Architectures and Synthesis Algorithms for Power-Efficient Bus Interfaces

Luca Benini, Alberto Macii, Enrico Macii, *Member, IEEE*, Massimo Poncino, *Member, IEEE*, and Ricardo Scarsi

***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 at no cost in communication throughput (i.e., one word is transmitted at each cycle). The distinguishing feature of our approach is that it does not rely on designer's intuition, but it automatically constructs low-transition activity codes and hardware implementation of encoders and decoders, given information on word-level statistics. We propose 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 approaches well out-perform specialized low-power encoding schemes presented in the past.**

***Index Terms*—Bus encoding, digital systems, low-power design.**

I. Introduction

OFF-CHIP and on-chip global bus lines in very large scale integrated (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 signaling, 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 schemes that reduce the average number of transitions.

Based on this observation, several researchers have proposed encoding schemes that reduce the average number of signal transitions. Some codes [1]–[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/temporal redundancy [5], [6].

Theoretical issues in bus encoding for low transition activity are investigated in [7]. In that 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] and [9], 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.

In [8] and [9], 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.

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. In addition, we offer the possibility of trading off bus activity reduction for interface complexity. In fact, designers can exploit our approach to rapidly explore the power-saving opportunities enabled by low-transition encodings.

Experimental results concerning both the quality of the encoding schemes and the efficiency of the encoding–decoding circuitry are very satisfactory for a large variety of data streams.

The rest of the paper is organized as follows. In Section II, we introduce our generic encoder–decoder architecture and we study its properties. In Section III, we describe the basic, exact encoding algorithm, in Section IV we outline approximate variants to be used for low-cost encoding/decoding of fast and wide buses, and in Section V we present the adaptive encoder–decoder architecture. Experimental results are reported in Section VI. Finally, Section VII concludes the work.

II. 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 $x =$

Manuscript received August 12, 1999; revised February 7, 2000. This paper was recommended by Associate Editor M. Pedram.

L. Benini is with the Università di Bologna, DEIS, 40136 Bologna, Italy.

A. Macii, E. Macii, M. Poncino, and R. Scarsi are with Politecnico di Torino, DAI, 10129 Torino, Italy (e-mail: enrico@athena.polito.it).

Publisher Item Identifier S 0278-0070(00)07472-8.

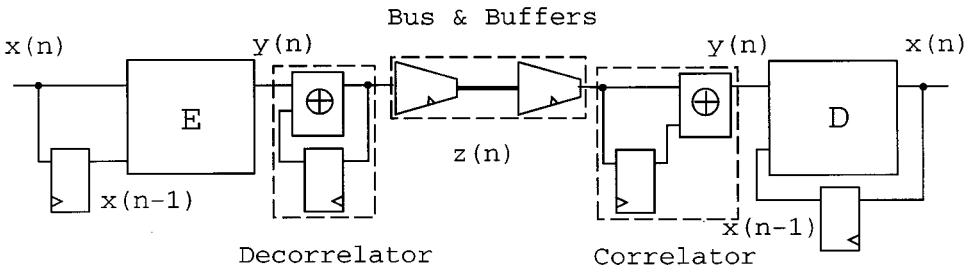

Fig. 1. General codec architecture.

$[b_1, b_2, \ldots, b_W]$. $\mathcal{X}$ is the Boolean space $\mathcal{B}^W$ such that every W-bit configuration has nonnull probability of being generated by the data source. Symbols x must be transmitted over time on a 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 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.

A. General Codec Architecture

We consider a general encoder–decoder (*codec*, for brevity) architecture, shown in Fig. 1, which is a specialization of the *source-coding encoder–decoder framework* introduced in [8], [9]. The encoder takes as input the stream of W-bit input words $x(n)$, $n = 0, 1, \ldots$. It consists of three block:

- a register, that stores $x(n-1)$ when the input is $x(n)$;
- a combinational *encoding function*, E: $\mathcal{B}^W \times \mathcal{B}^W \rightarrow \mathcal{B}$, that generates the encoded word $y(n)$ from $x(n), x(n-1)$;
- a *decorrelator*, that translates one-valued bits of $y(n)$ into transitions on the corresponding bus lines (zero-valued bits correspond to stationary values on the bus lines).

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:

- A *correlator*, that computes the inverse function of the decorrelator and reconstructs $y(n)$;
- 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 $x(n-1)$;
- A register, that stores $x(n-1)$ when the output of the decoder is $x(n)$.

Fig. 1 also shows the bus and its input/output buffers (this block is independent from the codec architecture). We assume that information is sent over the bus using standard CMOS level signaling. Also, notice that bus drivers and receivers are clocked (hence, glitches on the decorrelator outputs are filtered out), and that the "time of flight" delay of the bus may be nonnegligible. Additional latency introduced by bus and drivers is not a concern for our encoding scheme. In the following, wewill assume that the bus latency is zero time periods without loss of generality.

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 encoding and decoding.

Clearly, the architecture of Fig. 1 is a generic scheme that can be customized by defining functions E and D. It is possible to further generalize the architecture by considering more than one past input words for encoding and decoding. In the general case, function E takes k input words to compute a single output word. Similarly, function D takes k previously decoded words, as well as the newly received word to produce the decoded output word. We will describe the algorithms for computing functions E and D assuming a value of $k = 2$. This is for two main reasons; first, the case $k = 2$ can be illustrated in very simple terms, and generalization to arbitrary k is straightforward, once the basic concepts have been understood. Second, and most important, hardware complexity of E and D rapidly increases with k, and schemes with $k > 2$ may be inapplicable in practice.

III. Exact-Low Transition Encoding Algorithm

The algorithm we present 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., $Prob(x(n) = x_i \wedge x(n-1) = x_j)$ is independent of the time index n.

```
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);
}
```

Fig. 2. Code construction algorithm.

