Architectural Advances in the VLSI Implementation of Arithmetic Coding for Binary Image Compression

Gennady Feygin, Patrick Glenn Gulak, and Paul Chow[1] [2]
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, Canada M5S 1A4

Abstract

This paper presents some recent advances in the architecture for the data compression technique known as Arithmetic Coding. The new architecture employs loop unrolling and speculative execution of the inner loop of the algorithm to achieve a significant speed-up relative to the Q-Coder architecture. This approach reduces the number of iterations required to compress a block of data by a factor that is on the order of the compression ratio. While the speed-up technique has been previously discovered independently by researchers at IBM, no systematic study of the architectural trade-offs has ever been published. For the CCITT facsimile documents, the new architecture achieves a speed-up of approximately seven compared to the IBM Q-coder when four lookahead units are employed in parallel. A structure for fast Input/Output processing based on run length pre-coding of the data stream to accompany the new architecture is also presented.

1 Introduction

Data compression has found widespread use in data storage, and audio and video transmission. In all of the above cases it is advantageous to compress data to conserve scarce resources (disk space in storage, bandwidth in sound and image transmission).

Arithmetic Coding [1, 2] is a data compression technique that represents the source data as a fraction that assumes a value between zero and one. The encoding algorithm is based upon recursive subdivision of an interval and retaining one of the created subintervals as a new interval for the next step of the recursion. The advantages of Arithmetic Coding include working directly with symbols, without resorting to construction of multi-symbol sequences (extensions of the source alphabet). This makes Arithmetic Coding particularly well suited to the encoding of binary images, where the probabilities of occurrence of two symbols are frequently skewed.

[1]Many thanks to Dr. Joan Mitchell and Dr. Ronald Arps for their thoughtful comments on an earlier version of this paper. Financial support for this research was provided by NSERC and ITRC.
[2]e-mail:gfeygin@eecg.toronto.edu

A group at IBM has reported a CMOS realization of an Arithmetic encoder/decoder for binary images in [3]. An updated version of the system, known as the Q-coder, appeared in [4]. The simplified encoding and decoding processes are described in Figures 1 and 2 respectively. The variable C keeps track of the code string, while the variable A keeps track of the interval as it is being subdivided. Q_e is the probability of the Less Probable Symbol. An efficient mechanism for adaptively varying the value of Q_e based on the frequencies of occurrence of ones and zeros in the data stream is suggested in [5].

In Section 2 we apply loop unrolling to the Q-coder algorithm and demonstrate how the Q-coder algorithm can be made faster through processing multiple source pels at a time. Section 3 discusses the techniques that must be applied to eliminate the bottleneck at the input to the encoder (and symmetrically at the output of the decoder). Section 4 discusses the trade-offs inherent in designing and implementing a high-performance arithmetic coder. Finally, Section 5 gives a summary of research and indicates possible directions for future work.

2 Loop Unrolling

We present a scheme for achieving parallelism in Arithmetic Coding using a variant of a common technique used in compiler optimization called *Loop Unrolling* [6], which exposes more parallelism to the hardware. This technique was investigated in [7], and in this paper we extend their investigation and consider an optimum hardware implementation of the technique. For instance, the encoder described in [4, 5] considers the source symbols sequentially, one symbol at a time. We may place at the encoder a second unit that considers input symbols two at a time. When the probability of the *More Probable Symbol* (MPS) is much greater than the probability of the *Less Probable Symbol* (LPS), the probability of two MPS's in a row is much greater than the sum of the probabilities of all other two-symbol combinations. Thus the most likely sequence of events will be as follows:

1. Examine the first input bit; detect an MPS. Update A and C: $A \leftarrow A - Q_e$, $C \leftarrow C - Q_e$.
2. Compare A against $A_{\text{min}} = 0.75$. Find that $A > A_{\text{min}}$, therefore no renormalization (which causes Q_e to be updated) is required.
3. Examine the second input bit; detect an MPS. Update A and C: $A \leftarrow A - Q_e$, $C \leftarrow C - Q_e$.

