Synthesis of Low-Overhead Interfaces for Power-Efficient Communication over Wide Buses

L. Benini * A. Macii ‡ E. Macii ‡ M. Poncino ‡ R. Scarsi ‡

* Università di Bologna
Bologna, ITALY 40136

‡ Politecnico di Torino
Torino, ITALY 10129

Abstract

In this paper we present algorithms for the synthesis of encoding and decoding interface logic that minimizes the average number of transitions on heavily-loaded global bus lines. The approach automatically constructs low-transition activity codes and hardware implementation of encoders and decoders, given information on word-level statistics. We present an accurate method that is applicable to low-width buses, as well as approximate methods that scale well with bus width. Furthermore, we introduce an adaptive architecture that automatically adjusts encoding to reduce transition activity on buses whose word-level statistics are not known a-priori. *Experimental results demonstrate that our approach well outperforms low-power encoding schemes presented in the past.*

1 Introduction

Off-chip and on-chip global bus lines in VLSI circuits are generally loaded with large capacitances, up to three orders of magnitude larger than the average on-chip interconnect capacitance. When using standard CMOS signalling, the power dissipated by bus drivers is proportional to the product of average number of signal transitions and line capacitance. Hence, one way of reducing power dissipation on bus drivers is to *encode* the data sent on the bus with encoding schemes that reduce the average number of signal transitions.

Based on this observation, several researchers have proposed encoding schemes that reduce the average number of signal transitions. Some codes [1, 2, 3] exploit *spatial redundancy*, i.e., they increase the number of bus lines, while others exploit *temporal redundancy*, i.e., they increase the number of bits transmitted in successive bus cycles [4]. A few codes do not rely on spatial nor temporal redundancy [5, 6].

Theoretical issues in bus encoding for low transition activity are investigated in [7]. In this work, the authors introduce an information-theoretic framework for studying low-transition encoding, and prove a useful lower bound on minimum achievable average transition activity. Several redundant and irredundant codes are then analyzed and compared to the theoretical bounds to assess their quality. In [8], the same authors introduce a generic encoder-decoder architecture that can be specialized to obtain an entire class of low-transition coding schemes. A few personalizations of the generic architecture are described, and the reductions in transition activity are compared.

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 99, New Orleans, Louisiana
(c) 1999 ACM 1-58113-109-7/99/06..$5.00

In [8], no systematic method is provided for obtaining optimum codes from the generic architecture. Also, the hardware complexity and cost of encoders and decoders is not studied in detail. Finally, all presented encoding schemes assume some knowledge of the statistical properties of the streams that must be encoded. These issues are addressed in this work.

We propose a generic encoder-decoder architecture and we describe an algorithm for customizing it to obtain implementations that minimize bus transition activity, given a detailed statistical characterization of the target stream. We also introduce two heuristic approximations of the basic algorithm that produce low-transition codes and low-complexity encoders and decoders. These codes are tailored for fast and wide buses, where encoders and decoders are subject to tight performance and hardware cost constraints, and for streams whose statistical properties are not known exactly. Finally, we describe a *general-purpose*, efficient encoder-decoder architecture that can be used to reduce bus transition activity for generic data streams with completely unknown statistical properties. This architecture is capable of *on-line adaptation* of the encoding scheme to the data stream currently being transmitted over the bus.

One desirable feature of our approach is that not only the abstract specification, but also the circuit implementation of encoder and decoder is automatically synthesized. Moreover, we offer the possibility of trading off bus switching activity reduction for encoder-decoder complexity. Designers can exploit our interface synthesis approach to rapidly explore the power-saving opportunities enabled by low-transition encodings.

2 Basic Concepts

Consider a data source that generates symbols over alphabet $\mathcal{X}$. We assume that the cardinality of the alphabet is $|\mathcal{X}| = 2^W$. Each symbol $x \in \mathcal{X}$ is represented as a W-bit word $x = [b_1, b_2, \ldots, b_W]$. Notice that $\mathcal{X}$ is the Boolean space $\mathcal{B}^W$ such that every W-bit configuration has non-null probability of being generated by the data source. Symbols x must be transmitted over time on a communication bus of width W. We assume here a discrete-time setting, and we use the notation $x(n)$ to indicate the word transmitted at time period n.

The bus width W and the communication throughput $T = 1$ (one word transmitted in each time period) will be taken as tight constraints. Such constraints rule out the possibility of considering space and/or time redundant codes, as well as variable-length codes. The motivation for this assumption is that spatial redundancy is hardly tolerated in global bus organization because it changes pinout and interface specification. Temporal redundancy and variable-length coding do not change bus width, but introduce variable latency in communication, which may be unacceptable.