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 columns 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 pseudocode of the algorithm is shown in Fig. 2. Its only input parameter is the initial code table *CodeTab* (a matrix with 2^{2W} rows and three 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, discussed next, is the update rule for array *Conflicts*. The algorithm terminates when the code for the last row has been assigned and returns the complete code table.

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 $y(n)$. The decoder function takes as inputs $x(n-1)$ and $y(n)$ and produces as output the correct $x(n)$ value (Fig. 1).

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. If this is not done and, for instance, code y_k is assigned to another row of the table with x_k and x_j in the first two columns, then the decoder will not have a way to know if the original data word is x_k or x_i, because both are associated to the same pair $y(n) = y_k$ and $x(n-1) = x_j$. The nature of the decodability constraint is best illustrated through an example.

Example 1: Consider a simple data stream with four symbols ($\mathcal{X} = \mathcal{B}^2$ and $W = 2$). The code table has $2^{2\cdot 2} = 16$ rows. The first four rows of the initial code table are shown in Fig. 3(a). No encoded word is assigned yet. Rows are ordered for decreasing probability. The first and second codes are assigned in Fig. 3(b). Code word 00 is selected because it contains the minimum number of ones. The content of array *Conflicts*, when nonempty, is shown to the right of the table.

Assigning code 00 to the first row creates forbidden codes in rows 3 and 4, because they have the same value as row 1 in column $x(n-1)$. For instance, code 00 cannot be assigned to the third row because the decoder, observing $x(n-1) = 10$ and $y(n) = 00$ would not be capable of selecting between $x(n) = 01$ and $x(n) = 11$. Since 00 is forbidden, we must select a different code with as few ones as possible. We assign 01 to the third row. Notice that this choice adds one forbidden code for the fourth row of the table, as shown in Fig. 3(c).

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.

It is important to notice that procedure `BuildTable` always finds an encoding that can be uniquely decoded. To prove this claim, we observe that for each value x_i in the second column of the table, there are 2^W rows with the same second-column value. A conflict in code assignment arises only if we try to assign the same y_i to two rows with the same second-column value. In that case, we need to choose another y_j for the row that comes last in the ordering of the table. The worst case is when we are trying to assign the third-column value to the last row in the table that has a second-column value x_i (i.e., the 2^Wth row with second-column value x_i), and all previous rows have been assigned different values of y. Fortunately, there are only $2^W - 1$ rows before the last one, hence there are $2^W - 1$ conflicts in the worst case, while we have 2^W available configurations to assign the value of the third column. This guarantees that we can always find a valid code for the third column; thus, procedure `BuildTable` always terminates correctly with a complete code table.

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 functions 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.

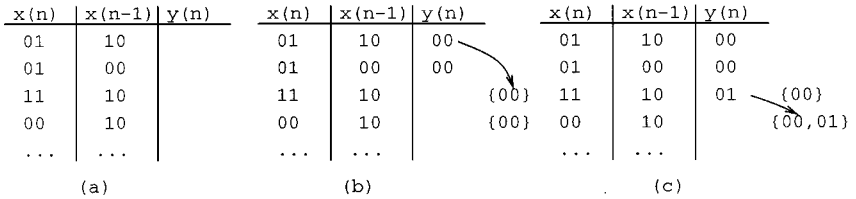

Fig. 3. Example of code construction.

IV. Approximate Low-Transition Encoding Algorithms

A. Clustered Encoding

The most intuitive approximation to the exact algorithm of Section III consists of 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. The criterion for cluster growth is then very important. Since breaking a word into clusters decreases the spatial correlation between bits, we must try to keep in the same cluster bits with high mutual spatial correlation. The clustering algorithm we have used exploits the calculation of various types of correlations and is similar to the one proposed in [6]; the interested reader may refer to that work for the details.

The architecture generated by the 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.

B. Discretized Encoding

An alternative approximate solution is to consider only the M most probable pairs of consecutive words in the code, where $M \ll 2^W$. 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 Fig. 4, 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 goes through a *background* function [denoted with $B(x(n), x(n-1))$].

The reason for the existence of the background function is that the architecture of Fig. 4 represents one realization of the block denoted with E in the general architecture of Fig. 1, whose output $y(n)$ feeds the decorrelator. The only constraint on B is

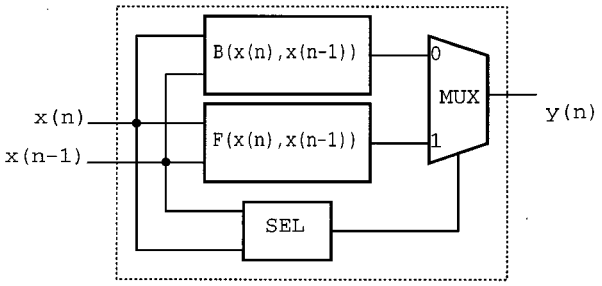

Fig. 4. Architecture of the discretized encoder.

that it should not violate the decodability constraint and it should be invertible. For instance, if we want to transmit on the bus the words outside $\mathcal{S}$ without any change, block B should implement a correlator ($B(x(n), x(n-1)) = x(n) \oplus x(n-1)$) to cancel the effect of the decorrelator that follows E. Other choices are possible: Identity and negation are two valid alternatives. Note that the choice of B is not critical because we are assuming that words outside $\mathcal{S}$ are transmitted with small probability.

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.

The construction of the encoding function F, unlike the clustered approximation, requires the modification of the basic algorithm of Section III.

In discretized encoding, constraints imposed by assigning a new code to a pair may create a conflict with another pair that 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 may affect 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 (assuming that B is a correlator).

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. A good way to attempt 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.

The removal of conflicts with background pairs is best illustrated through an example.

Example 2: Consider the table at the top, where codes have been replaced by symbols for the sake of clarity. The pair $P_1 = (A,\ B)$ represents the pair just assigned by the algorithm, and C is the chosen code. The horizontal line separates $\mathcal{S}$ from the background pairs. Below this line, P_2 denotes background pair that is conflicting with P_1

	x(n)	x(n − 1)	y(n)
	...	...	...
P_1 :	A	B	C
	...	...	...
...	...	...	
P_2 :	$B \oplus C$	B	C

$\Longrightarrow$

x(n)	x(n − 1)	y(n)
...	...	...
A	B	C
$B \oplus C$	B	$\boldsymbol{A}$
...	...	...
...	...	...

