table 17

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

Moreover, *delta(context)* = $p_{context} * \log_2(1/p_{context})/2$. See equation 2) above in Table 14.

Visual Aspects

Figure **12** illustrates a method **1200** for compressing data with chrominance temporal rate reduction, in accordance with one embodiment. In one embodiment, the present method **1200** may be carried out in the context of the transform module

102 of Figure **1** and the manner in which it carries out a reversible transform. It should be noted, however, that the method **1200** may be implemented in any desired context.

In operation **1202**, luminescence (luma) data of a frame is updated at a first predetermined rate. In operation **1204**, chrominance (chroma) data of the frame is updated at a second predetermined rate that is less than the first predetermined rate.

In a digital video compression system, it is thus possible to vary the effective rate of transmitting temporal detail for different components of the scene. For example, one may arrange the data stream so that some components of the transformed signal are sent more frequently than others. In one example of this, one may compute a three-dimensional (spatial + temporal) wavelet transform of a video sequence, and transmit the resulting luma coefficients at the full frame rate.

Moreover, one may omit one or more higher-frequency bands from the chroma signal, thus in effect lowering the temporal response rate – the temporal detail fidelity – of the video for chroma information. During reconstruction of the compressed video for viewing, one may fill in or interpolate the omitted information with an approximation, rather than showing a "zero level" where there was no information transmitted. This may be done in the same way as for omitted spatial detail. Most simply it can be done by holding the most-recently received level until new information is received. More generally, it can be done by computing an inverse wavelet filter, using zero or some other default value for the omitted information, to produce a smoothed variation with the correct overall level but reduced temporal detail.

One particular example of this sort of chroma rate compression is as follows: for the chroma components, one may compute an average across two frames (four fields) of the spatially transformed chroma values. This may be accomplished by applying a double Haar wavelet filter pair and discarding all but the lowest

frequency component. One may transmit only this average value. On reconstruction, one can hold the received value across two frames (four fields) of chroma. It has been found that viewers do not notice this, even when they are critically examining the compression method for flaws.

The following stage of the video compression process, quantization, discards information by grouping similar values together and transmitting only a representative value. This discards detail about exactly how bright an area is, or exactly what color it is. When one finds that a transformed component is near zero, zero is chosen for the representative value (denoting no change at a particular scale). One can omit sending zero and have the receiver assume zero as its default value. This helps compression by lowering the amount of data sent. Again, human visual sensitivity to levels is known to differ between luma and chroma.

Thus, one can take advantage of this fact by applying different levels of quantization to luma and chroma components, discarding more information from the chroma. When this different quantization is done following a temporal or 3-D transform, the effect is to reduce the temporal detail in the chroma band along with the spatial detail. In a typical case, for ordinary video material, the temporal transform results in low frequency components that are much larger than higher frequency components. Then, applying quantization to this transformed result groups the small values with zero, in effect getting them omitted from the compressed representation and lowering the temporal resolution of the chroma components.

Figure **12A** illustrates a method **1250** for compressing data with a high-quality pause capability during playback, in accordance with one embodiment. In one embodiment, the present method **1250** may be carried out in the context of the framework of Figure **1**. It should be noted, however, that the method **1250** may be implemented in any desired context.

In operation **1252**, video data is compressed. In one embodiment, the data compression may be carried out in the context of the coder portion **101** of the framework of Figure **1**. Of course, such compression may be implemented in any desired context.

In operation **1254**, pause information is inserted with the compressed data. In one embodiment, the pause information may be used to improve a quality of the played back video data. Moreover, the pause information may include a high-resolution frame. Still yet, the pause information may include data capable of being used to construct a high-resolution frame.

Thus, in operation **1256**, the pause information may be used when the video data is paused during the playback thereof. In the present method, the compressed video is equipped with a set of extra information especially for use when the video is paused. This extra information may include a higher-quality frame, or differential information that when combined with a regular compressed frame results in a higher-quality frame.

In order to keep the compression bit rate at a useful level, this extra information need not be included for every frame, but rather only for some frames. Typically, the extra information may be included for one frame of every 15 or so in the image, allowing a high-quality pause operation to occur at a time granularity of ½ second. This may be done in accord with observations of video pausing behavior. One can, however, include the extra information more often than this, at a cost in bit rate. One can also include it less often to get better compression performance, at a cost in user convenience. The tradeoff may be made over a range from two frames to 60 or more frames.

In one embodiment, the extra information may include a whole frame of the video, compressed using a different parameter set (for example, quantizing away less

information) or using a different compression method altogether (for example, using JPEG-2000 within an MPEG stream). These extra frames may be computed when the original video is compressed, and may be carried along with the regular compressed video frames in the transmitted or stored compressed video.

In another embodiment, the extra information may include extra information for the use of the regular decompression process rather than a complete extra frame. For example, in a wavelet video compressor, the extra information might consist of a filter band of data that is discarded in the normal compression but retained for extra visual sharpness when paused. For another example, the extra information might include extra low-order bits of information from the transformed coefficients, and additional coefficients, resulting from using a smaller quantization setting for the chosen pausable frames.

In another embodiment, the extra information may include data for the use of a decompression process that differs from the regular decompression process, and is not a complete frame. This information, after being decompressed, may be combined with one or more frames of video decompressed by the regular process to produce a more detailed still frame.

More information regarding an exemplary wavelet-based transformation will now be set forth, which may be employed in combination with the various features of Figures **1** and **12A**. It should be highly noted, however, that such wavelet-based transformation is set forth for illustrative purposes only and should not be construed as limiting in any manner. For example, it is conceived that the various features of Figures **1** and **12A** may be implemented in the context of a DCT-based algorithm or the like.

Figure **13** illustrates a method **1300** for compressing/decompressing data, in accordance with one embodiment. In one embodiment, the present method **1300** may be carried out in the context of the transform module **102** of Figure **1** and the

manner in which it carries out a reversible transform. It should be noted, however, that the method **1300** may be implemented in any desired context.

In operation **1302**, an interpolation formula is received (i.e. identified, retrieved from memory, etc.) for compressing data. In the context of the present description, the data may refer to any data capable of being compressed. Moreover, the interpolation formula may include any formula employing interpolation (i.e. a wavelet filter, etc.).

In operation **1304**, it is determined whether at least one data value is required by the interpolation formula, where the required data value is unavailable. Such data value may include any subset of the aforementioned data. By being unavailable, the required data value may be non-existent, out of range, etc.

Thereafter, an extrapolation operation is performed to generate the required unavailable data value. See operation **1306**. The extrapolation formula may include any formula employing extrapolation. By this scheme, the compression of the data is enhanced.

