Tunable and Energy Efficient Bus Encoding Techniques

Dinesh C. Suresh, Member, IEEE, Banit Agrawal, Student Member, IEEE, Jun Yang, Member, IEEE, and Walid A. Najjar, Fellow, IEEE

Abstract—Off-Chip buses constitute a significant portion of the total system power in embedded systems. Many research works have focused on reducing power consumption in the off-chip buses. While numerous techniques exist for reducing bus power in address buses, only a handful of techniques have been proposed for off-chip data bus power reduction. In this paper, we propose two novel data bus encoding schemes to reduce power consumption in the data buses. The first scheme called Variable Length Value Encoder (VALVE) is capable of detecting and encoding variable lengths of repeated bit patterns in the data. The second technique called Tunable Bus Encoder (TUBE) encodes repetition in contiguous as well as noncontiguous bit positions of data values. Both schemes require just one external control signal to encode data values. TUBE is the first proposed hardware-based bus encoding scheme capable of detecting and encoding both contiguous and noncontiguous bit patterns of varying widths. Experimental evaluation on a large set of benchmarks shows an energy reduction of 58 percent and 60 percent on average for VALVE and TUBE, respectively. We evaluate the performance penalty incurred due to the codec delay and it is found to be 0.45 percent of the total program execution time. We also quantify our hardware overhead in terms of area, delay, and energy consumption. In 0.18 μm technology, VALVE and TUBE require a modest area of 0.0486 mm² and 0.0521 mm², respectively.

Index Terms—VALVE, TUBE, power, data buses, encoding, bus switching, hardware design, internal capacitances.

1 INTRODUCTION

Embedded systems are special-purpose computer systems that repeatedly perform selected, predefined functions throughout their lifetime. Unlike general-purpose computers, embedded systems are usually battery-powered and operate under tight real-time constraints. Hence, power consumption becomes a very critical design constraint for an embedded system. Over the years, researchers have devised increasingly sophisticated techniques to throttle the power expenditure in major components of embedded systems. Off-chip bus lines are one of the key constituents that contribute significantly to the system’s total power consumption [22]. The off-chip bus energy is given by the following equation:

\[ E_{\text{off-chip}} = (C \times V^2 \times A), \]  \hspace{1cm} (1)

where,

\[ E_{\text{off-chip}} = \text{energy consumed in the off-chip bus}, \]
\[ C = \text{load capacitance of an off-chip bus line}, \]
\[ V = \text{supply voltage, and} \]
\[ A = \text{total number of transitions on all the bus wires during each cycle}. \]

From the equation, it is evident that reducing the supply voltage, bus capacitance, or the total number of signal transitions during each bus cycle, minimizes the off-chip bus energy. Dynamic Voltage Scaling (DVS) [5] is the process of varying the processor’s supply voltage based on the minimal performance level required by active processes. Kaul et al. [14] advocate the use of DVS to minimize on-chip bus energy consumption. While voltage scaling or reducing the equivalent line capacitance decreases the bus energy consumption, reducing signal transitions can further boost the energy savings. Bus encoding schemes are techniques that reduce the total number of transitions on the bus and consequently lower the overall energy consumption. The overhead associated with encoding and decoding of the bus lines is negligible compared to the energy saved during off-chip transmission [30].

Both address and data streams can be encoded effectively to minimize power. Address streams tend to be highly sequential in nature, and hence, address bus encoding schemes proposed in the literature have achieved significant energy savings [29], [4]. Data streams tend to be far less sequential than address streams and therefore, they are more challenging to encode. Data bus encoding schemes can be classified into two categories: non-table-based and table-based encoding schemes. Examples of non-table-based schemes include bus-invert scheme [28], where a combinatorial circuit encodes and decodes data without maintaining its state in a table-like storage. Table-based schemes maintain a finite storage at the encoder and decoder ends in order to capture interesting bit patterns in the data stream [30], [31], [36]. During each cycle, bit patterns from the incoming data stream are compared with the stored bit patterns to detect a
match. In the event of a match in the tables, the bit pattern is transmitted using minimal transition codes. Minimal transition codes are values that have a binary value of “1” in very few bit positions.

A full match in a data bus encoding scheme is when all the \( k \)-bits of an incoming data value matches with the corresponding bits of the stored data value. In a partial match event, only a small number of the \( k \)-bits do not match. Our analysis has shown that the occurrence of partial match events is more frequent than that of full match events. Variable Length Value Encoding (VALVE) [32] and Tunable Bus Encoder (TUBE) [33] are two efficient techniques that effectively encode both full and partial matches to reduce energy expenditure in off-chip bus data buses.

VALVE is capable of encoding both full matches and variable length partial matches in data streams. VALVE and TUBE use a Content Addressable Memory (CAM) table to store a finite set of fixed-width codes. TUBE encodes interesting bit patterns in contiguous as well as noncontiguous bit positions of the data stream. TUBE is the first proposed hardware-based bus encoding scheme capable of detecting both contiguous and noncontiguous bit patterns of varying widths. Noncontiguous bits include silent bits which incur less transitions and hot bits which incur more transitions.

Experimental evaluation on a large set of embedded system benchmarks have shown that VALVE and TUBE provide average energy reductions of 58 percent and 60 percent, respectively, over unencoded data. These encoding schemes achieve significant energy savings with a performance overhead of less than 0.45 percent of the total execution time, assuming a 2-cycle codec delay.

The key contributions of this paper can be summarized as follows. We present an exhaustive analysis of the hardware design, area, and energy consumption of VALVE and TUBE codecs. We evaluate the energy savings for a wide range of VALVE and TUBE configurations normalized over the best configuration. We give a breakdown of the switching activity in each segment of the VALVE and TUBE codecs. We also compare the bus encoding schemes based on the average switching activity during each interval. This helps us understand the behavior of the encoding schemes during different program phases. Finally, we also study the effect of our schemes on the switching activity along each bitline. We find that our encoding schemes tend to distribute the switching activity uniformly across all bus wires. TUBE is most useful in cases where the application can be profiled in advance to identify the hot/silent bit positions. VALVE is ideally suited for shared multiprocessor systems.