To remove the conflict, P_2 has to be taken into $\mathcal{S}$, and assigned a new value. This value X is obtained by solving the following equation for the unknown $X = y^{(P_2)}(n)$

$$y^{(P_2)}(n) \oplus x^{(P_2)}(n-1) = x^{(P_1)}(n).$$

This translates to $X \oplus B = (A \oplus B)$, that is $X = A$, as shown in the table at the bottom.

V. Adaptive Encoding

The solutions described in Sections III and IV require that word-pair statistics are known before synthesizing the codec. 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 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 III on the basis of approximate statistical information, that are collected by observation of the bit stream over a window of fixed size S. Clearly, the window size must be chosen as a compromise between adaptation speed and delay in updating the statistics. In particular, the optimal value of S depends on the streams of patterns that must be transmitted; therefore, it must be tuned by experimenting with different window sizes and by selecting the one that performs best on average for a given set of data. In our case, for each of the input streams that we have considered, we have tried values of S going from 16 to 1024, and we have plotted the corresponding value of the bus switching activity. From the diagram of Fig. 5, that summarizes the results of our analysis (on the y axis switching activity values are normalized), we can evince that the highest savings are achieved, on average, for a value of $S = 64$.

The application of the exact algorithm of Section III 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 analyze the frequency distribution, we note that not all the four occurrences are 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 zero-to-one and one-to-zero 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} differ by one 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}. Reducing the information that is required is beneficial from the hardware implementation point of view, because there are fewer quantities that need to be stored.

A. Encoder Architecture

The basic scheme of the architecture for the 1-bit encoder is shown in Fig. 6. 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

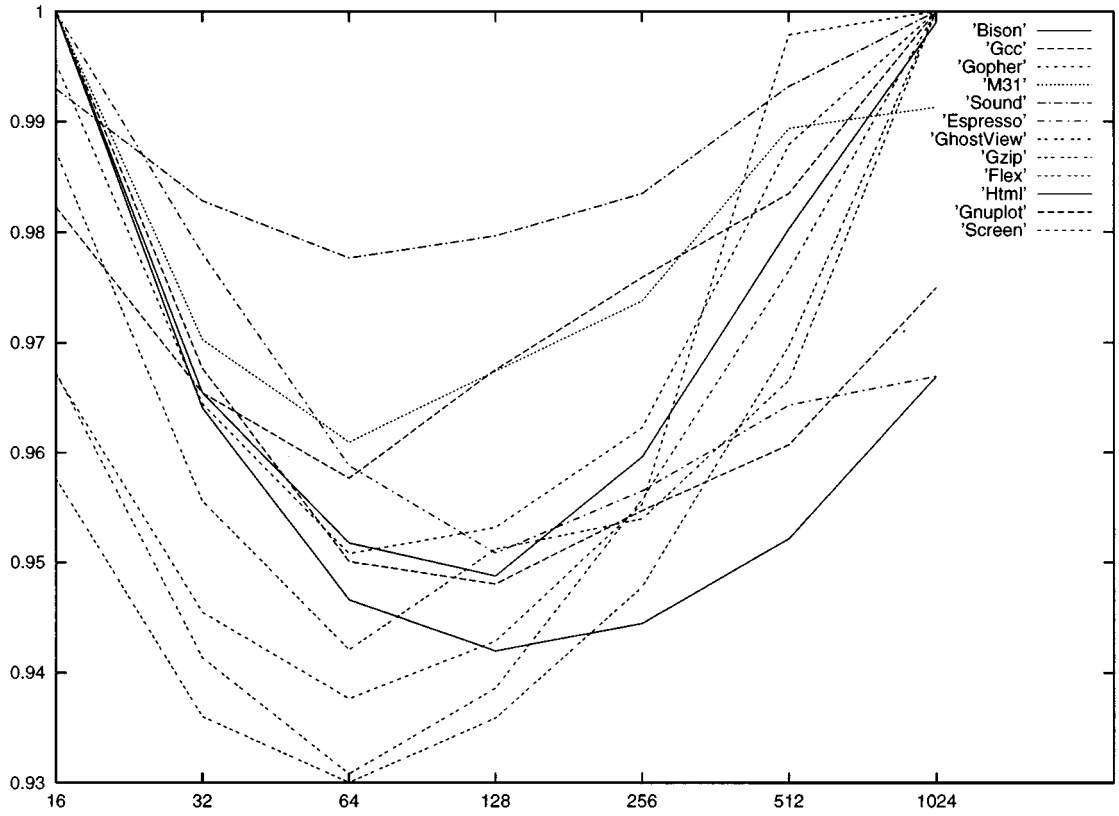

Fig. 5. Experimental search of the optimal window size (S).

properly resets the two counters. The window counter is shared across all the bits in the bus.

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 emphasize 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:

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

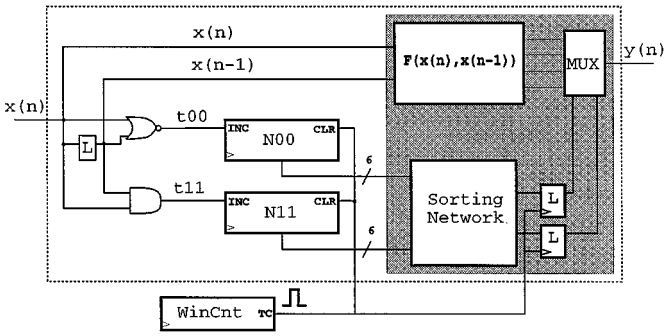

Fig. 6. Architecture of the adaptive encoder.