Figure **14** shows a data structure **1400** on which the method **1300** is carried out. As shown, during the transformation, a "best fit" **1401** may be achieved by an interpolation formula **1403** involving a plurality of data values **1402**. Note operation **1302** of the method **1300** of Figure **13**. If it is determined that one of the data values **1402** is unavailable (see **1404**), an extrapolation formula may be used to generate such unavailable data value. More optional details regarding one exemplary implementation of the foregoing technique will be set forth in greater detail during reference to Figure **15**.

Figure **15** illustrates a method **1500** for compressing/decompressing data, in accordance with one embodiment. As an option, the present method **1500** may be carried out in the context of the transform module **102** of Figure **1** and the manner in

which it carries out a reversible transform. It should be noted, however, that the method **1500** may be implemented in any desired context.

The method **1500** provides a technique for generating edge filters for a wavelet filter pair. Initally, in operation **1502**, a wavelet scheme is analyzed to determine local derivatives that a wavelet filter approximates. Next, in operation **1504**, a polynomial order is chosen to use for extrapolation based on characteristics of the wavelet filter and a numbers of available samples. Next, extrapolation formulas are derived for each wavelet filter using the chosen polynomial order. See operation **1506**. Still yet, in operation **1508**, specific edge wavelet cases are derived utlizing the extrapolation formulas with the available samples in each case.

Moreover, additional optional information regarding exemplary extrapolation formulas and related information will now be set forth in greater detail.

One of the transforms specified in the JPEG 2000 standard 1) is the reversible 5-3 transform shown in Equations #1.1 and 1.2.

Equations #1.1 and 1.2

$$Y_{2n+1} = X_{2n+1} - \left\lfloor \frac{X_{2n} + X_{2n+2}}{2} \right\rfloor \qquad eq\ 1.1$$

$$Y_{2n} = X_{2n} + \left\lfloor \frac{Y_{2n-1} + Y_{2n+1} + 2}{4} \right\rfloor \qquad eq\ 1.2$$

To approximate Y_{2N-1} from the left, one may fit a quadratic polynomial from the left. Approximating the negative of half the 2nd derivative at 2N-1 using the available values yields Equation # 1.1.R.

Equation # 1.1.R

$$Y_{2N-1} = -\frac{1}{3}\left(X_{2N-1} - \left\lfloor \frac{3X_{2N-2} - X_{2N-4} + 1}{2} \right\rfloor \right) \quad eq\ 1.1.R$$

Equation # 1.1.R may be used in place of Equation #1.1 when point one is right-most. The apparent multiply by 3 can be accomplished with a shift and add. The division by 3 is trickier. For this case where the right-most index is $2N-1$, there is no problem calculating Y_{2N-2} by means of Equation #1.2. In the case where the index of the right-most point is even (say $2N$), there is no problem with Equation #1.1, but Equation #1.2 involves missing values. Here the object is to subtact an estimate of Y from the even X using just the previously calculated odd indexed Y s, Y_1 and Y_3 in the case in point. This required estimate at index $2N$ can be obtained by linear extrapolation, as noted above. The appropriate formula is given by Equation #1.2.R.

Equation #1.2.R

$$Y_{2N} = X_{2N} + \left\lfloor \frac{3Y_{2N-1} - Y_{2N-3} + 2}{4} \right\rfloor \quad eq\ 1.2.R$$

A corresponding situation applies at the left boundary. Similar edge filters apply with the required extrapolations from the right (interior) rather than from the left. In this case, the appropriate filters are represented by Equations #1.1.L and 1.2.L.

Equations #1.1.L and 1.2.L

$$Y_0 = -\frac{1}{3}\left(X_0 - \left\lfloor \frac{3X_1 - X_3 + 1}{2} \right\rfloor\right) \qquad eq\ 1.1.L$$

$$Y_0 = X_0 + \left\lfloor \frac{3Y_1 - Y_3 + 2}{4} \right\rfloor \qquad eq\ 1.2.L$$

The reverse transform fiters can be obtained for these extrapolating boundary filters as for the original ones, namely by back substitution. The inverse transform boundary filters may be used in place of the standard filters in exactly the same circumstances as the forward boundary filters are used. Such filters are represented by Equations #2.1.Rinv, 2.2.Rinv, 2.1.L.inv, and 2.2.L.inv.

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

$$X_{2N-1} = -3Y_{2N-1} + \left\lfloor \frac{3X_{2N-2} - X_{2N-4} + 1}{2} \right\rfloor \qquad eq\ 2.1.R\ \ inv$$

$$X_{2N} = Y_{2N} - \left\lfloor \frac{3Y_{2N-1} - Y_{2N-3} + 2}{4} \right\rfloor \qquad eq\ 2.2.R\ \ inv$$

$$X_0 = -3Y_0 + \left\lfloor \frac{3X_1 - X_3 + 1}{2} \right\rfloor \qquad eq\ 2.1.L\ \ inv$$

$$X_0 = Y_0 - \left\lfloor \frac{3Y_1 - Y_3 + 2}{4} \right\rfloor \qquad eq\ 2.2.L\ \ inv$$

Thus, one embodiment may utilize a reformulation of the 5-3 filters that avoids the addition steps of the prior art while preserving the visual properties of the filter. See for example, Equations #3.1, 3.1R, 3.2, 3.2L.

Equations #3.1, 3.1R, 3.2, 3.2L

$$Y_{2n+1} = (X_{2n+1} + 1/2) - \left\lfloor \frac{(X_{2n} + 1/2) + (X_{2n+2} + 1/2)}{2} \right\rfloor \qquad eq\ 3.1$$

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

$$(Y_{2n} + 1/2) = (X_{2n} + 1/2) + \left\lfloor \frac{Y_{2n-1} + Y_{2n+1}}{4} \right\rfloor \qquad eq\ 3.2$$

$$(Y_0 + 1/2) = (X_0 + 1/2) + \left\lfloor \frac{Y_1}{2} \right\rfloor \qquad eq\ 3.2L$$

In such formulation, certain coefficients are computed with an offset or bias of ½, in order to avoid the additions mentioned above. It is to be noted that, although there appear to be many additions of ½ in this formulation, these additions need not actually occur in the computation. In Equations #3.1 and 3.1R, it can be seen that the effects of the additions of ½ cancel out, so they need not be applied to the input data. Instead, the terms in parentheses $(Y_0 + 1/2)$ and the like may be understood as names for the quantities actually calculated and stored as coefficients, passed to the following level of the wavelet transform pyramid.

Just as in the forward case, the JPEG-2000 inverse filters can be reformulated in the following Equations #4.2, 4.2L, 4.1, 4.1R.

Equations #4.2, 4.2L, 4.1, 4.1R

$$(X_{2n} + 1/2) = (Y_{2n} + 1/2) - \left\lfloor \frac{Y_{2n-1} + Y_{2n+1}}{4} \right\rfloor \qquad eq\ 4.2$$

$$(X_0 + 1/2) = (Y_0 + 1/2) - \left\lfloor \frac{Y_1}{2} \right\rfloor \qquad eq\ 4.2L$$

$$(X_{2n+1} + 1/2) = Y_{2n+1} + \left\lfloor \frac{(X_{2n} + 1/2) + (X_{2n+2} + 1/2)}{2} \right\rfloor \qquad eq\ 4.1$$

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

As can be seen here, the values taken as input to the inverse computation are the same terms produced by the forward computation in Equations #3.1 ~ 3.2L and the corrections by ½ need never be calculated explicitly.