2.1 General Codec Architecture

We propose the general encoder-decoder (*codec*, for brevity) architecture shown in Figure 1. The encoder takes as input the stream of W-bit input words $x(n), n = 0, 1, \ldots$. It consists of three blocks: (i) A register that stores $x(n-1)$ when the input is $x(n)$; (ii) A combinational *encoding function* $E : \mathcal{B}^W \times \mathcal{B}^W \rightarrow \mathcal{B}$, that generates the encoded word $y(n)$ as a function of $x(n)$, $x(n-1)$; (iii) A *decorrelator* [4, 8], that simply translates 1-valued bits of $y(n)$ into transitions on the corresponding bus lines (0-valued bits correspond to stationary values on the bus lines).

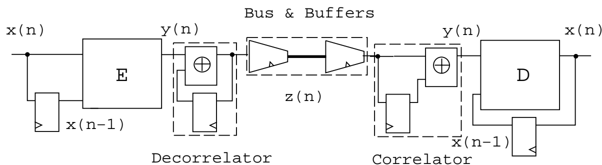

Figure 1: General Codec Architecture.

The decoder takes as input the word $z(n)$ transmitted over the bus and computes the original input word $x(n)$. It consists of three blocks: (i) A *correlator* which computes the inverse function of the decorrelator, and reconstructs $y(n)$; (ii) A combinational *decoding function* $D : \mathcal{B}^W \times \mathcal{B}^W \rightarrow \mathcal{B}$, that reconstructs input word $x(n)$ from $y(n)$ and $x(n-1)$; (iii) A register that stores $x(n-1)$ when the output of the decoder is $x(n)$.

Before describing the salient features of functions E and D, we briefly review the operation of decorrelator and correlator. These two blocks have transfer functions $out(n) = in(n) \oplus out(n-1)$ and $out(n) = in(n) \oplus in(n-1)$, respectively (we use symbol "$\oplus$" to denote the *exclusive-or* operation). It is assumed that when $n = 0$, $in(n-1) = out(n-1) = 0$. The transfer functions of the two blocks are one the inverse of the other. The main advantage of using correlator and decorrelator is that they transform the problem of minimizing the number transitions on the bus into the problem of minimizing the number of ones on the decorrelator's input [4].

Encoding function E should minimize the average number of ones at its output while guaranteeing that the encoded value $y(n)$ can still be uniquely decoded by D. The sole purpose of D is to compute the correct value of $x(n)$. Note that both E and D exploit past values of the input stream for decoding and encoding. Clearly, the architecture of Figure 1 is a generic scheme that can be customized by defining functions D and E. It is possible to further generalize the architecture by considering more than one past input words for encoding and decoding. Unfortunately, hardware complexity of D and E rapidly increases with the number of inputs, thus we will not consider more complex schemes.

3 Low-Transition Encoding Algorithm

The algorithm presented in this section moves from the assumption that a detailed statistical characterization of the data source is available. More specifically, we assume the availability of the complete probability distribution of all pairs of consecutive values in the input stream x. In symbols, the probability:

$$P_{x_i,x_j} = Prob(x(n) = x_i \wedge x(n-1) = x_j)$$

is known $\forall x_i, x_j \in \mathcal{X}$. We call this distribution *joint probability distribution* (JPD). Furthermore, we assume that JPD is stationary, i.e., $Prob(x(n) = x_i \wedge x(n-1) = x_j)$ does not depend on the time index n.

The encoding algorithm builds the specification (i.e., the truth table) of function E in an enumerative fashion. Function D is obtained as a by-product. The starting point of the algorithm is a table (called *code table*) with three columns, labeled $x(n)$, $x(n-1)$ (current and past input words) and $y(n)$ (current encoded word), respectively. The table has 2^{2W} rows, one for each pair of input words. Initially, the third column is empty (i.e., no encoded word is specified), while the first and second column contain all (x_i, x_j) pairs, ordered for decreasing P_{x_i,x_j}. The value of the encoded word y corresponding to each pair (x_i, x_j) is computed starting from the first row of the code table.

The pseudo-code of the algorithm is shown in Figure 2. Its only input parameter is the initial code table *CodeTab* (a matrix with 2^{2W} rows and 3 columns). First, *Conflicts* is initialized. This array has one element for each row of *CodeTab* and it will be used to store forbidden values of the encoded word y. Initially, any value can be assigned to any row. The external loop scans the table from the top. For each row, the encoded word y (i.e., the third column of the table) is assigned by calling function *MinOneCode*. This function assigns to y the W-bit word containing the minimum number of ones that does not belong to the set of forbidden codes for the row under consideration. As the algorithm scans the table from top to bottom and assigns values to the third column, the *Conflicts* array is updated. The key point of the algorithm is the update rule for array *Conflicts*, that will be discussed next. The algorithm terminates when the code for the last row has been assigned and returns the complete code table.

```
procedure BuildTable(CodeTab) {
  for (row = 0; row < 2^{2W}; row++) Conflicts[row] = ∅;
  for (row = 0; row < 2^{2W}; row++) {
    CodeTab[row][2] = MinOneCode(Conflicts[row]);
    foreach (r s.t. CodeTab[r][1] == CodeTab[row][1]) {
      Conflicts[r] = Conflicts[r] ∪ CodeTab[row][2];
    }
  }
  return (CodeTab);
}
```

Figure 2: Code Construction Algorithm.