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 zero in the stream will result in no transition after the decorrelator in the scheme of Fig. 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 ones, 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], [9] as a particular case.

It is important to observe that the sorting rules of (1) rely on two approximations, that may lead to suboptimal results in some special cases. The first one is due to the assumption that $N_{01} = N_{10}$ which is only asymptotically true. The second is that the above inequalities are always considered as *strict* inequalities.

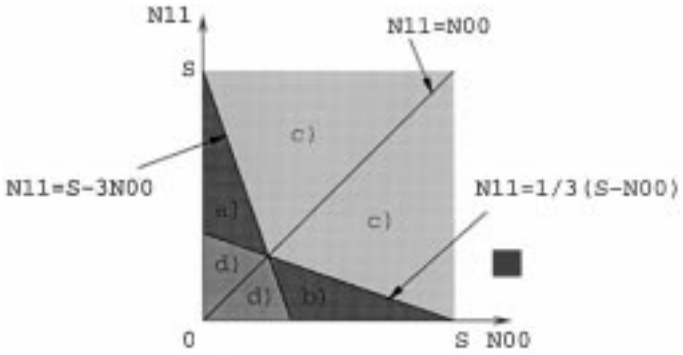

Fig. 7. Space of the sorting network.

Consider for example the following JPD over a given window of size $S = 64$: $N_{00} = 25$, $N_{01} = 12$, $N_{10} = 13$, and $N_{11} = 13$. The exact ordering is clearly $(N_{00} < N_{10} = N_{11} < N_{01})$. According to our computation, however, we will infer N_{10} and N_{01} from N_{00} and N_{11}, and get another JPD: $N_{00} = 25$, $N_T = 13$, and $N_{11} = 13$. Therefore, we are not able to distinguish the two orderings $(N_{00} < N_T < N_{11})$ from $(N_{00} < N_{11} < N_T)$, that require different encoding functions, according to (1).

The decision rules described in (1) can be graphically represented as in Fig. 7, 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:

$$\begin{aligned} N_T > N_{00} &\rightarrow \frac{(S-(N_{00}+N_{11}))}{2} \\ &> N_{00} \rightarrow N_{11} + 3N_{00} - S < 0 \\ N_T > N_{11} &\rightarrow \frac{(S-(N_{00}+N_{11}))}{2} \\ &> N_{11} \rightarrow 3N_{11} + N_{00} - S < 0. \end{aligned}$$

Notice that regions a) and b) are symmetric around the line $N_{11} = N_{00}$, denoting the fact that in these two regions the relative magnitude of N_{00} and N_{11} is irrelevant.

B. 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 Fig. 7. We observe that counters N_{00} and N_{11} and the sorting network can be merged together. The inequalities of Section V-A can be rewritten as

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

Instead of computing N_{00} and N_{11}, and derive the two left-hand sides 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 left-hand sides of the inequalities of (2) is bounded by $4S$, and can then be stored in a register with $(\log_2 S) + 2$ bits.

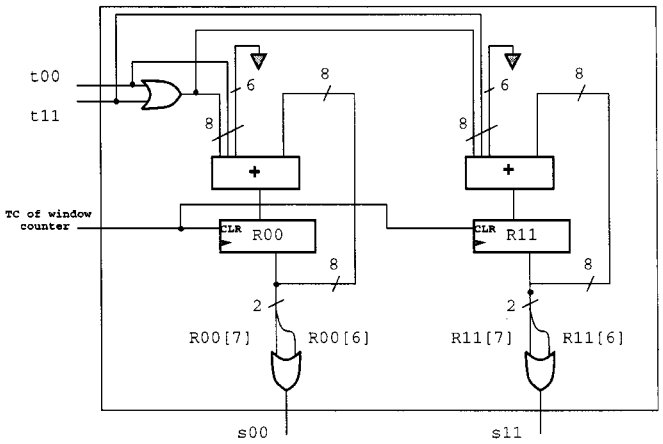

Fig. 8. Efficient encoder implementation.

The dashed box in Fig. 8 shows the optimized schematic that merges the two counters of Fig. 6 and the sorting network for a window of size $S = 64$.

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 zero-to-zero and one-to-one transitions. For example, for register R_{00}:

- $C = 0$, if $(t_{00}, t_{11}) = 00$;
- $C = 1$, if $(t_{00}, t_{11}) = 01$;
- $C = 3$, 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 conditions of (2) can be obtained by looking at the value contained in the two registers. The values of their two most significant bits (bit 6 and 7) express the condition that the number stored in the register exceeds the value of $S = 64$ (and cannot thus be represented using only six bits). By OR-ing these two bit pairs we obtain two selection signals s_{00} and s_{11} that can be used to directly drive the output multiplexor of Fig. 6. More precisely, $s_{00} = 1$ implies $R_{00} > S$, that is, $N_T < N_{00}$; similarly, $s_{11} = 1$ implies $N_T < N_{11}$.

The four combinations of (s_{00}, s_{11}) select the proper encoding function, as follows:

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

Concerning the performance of the encoder, the critical path runs through the block F and the multiplexor, in the upper part of Fig. 6. Since the encoding functions F consist of at most one gate, we can conclude that in the worst case we have two

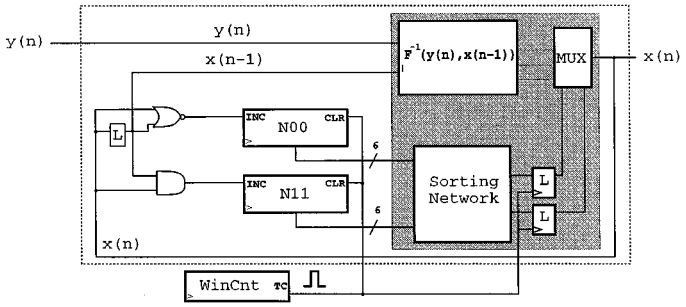

Fig. 9. Architecture of the adaptive decoder.

or three equivalent gates on the critical path, depending on the multiplexor implementation.