In this way, the total number of arithmetic operations performed during the computation of the wavelet transform is reduced.

While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

CLAIMS

What is claimed is:

1. A method for processing exceptions, comprising:
processing computational operations in a loop;
identifying exceptions while processing the computational operations;
storing the exceptions while processing the computational operations; and
processing the exceptions separate from the loop.

2. The method as recited in claim 1, wherein the computational operations including non-significant values.

3. The method as recited in claim 2, wherein the computational operations include counting a plurality of zeros.

4. The method as recited in claim 1, wherein the computational operations include at least one of clipping and saturating.

5. The method as recited in claim 1, wherein the exceptions include significant values.

6. The method as recited in claim 5, wherein the exceptions include non-zero data.

7. The method as recited in claim 1, wherein the computational operations are processed at least in part utilizing a transform module.

8. The method as recited in claim 1, wherein the computational operations are processed at least in part utilizing a quantize module.

9. The method as recited in claim 1, wherein the computational operations are processed at least in part utilizing an entropy code module.

10. The method as recited in claim 1, wherein the storing includes storing data required to process the exceptions.

11. The method as recited in claim 1, wherein the processing is carried out to compress data.

12. The method as recited in claim 11, wherein the data is compressed utilizing a de-correlating transform.

13. The method as recited in claim 11, wherein the data is compressed utilizing a wavelet transform.

14. The method as recited in claim 11, wherein the data is compressed utilizing a discrete cosine transform.

15. A computer program product for processing exceptions, comprising:

computer code for processing computational operations in a loop;

computer code for identifying exceptions while processing the computational operations;

computer code for storing the exceptions while processing the computational operations; and

computer code for processing the exceptions separate from the loop.

16. A system for processing exceptions, comprising:

at least one data compression module selected from the group consisting of a transform module, a quantize module and an entropy code module, the at least one data compression module adapted for processing computational operations in a loop, identifying exceptions while processing the computational operations, storing the exceptions while processing the computational operations, and processing the exceptions separate from the loop.

17. A coder comprising a variable modulus.

18. The coder as recited in claim 17, wherein the modulus reflects a steepness of a probability distribution curve associated with a compression algorithm.

19. The coder as recited in claim 18, wherein the modulus includes a negative exponential of the probability distribution.

20. The coder as recited in claim 18, wherein the probability distribution is associated with a codec.

21. The coder as recited in claim 18, wherein the codec is designed to utilize a minimal computational complexity given a predetermined performance level.

22. The coder as recited in claim 17, wherein the modulus depends on a context of a previous set of data.

23. The coder as recited in claim 17, wherein the modulus avoids increasing as a function of a run length.

24. The coder as recited in claim 23, wherein the run length includes a plurality of identical bits in a sequence.

25. The coder as recited in claim 17, wherein the coder includes an entropy coder.

26. A decoder comprising a variable modulus.

27. The decoder as recited in claim 26, wherein the modulus reflects a steepness of a probability distribution curve associated with a compression algorithm.

28. The decoder as recited in claim 27, wherein the modulus includes a negative exponential of the probability distribution.

29. The decoder as recited in claim 27, wherein the probability distribution is associated with a codec.

30. The decoder as recited in claim 27, wherein the codec is designed to utilize a minimal computational complexity given a predetermined performance level.

31. The decoder as recited in claim 26, wherein the modulus depends on a context of a previous set of data.

32. The decoder as recited in claim 26, wherein the modulus avoids increasing as a function of a run length.

33. The decoder as recited in claim 32, wherein the run length includes a plurality of identical bits in a sequence.

34. The decoder as recited in claim 26, wherein the decoder includes an entropy decoder.

35. A method for using a codec including a variable modulus that reflects a steepness of a probability distribution curve associated with a compression algorithm and does not increase with a run length.

36. A method for compressing video data, comprising:

updating luminescence data of a frame at a first predetermined rate; and

updating chrominance data of the frame at a second predetermined rate that is less than the first predetermined rate.

37. The method as recited in claim 36, wherein one or more frequency bands of the chrominance data is omitted.

38. The method as recited in claim 37, wherein the one or more frequency bands are omitted utilizing a filter.

39. The method as recited in claim 38, wherein the filter includes a wavelet filter.

40. The method as recited in claim 37, and further comprising interpolating omitted portions of the chrominance data upon the decompression of the video data.

41. A computer program product for compressing video data, comprising:

computer code for updating luminescence data of a frame at a first predetermined rate; and

computer code for updating chrominance data of the frame at a second predetermined rate that is less than the first predetermined rate.

42. The computer program product as recited in claim 41, wherein one or more frequency bands of the chrominance data is omitted.

43. The computer program product as recited in claim 42, wherein the one or more frequency bands are omitted utilizing a filter.

44. The computer program product as recited in claim 43, wherein the filter includes a wavelet filter.

45. The computer program product as recited in claim 42, and further comprising interpolating omitted portions of the chrominance data upon the decompression of the video data.

46. A method for compressing video data, comprising:
compressing video data;
inserting pause information with the compressed data; and
wherein the pause information is used when the video data is paused during the playback thereof.

47. The method as recited in claim 46, wherein the pause information is used to improve a quality of the played back video data.

48. The method as recited in claim 47, wherein the pause information includes a high-resolution frame.

49. The method as recited in claim 47, wherein the pause information includes data capable of being used to construct a high-resolution frame.