Thus the most likely outcome of the sequence of processing two input bits is simply the following update: $A \leftarrow A - 2 \times Q_e$, $C \leftarrow C - 2 \times Q_e$. These computations are easy to perform (multiplication by 2 is implemented by wiring, with no circuitry required). It is natural to combine the two updates into one, thus saving one iteration. This is analogous to *loop unrolling*, with two iterations of the updates of the variables A and C combined into a single iteration of the new algorithm. What makes this modification of the basic algorithm particularly appealing is the possibility of combining the two operations that must be performed on each of the variables A and C into only a single operation on each variable. Of course there will be a small number of cases where advancing by two bits is not possible, due either to the presence of an LPS in the second bit position, or the need to re-normalize A and C, and update Q_e following the $A \leftarrow A - Q_e$ computation. Thus we must preserve the ability to perform the original one-bit update — our attempt to combine two iterations into one must be treated as *speculative*. The best technique is to compute both $A_1 \leftarrow A - Q_e$ and $A_2 \leftarrow A - 2 \times Q_e$, while simultaneously checking for the LPS and the re-normalization required, and selecting A_1 or A_2 as the new, updated value of A. The input data stream is also updated by removing one or two bits from the input stream, as appropriate. If $p(\text{MPS}) \gg p(\text{LPS})$, then the two-bit updates will dominate, allowing the processing rate to increase by a factor of almost two.

Of course there is nothing magical about examining the input stream two bits at a time. In principle, we could choose to examine k bits at a time and increase the processing speed by a factor of approximately k. There are obvious practical limitations on what values of k are worth considering — for instance in computing $A_k \leftarrow A - k \times Q_e$ we must restrict k to be a power of two, if we are to avoid the expense and delay of a true multiplication. As k grows larger it becomes increasingly difficult to get k input bits and to verify whether or not they are all MPS's. Yet this verification must be fast enough to keep up with the circuitry used to update A and C. The technique for providing fast I/O will be discussed at length in Section 3.

A technique described in [7] requires that an integer quotient $(A - 0.75)/Q_e$ be computed to calculate the value k. Although this may be perfectly acceptable in a software implementation running on a general-purpose computer, a fast division operation is expensive in hardware[3] and would add significantly to the cycle time (since it must precede the actual update of the A and C registers).

Thus we propose a modification of the approach of [4]. This modification consists of allowing simultaneous lookahead by multiple powers of two $(2, 4, \ldots 2^{k_{\max}})$ to be performed. While occurrences of a large number of MPS's in a row are less frequent than those of the smaller number of MPS's in a row, each incidence of accepting a long string gives greater benefit than accepting a short string. Thus maintaining separate circuitry for large lookahead values may be advantageous, even if used only infrequently. Any string of length $N < 2^{k_{\max}+1}$ can be processed in $O(\lfloor \log_2(N) \rfloor)$ iterations using $O(\lfloor \log_2(N) \rfloor)$ adders for updating A and C. In the original Q-Coder, N iterations are required using one adder for A and one for C.

The pseudo-code describing the operation of the proposed encoder is presented in Figure 3. Note that the operation of lookahead by two is slightly different from all other powers of two. Since we know that $Q_e < 0.75$, we can accept lookahead by two even when re-normalization is caused by *the second MPS*. This is easily verified by considering A_1. If $A_1 \geq 0.75$ then we can guarantee that the re-normalization is not required until at least the second MPS. Thus we can always take advantage of lookahead by two. A similar enhancement for any other power of two is impossible:

[3]Recall that the Q-coder is designed to be multiplication-free.

for instance, to guarantee that lookahead by eight is acceptable, we must ensure that the re-normalization occurs after the eighth bit at the earliest. For this we must know $A_7 = A - 7 \times Q_e$. Since computing A_7 requires (expensive) multiplication, we avoid it by slightly tightening the constraint and verifying that re-normalization is not required until the ninth bit at the earliest ($A_8 \geq 0.75$). Occasionally, we may miss an opportunity to perform lookahead by the maximum amount possible, but we avoid the penalties (time and circuitry) required to perform the multiplication. For the case of lookahead by two, where the result of the "multiplication" is available for free, we use it.

As we process our input data stream, we frequently encounter long strings of MPS's with only a few LPS's, however, each LPS contains a large amount of information, while each MPS contains only a little. In the original Q-Coder, either a single MPS or a single LPS was processed in one iteration, resulting in an encoded output whose rate was uneven. In our scheme with lookahead, we process a single LPS or *multiple* MPS's in one iteration. Thus the amount of *information* being processed per iteration is more nearly balanced. Since the number of output bits must be (in the ideal case) proportional to the amount of information being transmitted, our design with lookahead results in reduced jitter at the encoded output end. The situation at the decoder end is similar.