C. Decoder Architecture

The architecture of the decoder, shown in Fig. 9, 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 differences with respect to the encoder. First, according to the architectural scheme of Fig. 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 (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.

VI. Experimental Results

In this section, we report the data we have collected through extensive experimentation of the encoding schemes proposed in this paper. We first consider exact and approximate encodings, then we focus on the adaptive solution.

A. Exact and Approximate Encoding

We have applied the exact (see Section III) and the approximate variants (see Section IV) of the new encoding algorithm to a set of real-life streams with various statistical profiles. The streams we have considered are the following, and we assume they have to be transmitted over a 32-bit data bus:

- *sound*: A `.wav` file;
- *m31*: An image in the `.ppm` format;
- *screen*: An image in raw format captured from a screen;
- *html*: A HTML page containing some images;
- *gopher*, *gzip*, *gcc*, *bison*, *espresso*, *ghostview*, *gnuplot*, *flex*: Executable files.

Table I collects the data of the experiments regarding exact encoding. Column *Initial* gives the number of transitions occurring in the stream when no encoding is applied. Column *Exact* reports the number of transitions in the stream after encoding is applied (*Trans*) and the percentage of achieved transition savings (*Sav*). Columns *xor-pbm* and *dbm-pbm* show similar data as obtained from application of the *xor-pbm* and *dbm-pbm* methods of [8] and [9]. The motivation for choosing *xor-pbm* for the comparison is that it seems the most promising for the case of data buses. On the other hand, *dbm-pbm* has been picked because, as pointed out in [8] and [9], it is the one that, in general, works best.

TABLE I
Results: Transition Savings for Exact Encoding

Stream	*Initial*	*Exact*		*xor-pbm*		*dbm-pbm*	
	Trans	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*
sound	1391981	13693	99.0%	398373	71.4%	392445	71.8%
m31	1014066	26193	97.4%	375960	62.9%	361786	64.3%
screen	211298	44912	78.7%	69695	67.0%	81441	61.4%
html	281230	4571	98.3%	67380	76.0%	66442	76.3%
gopher	586796	10381	98.2%	162825	72.2%	158004	73.1%
gzip	250782	4343	98.2%	69823	72.1%	68828	72.5%
gcc	191888	8106	95.7%	51826	72.9%	50236	73.8%
bison	600006	20700	96.5%	187870	68.7%	183018	69.4%
espresso	793074	55845	92.9%	282238	64.4%	266963	66.3%
ghostview	501286	23215	95.4%	140703	71.9%	137870	72.5%
gnuplot	2136288	127213	94.0%	709494	66.8%	679348	68.2%
flex	769122	51651	93.3%	244272	68.2%	234046	69.6%
Average			94.8%		69.6%		69.9%

The numbers clearly support the claim that our exact algorithm out-performs both *xor-pbm* and *dbm-pbm*. The average savings is, in fact, 94.8% as opposed to 69.6% of *xor-pbm* and 69.9% of *dbm-pbm*.

Results of the application of our approximate encoding methods are shown in Table II. Data for the discretized algorithm (column *Discretized*) consist of three sets of numbers (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: The first refers to the case of eight clusters of size 4, the second to the case of four clusters of size 8.

A key observation about the data in Table II is that the clustered algorithms perform almost as well as *xor-pbm* and *dbm-pbm* when the latter are applied to the entire 32-bit bus (see data in Table I). This is an important result, because our exact algorithm, as well as *xor-pbm* and *dbm-pbm* are of limited practical applicability, due to the size and complexity of the encoders and decoders they necessitate. In fact, any code for which the codec contains a combinational function with a number of inputs larger than (as for our exact algorithm) or equal to (as for *xor-pbm* and *dbm-pbm*) the bus width, may be unusable on wide buses. This is because, as bus width grows, it becomes more difficult building the encoding and decoding functions with a reasonably small digital circuit.

In our specific case, since the data bus we considered was 32-bit wide, the *pbm* part of the encoder could not be synthesized for any of the benchmarks we used in the experiments. Obviously, the same thing happened for our exact encoder, since it encompasses a 64-bit combinational function. Conversely, as the results of Section VI-A1 will show, both the clustered and the discretized methods have been successfully implemented in hardware; therefore, the reported savings will translate to realistic power reductions.

We note also that the discretized encoding is typically less effective than the clustered one; this indicates that, for the streams

TABLE II
RESULTS: TRANSITION SAVINGS FOR APPROXIMATE ENCODING

Stream	*Initial Trans*	*Discretized*						*Clustered*			
		20		*50*		*100*		*8*		*4*	
		Trans	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*
sound	1391981	1380043	0.8%	1374548	1.2%	1371364	1.4%	753472	45.0%	701599	49.5%
m31	1014066	1003540	1.0%	999420	1.4%	995816	1.7%	590921	41.7%	584258	42.3%
screen	211298	191204	9.5%	156806	25.7%	142586	32.5%	128817	39.0%	116772	44.7%
html	281230	268634	4.4%	255102	9.2%	240009	14.6%	165642	41.1%	112362	60.0%
gopher	586796	571718	2.5%	565205	3.6%	557220	5.0%	276592	52.9%	223502	61.9%
gzip	250782	246986	1.5%	244358	2.5%	242278	3.3%	122428	51.1%	100716	59.8%
gcc	191888	187176	2.4%	182168	5.1%	175887	8.3%	93489	51.2%	66575	65.3%
bison	600006	582781	2.9%	576739	3.9%	569141	5.1%	290447	51.6%	224925	62.5%
espresso	793074	769344	2.9%	763399	3.7%	753012	5.0%	394497	50.3%	336755	57.5%
ghostview	501286	487255	2.8%	477910	4.7%	468652	6.5%	249519	50.2%	198370	60.4%
gnuplot	2136288	2051129	3.9%	2011647	5.8%	1971949	7.7%	1076603	49.6%	887030	58.5%
flex	769122	735821	4.3%	724072	5.8%	716063	6.9%	375325	51.2%	298656	61.2%
Average			3.2%		6.1%		8.2%		47.9%		56.9%