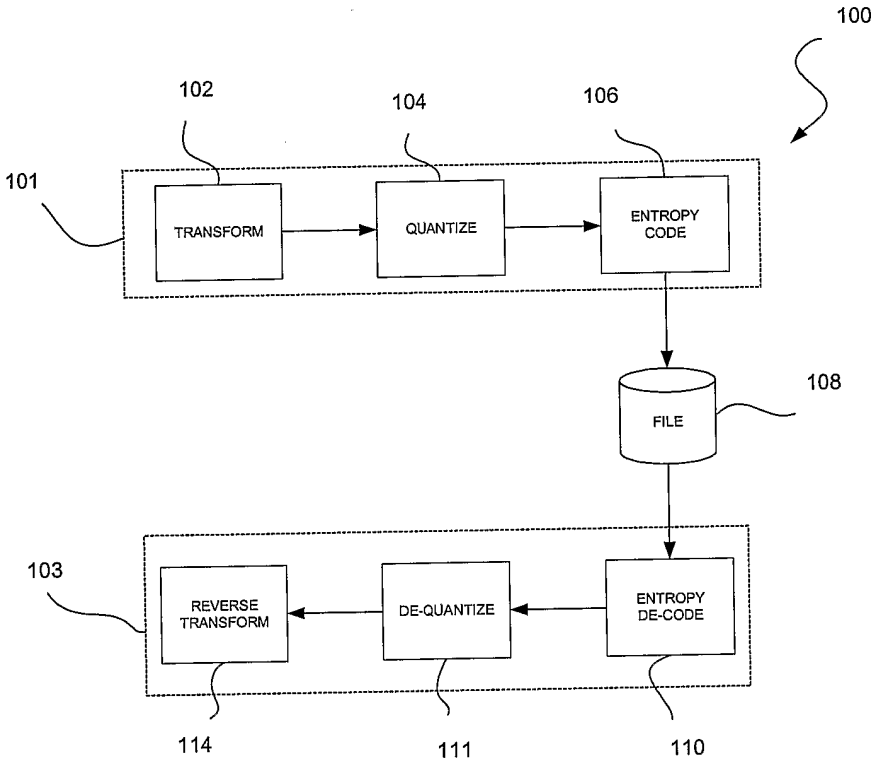

100
102
104
106
101
TRANSFORM
QUANTIZE
ENTROPY CODE
108
FILE
103
REVERSE TRANSFORM
DE-QUANTIZE
ENTROPY DE-CODE
114
111
110

FIG. 1

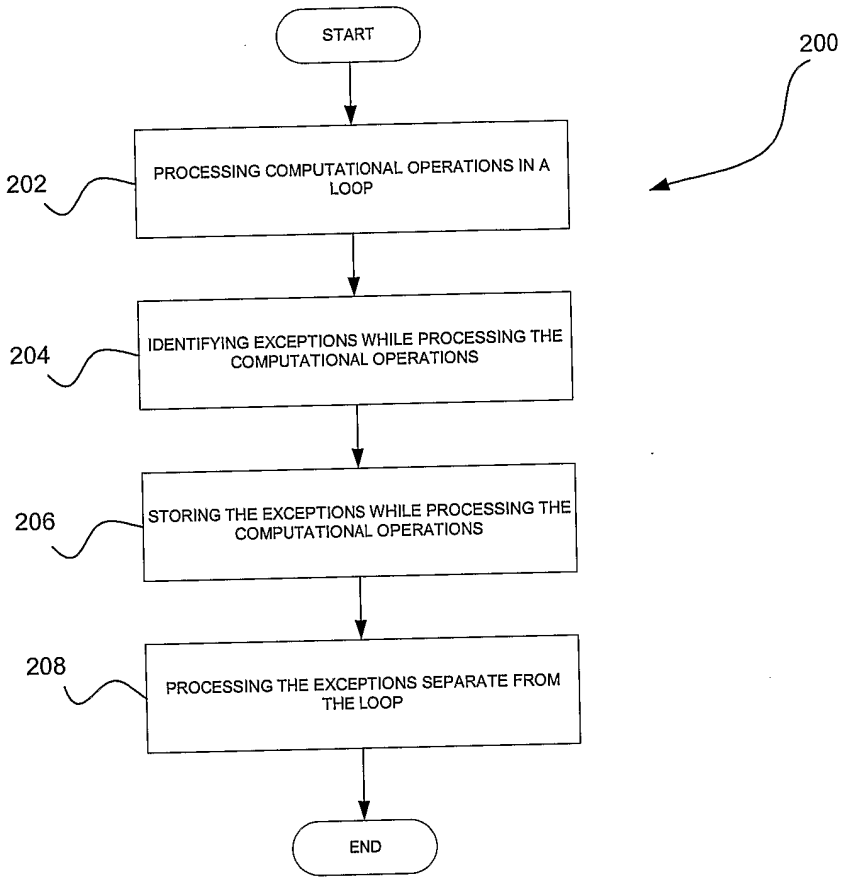

START
200
PROCESSING COMPUTATIONAL OPERATIONS IN A LOOP
202
IDENTIFYING EXCEPTIONS WHILE PROCESSING THE COMPUTATIONAL OPERATIONS
204
STORING THE EXCEPTIONS WHILE PROCESSING THE COMPUTATIONAL OPERATIONS
206
PROCESSING THE EXCEPTIONS SEPARATE FROM THE LOOP
208
END