The remainder of this paper is organized as follows: We present the related work in Section 2. We describe the bus encoding schemes in Section 3. Section 4 shows our experimental framework used to evaluate the encoding schemes. In Section 5, we discuss the results, hardware analysis, and evaluate the energy consumed by the off-chip bus and study the effect of codecs on the overall performance. In Section 6, we conclude.

2 RELATED WORK

A small number of data and address bus encoding techniques have been proposed which exploit various properties of address and data bus values. Bus-invert coding [28], [27] transfers a data value either in its original form or in its complement form depending on whose hamming distance with the previous bus transmission is smaller. An external complement signal is used to let the destination know that the value sent on the bus is in 1s-complement form, and hence, it should not be interpreted as is. It is a simple method that assumes values are uniformly distributed across the entire value space. The adaptive encoding scheme [18] goes a step further and is capable of online adaptation to the value streams by learning the statistics on the fly. As collecting the accurate statistics for the value streams can be very expensive, the proposed adaptive encoding operates bitwise rather than wordwise. Thus, it looses the correlation among the bits within a single value. Petrov and Orailoglu propose an address bus encoding scheme [21] that utilizes application-specific hot-spot information to encode instruction buses. Aghagiri et al. [2] propose an efficient address bus encoding scheme that reduces the power consumption by 83 percent for address streams.

Data bus encoding schemes such as bus-invert coding [28], adaptive coding [18], power protocol [3], frequent value encoding [36], FV-MSB encoding [31], and FV-MSB-LSB encoding [30] do not assume any prior knowledge of the application. A scheme that operates without prior knowledge of input data is highly desirable because in many application domains, knowing the data in advance might prove to be a very stringent requirement.

Gray code encoding [29] capitalizes on the observation that consecutive values are often sent during successive bus cycles. If gray code was used for representing addresses, sending consecutive values would result in only one transition on the bus. In T0 encoding [4], an external control signal is used to indicate that the current and previous bus values differ by one and there is no transition activity in the bus wires while sending the second value. Although these schemes work well with address streams, they do not work well with data streams because sequential data values are rarely sent on successive bus cycles.

Bus Expander [9] and Dynamic Base Register Caching (DBRC) [12] propose compaction techniques to increase the effective bus width. DBRC uses dynamically allocated base registers to cache the higher order bits of address values. Self-organizing list-based encoding [18] minimizes the transition activity between the codes assigned to the most frequent incoming symbols. Their technique efficiently exploits the sequential nature of address streams and the locality of addresses in multiplexed address bus values. Working Zone Encoding (WZE) [20] keeps track of a few working zones that are favored by the application. Whenever possible, the addresses are expressed as a working zone offset along with an index to the working zone. The encoder and decoder have a few registers to keep track of the working zones and the index selects the current working zone’s value from one of the registers. They also extended the WZE scheme for data buses. The working zone offsets are encoded using one-hot codes. However, this technique requires extra bitlines leading to redundancy in space.

Lv et al. [16] proposed a dictionary-based encoding scheme where in the upper few lines of the bus wires are kept in a high impedance state and the lower bits are encoded. This scheme fails to exploit the occurrences of
entire data values and consequently, the reduction in switching activity is not significantly high. We observed that any scheme that exploits data value locality should be able to exploit entire as well as partial data value locality in order to achieve optimal energy savings. Wen et al. [34] investigate the use of value prediction techniques to reduce transition activity on the data buses. Pappalardo et al. [17] tentatively encode, cluster, and reorder wide data buses according to a fixed coding function and a reordering pattern. The high hardware complexity limits the application of their novel encoding scheme in high-capacity off-chip buses. Shin and Choi [23] have combined Bus-Invert with transition signaling to achieve minimal energy savings on data streams.

Past research has extensively dealt with minimizing the static power consumption in off-chip buses [28], [11], [8]. In deep submicron technologies, besides static power, the power consumed due to crosstalk in adjacent on-chip bus lines becomes very significant. Recent research works have addressed the problem of effectively minimizing the crosstalk between adjacent on-chip bus wires [10], [35], [11]. Kaul et al. investigate the use of dynamic voltage scaling for on-chip buses in order to minimize power. These works have focused on on-chip traffic and are orthogonal to our proposed schemes.

Frequent Value Encoding (FVE) [36], [37] is a data bus encoding scheme that exploits temporal data value locality to encode data. The FV codec has a \( k\)-bit, \( k\)-entry table to store previously seen data values, where \( k\) is the width of the data bus. Before placing a data value on the data bus, the encoder compares the current data value with the values stored in the table. In the event of a hit, the codec sends a one-hot code on the bus. One-hot code denotes a value whose binary representation has a high value in only one of the bit positions. If the data value is not found, the codec adds the value to the table and then sends the value unencoded.

Suresh et al. [30] exploited the locality in the partial data values by sending one-hot codes for the Most Significant Bits (MSB), Least Significant Bits (LSB), or both portions of the data value. The width of these encoded portions was fixed at design time. However, program behavior tends to vary during different phases of the program and consequently, the opportunities to exploit data locality also vary depending on the current phase of program execution. If an encoding scheme can encode variable length prefixes, then it can adapt and optimize its behavior to suit different program phases.

Our work differs from all of the aforementioned works in the following aspect. Our data bus encoding scheme can capture partial data values of varying widths, thereby adapting itself to different phases of the program. Control signals require the availability of a free pin on the chip and are, hence, very expensive to provide. Our technique uses just one external control signal to indicate the presence of encoded values on the data bus. Our encoding schemes are also capable of maintaining a larger history of data values than the maximum possible history length in the FVE scheme, and hence, our schemes have a higher probability of encoding incoming data values in the presence of data locality.

\[
\begin{array}{cccccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

(a) Traditional encoding schemes encode bit patterns of a single, fixed width in incoming data streams. These bit patterns could be of entire or partial width.

\[
\begin{array}{cccccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

(b) Pattern in first 7 bits (MSB)

Pattern in first 9 bits (MSB)

Pattern in all bits

Pattern in first 12 bits (MSB).