The need for storing and updating forbidden codes stems from a fundamental *decodability* constraint. The encoding function cannot be a 1-to-1 mapping, because its domain is $\mathcal{B}^{2W}$ while its co-domain is $\mathcal{B}^W$. Thus, many input pairs $(x(n), x(n-1))$ are necessarily associated to a single output value $y(n)$. However, this association cannot be arbitrary, because we need to decode $y(n)$. Decodability is ensured if any pair $(x(n-1), y(n))$ uniquely identifies a single value $x(n)$. This constraint must be satisfied for each row of the table. Hence, whenever we assign a code y_k to the table row with code x_i and x_j in the first two columns, we must guarantee that the same code is not used for any other row with the same value of x_j.

The complete code table is the truth table for function E. The first two columns are input minterms, the third column is the output value. The coding function minimizes the probability of generating ones on its outputs, within the constraints imposed by unique decodability. Function D is obtained by taking columns $y(n)$ and $x(n-1)$ as inputs, and column $x(n)$ as output. The complexity of the algorithm is exponential in W, because the number of rows in the code table is 2^{2W}. Clearly, computation of the complete table becomes infeasible for large bus widths. Besides the obvious computational bottleneck, there are a few more limitations. First, the knowledge of the JPD may be incomplete or approximate. For instance, obtaining a reasonably accurate estimate of P_{x_i,x_j} for every pair of input words

may be infeasible for large streams. Second, the implementation of function E and D in hardware may be unacceptably large, slow or power-consuming. In summary, the encoding algorithm may become impractical for wide global buses in current VLSI circuits. Hence, we need to resort to approximate algorithms that scale well with bus width.

4 Approximate Algorithms

4.1 Clustered Encoding

The most intuitive approximation to the exact algorithm of the previous section consists in partitioning the set of bus lines in smaller clusters and apply the exact algorithm to each cluster. We call this solution *clustered* encoding. In this scheme, we privilege *temporal* correlation with respect to *spatial* correlation, since we still base the encoding/decoding process on the statistics of all possible input pairs, yet smaller than the total bus width.

This solution exhibits an evident trade-off between accuracy and complexity; the smaller the clusters, the smaller the reduction in the number of transitions, because the spatial correlation between bits is partially lost. On the other hand, larger clusters imply larger encoders and decoders and longer code construction times, as in the case of the exact algorithm.

An important issue here is the criterion used to grow the clusters. Since breaking a word into clusters of bits decreases the spatial correlation between bits, it is reasonable trying to keep in the same cluster bits with high mutual spatial correlation. Our clustering algorithm we have used is similar to the one proposed in [6], and it is not reported here for space reasons.

The architecture generated by clustered encoding consists of a set of encoder/decoder pairs, one for each cluster. The encoder/decoder logic is synthesized from a two-level description that represents the code table of each cluster.

4.2 Discretized Encoding

An alternative approximate solution is to consider only the M most probable pairs of consecutive words in the code, where $M << 2^W$. Let us denote such set as $\mathcal{S}$. We call this approximate solution *discretized* encoding. The optimality loss in this solution is due to the fact that we consider all pairs outside the first most probable M as equiprobable. In this method spatial correlation is privileged, since the statistics are computed on full words; conversely, we neglect some temporal correlation because the encoding/decoding process is driven only by a small set of code-words.

The implementation of the discretized scheme can be realized according to the conceptual architecture of Figure 3, where the encoder E is shown. The block $F(x(n), x(n-1))$ implements the encoding function for set $\mathcal{S}$. The rest of the words in the alphabet is left unchanged. This is realized by a generic *background* function (denoted with $B(x(n), x(n-1))$).

The reason for the existence of the background function is that the architecture of Figure 3 represents one realization of the block denoted with E in the general architecture of Figure 1, whose output $y(n)$ feeds the decorrelator. If the upper path of Figure 3 must realize the identity function, the block B must cancel the effect of the decorrelator that follows the block. Therefore, block B actually implements a correlator, that is, $B(x(n), x(n-1)) = x(n) \oplus x(n-1)$.

The block *Sel* determines which of the two functions, F or B has to be applied to the current pair of words. In other terms, *Sel* represents the characteristic function of the pairs that belong to set $\mathcal{S}$.

In the clustered architecture, splitting the bus width in smaller blocks implies smaller encoding and decoding logic. Conversely, in the discretized solution, encoder and decoder must still be $2W$-input, W-output functions. The simplification in the hardware implementation of encoding and decoding functions comes from the fact that the specification has a large don't care set, namely the set of all word pairs that are not encoded.