FIG. 2

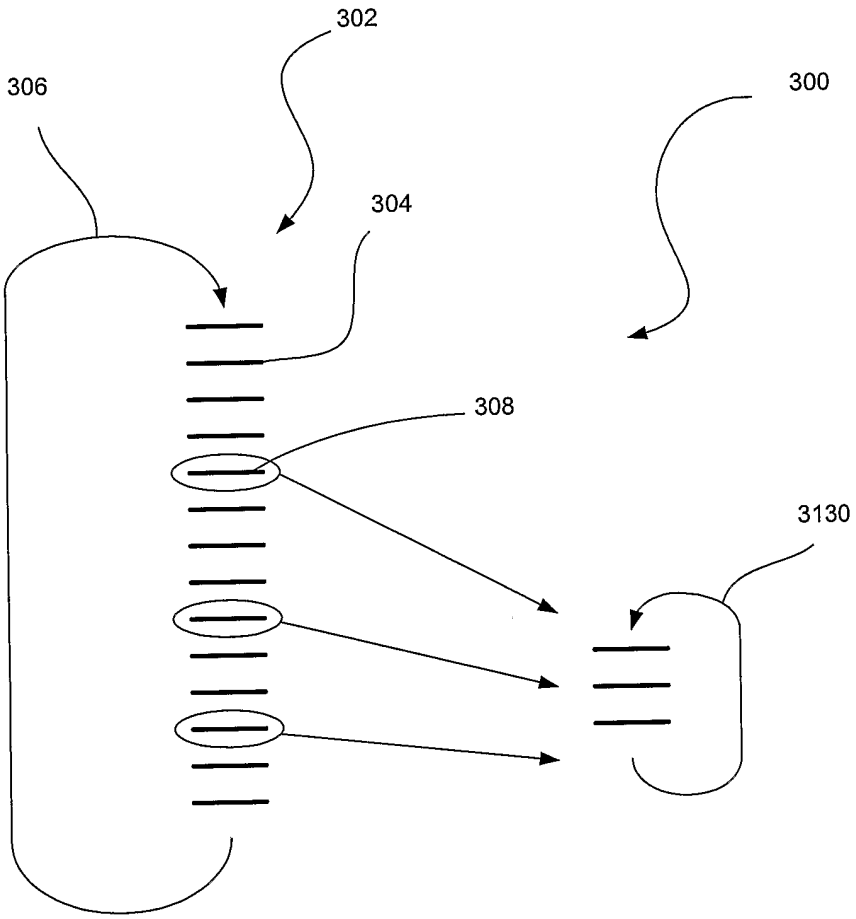


FIG. 3

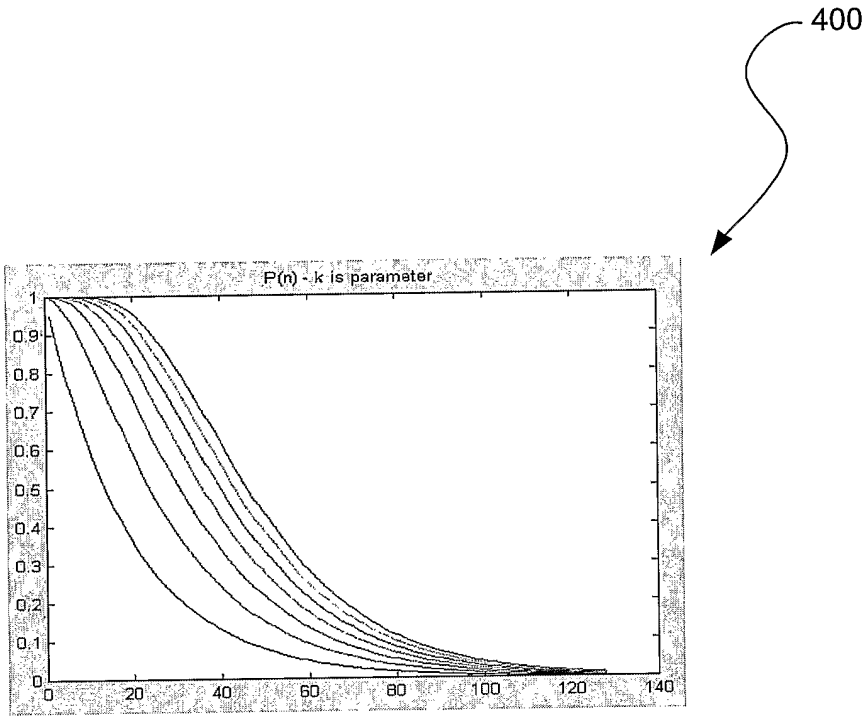

400
P(n) - k is parameter
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
20
40
60
80
100
120
140

FIG. 4

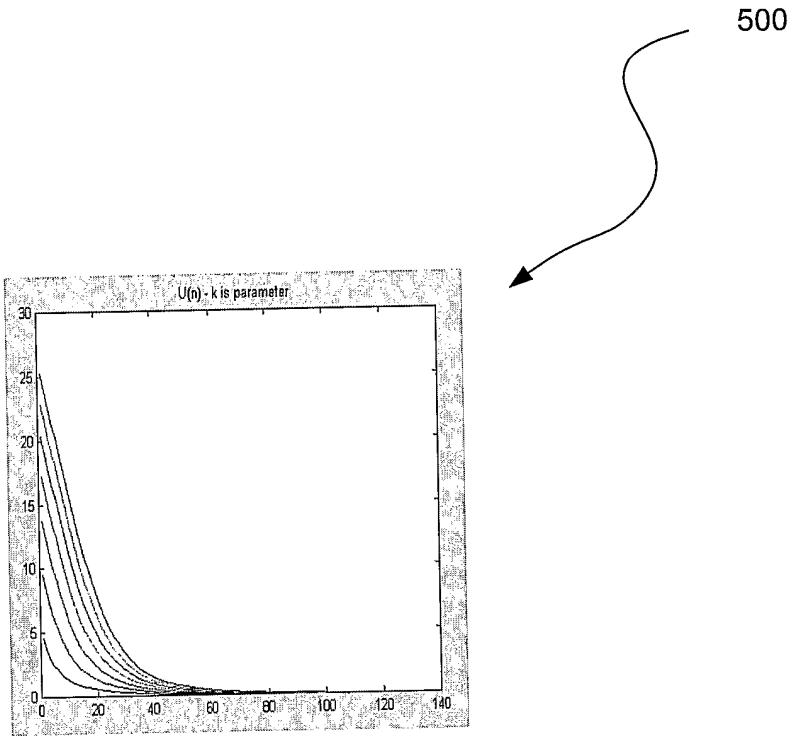


FIG. 5

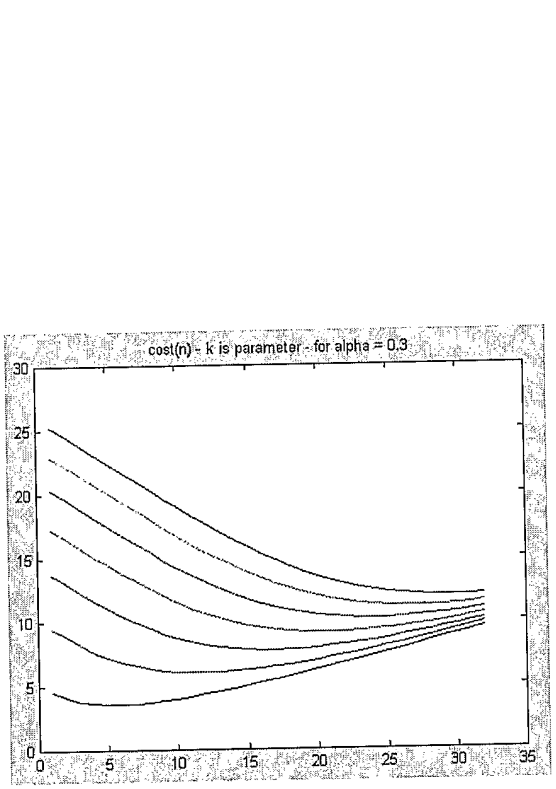

600
cost(n) - k is parameter - for alpha = 0.3
30
25
20
15
10
5
0
0
5
10
15
20
25
30
35

FIG. 6

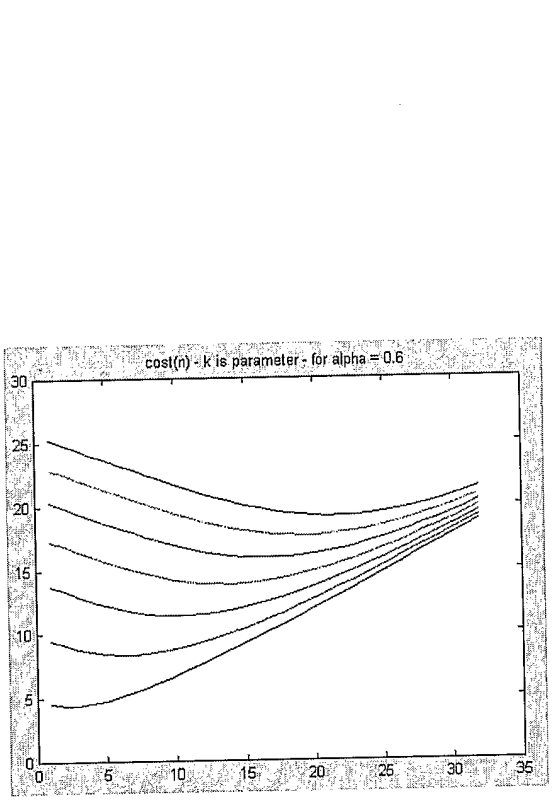

700
cost(n) - k is parameter - for alpha = 0.6
30
25
20
15
10
5
0
0
5
10
15
20
25
30
35