VALVE encodes multiple width bit patterns in the incoming data value. For each incoming pattern, VALVE captures patterns of different widths and encodes subsequent occurrences of the pattern using m-hot codes.

\[
\begin{array}{cccccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

(c) Pattern in 6 non-contiguous bits

Pattern in 14 contiguous bits

Pattern in last 7 contiguous bits

TUBE encodes bit patterns in contiguous bit as well as non-contiguous positions of incoming data value.

Fig. 1. Bit pattern detection in (a) conventional encoding scheme, (b) VALVE, and (c) TUBE.

3 Codec Design

In this section, we present two table-based data bus encoding schemes called VALVE and TUBE. Table-based encoding schemes maintain storage to encode finite set of patterns using low transition code. The locality of patterns tends to change with different program phases and hence, a scheme that encode patterns of multiple widths yields higher energy savings. Past works have focused on encoding patterns of a single width.

Fig. 1 shows how VALVE and TUBE encode patterns compared to conventional data bus encoding schemes. Existing data bus encoding schemes encode patterns of a fixed width. VALVE uses segments to encode patterns of different widths. TUBE is capable of identifying bit patterns in noncontiguous bit positions as well as in contiguous bit positions. VALVE codec is just a special case of the TUBE encoder where all the encoded bit patterns are in contiguous bit positions (MSBs).

In the following paragraph, we begin by illustrating our base codec design and proceed to further demonstrate the operation of two off-chip bus encoders that are derived from our base codec.

3.1 Base Codec Design

Our proposed codec design uses a table to encode and decode data values. Each table consists of a finite set of table entries and each table entry, in turn, comprises of a code field and a data field. The code field in a table entry contains an \( m\)-hot code—a value whose binary representation has a high value (logic “1”) only in \( m\) different bit positions, where \( m\) is a small number (usually one or two).

For a table storing codes of width \( k\)-bits, containing up to \( m\)-hot codes, the maximum number of allowable table entries is given by

\[ \text{Maximum number of entries} = 2^k - 1 \]
\[ \sum_{i=1}^{m} C_i^k = \sum_{i=1}^{m} \frac{k!}{(k-i)!i!} \]  

Group of table entries that extract the same set of bit positions constitute a segment. During the first occurrence of a full or partial bit pattern, the encoder stores the bit pattern in its segments and transmits the data value over the data bus without encoding. Upon receiving the unencoded value, the decoder stores the bit pattern from the data value in its segments. Thus at the end of every bus cycle, the encoder and decoder table contents are exact replicas of each other. For subsequent occurrences of the repeating bit pattern, the codec sends a \( m \)-hot code for the repeating portion. When using codes of width \( w \), all bit patterns of width \( w \) or greater can be encoded by the codec. For each unique bit-pattern-width, the codec requires a new segment. While sending encoded values on the bus, an external control signal is used to let the destination know that the first \( k \)-bits of the data bus value correspond to a code.

In order to facilitate easy decoding of bit positions, the code is of a fixed width and is always sent in a predetermined set of bus wires. For example, irrespective of the width of the portion being encoded, the codec can always choose to send a code of width 12-bits in the upper 12 bus wires. In order to ensure integrity of the encoded data values, the width of the segment code should always be smaller than that of the segment data field.

The codec associates a three-bit time stamp with the table entries and evicts stale entries using Least Recently Used (LRU) replacement policy. As used in other techniques [36], a pair of correlator and decorrelator is added to the two ends of the buses. They are inverse functions of each other and their purpose is to reduce the correlation between successive values.

In the following sections, we explore two variants of our base codec design, VALVE and TUBE codecs, and illustrate their design in greater detail.

### 3.2 VALVE Description

VALVE exploits full and partial data match by encoding repeating bit patterns of varying width. The repeating bit patterns are stored in Content Addressable Memory (CAM) tables maintained at the encoder and the decoder. VALVE table comprises of a set of segments of varying width, where segment denotes a group of table entries that store data values of same width. In order to extract bit pattern of a particular width, VALVE associates a segment mask with each segment.

While using a code of width \( k \)-bits, the first \( k \) locations of the table contain one-hot codes of width \( k \). These locations are also used to store full-width data values (by setting the mask value to all 1s). The remaining \( k^*(k-1)/2 \) locations use two-hot codes. Hence, for codes of width 16-bits, the VALVE table can have up to 16 locations with one-hot codes and 120 locations (16*16 - 1)/2 with two-hot codes.

During execution, the VALVE codec dynamically maps the incoming data value’s bit pattern to one of the available codes stored in the table. A three-bit time stamp is associated with the VALVE table entries, and stale entries are evicted from the table using Least Recently Used (LRU) replacement policy.

To store and retrieve data values, we use content addressable memories (CAM) of different widths. For each segment, we use a code-CAM to store the code and a separate data-CAM to store the corresponding data values. Addition and removal of new entries to the data-CAM is accomplished through a codec controller. Selection logic is used to generate encode signal and to arbitrate hits in various segments of the codec. During a hit in multiple segments, the selection logic gives priority to the segment hit with the largest bit width. The selection logic also generates 3-bit multiplexer control signals to select the appropriate 32-bit data bus value from the previous stage. In the following paragraphs, we describe the encoder and decoder algorithms.

#### 3.2.1 VALVE Encoder Algorithm

The algorithm for the VALVE encoder is described in Algorithm 1. The VALVE encoder, shown in Fig. 2, can encode bit patterns of width 32-bits, 24-bits, and 16-bits. For every data value, masks are applied to extract the 32, 24, and 16 bit patterns. These bit patterns are then looked up in the appropriate segments of the VALVE table. In the event of a hit in multiple segments, the segment selector picks the hit from a segment with the largest segment mask. The \( w \)-bit code from the hit location forms the upper \( w \)-bits of the current data bus value. The complement of the hit segment’s mask is logically AND-ed with the data value in order to get the low-order bits of the encoded data bus value. The code is OR-ed with the low-order data bits and the final 32-bit value is sent to the data bus after correlating with the previous data value.