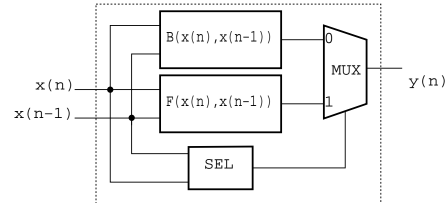

Figure 3: Architecture for Discretized Encoding.

The construction of the encoding function F, unlike the clustered approximation, requires the modification of the basic algorithm of Section 3. Due to space limitations, we will only outline the differences: In discretized encoding, constraints imposed by assigning a new code to a pair may create a conflict with another pair which is not expected to be encoded, because it does not belong to $\mathcal{S}$.

The modified algorithm proceeds as in the exact case for what concerns the assignment of a code to a given pair. After the lists of conflicts have been updated, however, the newly assigned code always affects one of the *background pairs*, i.e., those outside $\mathcal{S}$. Consider the table row r identified by the pair $P_1 = (x_i, x_j)$, and assume that it has been assigned code y_k. The conflict mechanism guarantees that this assignment is uniquely decodable with respect to the pairs in $\mathcal{S}$. However, such assignment affects one of the background pairs, and precisely the one that has the last two columns equal to those of r, i.e., (x_j, y_k). This pair is $P_2 = ((x_j \oplus y_k), x_j)$, since it implements the background function.

Because of this conflict, we are forced to change the code assigned to P_2, otherwise P_1 and P_2 will not be distinguishable by the decoder. Changing the code for P_2 (a background pair) means bringing P_2 into $\mathcal{S}$, because it will not be encoded according to the background function anymore.

When bringing P_2 into $\mathcal{S}$, we assign it a new code, say y_l. Obviously, code y_l must neither conflict with any other previously assigned pair, nor with other background pairs. The only way of satisfying these two requirements is to assign y_l in such a way that $y_l \oplus x_j = x_i$, that is, $y_l = x_i \oplus x_j$. The resulting line of the code table for P_2 would then be: $((x_j \oplus y_k), x_j, y_l)$.

The rationale is that the entry for P_2 is now potentially conflicting with the background entry $((x_j \oplus y_l), x_j, y_l)$, because they share the (x_j, y_l) in the last two columns. After some computations, this conflicting entry can be simplified to (x_i, x_j, y_l), which cannot belong to the background pairs, since (x_i, x_j) is exactly P_1. In some cases not described here, conflict resolution with background pairs requires complex operations.

5 Adaptive Encoding

The solutions described in Sections 3 and 4 require that word-pair statistics are known before synthesizing the encoder and decoder logic. This assumption may not hold in some application domains. In this section, we present an encoding scheme that does not require any *a-priori* knowledge of the input statistics, and is capable of on-line adaptation of the the encoding to stream statistics.

The proposed solution is approximate in the sense that it realizes an adaptive scheme that operates *bit-wise* rather than word-wise, and therefore ignores the spatial correlation between bits of the same code-word. Such approximate solution is needed to allow a low-cost implementation of the encoding and decoding logic in terms of area, delay and power.
The basic idea behind the adaptive method is to apply the algorithm of Section 3 on the basis of approximate statistical information, that are collected by observation of the bit stream over a window of fixed size S. There is a trade-off between the window size S and the accuracy in the prediction of the bit-wise JPD. The larger S, the more we will approach the exact bit-wise joint probabilities. At the same time, increasing the window size has a direct impact on the complexity of the hardware implementation. Experimental data have shown that a window size of $S = 64$ offers a good compromise between complexity and accuracy.
The application of the exact algorithm of Section 3 on a single bit requires the knowledge of the four joint probabilities $P_{0,0}, P_{0,1}, P_{1,0}$, and $P_{1,1}$, whose ranking determines the optimal 1-bit code. In order to deal with integer quantities, that simplifies the hardware, we will use the occurrence frequencies N_{00}, N_{01}, N_{10} and N_{11} instead of the joint probabilities. Clearly, since the window size is fixed, the joint probabilities can always be computed by dividing the occurrence frequencies by $S-1$, e.g. $P_{0,0} = N_{00}/(S-1)$.
If we closely analyze the frequency distribution, we can observe that not all the four occurrences are actually required. First, the sum of the four occurrence frequencies is known; since there are only $S-1$ pairs over a window of size S, $N_{00} + N_{01} + N_{10} + N_{11} = S - 1$. For practical window sizes, we can then assume that $S-1 \approx S$. Second, the number of 0-to-1 and 1-to-0 transitions must be balanced over the observation window, that is $N_{01} = N_{10}$. The equality should be interpreted loosely; in fact, N_{01} and N_{10} actually differ by 1 at most.
In conclusion, it is sufficient to consider only two joint probabilities to fully characterize the JPD, since their knowledge implies the other two. Without loss of generality, we choose N_{00} and N_{11}.