FIG. 7

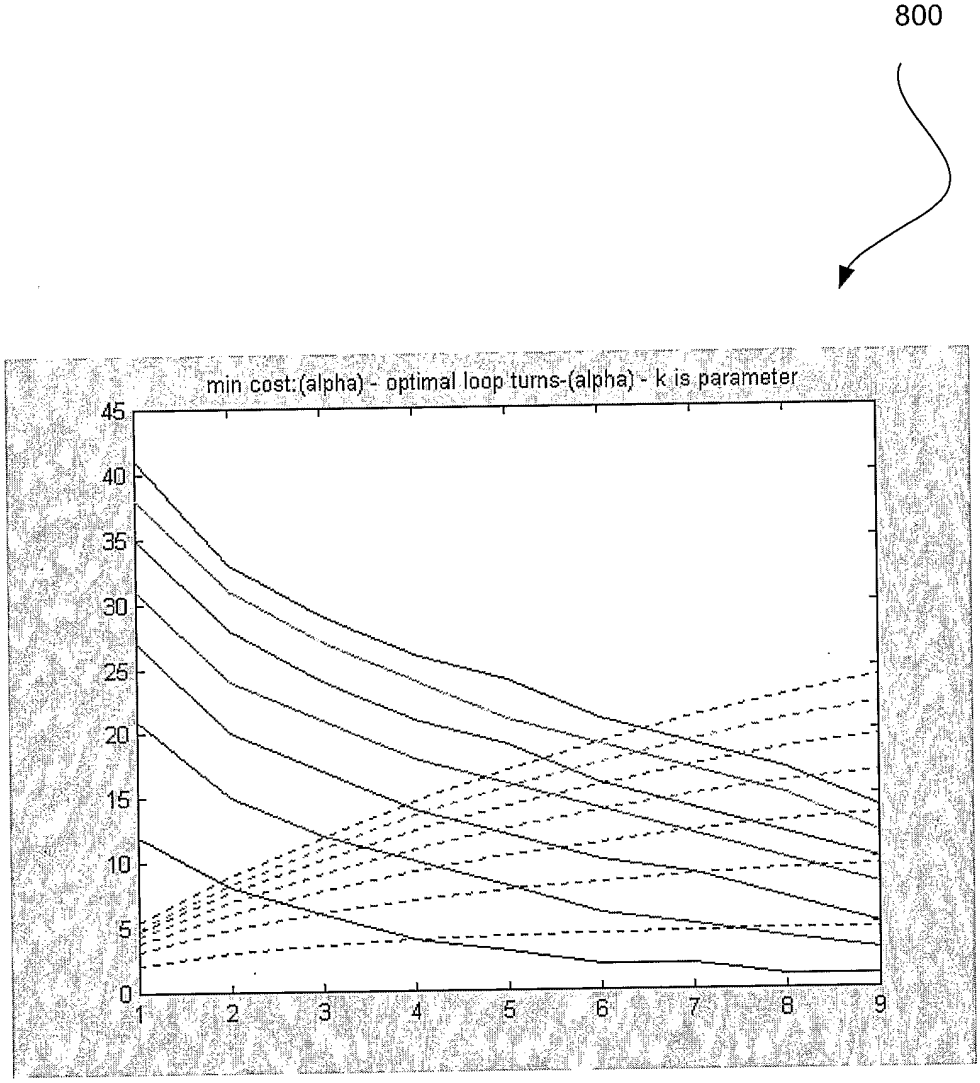

800
min cost:(alpha) - optimal loop turns-(alpha) - k is parameter
45
40
35
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8
9

FIG. 8

900



	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

FIG. 9

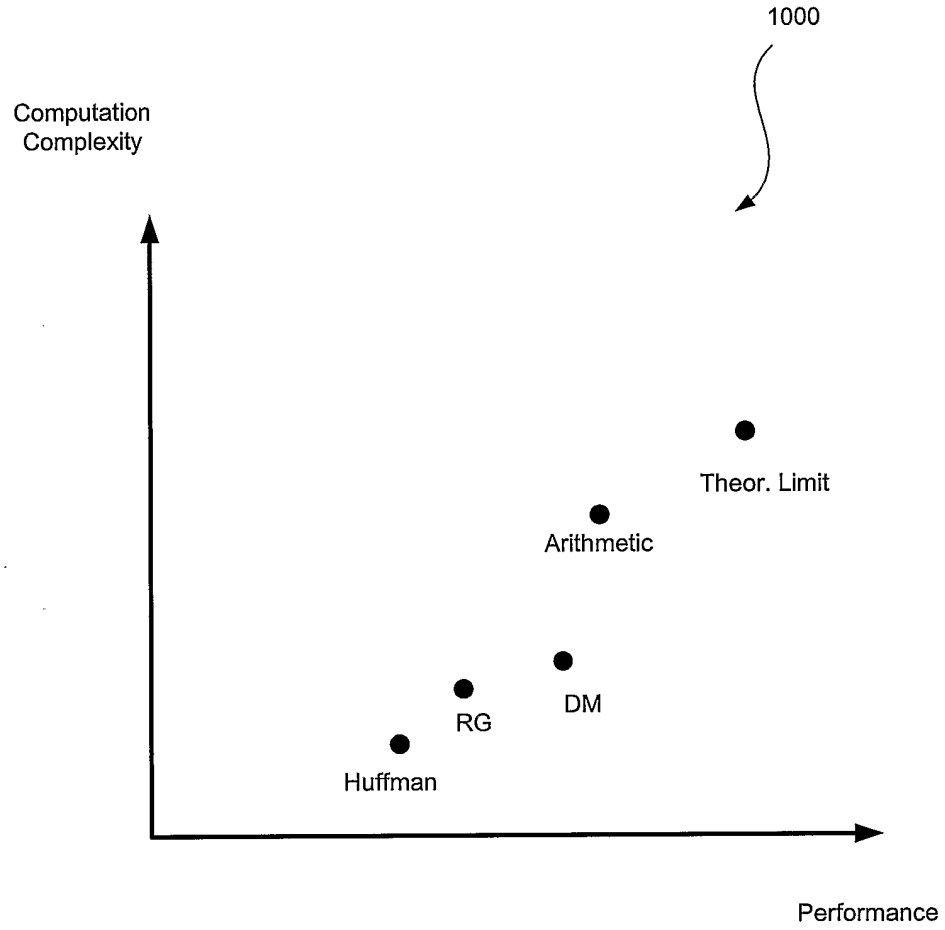

1000
Computation Complexity
Theor. Limit
Arithmetic
DM
RG
Huffman
Performance

FIG. 10

	A	B	C	D	E	F	G	H	I	J	K	L
1	COMBINED ENCODE AND DECODE											
2										ENCODE ACTIONS	DECODE ACTIONS	
3	PREDICATES					COMMON 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)

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

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

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.