**Algorithm 1. VALVE Encoding Algorithm**

1: encode signal ← 0
2: for each data value do
3: if hit in valve table then
4: encode signal ← 1
5: current data bus value ← code[hit.index] OR
6: (complement(mask) AND data value)
7: else
8: encode signal ← 0
9: current data bus value ← data value
10: insert data value in valve table
11: end if
12: end for
3.2.2 VALVE Decoder Algorithm

Algorithm 2 shows the VALVE decoder algorithm. Fig. 3 shows a VALVE decoder with three segments of widths 32-bits, 24-bits, and 16-bits, respectively. The upper w-bits of the data bus value contain the code. Upon receiving an encoded data bus value, the decoder searches for the upper w-bits in its VALVE tables. In the event of a hit, the VALVE table returns a data value associated with the search code. This returned data value constitutes the high-ordered bits of the decoded data value. The complement of the hit segment’s mask is applied to the current data bus value in order to obtain the lower order bits of the decoded data value. Then, higher order and lower order bits are OR-ed together to get the final 32-bit data value.

Algorithm 2. VALVE Decoding Algorithm
1: for each data bus value do
2:   if encode signal = 1 then
3:     code = code portion [data bus value]
4:     data value ← data[code] OR
5:     (complement(mask) AND data bus value)
6:   else
7:     data value ← data bus value
8:     insert data value in valve table
9:   end if
10: end for

3.3 VALVE—An Example

Fig. 4 illustrates the operation of a VALVE codec that uses three segments: a 32-bit segment, 24-bit segment, and a 20-bit segment. Let the contents of the VALVE encoder segments after t cycles be as shown in Fig. 4a. For the next four consecutive transactions, Fig. 4a shows the contents of the VALVE table at the encoder end. It also illustrates the results of segment search and shows how the data value is encoded. Fig. 4b shows the VALVE table contents at the decoder end and illustrates the decoding operation.

Let us consider the operations at the encoder end for the next four consecutive transactions. During the first cycle, the value 0xFFF80145 is given as input to the encoder. The encoder searches its segment in order to find out if the current value was encountered in the recent past. The encoder finds a hit in its 32-bit segment. The corresponding 16-bit code (0 × 1000) is the output of the encoder. The encoder asserts the control signal to let the destination know that a code is being sent on the upper 16 bus wires. The decoder decodes the upper 16 bus wires (0 × 1000) and obtains a 32-bit value. During the second transaction (0 × 000F8752), the search value misses in the 32-bit and 24-bit segment but is found in the 20-bit segment of the encoder table. The 16-bit code corresponding to the hit location (0 × 0011) is logically OR-ed with the lower 12 bits of the data value (complement of the hit segment mask) in order to obtain the current encoded data bus value. The encode signal is raised to inform the destination that a code is sent in the upper 16 bus wires. Upon decoding the value, the decoder observes the mask in the hit location and concludes that the 16-bit code corresponds to a 20-bit data and hence, it interprets the lower 12 bus wires as unencoded portion. Likewise, a 16-bit code is sent for a 24-bit hit in transaction 3. During transaction 4, the encoder finds a miss in all of its segments, and hence, it sends the data unencoded after lowering the encode signal. When the encode signal is low, the decoder interprets the data “as is”.

![Fig. 3. VALVE decoder block diagram. The figure shows three segments: 32-bit segment, 24-bit MSB segment, and 16-bit MSB segment. The selection logic takes all the combined values based on the code read from data bus and send it to processor/memory controller.](image)

![Fig. 4. This illustrates the encoding and decoding operations using VALVE tables. X indicates a don’t care value. Segment Value field corresponds to the value returned by the VALVE table during a segment search operation. (a) Table content and state at the encoder. (b) Table content and state at the decoder.](image)
At the end of each bus cycle, the encoder and decoder tables are exact replicas of each other. In order to ensure the integrity of the encoding/decoding operation, the code space of different segments are disjoint.

### 3.3.1 Design Parameters

In order to find out the optimal VALVE configuration, we tune the following design parameters:

- **Code width**: The code width determines the minimum width of the bit pattern that can be encoded using the VALVE codec. While using codes of width \(k\), only bit patterns of width greater than or equal to \(k\) can be encoded. If we try to encode bit patterns of width less than \(k\), then the lower portion of the unencoded data corrupts the code portion, thereby resulting in incorrect decoding of the encoded data value. The code width also determines the maximum number of table entries that can be supported by the VALVE table. We varied the code width from 10-bits to 22-bits and observed the codec performance in each case.

- **Number of segments**: The number of segments in the VALVE design determines the number of different bit patterns that can be encoded using a given code width. Lesser number of segments implies that lesser number of bit-pattern-widths can be encoded. Having large number of segments increases the redundancy amongst the stored table entries, thereby hampering the effectiveness of the table.

### 3.4 TUBE Description

The TUBE codec encodes data by storing selective bit positions of data values in its segments. Each segment is capable of encoding repeating data portions of a fixed width. We define silent bits and hot bits as follows. For a given class of applications, silent bits are those bit positions that incur the least amount of transitions. Hot bits are bit positions that incur the most number of transitions. TUBE also has noncontiguous segments that encode data by identifying the silent bits and hot bits in data values. By profiling the applications in advance, TUBE’s noncontiguous segment is loaded with a mask for the silent bits and hot bits.

During execution, the TUBE codec dynamically maps the incoming data values bit pattern to one of the available codes stored in the table. TUBE uses two kinds of segments in order to encode data: contiguous segment and noncontiguous segment. Contiguous segments exploit repetition in contiguous bit positions across different data values, while noncontiguous segments exploit repetition in noncontiguous bit positions. The contiguous segments store the MSB, LSB, and entire data values (32-bit). Each contiguous segment encodes data value of a fixed width. The noncontiguous segments are tuned to the characteristics of the application. The noncontiguous segments can capture two different kinds of bits: hot bits and silent bits. Hot bits tend to have a lot of transitions and hence, encoding them would yield significant energy savings. Silent bits are bit positions that incur fewer transitions than the rest. However, fewer bit patterns tend to repeat in silent bit positions and they can, hence, be encoded more often than hot bits.