TABLE III
RESULTS: CODEC IMPLEMENTATION FOR APPROXIMATE ENCODING

Stream		*Discretized*									*Clustered*		
		20			*50*			*100*			*8*		
		A	*P*	*D*	*A*	*P*	*D*	*A*	*P*	*D*	*A*	*P*	*D*
sound	Enc	5895	2.40	1.99	20079	8.07	3.38	49761	22.17	4.88	43623	27.64	2.19
	Dec	5865	2.39	1.96	19867	7.97	3.19	49731	21.73	4.67	47520	30.80	2.12
m31	Enc	6606	3.28	2.55	18747	9.72	3.18	56115	32.01	3.66	42390	25.95	1.93
	Dec	6550	3.24	2.54	18286	9.44	3.26	55880	31.75	3.64	42561	28.09	2.06
screen	Enc	3654	1.87	1.79	9981	5.32	2.23	31761	16.50	3.50	39393	25.27	2.47
	Dec	3649	1.82	1.79	10006	5.29	2.35	31750	16.49	3.51	37300	24.55	1.89
html	Enc	26523	13.04	3.15	61668	35.32	3.68	118512	74.13	4.73	46708	29.64	2.14
	Dec	26506	13.02	3.03	58631	34.85	3.61	118445	73.86	4.69	47673	31.34	2.23
gopher	Enc	12348	5.28	2.76	37782	19.09	3.70	77022	45.11	4.35	45214	27.69	2.11
	Dec	12270	5.32	2.76	37582	18.69	3.61	76965	44.66	4.35	45684	29.86	2.23
gzip	Enc	20124	9.02	3.03	49392	25.79	3.43	96003	56.77	4.78	43374	27.29	2.17
	Dec	20062	9.03	3.01	49993	25.84	3.45	96060	57.02	4.80	42759	27.79	2.20
gcc	Enc	11592	5.32	2.65	42273	23.07	3.38	101457	57.46	4.97	43560	27.28	1.99
	Dec	11603	5.34	2.62	42837	22.74	3.33	101253	57.02	4.94	44417	27.84	2.05
bison	Enc	13518	5.83	3.07	25713	12.68	3.51	56178	30.30	4.29	43362	27.25	2.02
	Dec	13289	5.81	3.04	26188	12.85	3.58	56567	30.22	4.30	42284	27.62	1.99
espresso	Enc	6201	2.88	2.36	8469	4.19	2.48	27180	10.50	4.91	40950	25.14	5.44
	Dec	6312	2.89	2.28	8463	4.18	2.53	27642	10.37	4.85	41013	25.28	5.52
ghostview	Enc	6444	2.61	2.21	13203	4.94	2.58	30343	12.99	5.07	42426	26.66	5.64
	Dec	6378	2.59	2.15	13198	4.93	2.32	30789	13.22	5.18	42131	26.40	5.47
gnuplot	Enc	4968	2.00	2.01	14895	6.53	3.18	34176	19.12	5.17	41022	25.59	5.70
	Dec	5043	2.02	2.03	14882	6.51	3.35	34864	19.27	5.37	41718	25.70	5.84
flex	Enc	5490	2.00	2.35	16748	9.52	4.37	39033	16.64	5.61	42318	25.52	4.87
	Dec	5504	2.02	2.27	16771	9.54	4.12	39873	16.77	5.87	43004	25.70	5.01

we considered, preserving temporal correlation is more important than preserving spatial correlation.

1) Codec Implementation: Table III provides data about codec implementation for the cases of approximate methods. Synthesis has been carried out using Synopsys Design Compiler, with a 0.25 μm, 2.5-V technology library from ST Microelectronics. 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 milliwatts), and delay (in nanoseconds) for both encoders and decoders of each stream. Power estimates have been obtained with Synopsys DesignPower with a clock frequency of 400 MHz.

For the discretized method, all three versions (i.e., $M =$ 20, 50, and 100) were successfully implemented. As expected,

TABLE IV
RESULTS: TRADE-OFF ANALYSIS OF BUS-LINE CAPACITANCE

Stream	*Discretized*			*Clustered*
	20	*50*	*100*	*8*
sound	30.77	70.57	163.31	7.02
m31	45.05	95.15	254.10	9.29
screen	38.08	40.38	99.57	8.27
html	37.30	48.41	64.72	9.51
gopher	28.80	71.67	124.32	7.60
gzip	93.49	158.01	263.08	8.44
gcc	37.07	77.22	117.22	9.18
bison	33.22	53.93	96.38	8.71
espresso	16.24	18.84	34.79	8.45
ghostview	13.97	15.91	30.27	7.94
gnuplot	7.73	17.14	38.27	7.93
flex	7.71	27.03	40.23	8.31

TABLE V
RESULTS: COMPARISON OF EXACT AND BEACH CODING

Stream	*Initial*	*Exact*		*Beach*		*dbm-pbm*		*inc-xor*	
	Trans	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*
dashb	619690	32070	94.8%	443115	28.4%	128203	79.3%	559030	9.8%
dct	48917	2917	94.0%	31472	35.6%	9622	80.3%	41464	15.2%
fft	138526	7107	94.9%	85653	38.1%	24044	82.6%	120502	13.0%
mat_mul	105947	5934	94.4%	60654	42.7%	16669	84.3%	85153	19.6%
vxv_mul	133272	5576	95.8%	46838	64.8%	16588	87.6%	121744	8.6%
Average			94.8%		41.8%		82.8%		13.2%