FIG. 11

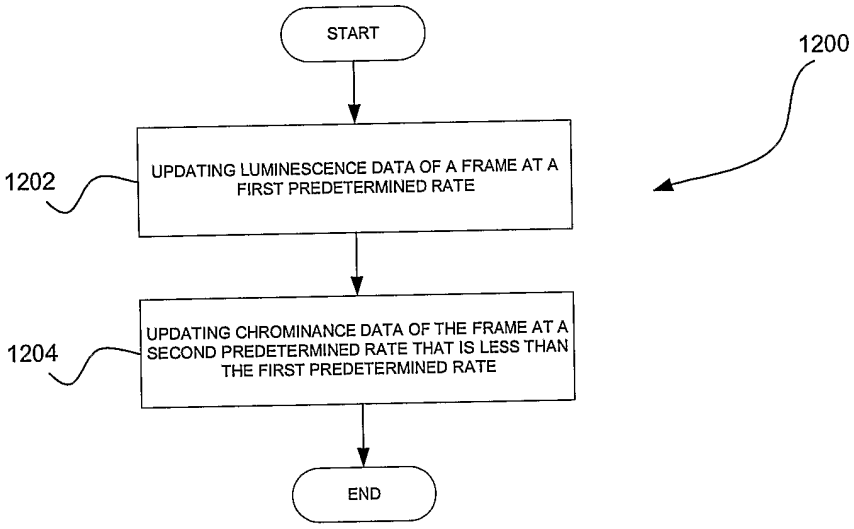

START
1200
1202
UPDATING LUMINESCENCE DATA OF A FRAME AT A FIRST PREDETERMINED RATE
1204
UPDATING CHROMINANCE DATA OF THE FRAME AT A SECOND PREDETERMINED RATE THAT IS LESS THAN THE FIRST PREDETERMINED RATE
END

FIG. 12

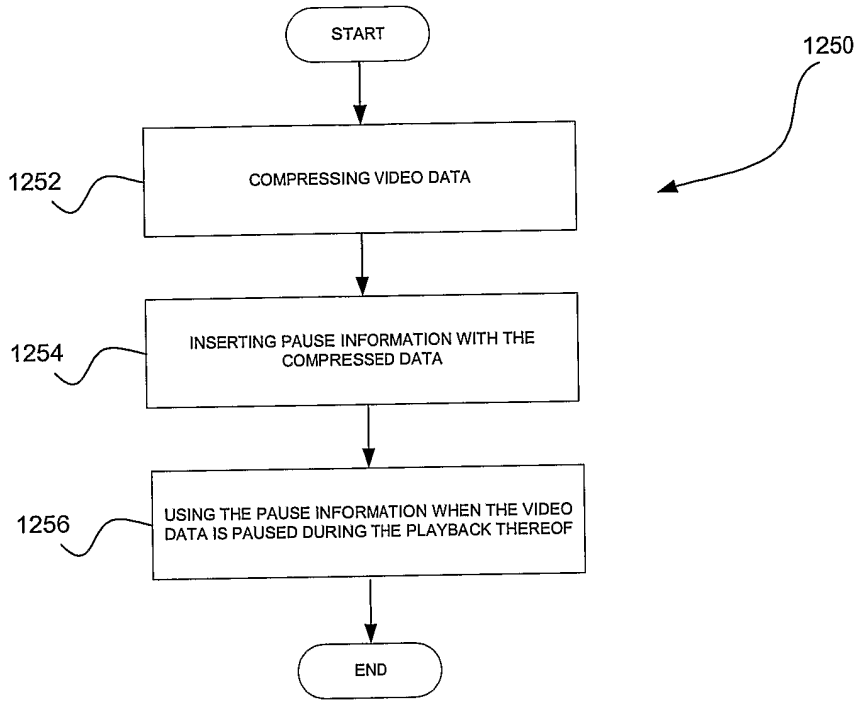

START
1250
1252
COMPRESSING VIDEO DATA
1254
INSERTING PAUSE INFORMATION WITH THE COMPRESSED DATA
1256
USING THE PAUSE INFORMATION WHEN THE VIDEO DATA IS PAUSED DURING THE PLAYBACK THEREOF
END

FIG. 12A

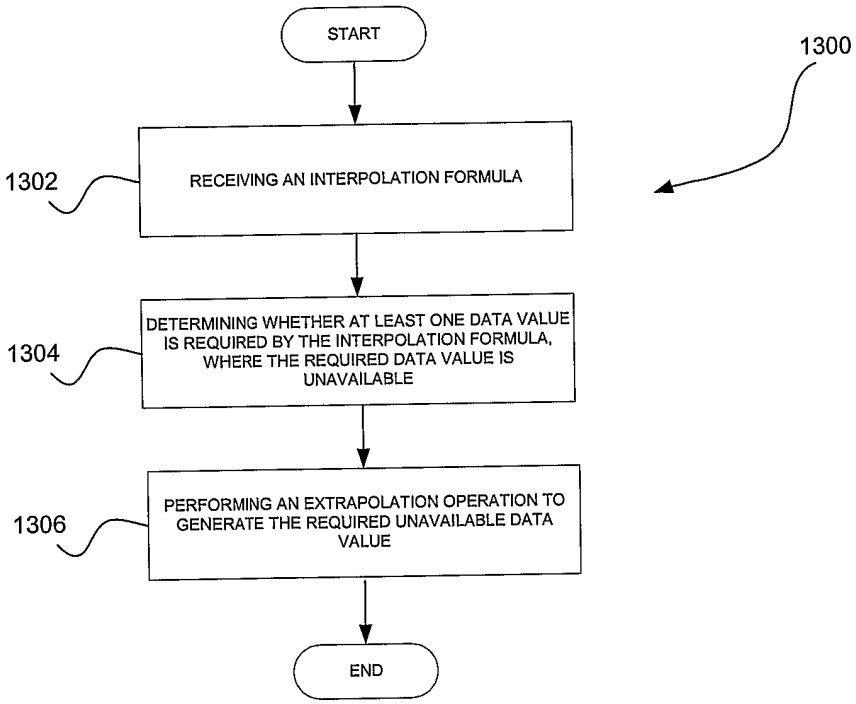

START
1300
1302
RECEIVING AN INTERPOLATION FORMULA
1304
DETERMINING WHETHER AT LEAST ONE DATA VALUE IS REQUIRED BY THE INTERPOLATION FORMULA, WHERE THE REQUIRED DATA VALUE IS UNAVAILABLE
1306
PERFORMING AN EXTRAPOLATION OPERATION TO GENERATE THE REQUIRED UNAVAILABLE DATA VALUE
END