5.1 Encoder Architecture

The basic scheme of the architecture for the 1-bit encoder is shown in Figure 4 (for $S = 64$). The input $x(n)$, and its previous value $x(n-1)$ feed some glue logic that triggers the two counters that store the number of occurrences of the two consecutive pairs (N_{00}, N_{11}). The counters count over a window size, and are reset after each S cycles. This is realized by a *window counter* (*WinCnt*) that properly resets the two counters. The window counter is shared across all the bits in the bus.

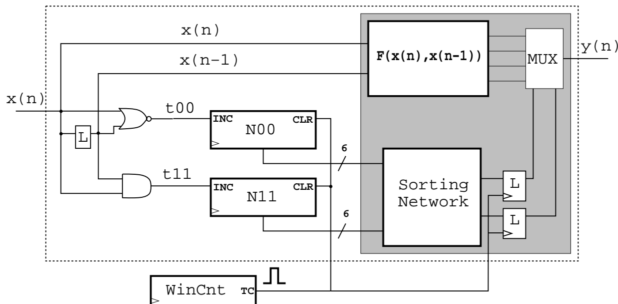

Figure 4: Conceptual Architecture for Adaptive Encoder.

The shaded block on the right computes the encoding $y(n)$ based on the knowledge of $x(n), x(n-1)$, and the values of N_{00} and N_{11}. Since there are only four possible combinations of $(x(n), x(n-1))$ we can explicitly enumerate all the possible orderings of these four configurations, that corresponds to consider $4! = 24$ cases. These orderings can be further reduced by observing that $N_{01} = N_{10}$. We actually need to consider only $3! = 6$ cases, corresponding to all the possible orderings of three quantities: N_{00}, N_{11}, and one of N_{01} and N_{10}. For ease of notation, we will denote both N_{01} and N_{10} with the symbol N_T, to denote the fact that they are indistinguishable.
The enumeration of the six orderings results in only four different encoding functions $F(x(n), x(n-1))$:

a) $y(n) = x(n)$ b) $y(n) = x(n)'$
c) $y(n) = x(n) \oplus x(n-1)$ d) $y(n) = x(n) \overline{\oplus} x(n-1)$

The block inside the shaded area denoted with *Sorting Network* serves the purpose of selecting the proper encoding function $F(x(n), x(n-1))$ according to the JPD of the current window. Such decision is taken as follows:

$$\begin{cases} y(n) = x(n) & \text{when } N_{00} > N_T > N_{11} \\ y(n) = x(n)' & \text{when } N_{11} > N_T > N_{00} \\ y(n) = x(n) \oplus x(n-1) & \text{when } N_T < \{N_{11}, N_{00}\} \\ y(n) = x(n) \overline{\oplus} x(n-1) & \text{when } N_T > \{N_{11}, N_{00}\} \end{cases} \quad (1)$$

This selection mechanism of the encoding functions has an intuitive interpretation; for example, in the first case, since the most probable pair of symbols is 00, it is reasonable to leave the bits unchanged, since a 0 in the stream will result in no transition after the decorrelator in the scheme of Figure 1. Similarly, when N_T (i.e., a transition) is the most probable symbol, the transitions are first eliminated by XOR-ing two consecutive bits (in other terms, by using a correlator). This yields a sequence of 1's, that has to be complemented before being fed to the decorrelator. The latter example clearly shows how the general scheme proposed includes the general framework structure of [8] as a particular case.
The decision rules described in Equation 1 can be graphically represented as in Figure 5, where the four regions denoted with a), b), c) and d) correspond to the four different encoding functions. The regions are delimited by the square of size S in the plane (N_{00}, N_{11}), and by three lines, that identify the possible relations between (N_{00}, N_{11}), and N_T.
The boundary lines are obtained by expressing all the inequalities in terms of (N_{00} and N_{11}), replacing thus N_T with $(S - (N_{00} + N_{11}))/2$. The line equations are derived as follows:

$$N_T > N_{00} \;\rightarrow\; \frac{(S - (N_{00} + N_{11}))}{2} > N_{00} \;\rightarrow\; N_{11} + 3N_{00} - S < 0$$

$$N_T > N_{11} \;\rightarrow\; \frac{(S - (N_{00} + N_{11}))}{2} > N_{11} \;\rightarrow\; 3N_{11} + N_{00} - S < 0$$

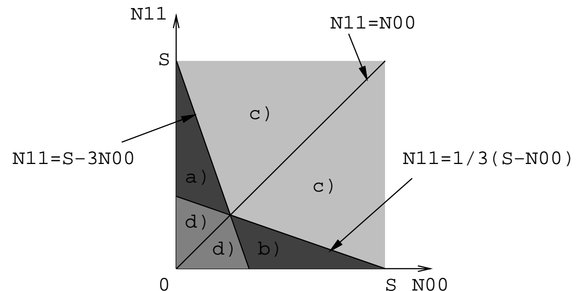

Figure 5: Space of the Sorting Network.