While sending encoded values on the bus, an external control signal is used to let the destination know that a fixed set of bit positions in the data bus value is a code. Using just one external control signal, TUBE encodes varying widths of MSB, LSB portions of data. It also encodes the silent bits and hot bits of data values. We found the number of simultaneous MSB and LSB hits to be significantly higher than the hits in LSB table only (miss in MSB). Hence, we choose to encode an LSB hit only if it happens to be an MSBLSB hit. Likewise, a hot segment hit is encoded only if there is also a hit in the silent segment. TUBE uses codes of two different widths and these codes are sent in the upper and lower bus wires of the off-chip data bus. The code for the MSB, entire data portion, and silent bit segment hits are sent on the upper portion of the data bus, while the code for the LSB and hot bits are sent on the lower portion of the data bus. Hence, during an MSBLSB hit or a silent/hot hit, a code is sent in both the upper and lower bus wires. For the remainder of this paper, we will refer to the segments that send code in the upper bus wires as upper segments. Likewise, lower segments are segments whose code is sent along the lower set of bus wires. In order to ensure that the destination can decode data without any ambiguity, the upper segments and the lower segments do not have any overlapping bus wires for code transmission. To ensure the integrity of the codec’s operation, all upper segments use the same code width and all the codes are disjoint. Likewise, all the lower segments use the same code width and codes are disjoint.

The biggest design challenge that we managed to solve while designing the TUBE codec was to accomplish the coding operation using just one external control signal. TUBE uses an external control signal to indicate the presence of encoded values on the bus. However, we need an additional control signal to let the destination know whether a code is being sent in the upper segment or if it is being sent in both segments. External control signals require the availability of a free pin on the chip and are, hence, very expensive to provide. Hence, we choose to use one of the bus wires as a control signal. For the rest of this paper, we will refer to this internal bus wire as an internal control signal.

We set the internal control signal to high whenever an MSBLSB hit is sent on the off-chip data bus. If the encode signal is high and the internal control signal is low, then it corresponds to an MSB hit only. The silent bit segment and the hot bit segment capture mutually exclusive bit positions. Hence, there is a likelihood of simultaneous hit in both segments. Since the code spaces of all these segments are mutually exclusive, we choose to combine an MSBLSB hit with a silent/hot simultaneous hit. Hence, during an MSBLSB hit or a silent/hot hit, we will end up searching the MSB, LSB, silent and hot bit segments.

By using two different code widths instead of one, TUBE sends one-hot code for more number of table hits. This also minimizes the required number of segment searches. If we used just one code width, then the decoder should lookup the incoming code in all of the segments. Since two different code widths are used, a code in the lower portion would initiate a decoder lookup only in the LSB and the hot bit segments. This saves the overall decoder energy.

### 3.4.1 TUBE Encoder Algorithm

Algorithm 3 shows the encoder algorithm for the TUBE encoder. Since the encoder encodes values of varying widths, the code width is kept constant. For every incoming value, the encoder searches its segments to see if the data value or its bit positions were encountered before. In the
event of a hit, the encoder sends the corresponding code along the upper bus wires and raises the external control signal. During a hit in multiple segments, the code from the segment with largest number of bit positions is chosen by the selection logic. The encoder also searches its lower segments to see if the bit positions were encountered in the recent past. During a match, the encoder raises the internal control signal to let the destination know that a code is being sent in both the upper and lower set of bus wires. When the encoder does not find a match in any of its segments, the encoder lowers the external control signal and sends the value unencoded.

Algorithm 3. TUBE Encoding Algorithm
1: encode signal ← 1
2: for each data value do
3:   search for data value in all segments
4:   if hit in full data segment then
5:     current data bus value ← code for hit entry in the segment
6:   else
7:     if hit in upper segments then
8:       current data bus value ← upper code
9:     if hit in lower segments then
10:    current data bus value ← upper code OR lower code
11:   internal control signal ← 1
12:   else
13:    current data bus value ← upper code OR lower(data value)
14:   internal control signal ← 0
15: end if
16: else
17:   encode signal ← 0
18:   current data bus value ← data value
19: insert data value in tube segments
20: end if
21: end if
22: end for

Fig. 5 shows a tube encoder that uses five segments to encode data. The codec has two MSB segments, one LSB, one silent segment, and one hot segment. The MSB and the silent segments send code along the upper bus wires during a hit, while the LSB and the hot segments send code along the lower set of bus wires in the event of a segment hit. The segment selector picks the code to be sent on the bus wires during a hit in multiple segments. MSB and silent segment hits are encoded by themselves. However, LSB or hot segment hits are encoded only during a MSB or a silent hit, respectively.

3.4.2 TUBE Decoder Algorithm

Fig. 6 shows a five-segment TUBE decoder. Algorithm 4 shows the decoder algorithm for the TUBE decoder. When the external control signal is low, the data is interpreted as is by the decoder. When the external control signal is high, the decoder searches the upper segments. The decoder searches the lower segments when the value of the internal control signal is set to 1. When the decoder encounters a hit, the data at the hit location is used to obtain the decoded data value.

Algorithm 4. TUBE Decoding Algorithm
1: for each data bus value do
2:   if encode signal = 1 then
3:     upper code = upper_code_portion [data bus value]
4:     lower code = lower_code_portion [data bus value]
5:     data value ← data[upper code]
6:     if internal control signal = 1 then
7:       data value ← data value OR data[lower code]
8:     else
9:       data value ← data value OR lower(data bus value)
10: end if
11: else
12:   data value ← data bus value
13:   insert data value in TUBE table
14: end if
15: end for

3.4.3 TUBE Hardware Design

In this section, we present the 2-stage pipelined codec design for TUBE. We use content addressable memories (CAMs) to store and search the data bus values of different widths. We store the data bus value of each segment in a data-CAM and store the corresponding code in a code-CAM. A codec
controller controls the addition and deletion of new entries to the data-CAM. There is a selection logic block to arbitrate among hits in various segments and to give priority for the segment hit with largest bit width. It generates 3-bit multiplexer control signals to select the appropriate 32-bit data bus value from the previous stage. The selection logic block is also responsible for generating the encode signal. Our hardware design is symmetric in nature to handle both encoding and decoding operations.