FIG. 13

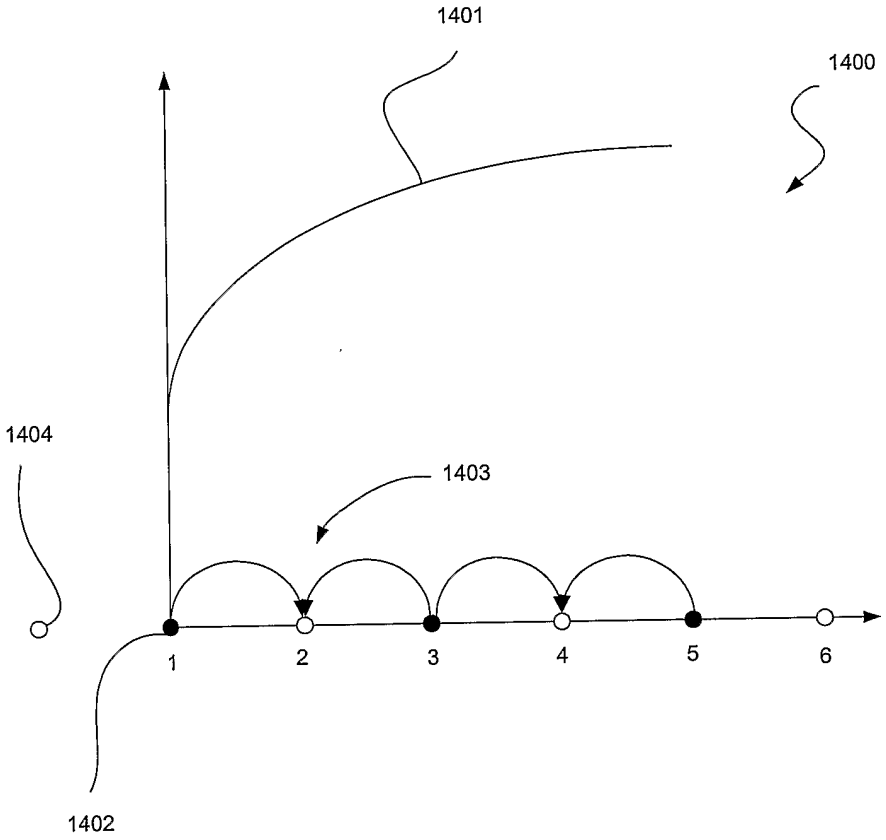


FIG. 14

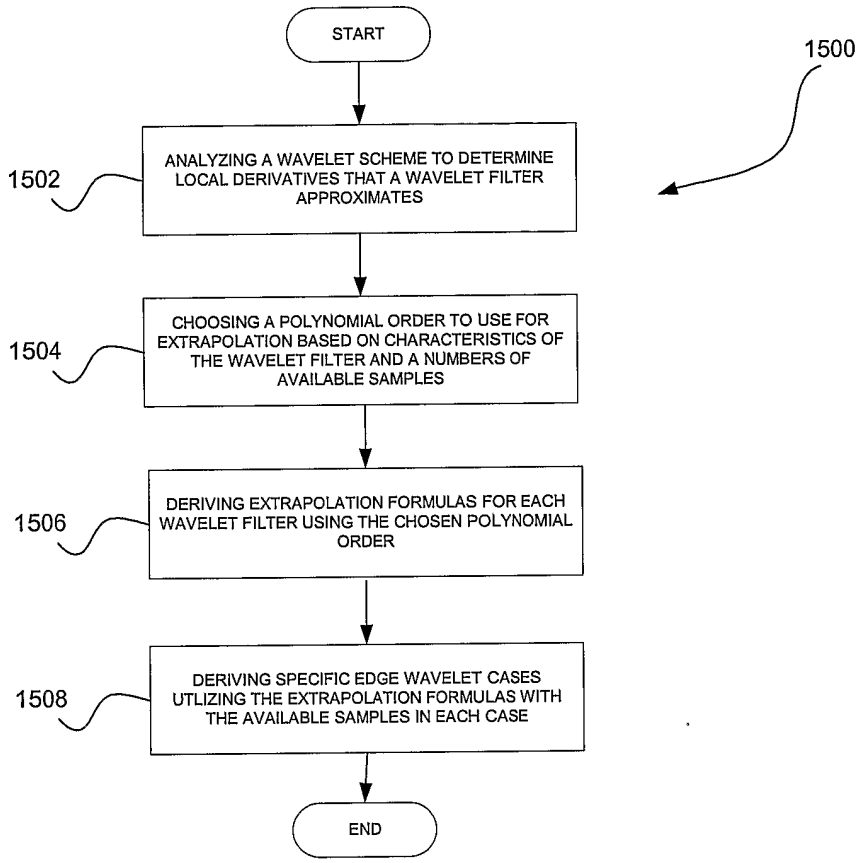

START
1500
ANALYZING A WAVELET SCHEME TO DETERMINE LOCAL DERIVATIVES THAT A WAVELET FILTER APPROXIMATES
1502
CHOOSING A POLYNOMIAL ORDER TO USE FOR EXTRAPOLATION BASED ON CHARACTERISTICS OF THE WAVELET FILTER AND A NUMBERS OF AVAILABLE SAMPLES
1504
DERIVING EXTRAPOLATION FORMULAS FOR EACH WAVELET FILTER USING THE CHOSEN POLYNOMIAL ORDER
1506
DERIVING SPECIFIC EDGE WAVELET CASES UTLIZING THE EXTRAPOLATION FORMULAS WITH THE AVAILABLE SAMPLES IN EACH CASE
1508
END

FIG. 15

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US03/16908

A. CLASSIFICATION OF SUBJECT MATTER

IPC(7) : G06F 17/14; G06F 12/00
US CL : 708/401; 711/125

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
U.S. : 708/401; 711/125

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5,893,145 A (THAYER ET AL.) 06 April 1999 (06.04.1999), column 6, lines 1-67; column 7, lines 1-67; column 8, lines 1-67; column 9, lines 1-67; column 10, lines 1-36	1-49
A	US 6,141,673 A (THAYER ET AL.) 31 October 2000 (31.10.2000), column 6, lines 1-67; column 7, lines 1-67; column 8, lines 1-67; column 9, lines 1-67; column 10, lines 1-45	1-49

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
29 July 2003 (29.07.2003)	29 SEP 2003
Name and mailing address of the ISA/US Mail Stop PCT, Attn: ISA/US Commissioner for Patents P.O. Box 1450 Alexandria, Virginia 22313-1450 Facsimile No. (703)305-3230	Authorized officer Apu Mofiz for James R. Matthews Telephone No. 7036054240

Form PCT/ISA/210 (second sheet) (July 1998)