TABLE VI
RESULTS: COMPARISON OF APPROXIMATE AND BEACH CODING

Stream	*Initial*	*Discretized*						*Clustered*				*Beach*	
	Trans	*20*		*50*		*100*		*8*		*4*		*Trans*	*Sav*
		Trans	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*		
dashb	619690	562253	9.3%	513419	17.1%	479680	22.6%	282853	54.4%	143904	76.8%	443115	28.4%
dct	48917	44099	9.8%	36860	24.6%	26934	44.9%	19317	60.5%	11681	76.1%	31472	35.6%
fft	138526	128800	7.0%	118800	14.2%	103464	25.3%	54617	60.6%	26287	81.0%	85653	38.1%
mat_mul	105947	80704	23.8%	34116	67.8%	22092	79.1%	34313	67.6%	19699	81.4%	60654	42.7%
vxv_mul	133272	32005	76.0%	31618	76.3%	31046	76.7%	32786	75.4%	14654	89.0%	46838	64.8%
Average			25.2%		40.0%		49.7%		63.7%		80.9%		41.8%

the complexity of the codec tends to increase rapidly for larger values of M. Concerning the clustered scheme, only the version with eight clusters of size 4 resulted in a compact implementation. Notice that the values of area, power and delay for the clustered method are comparable to those of the discretized method with $M = 50$ and $M = 100$, although the latter provide sensibly smaller savings.

We complete our analysis of the synthesized codecs by calculating the break-even point of the trade-off curve between capacitance per bus line and bus power savings. In other words, for each data stream, we determine the minimum capacitance each bus line should have in order for the encoding to pay off. (The interested reader may find a similar investigation for a number of existing encoding schemes in [10].) From the data in Table IV we can evince that, while discretized encoding may be applicable only at the interface of off-chip buses (capacitance per bus line is within a few tens of pF), the cost of the clustered codec is affordable also in the case of on-chip buses, since capacitance per bus line never exceeds 10 pF.

2) Comparison to Beach: As our exact and approximate encodings, also the Beach scheme introduced in [6] is applicable in cases where statistics about the stream being transmitted over the bus are available up-front. On the other hand, the Beach code is specifically thought for encoding address buses, since code construction is based on the identification of correlations that are typical of address sequences.

Tables V and VI compare the results of the application of exact and approximate encodings to those achieved by the Beach code when the address streams being transmitted are those described in [6]. For completeness in the comparison, Table V also includes the data obtained by using *dbm-pbm* and *inc-xor*, the latter being the version of the source-coding framework of [8] and [9] which is most appropriate for address buses.

We observe that, although more general, the encoding schemes introduced in this paper are clearly superior to the Beach method. It is also worth noting the limited effectiveness of *inc-xor*. This behavior is justified by the fact that the address streams we have used are characterized by a low sequentiality; this in sharp contrast with the assumption on which *inc-xor* is based (i.e., most addresses are consecutive).

TABLE VII
RESULTS: ADAPTIVE ENCODING

Stream	*Initial*	*Adaptive*					
	Trans	*Trans*	*Sav*	*Area*	*Power*	*Delay*	*Min Cap*
sound	1391981	1361505	2.1%				161.67
m31	1014066	977862	3.5%				129.05
screen	211298	218482	-3.4%				—
html	281230	255075	9.3%				44.28
gopher	586796	475075	19.0%				23.55
gzip	250782	209846	16.3%	99360	32.12	0.36	30.85
gcc	191888	161885	15.6%				35.08
bison	600006	495198	17.4%				30.13
espresso	793074	632667	20.2%				26.75
ghostview	501286	404854	19.2%				25.10
gnuplot	2136288	1799620	15.8%				31.26
flex	769122	595088	22.6%				23.59
Average			13.1%				

B. Adaptive Encoding

The same streams described in Section VI-A have been used to benchmark the performance of the adaptive encoding scheme introduced in Section V. Table VII collects all the results, including savings in number of bus transitions, encoder implementation data (numbers for the decoder are not reported since it has roughly the same realization of the encoder, as mentioned in Section V-C), and minimum bus-line capacitance required to guarantee an advantage in the usage of the codec.

Concerning codec implementation (one number only is reported for area, power dissipation and delay, since the synthesized circuit is the same for all streams), we observe that the adaptive encoder is typically larger than the approximate solutions. One desirable characteristics of the adaptive interface logic, however, is its negligible delay (0.36 ns), which is at least five times smaller than the fastest among the other encoders/decoders.

TABLE VIII
RESULTS: COMPARISON OF ADAPTIVE AND BUS-INVERT ENCODING

Stream	*Initial*	*Adaptive*		*Bus-Invert*		*Bus-Invert (2 Cl.)*		*Bus-Invert (4 Cl.)*	
	Trans	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*	*Trans*	*Sav*
sound	1391981	1361505	2.1%	1276816	8.3%	1000505	28.1%	837599	39.8%
m31	1014066	977862	3.5%	1006144	0.8%	935863	7.7%	884509	12.8%
screen	211298	218482	-3.4%	211245	0.1%	209894	0.7%	209212	1.0%
html	281230	255075	9.3%	279201	0.7%	255019	9.3%	235721	16.2%
gopher	586796	475075	19.0%	563765	3.9%	482080	17.8%	441220	24.8%
gzip	250782	209846	16.3%	242115	3.5%	211576	15.6%	190198	24.2%
gcc	191888	161885	15.6%	185109	3.5%	165572	13.7%	152422	20.5%
bison	600006	495198	17.4%	574985	4.2%	501732	16.4%	454536	24.2%
espresso	793074	632667	20.2%	755218	4.8%	668879	15.7%	615719	22.4%
ghostview	501286	404854	19.2%	487087	2.8%	426871	14.8%	388463	22.5%
gnuplot	2136288	1799620	15.8%	2044307	4.3%	1807634	15.4%	1639940	23.2%
flex	769122	595088	22.6%	745663	3.1%	634596	17.5%	576974	25.0%
Average			13.1%		3.3%		14.4%		21.4%

1) Comparison to Bus-Invert: To fairly evaluate the effectiveness of the adaptive scheme, we have compared its performance against the Bus-Invert code [4]. Although spatially redundant codes were excluded from our analysis because of our tight constraint 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.