5.2 Implementation

Concerning the hardware implementation of the sorting network, we face two possibilities. The most intuitive choice is to generate a two-level cover of the sorting network with a software program, by exploring all the possible orderings and associating an output value to each of them. This solution may result in excessively large circuits.

Another option is to realize the sorting network by directly implementing the decision regions of Figure 5. We observe that counters N_{00} and N_{11} and the sorting network can be merged together. The inequalities of Section 5.1 can be rewritten as:

$$N_{11} + 3N_{00} < S \qquad 3N_{11} + N_{00} < S \tag{2}$$

Instead of computing N_{00} and N_{11}, and derive the two l.h.s. of the above inequalities from them arithmetically, we can directly store in a register the quantities needed to take the decision, i.e., $N_{11} + 3N_{00}$ and $3N_{11} + N_{00}$. The magnitude of the l.h.s. of the inequalities of Equation 2 are bounded by $4S$, and can then be stored in a register with $(\log_2 S + 2) = 8$ bits.

Figure 6 shows the optimized implementation that merges the two counters of Figure 4 and the sorting network.

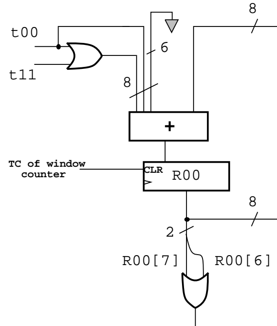

Figure 6: Efficient Encoder Implementation.

Each counter is replaced by a cheaper register: Register R_{00} is used to store $N_{11} + 3N_{00}$, while register R_{11} stores $3N_{11} + N_{00}$. Each register computes $R = R + C$, where C is determined by the values that are present on the signals t_{00} and t_{11} that detect the 0-to-0 and 1-to-1 transitions. For example, for register R_{00}: $C = 0$, if $(t_{00}, t_{11}) = 00$, while $C = 1$, if $(t_{00}, t_{11}) = 01$. Finally, $C = 3$, if $(t_{00}, t_{11}) = 10$. The operations for R_{11} are similar, and are obtained by exchanging the last two conditions. At the end of the window, the condition of Equation 2 can be obtained by looking at the second most significant bit of the output (bit 7). If this is 1, we have exceeded the value S that can be stored in 6 bits. The four combinations of $(R_{00}[7], R_{11}[7])$ can be used to directly drive the output multiplexor, to select the proper encoding function, as follows:

$R_{11}[7]$	$R_{11}[7]$	*Condition*	*Function*
0	0	$N_T < \{N_{00}, N_{11}\}$	$y(n) = x(n) \oplus x(n-1)$
0	1	$N_{11} > N_T > N_{00}$	$y(n) = x(n)'$
1	0	$N_{00} > N_T > N_{11}$	$y(n) = x(n)$
1	1	$N_T > \{N_{11}, N_{00}\}$	$y(n) = x(n) \overline{\oplus} x(n-1)$

Concerning the performance of the encoder, the critical path runs through the block F and the multiplexor, in the upper part of Figure 4. Since the encoding functions F consist of at most one gate, we can conclude that in the worst case we have two or three equivalent gates on the critical path, depending on the multiplexor implementation.

5.3 Decoder Architecture

The architecture of the decoder is very similar to that of the encoder, and is not shown here for space reasons. It computes the same statistics as the encoder, that is N_{00} and N_{11}, that are derived by observing pairs of consecutive values of the decoded output $(x(n), x(n-1))$.

There are two main difference with respect to the encoder. First, according to the architectural scheme of Figure 1, the "true" decoder D must take as inputs $y(n)$ and $x(n-1)$. Second, the decoding functions (block $F^{-1}(y(n), x(n-1))$) must compute the *inverse* of the functions F in the encoder.

In this case, all the encoding functions of Equation 1 are exactly the same as their inverse. For example, if $y(n) = x(n) \oplus x(n-1)$ is selected in the encoder, $x(n) = y(n) \oplus x(n-1)$ is selected in the decoder. Notice that the same hardware optimization employed for the encoder can be used in the decoder as well.

6 Experimental Results

We have applied the proposed encoding scheme, in the exact, approximate, and adaptive variants, to a set of ten real-life streams with various statistical profiles. The streams we have considered are the following:

- *DCT*, *FFT*: Traces obtained from a code profiler;
- *Sound*: A `.WAV` file;
- *M31*: Image in `.PPM` format;
- *SCREEN*: Image in raw format captured from screen;
- *HTML*: HTML page containing some images;
- *GOPHER*, *GZIP*, *GCC*, *BISON*: Executable files.