3 Input Data Stream Processing

As lookahead is introduced into the *Interval Update Core* (IUC) of the encoder, the speed of the encoder will become increasingly limited by the speed at which uncompressed data can be supplied and examined for the presence of an LPS in the data stream. Some kind of buffering must be provided to examine up to $2^{k_{max}}$ input symbols at one time. A simple solution is to use a simple form of run-length codes. Whenever a run is terminated, a number indicating the length of the run is deposited in a FIFO that is placed between the run-length encoder and the IUC. This also removes input processing from the critical path of the IUC iteration, and allows all input processing to be pipelined. The IUC "consumes" run lengths from the FIFO and loads them into its own counter, and proceeds to encode the run. At each step of the IUC operations an internal counter is decremented by an appropriate amount. It is advantageous to generate packets that correspond to the number of MPS's followed by an LPS in a manner similar to that of [8]. Thus i MPS's followed by one LPS followed by j MPS's followed by one LPS will require only two packet locations inside the FIFO (i, j). This scheme is superior to the usual run-length encoding, since the probability of two or more LPS's in a row is very low (if the source information is compressible), and since processing more than one LPS at a time is impossible (the IUC performs re-normalization following every LPS). This scheme also has certain shortcomings, primarily the need for an escape sequence to indicate the occurrence of a string of MPS's that is longer than can be stored inside the counter.

4 Performance Driven Design

As discussed previously, IUC performance can be immproved by employing lookahead into the input sequence. To determine precisely how much speed-up can be obtained, we have encoded the eight CCITT FAX images [9, 10]. The seven-pel predictor with *context* used in [10] was employed.

4.1 Speed-up Evaluation

The methodology of examining the speed-up that can be obtained with various arrangements of the IUC is quite simple. We begin by running a simulation that assumes a fully parallel IUC, with a lookahead by every power of 2 up to a maximum[4] of 1024. We record the number of iterations in which lookahead by $1, 2, \ldots, 1024$ MPS's was employed. The results for such an experiment are summarized in Table 1 for the CCITT1 image.

Note that the three largest lookahead values account for for well over 50% of the processed bits, but under 2% of the number of iterations. This gives a clear indication that the better encoder configurations will not be confined to using only the smallest lookahead values. It is also important to note that the processing of single LPS's and single MPS's together account for over 75% of the iterations of the encoder. Thus any additional speed-up improvement must involve reducing the number of iterations spent processing single pels.

It is convenient to represent an encoder by assigning a 12-bit binary number that has a one in position i if lookahead by 2^i is provided and a zero if lookahead by 2^i is not provided. Thus, under this classification, the Q-coder encoder is 000000000001 (or $0X001$ in hexadecimal notation), the fully parallel encoder/decoder is 111111111111 or $0XFFF$, and the decoder with lookahead by $1024, 64, 8, 1$ pel is 010001001001 or $0X449$. It is then relatively simple to perform an exhaustive search of all possible combinations, noting the best encoder and the number of iterations required when using the system with $1, 2, \ldots,$ lookahead units for each of the eight CCITT images. We have determined the "best" encoders for each CCITT image when $1, 2, \ldots, 12$ lookahead units are employed. The results are summarized in Figure 4, with each curve corresponding to one of the CCITT documents (indicated by a number).

We make the following observations: even in the worst case (least-compressible, CCITT7 document) up to four lookahead units may be employed with approximately linear speed-up and is therefore a reasonable choice for a lookahead value. Only one question remains outstanding: which particular encoder with four lookahead units is the best? Unfortunately, there is no one encoder for a particular value of lookahead[5] that is best for all eight CCITT documents. We can narrow our choice by looking at candidate encoder configurations that have been found to be optimum for at least one of the eight CCITT documents. The results are summarized in Table 2.

[4]The maximum is the highest power of two that is less than or equal to $\frac{A_{\min}}{\min(Q_e)}$

[5]Except, of course, for the trivial cases of one and twelve lookahead units.