We have considered three versions of the Bus-Invert code: The base case (i.e., no clustering of the 32-bit data bus), a two-cluster solution (each cluster contains 16 bus lines) and a four-cluster solution (each cluster is of size 8). Data are collected in Table VIII.

The adaptive code clearly out-performs the nonclustered Bus-Invert code, it provides results similar to those of the two-cluster version and it is less effective than the four-cluster implementation. However, as already mentioned, the Bus-Invert code is redundant by nature. The clustered versions imply a further increase in the number of bus lines that have to be added to the bus interface (i.e., one line for each cluster). Since pin count is normally a scarse resource, these schemes may not be usable in practice.

VII. CONCLUSION

We have presented novel algorithms for the automatic synthesis of bus interface logic that targets the minimization of the switching activity on global buses. In particular, we have addressed the problem of minimizing the bus activity in three different situations.

For buses with limited width and known statistics of the streams being transmitted, we have proposed a method that allows us to exactly determine the optimum encoding scheme. For wider buses, the exact approach is no longer applicable, since the complexity of the synthesized encoding/decoding logic may become unmanageable. We have solved this problem by means of two classes of encoding algorithms. The first one reduces the size of the codec by separately considering clusters of bus lines instead of the entire bus width; the second scheme re-encodes only a subset of the patterns being transmitted, namely, those with higher occurrence probability. Both these methods still rely on the assumption of full knowledge of the input statistics. When this assumption is no longer satisfied, a different path must be followed. Thus, we have introduced an adaptive strategy that dynamically modifies the encoding function depending on the patterns that are being transmitted.

We have benchmarked the capabilities of the proposed encoding techniques on a set of data streams; we have also investigated the trade-off between optimality of the encoding scheme and complexity of the encoding/decoding circuitry, and we have pointed out possible domains of applicability of the presented encoding approaches. The results we have obtained from an extensive experimentation are satisfactory, since they have outperformed those produced by the most effective schemes that are currently available.

REFERENCES

[1] M. R. Stan and W. P. Burleson, "Bus-invert coding for low-power I/O," *IEEE Trans. VLSI Syst.*, vol. 3, pp. 49–58, Mar. 1995.
[2] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems," in *Proc. GLS-VLSI-97: IEEE/ACM 7th Great Lakes Symp. VLSI*, Urbana-Champaign, IL, Mar. 1997, pp. 77–82.
[3] E. Musoll, T. Lang, and J. Cortadella, "Working-zone encoding for reducing the energy in microprocessor address buses," *IEEE Trans. VLSI Syst.*, vol. 6, pp. 568–572, Dec. 1998.
[4] M. R. Stan and W. P. Burleson, "Low-power encodings for global communication in CMOS VLSI," *IEEE Trans. VLSI Syst.*, vol. 5, pp. 444–455, Dec. 1997.
[5] H. Mehta, R. M. Owens, and M. J. Irwin, "Some issues in gray code addressing," in *Proc GLS-VLSI-96: IEEE/ACM 6th Great Lakes Symp. VLSI*, Ames, IA, Mar. 1996, pp. 178–180.
[6] L. Benini, G. De Micheli, E. Macii, M. Poncino, and S. Quer, "Reducing power consumption of core-based systems by address bus encoding," *IEEE Trans. VLSI Syst.*, vol. 6, pp. 554–562, Dec. 1998.
[7] S. Ramprasad, N. R. Shanbhag, and I. N. Hajj, "Information-theoretic bounds on average signal transition activity," *IEEE Trans. VLSI Syst.*, vol. 7, pp. 359–368, Sept. 1999.
[8] ——, "A coding framework for low power address and data busses," *IEEE Trans. VLSI Syst.*, vol. 7, pp. 212–221, June 1999.
[9] ——, "Signal coding for low power: Fundamental limits and practical realizations," *IEEE Trans. Circuits Syst. II*, vol. 46, pp. 923–929, July 1999.
[10] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Address bus encoding techniques for system-level power optimization," in *Proc. DATE-98: IEEE Design Automation and Test in Europe*, Paris, France, Feb. 1998, pp. 861–866.

Luca Benini received the Dr.Eng. degree in electrical engineering from the Università di Bologna, Bologna, Italy, in 1991, and the M.S. and Ph.D degrees in electrical engineering from Stanford University, Stanford, CA, in 1994 and 1997, respectively.

Currently he is an Assistant Professor at the Università di Bologna. His research interests are in all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications.

Alberto Macii received the Dr.Eng. degree in computer engineering from Politecnico di Torino, Torino, Italy, in 1996. Currently, he is working toward the Ph.D. degree in computer engineering at the same institution.

His research interests include several aspects of the development of algorithms, methods, and tools for low-power digital design.

Enrico Macii (M'91) received the Dr.Eng. degree in electrical engineering from Politecnico di Torino, Torino, Italy, in 1990, and the Dr.Sc. degree in computer science from the Universitè di Torino in 1991 and the Ph.D. degree in computer engineering from Politecnico di Torino in 1995, respectively.

From 1991–1994, he was an Adjunct Faculty at the University of Colorado at Boulder.Currently he is an Associate Professor at Politecnico di Torino. His research interests include synthesis, verification, simulation, and testing of digital circuits and systems.

Dr. Macii is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.

Massimo Poncino (M'97) received the Dr.Eng. degree in electrical engineering in 1989 and the Ph.D. degree in computer engineering in 1993, both from Politecnico di Torino, Torino, Italy.

From 1993–1994, he was a Visiting Faculty at the University of Colorado at Boulder. Currently he is an Assistant Professor at Politecnico di Torino. His research interests include synthesis, verification, simulation, and testing of digital circuits and systems.

Riccardo Scarsi received the Dr. Eng. degree in electrical engineering from Politecnico di Torino, Torino, Italy, in 1997. Currently, he is working toward his Ph.D. degree in computer engineering at the same institution.

His research interests include several aspects of the development of algorithms, methods, and tools for low-power digital design.