Table 1 reports the comparison between the proposed schemes and the method of [8], in terms of reduction in the number of transitions with respect to the original streams. To enable a fair comparison with the method of [8], we implemented the algorithm called in their work ***dbm-pbm***, which provided the best results for most of the streams used.

The results for the discretized method (column *Discretized*) consist of three sets of data (columns *M=20, M=50, M=100*), corresponding to different numbers of words considered. For the clustered method (column *Clustered*), two sets of results are shown, one for 8 clusters of size 4, the second for 4 clusters of size 8.

The results support the claim that our exact algorithm outperforms the method of [8], because it is able to generalize the scheme realized by their general framework. The average savings of our method is 93.9%, as opposed to the 67.5% of [8].

Another important observation is that the clustered algorithms perform almost as well as the ***dbm-pbm***. This is an important result, because both our exact algorithm and the ***dbm-pbm*** have only theoretical interest, but are of limited practical use due to the size and complexity of the corresponding encoders and decoders. Conversely, as the circuit implementation results of Table 2 show, both the clustered and the discretized methods can be successfully implemented in hardware; therefore, the reported savings represent realistic power reductions.

The discretized encoding is, on average, less effective than the clustered one, suggesting that preserving temporal correlation is more important than preserving spatial correlation. Finally, as expected, the adaptive method is obviously the one that provides the smallest savings for the reasons already discussed in Section 5.

Stream	Exact	[8]	Discretized			Clustered		Adaptive	Bus Invert
			20	50	100	8	4		
DCT	86.8%	55.9%	37.2%	62.3%	67.4%	58.2%	75.4%	15.2%	0.01%
FFT	90.3%	56.9%	37.7%	52.7%	64.3%	55.9%	74.3%	5.5%	0.00%
Sound	99.0%	71.8%	0.8%	1.2%	1.4%	45.0%	49.5%	2.1%	8.27%
M31	97.4%	64.3%	1.0%	1.4%	1.7%	41.7%	42.3%	3.5%	0.78%
SCREEN	78.7%	61.4%	9.5%	25.7%	32.5%	39.0%	44.7%	-3.4%	0.01%
HTML	98.3%	76.3%	4.4%	9.2%	14.6%	41.1%	60.0%	9.3%	0.72%
GOPHER	98.2%	73.1%	2.5%	3.6%	5.0%	52.9%	61.9%	19.0%	3.92%
GZIP	98.2%	72.5%	1.5%	2.5%	3.3%	51.1%	59.8%	16.3%	3.45%
GCC	95.7%	73.8%	2.4%	5.1%	8.3%	51.2%	65.3%	15.6%	3.53%
BISON	96.5%	69.4%	2.8%	3.9%	5.1%	51.6%	62.5%	17.4%	4.17%
Average	93.9%	67.5%	9.9%	16.7%	20.4%	48.7%	59.5%	10.1%	2.48%

Table 1: Comparison of Transition Reduction.

Stream		Discretized									Clustered			Adaptive		
		20			50			100			8					
		A	P	D	A	P	D	A	P	D	A	P	D	A	P	D
DCT	E	5472	6.86	2.31	10734	14.37	2.96	23670	30.69	3.67	29988	41.80	2.89			
	D	5468	6.86	2.31	11034	15.24	3.06	23730	31.02	3.62	32904	50.57	2.46			
FFT	E	7380	9.33	3.45	15408	18.61	3.86	31842	40.37	4.40	37242	52.91	2.95			
	D	7378	9.28	3.44	14356	17.84	3.81	31678	40.04	4.32	37386	55.52	2.48			
Sound	E	11790	12.02	2.85	40158	40.38	4.84	99522	110.88	6.98	87246	138.21	3.13			
	D	11730	11.98	2.80	39734	39.87	4.56	99462	108.68	6.68	95040	154.03	3.03			
M31	E	13212	16.43	3.65	37494	48.63	4.55	112230	160.05	5.24	84780	129.77	2.77			
	D	13101	16.23	3.63	36573	47.23	4.67	111760	158.76	5.20	85122	140.47	2.95			
SCREEN	E	7308	9.37	2.57	19962	26.60	3.19	63522	82.53	5.00	78786	126.37	3.54			
	D	7298	9.12	2.56	20013	26.45	3.36	63500	82.48	5.00	74601	122.78	2.70			
HTML	E	53046	65.24	4.50	123336	176.64	5.26	237024	370.68	6.77	93416	148.23	3.06	187200	81.6	0.60
	D	53012	65.13	4.34	117263	174.26	5.16	236890	369.31	6.70	95346	156.70	3.19			
GOPHER	E	24696	26.42	3.95	75564	95.45	5.29	154044	225.57	6.22	90428	138.45	3.02			
	D	24540	26.60	3.95	75164	93.45	5.17	153930	223.32	6.22	91368	149.33	3.19			
GZIP	E	40248	45.12	4.33	98784	128.98	4.91	192006	283.50	6.84	86748	136.46	3.11			
	D	40124	45.18	4.31	99987	129.23	4.93	192120	285.10	6.86	85518	138.95	3.15			
GCC	E	23184	26.63	3.79	84546	115.36	4.84	202914	287.33	7.10	87120	136.42	2.85			
	D	23206	26.70	3.75	85674	113.71	4.76	202506	285.12	7.06	88834	139.21	2.93			
BISON	E	27036	29.15	4.39	51426	63.44	5.02	112356	151.53	6.13	86724	136.25	2.89			
	D	26578	29.05	4.35	52376	64.25	5.12	113134	151.12	6.15	84569	138.13	2.85			