The configuration $0X95$ is a clear winner, with the best average speed-up of 8.48. It is the optimum encoder configuration for *three* out of the eight CCITT documents. It is also important to note that the worst encoder configuration, $0X2b$, is on average only 8% slower than the best configuration. Thus any one of the four configurations shown in Table 2 can be expected to perform reasonably well. Yet another criterion that must be considered when choosing an encoder configuration is keeping the largest lookahead value used in an encoder as small as possible, since it reduces the wiring required to provide shifted (i.e., multiplied by powers of two) copies of the Q_e. The configurations $0X4b$ and $0X55$ require one bit less shifting than the configuration $0X95$. The diagram of a four-lookahead unit encoder with configuration $0X95$ is shown in Figure 5.

4.2 Single-Cycle Processing of Multiple Pels in an Area with Changing Context

Consider again Table 1. We have already pointed out that the lion's share of the clock cycles are spent in processing the pels one at a time. Even if all the time spent processing two, four, etc. pels at a time was reduced to zero, 75% of the clock cycles would still be required to continue processing single MPS's and single LPS's (Amdahl's Law [11]). Table 3 gives a summary of the causes that prevent the processing of multiple pels.

In those cases where the context is changing and where an MPS is followed by another MPS, it may be possible to process two MPS's in a single clock cycle, providing the Q_e values are available in time for both contexts. This may be accomplished through employing a double-ported storage for the Q_e values. A better alternative may be pre-fetching of the Q_e values. For instance, suppose an encoder is processing a run of zeros and the present context[6] is also zero $\overset{00000}{00}$. There is only one possible context following an all-zero context, namely $\overset{00001}{00}$, that might allow two MPS's to be processed in the same clock cycle. Thus, when processing an all-zero run, the encoder begins to fetch the Q_e value of the contexts $\overset{00001}{00}$, $\overset{00011}{00}$, $\overset{00111}{00}$ etc. always selecting the more likely successor context to pre-fetch. As successive Q_e values are fetched, they are stored in a cache and are made available to the interval and code point update circuitry as required. Note that the Q_e values cannot change in the interim and the cache is guaranteed to have the correct Q_e values.

The speed-up that may be achieved if this change is implemented is shown and compared to the default scheme in Table 4. A speed-up gain of up to thirty percent is consistent with our analysis of the default configuration, where advancing by a single MPS accounted for approximately 60% of the cycles, with the lion's share of these being convertible into pairs of MPS pels that can be processed in a single cycle.

[6]We adopt the notation $\overset{XXXXX}{XX}$ to describe the seven-pel context determined by the values of five pels in the previous line and two pels in the current line of the image being processed

5 Summary

This paper has presented new architectures for the data compression technique known as Arithmetic Coding based on loop unrolling and speculative execution of the inner loop of the algorithm that is a generalization of the Q-Coder [4] architecture. Multiple lookahead units are employed, with each unit attempting to process a different number of input bits (output bits in the case of the decoder) in one iteration. For the set of CCITT documents, the coder achieves a speed-up of approximately eight compared to the IBM Q-coder when four lookahead units are employed in parallel. A structure for fast Input/Output processing based on Run-Length encoding of the data stream to accompany the new, faster IUC is also suggested.

Additional speed-up using single-cycle processing of multiple pels with different contexts was also investigated.

References

[1] N. Abramson. *Information Theory and Coding*, pages 61–62. McGraw-Hill, New York, NY, 1963. Refers to unpublished work of Elias.

[2] J. J. Rissanen and G. G. Langdon. Universal Modelling and Coding. *IEEE Transactions on Information Theory*, 27(12):12–23, 1981.

[3] G. G. Langdon and J. J. Rissanen. A Simple General Binary Source Code. *IEEE Transactions on Information Theory*, 28(5):800–803, 1982.

[4] J. L. Mitchell and W. B. Pennebaker. Optimal hardware and software arithmetic coding procedures for the Q-Coder. *IBM Journal of Research and Development*, 32(6):727–736, November 1988.

[5] W. B. Pennebaker et al. An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder. *IBM Journal of Research and Development*, 32(6):717–726, November 1988.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture A Quntitative Approach*, chapter 6, page 315. Morgan Kaufmann Publishers, Inc., 1990.

[7] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*, chapter 13.7, pages 231–232. Van Nostrand Reinhold, 1993.

[8] R. B. Arps. Binary image compression. In William K. Pratt, editor, *Image Transmission Techniques*, pages 222–276. Academic Press Inc, New York, 1979.