### 3.4.4 TUBE—An Example

Fig. 7 illustrates the operation of the TUBE codec with an example. The TUBE codec shown in the example uses three upper segments: an MSB segment of width 32-bits (full data), MSB segment of width 20-bits, and a silent segment. It also uses two lower segments: a LSB segment of width 12-bits and a hot segment. We will illustrate the encoding and decoding operation for four consecutive transactions assuming the current state of the segments and the silent/hot bit positions to be as shown in Fig. 7a. Fig. 7b shows the contents of the TUBE segments and the control signal status for the next four consecutive transactions. The decoder state for the same set of transactions are shown in Fig. 7d.

During the first transaction, the codec encounters a hit in the upper segment only. Hence, the internal control signal is set to 0 and the external control signal is set to 1. A code is sent along the upper bus wires only. The decoder checks the internal control signal and concludes that the encoded portion is present only in the upper bus wires. During the second transaction, TUBE finds an MSBLSB hit and hence, it sends a code along both the upper and lower bus wires. The internal control signal is set to 1 so that the destination can resolve the code without any ambiguity. As shown in Fig. 7c, while searching the silent segment, the values in the silent bit positions are concatenated in order to form the search string. LSB hits are always encoded with MSBs or else they are sent unencoded. Likewise, hot segment hits are always encoded with silent segment hits.

### 4 Experimental Evaluation

We modify the sim-outorder simulator in the SimpleScalar toolset [6] to incorporate our bus encoding techniques. In order to evaluate the effectiveness of our encoding schemes, we used a wide range of benchmarks that are representative of both embedded and desktop applications. Our test programs consisted of benchmarks from the MediaBench [15], NetBench [19], benchmark suites, and three applications from SPECINT2000 [26].

We modeled two different architectures: an embedded system-like architecture with a small L1 cache and a desktop-like architecture with both L1 and L2 cache sizes. For the embedded system-like architecture, we evaluated data caches of the following sizes: 4 KB, 8 KB, 16 KB, and...
32 KB. For the desktop-like architecture, we fixed the L1 and L2 cache sizes at 64 KB and 512 KB, respectively. For each of these cache configurations, we fixed the block size of the instruction and data caches at 32 bytes. Both instruction and data caches have on-chip and off-chip latencies of 1 cycle and 100 cycles, respectively. The off-chip data bus is 32-bits wide. The off-chip data trace consisted of both instruction and data values.

### 4.1 Bus Power Model

We use a bus power model similar to the one discussed by [7]. In general, estimating the energy used in the off-chip interconnects is difficult. We can approximate the capacitance for the bus using the formula:

$$C_{bus} = C_{metal} \times \text{No. of Bus lines}. \quad (3)$$

In this expression, $C_{metal}$ is the capacitance of the metal interconnect for each bus line. Using the numbers given in [7], it is estimated to be 20 pF. $C_{bus}$ gives the effective capacitive load to be driven during a bus transaction. We calculate the bus total energy per cycle using the following formula:

$$E_{total} = E_{enc} + \left( \frac{T_r \cdot C_L \cdot V^2}{\# \text{of cycles}} \right) + E_{dec}, \quad (4)$$

where,

- $T_r =$ total number of transitions in the off-chip bus,
- $C_L =$ load capacitance of the off-chip bus line,
- $V =$ supply voltage,
- $E_{enc} =$ energy consumed by the encoder per cycle, and
- $E_{dec} =$ energy consumed by the decoder per cycle.

There are low-power memories offered by NEC where the memory chip operates between 2.5 and 3.6 volts and I/O buffer component operates at 3.3 Volts [1]. Hence, we picked a value of 3.3 Volts for supply voltage and 20 pF for bus capacitance.

### 4.2 VALVE Configurations

To find the best VALVE configuration for a comparative study with other schemes, we evaluate a set of VALVE configurations. For each code width, we vary the number of segments with widely separated width from two to six and observe the codec performance in each case. Table 1 shows the normalized energy reduction for different configurations normalized against the best VALVE configuration. For lower code widths, even though the codec registers a large number of hits, the reduction in switching due to each hit is low. As we increase the code width beyond 18 bits, the number of bits in the table decreases. As shown in the table, energy consumption is lowest when the code width is 16-bits and when four VALVE segments of width 32, 24, 20, and 16-bits are used.

### 4.3 Silent and Hot Bits

We executed the embedded and desktop applications in SimpleScalar simulator to find the switching activity of each bit in the off-chip data bus. The switching is then averaged over all applications belonging to a single benchmark suite. We refer to the first 16 bit positions with highest switching activity as hot bits, and the remaining bits are classified as silent bit positions. The silent and hot bits for each benchmark suite are shown in Fig. 8. In the figure, silent bits are shown in dark gray while hot bits are shown in light gray. For all the benchmark suites, bit position 32 is the most silent (least switching activity) and bit position 1 is the least silent (highest switching activity).

### 4.4 TUBE Configurations

Once the silent and hot bits were determined, we evaluate different TUBE configurations to select the best configuration. We vary the number of segments, size of the upper code width, and the lower code width. The segment entries are equally divided among the segments. The normalized energy reduction for a set of TUBE configurations is shown in Table 2. The results are normalized with respect to the best configuration. The best TUBE configuration is found to have one Full Value segment (FV segment) to store 32-bit values, one MSB segment of width 18-bits, two silent segments of width 24 and 18-bits, one LSB segment of width 14-bits, and one hot segment of width 14-bits. For the rest of this paper, we pick this tube configuration for analysis.

### 5 Results

In this section, we present the delay and area analysis for VALVE and TUBE codecs. We then highlight the switching and energy reduction obtained using VALVE and TUBE codec while executing embedded system and desktop applications. We also evaluate the impact of cache size on the energy savings and study the impact of our codec on the overall performance of the system.

#### 5.1 Delay and Area Analysis

In this section, we analyze area overheads of our hardware design and quantify the delay and energy consumption. We use Cacti-3.0 [24] to model most of hardware components and some previously published results for some hardware components. We use 0.18 µm CMOS technology and we scale various parameters accordingly. Our hardware design can be classified into two main components such as Content Addressable Memories (CAMs), pipelined registers, $128 \times 32$ MUX (selects the segment values based on control signals from the selection logic), and selection logic.