Table 2: Comparison of Hardware Implementations for the Proposed Algorithms.

To fairly evaluate the effectiveness of the adaptive scheme, we have compared its performance to the Bus Invert code [4]. Although spatially redundant codes were excluded from our analysis because of our tight constraints on the bus width, we have included these experiments because the Bus Invert is the only low-power coding scheme that does not require any *a-priori* information about the stream that is transmitted, and can be reasonably compared to our general-purpose, adaptive scheme. The results are in favor of our algorithm. In fact, for most of the streams we have considered, the Bus Invert yields negligible savings, and the peak improvement is 8.27%. The proposed adaptive method is always better but in one case, where the peculiar structure of the stream (long patterns of 0's and 1's) results in a slight increase in the number of transitions.

Table 2 collects the implementation results of the encoders and decoders (E and D in the table) for the cases of approximate and adaptive schemes. The synthesis has been carried out using Synopsys DesignCompiler on a $0.35\mu m$ industrial library. The circuits have been optimized for speed, because we assumed that the latency of the encoders and decoders is the most stringent constraint. The results report the values of area (in μm^2), power (in mW), and delay (in ns) for both encoders and decoders of each stream. Power estimates are obtained from DesignPower with a clock frequency of $500MHz$.

For the discretized method, all three versions ($M = 20, 50, 100$) were successfully implemented. As expected, the complexity of the encoder and decoder logic tends to rapidly increase for larger values of M. Concerning the clustered scheme, only the version with 8 clusters of size 4 resulted in a cost-effective implementation. Notice that the values of area, power and delay for the clustered method are comparable to those of the discretized method with $M = 50$, that provides sensibly smaller savings.

The adaptive encoder (the decoder has a roughly identical implementation) is typically larger than those of the approximate solutions (except for the $M = 50$ discretized case, which is comparable). One desirable characteristics of this encoder is the negligible delay ($0.6ns$), which is at least four times less than the fastest among the other encoders/decoders.

7 Conclusions

We have presented novel algorithms for the automatic synthesis of bus interface logic that targets the minimization of the switching activity on global buses. We have benchmarked the capabilities of the proposed encoding/decoding techniques on a large set of data streams. In addition, we have investigated the trade-off between optimality of the encoding scheme and the complexity of the encoding/decoding circuitry.

References

[1] M. R. Stan, W. P. Burleson, "Bus-Invert Coding for Low-Power I/O,"

[2] L. Benini, G. De Micheli, E. Macii, D. Sciuto, C. Silvano, "Address Bus Encoding Techniques for System-Level Power Optimization," DATE-98, pp. 861-866, Feb. 1998.

[3] E. Musoll, T. Lang, J. Cortadella, "Working-Zone Encoding for Reducing the Energy in Microprocessor Address Buses," IEEE Trans. on VLSI Systems, Vol. 6, No. 4, pp. 568-572, Dec. 1998.

[4] M. R. Stan, W. P. Burleson, "Low-Power Encodings for Global Communication in CMOS VLSI," IEEE Trans. on VLSI Systems, Vol. 5 No. 4, pp. 444-455, Dec. 1997.

[5] H. Mehta, R. M. Owens, M. J. Irwin, "Some Issues in Gray Code Addressing," GLS-VLSI-96, pp. 178-180, Mar. 1996.

[6] L. Benini G. De Micheli, E. Macii, M. Poncino, S. Quer, "Reducing Power Consumption of Core-Based Systems By Address Bus Encoding", IEEE Trans. on VLSI Systems, Vol. 6, No. 4, pp. 554-562, Dec. 1998.

[7] S. Ramprasad, N. R. Shanbhag, I. N. Hajj, "Achievable Bounds on sIgnal Transition Activity," ICCAD-97, pp. 126-131, Nov. 1997.

[8] S. Ramprasad, N. R. Shanbhag, I. N. Hajj, "Signal Coding for Low Power: Fundamental Limits and Practical Realizations," ISCAS-98, pp. 1-4, Jun. 1998.