[9] R. Hunter and A. Robinson. International Digital Facsimile Coding Standards. *Proceedings of the IEEE*, 68, July 1980.

[10] R. B. Arps, T. K. Truong, D. J. Lu, R. C. Pasco, and T. D. Friedman. A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images. *IBM Journal of Research and Development*, 32(6):775–794, November 1988.

[11] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS 1967 Spring Joint Computer Conference*, pages 483–485, Atlantic City, New Jersey, April 1967.

```
if MPS is encoded
  $C \leftarrow C + Q_e$    $A \leftarrow A - Q_e$
  while $A \leq 0.75$ do
      $A \leftarrow A \times 2$    $C \leftarrow C \times 2$
      decrement $Q_e$
  end do
else    LPS encoded
      $A \leftarrow Q_e$
  while $A \leq 0.75$ do
      $A \leftarrow A \times 2$    $C \leftarrow C \times 2$
      increment $Q_e$
  end do
end if
```

Figure 1: Pseudo-code Description of Arithmetic Encoding

```
if $C \geq Q_e$    MPS decoded
  $C \leftarrow C - Q_e$    $A \leftarrow A - Q_e$
  while $A \leq 0.75$ do
      $A \leftarrow A \times 2$    $C \leftarrow C \times 2$
      decrement $Q_e$
  end do
else    LPS decoded
      $A \leftarrow Q_e$
  while $A \leq 0.75$ do
      $A \leftarrow A \times 2$    $C \leftarrow C \times 2$
      increment $Q_e$
  end do
end if
```

Figure 2: Pseudo-code Description of Arithmetic Decoding

```
do    par
        $C_1 \leftarrow C + Q_e$    $A_1 \leftarrow A - Q_e$    $\mathrm{FLAG}_{1\mathrm{MPS}} \leftarrow$ 1 symb. $\overset{?}{=}$ MPS
        $C_2 \leftarrow C + 2 \times Q_e$    $A_2 \leftarrow A - 2 \times Q_e$    $\mathrm{FLAG}_{2\mathrm{MPS}} \leftarrow$ 2 symb. $\overset{?}{=}$ MPS
        $C_4 \leftarrow C + 4 \times Q_e$    $A_4 \leftarrow A - 4 \times Q_e$    $\mathrm{FLAG}_{4\mathrm{MPS}} \leftarrow$ 4 symb. $\overset{?}{=}$ MPS
                                  $\vdots$
         $C_{2^{k_{\max}}} \leftarrow C + 2^{k_{\max}} \times Q_e \ldots \mathrm{FLAG}_{2^{k_{\max}}\mathrm{MPS}} \leftarrow$ $2^{k_{\max}}$ symb. $\overset{?}{=}$ MPS
end   par
if        ( $\mathrm{FLAG}_{2^{k_{\max}}\mathrm{MPS}} \wedge A_{2^{k_{\max}}} \geq 0.75$ )    { $C \leftarrow C_{2^{k_{\max}}}$    $A \leftarrow A_{2^{k_{\max}}}$ }
else if   ( $\mathrm{FLAG}_{4\mathrm{MPS}} \wedge A_4 \geq 0.75$ )                    { $C \leftarrow C_4$    $A \leftarrow A_4$ }
else if   ( $\mathrm{FLAG}_{2\mathrm{MPS}} \wedge A_2 \geq 0.75$ )                    { $C \leftarrow C_2$    $A \leftarrow A_2$ }
else if   ( $\mathrm{FLAG}_{1\mathrm{MPS}}$ )                                       { $C \leftarrow C_1$    $A \leftarrow A_1$ }
else                                                                  { $C \leftarrow C$    $A \leftarrow Q_e$ }
end if
do    par
        if (( $\mathrm{FLAG}_{2^{k_{\max}}\mathrm{MPS}} \vee \ldots \vee \mathrm{FLAG}_{2\mathrm{MPS}} \vee \mathrm{FLAG}_{1\mathrm{MPS}}) \wedge A < 0.75$ )    decr $Q_e$
        else if ( $A < 0.75$ )                                                       incr $Q_e$
        end if
        while ( $A \leq 0.75$ ) do { $A \leftarrow A \times 2$    $C \leftarrow C \times 2$ }
        end do
end   par
```

Figure 3: Pseudo-code Description of an Arithmetic Encoder with Lookahead by Multiple Powers of 2.FLAG $\leftarrow a \overset{?}{=} b$ means that FLAG will be set to TRUE if a equals b.