We take the results of CAM from [13] and we find that CAM requires 45.5 $fJ/bit/search$ in 0.35 µm technology. We estimate the area of a CAM cell from the results in [13].

<table>
<thead>
<tr>
<th>Code width</th>
<th>Number of segments</th>
<th>Segment width</th>
<th>Maximum Size</th>
<th>Normalized % Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>10,16,32</td>
<td>45</td>
<td>70.19</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>11,17,21,32</td>
<td>55</td>
<td>78.79</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>12,20,32</td>
<td>66</td>
<td>78.23</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>13,17,21,32</td>
<td>78</td>
<td>87.70</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>14,20,32</td>
<td>91</td>
<td>89.81</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>15,19,24,32</td>
<td>105</td>
<td>91.13</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16,20,24,32</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>17</td>
<td>5</td>
<td>18,20,26,30,32</td>
<td>120</td>
<td>92.51</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>18,26,32</td>
<td>120</td>
<td>89.10</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>19,23,26,32</td>
<td>120</td>
<td>91.42</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>20,24,28,32</td>
<td>120</td>
<td>90.88</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>21,23,25,27,32</td>
<td>120</td>
<td>90.90</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>22,25,28,32</td>
<td>120</td>
<td>89.80</td>
</tr>
</tbody>
</table>
and it is found to be 11 $\mu m^2$ in 0.18 $\mu m$ technology. The delay for $32 \times 32$ CAM is estimated to be 5.6 ns in 0.18 $\mu m$ technology. The energy, delay, and size requirements of different components of the VALVE and TUBE codec are shown in Table 3. We calculate the clock frequency based on the CAM delay. VALVE and TUBE codecs had clock frequencies of 220 MHz and 175 MHz, respectively. The clock frequency of VALVE and TUBE hardware design is determined by the CAM pipeline stage. The observed clock frequencies are sufficient for most of the embedded SRAM/SDRAM memories. It can be increased further by selecting delay-optimized CAMs.

### 5.1.1 Codec versus Increasing the Cache Size

Results from CACTI [25] show that the area requirement of a $192 \times 32$ CAM is comparable to that of a 1 KB, 8 byte block, 2-way set associative cache. Like most of the modern day embedded platforms, our simulated architecture also has L1 cache size of at least 4 KB (all the way up to 32 KB). Since cache sizes must be an integral power of two, the extra real estate used by our chip cannot be used to augment the cache size on such platforms.

### 5.2 Switching Reduction

Fig. 9 shows the percentage reduction in switching activity for FVE, VALVE, and TUBE encoding schemes. As evident from the figure VALVE and TUBE achieve switching reduction of 19.5 percent and 21.2 percent, respectively, more than FVE, on an average. Later, we take into account the energy of codec for all the schemes and present the energy savings compared to no-encoding case.

### 5.2.1 Average Switching Activity

For mpeg2dec benchmark in the MediaBench benchmark suite, Fig. 10 shows the average reduction in switching activity during each cycle for Bus-Invert, FVE, VALVE, and TUBE encoding schemes. The performance of Bus-Invert scheme deteriorates during the second half of the execution time. However, VALVE and TUBE schemes consistently achieve reduction in switching activity over FVE. The efficiency of our schemes in reducing the switching activity uniformly across different phases of the program justifies the use of encoding varying widths of contiguous and noncontiguous bit patterns.

### 5.2.2 Individual Bus Wire Switching

Fig. 12 shows the activity number for each bit position in the NetBench benchmark suite. Activity number ranks the bit wires based on transition activity. An activity number of 31 indicates the hottest bit position while an activity number of 0 denotes the most silent bit position. From the figure, it is evident that the highest transition activity takes place in bit positions 0-3, bitline 21, and in bit wires 25-27. The lowest switching activity is along the bit wires 7-12, 20-24, and 28-31. For route benchmark in the NetBench benchmark suite, Fig. 11 shows the ratio of switching activity on a bus line while using an encoding scheme to the switching activity on the bus line without any encoding schemes. VALVE and TUBE significantly reduce the transition activity on all the hot bitlines described in Fig. 12. This is achieved at the expense of increasing the switching activity in silent bit positions. However, since the contribution of silent bits to total switching activity is insignificant, increased switching activity in silent bit positions does not adversely impact the overall energy savings.

### TABLE 2

Percentage Switching Reduction of Different Configurations Normalized over the Best TUBE Configuration

<table>
<thead>
<tr>
<th>U-code width</th>
<th>L-code width</th>
<th>Description of segments</th>
<th>Max. Size</th>
<th>Normalized % reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>10</td>
<td>FV, 2 MSB(24,16), 1 silent(20), 1 LSB(16), 1 hot(12)</td>
<td>167</td>
<td>67.45</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>FV, 1 MSB(18), 2 silent(24,16), 1 LSB(16), 1 hot(16)</td>
<td>167</td>
<td>72.13</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>FV, 2 MSB(24,18), 1 silent(20), 1 LSB(14), 1 hot(12)</td>
<td>218</td>
<td>86.23</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>FV, 2 MSB(22,16), 1 silent(18), 1 LSB(16), 1 hot(14)</td>
<td>218</td>
<td>86.54</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>FV, 1 MSB(16), 2 silent(22,16), 1 LSB(16), 1 hot(16)</td>
<td>218</td>
<td>87.43</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>FV, 1 MSB(18), 2 silent(24,18), 1 LSB(14), 1 hot(14)</td>
<td>218</td>
<td>91.10</td>
</tr>
<tr>
<td>18</td>
<td>13</td>
<td>FV, 2 MSB(24,18), 1 silent(20), 1 LSB(14), 1 hot(12)</td>
<td>263</td>
<td>95.32</td>
</tr>
<tr>
<td>18</td>
<td>13</td>
<td>FV, 1 MSB(16), 2 silent(22,16), 1 LSB(16), 1 hot(16)</td>
<td>263</td>
<td>92.51</td>
</tr>
<tr>
<td>18</td>
<td>13</td>
<td>FV, 1 MSB(18), 2 silent(24,18), 1 LSB(14), 1 hot(14)</td>
<td>263</td>
<td>100.0</td>
</tr>
<tr>
<td>18</td>
<td>13</td>
<td>FV, 2 MSB(22,16), 1 silent(20), 1 LSB(16), 1 hot(12)</td>
<td>263</td>
<td>91.42</td>
</tr>
</tbody>
</table>
5.2.3 Contributions of Various Segments