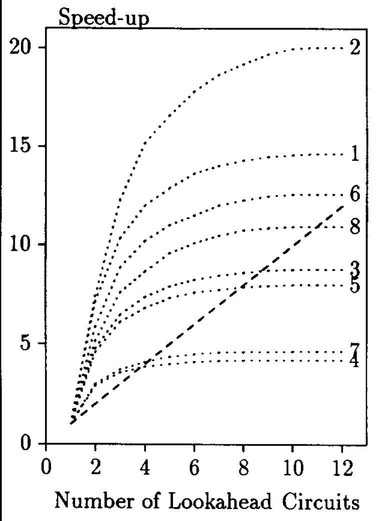

Figure 4: Maximum Speed-up versus Number of Lookahead Units for all eight CCITT FAX documents The dashed line corresponds to Speed-up linear in number of Lookahead Units.

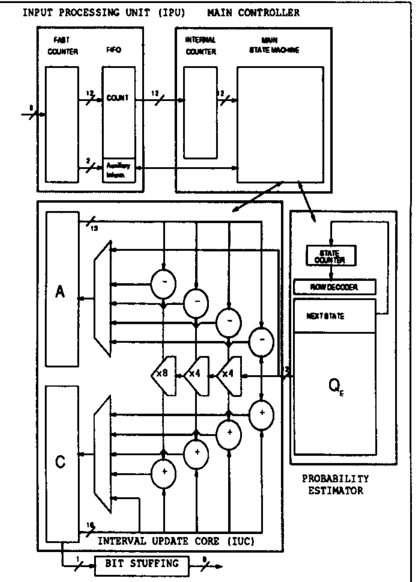

Figure 5: A block diagram of an arithmetic encoder with four lookahead units (configuration $0X95$).

Advance by	No. of Iterations	% of total	No. of Bits Encoded	% of total
1 LPS	34640	12.35	34640	0.84
1 MPS	182233	64.98	182233	4.44
2 MPS	22174	7.91	44348	1.08
4 MPS	12946	4.62	51784	1.26
8 MPS	7307	2.61	58456	1.42
16 MPS	5574	1.99	89184	2.17
32 MPS	3835	1.37	122720	2.99
64 MPS	3609	1.29	230976	5.63
128 MPS	3030	1.08	387840	9.45
256 MPS	1698	0.61	434688	10.59
512 MPS	2008	0.72	1028096	25.04
1024 MPS	1407	0.50	1440768	35.09
Total	280461	100%	4105728	100%

Table 1: Applying Lookahead to the CCITT1 image encoded with a seven-pel predictor employing context.

Speed-up for Coder Configuration	CCITT Document								Average
	1	2	3	4	5	6	7	8	
0X95	**12.00**	**15.17**	7.37	3.76	6.79	**10.21**	3.98	8.53	8.48
0x4b	11.61	14.45	**7.39**	**3.87**	**6.82**	10.10	4.02	8.31	8.32
0x2b	10.33	12.81	7.10	3.81	6.63	9.39	**4.09**	8.59	7.84
0x55	11.50	14.78	7.35	3.76	6.79	10.17	4.04	**8.68**	8.38
Compression Ratio	34.0	57.4	21.7	9.1	19.0	36.5	8.8	32.8	

Table 2: Speed-up for all eight CCITT documents that is obtained using four different encoder configurations and a seven pel predictor with context. Speed-up values corresponding to the optimum encoder configuration for a particular CCITT document are highlighted. The compression ratio is independent of the encoder configuration.

Advance by	No. of bits processed
Single LPS	34640
MPS followed by renormalization	25085
MPS followed by LPS	2634
MPS in a changing context followed by an LPS	23580
MPS in a changing context followed by an MPS	130934

Table 3: Frequency breakdown of the case of advancing by a single pel for the CCITT1 image encoded with a seven-pel predictor employing context.

	Average Speed-up			
Configuration	$0X95+$	$0X95$	$0XFFF+$	$0XFFF$
CCITT (1-8)	10.34	8.48	13.46	10.49

Table 4: Speed-up averaged over eight CCITT documents for the fastest four-adder configuration, $0X95$, and twelve-adder configuration $0XFFF$. The "+" indicates that pairs of MPS's with different context are processed in parallel, where possible.