Fig. 13 shows the breakdown of segment hits and their corresponding reduction in switching for epic benchmark from the MediaBench Suite. The hits shown in the figure correspond to the cases in which a segment is chosen by the selection logic for encoding. In our design, we give highest priority to FV hits, and hence, the hits in silent-hot segments are fewer than that of the hit in FV segment. The FV segment uses only one-hot codes while the silent-hot and MSB-LSB segments use both one-hot and two-hot codes. Hence, the hits in FV segment correspond to a higher percentage reduction in switching activity.

Fig. 14 shows the hits in a VALVE codec that uses three segments. As shown in the figure, the hit in entire data segment (32-bit width) constitutes only 23 percent of the total segment hits. 16-bit segment has nearly 40 percent of all the hits in VALVE segments. However, while encoding a 16-bit segment hit, the lower 16 bits of the data are sent unencoded and during a 24-bit segment hit, the lower 8 bits of the data are sent unencoded. Hence, in spite of the lesser number of hits, segments that encode wider portion of the data value are highly effective in reducing the transition activity on the data bus. The high hit rate in segments of different widths justifies the use of variable width segments to encode data.

5.3 Energy Savings

We compute the energy savings using the formula given in (4). Assuming a capacitance value of 20 pF and a supply voltage of 3.3 V, the energy spent in a single bus cycle where the bus undergoes 10 transitions is of the order of 2000 pJ. The energy of the encoder and decoder as computed in the earlier sections is of the order of 100 pJ. Hence, the observed energy savings closely mirrors the reduction in switching activity of the off-chip data bus. Fig. 15 shows the percentage reduction in bus energy for FVE, VALVE, and TUBE encoding schemes. VALVE and TUBE consistently achieve energy savings over FVE for all applications. For some applications like parser, the observed energy gain over FVE is as high as 25 percent. VALVE and TUBE achieve energy savings of 58 percent and 60 percent, respectively, over unencoded data, on an average.

5.4 VALVE and TUBE—Comparison

VALVE encoder can be considered as a special case of TUBE encoder where all the segments are contiguous and only bit patterns in MSB positions are encoded. Our best TUBE configuration yields 60 percent energy savings using nearly 260 entries, while our best VALVE configuration uses 120 table entries to provide energy savings of 58 percent. In our experiments with TUBE codec, we profiled the applications and fixed the silent and hot bits for each benchmark suite. However, the energy savings
reported in TUBE could be further extended by tuning the silent and hot bits for each application. The advantage of VALVE is that it can be easily extended to multiprocessor systems and has lesser hardware overhead than TUBE. However, the extra hardware overhead in TUBE is insignificant when compared to the energy savings due to reduction in switching activity.

### 5.5 Effect of Varying Cache Parameters

The codec deals with off-chip values at the granularity of cache blocks. Hence, the cache size and cache block size influence the codec’s performance. We observed the energy savings of our schemes for two different block sizes: 32 and 64 bytes. While the embedded system applications did not show significant change in energy savings with increase in block size, the SPECINT applications showed 0.19 percent increase in energy savings. We varied the L1 cache size and studied its effect on the overall energy savings. We find that with increasing cache size, there is no consistent increase or decrease in energy savings. This implies that the temporal locality in contiguous/noncontiguous positions of data values within a block does not change significantly for the cache sizes under consideration. Fig. 16 shows the percentage reduction in energy for different cache configurations while running embedded benchmarks and SPECINT applications. On an average, VALVE and TUBE reduce the off-chip data bus energy by 58 percent and 60 percent, respectively.

### 5.6 Impact on Performance

The encoding and decoding operations add extra latency in the processor-memory transaction and hence, there is a slight decrease in the overall performance. Using the proposed pipelined architecture, the codec can be easily implemented with a delay of 2 clock cycles, which amounts to a single cycle delay at both the encoder and decoder ends. We take the codec delay to be 1 cycle, 2 cycles, and 4 cycles to evaluate the performance penalty. Table 4 shows the performance penalty for different values of codec delays. We instrumented the SimpleScalar simulator to measure the performance penalty for a set of benchmarks and we assumed an off-chip memory latency of 100 cycles. On an average, VALVE and TUBE incur 0.29 percent, 0.76 percent, and 1.47 percent performance penalty with codec delay of 2 cycles for MediaBench, NetBench, and SPECINT applications, respectively. However, we achieve...
58 percent and 60 percent energy savings on an average with little performance overhead, i.e., less than 1 percent among the chosen set of benchmarks.

6 Conclusion

Off-Chip buses constitute a significant portion of the total system power in embedded systems. We proposed and evaluated two novel data bus encoding schemes to reduce power consumption in the data buses. Program behavior tends to vary during different phases and consequently, the opportunities to exploit data locality also vary depending on the current phase of program execution. VALVE technique, which can encode variable length prefixes, thereby adapting its behavior to suit different program phases. VALVE provides an average energy reduction of 58 percent over unencoded data. Our extensive analysis on data traces has shown that the switching activity in different bit wires is highly nonuniform. The switching activity in hot bit positions is significantly higher than that of the silent bit positions. To better exploit the repetition in silent and hot bit positions, we proposed TUBE. This encoding technique achieved an average energy reduction of 60 percent over unencoded data. TUBE is the first proposed hardware-based bus encoding scheme capable of detecting contiguous and noncontiguous bit patterns of varying widths.

Using just one control signal, VALVE and TUBE provide significant energy savings at the expense of a very small performance overhead. We evaluated the performance penalty incurred due to the codec delay and it was found to be 0.45 percent of the total program execution time. While using 0.18 μm process technology, VALVE and TUBE require minimal area of 0.0486 mm² and 0.0521 mm², respectively. Our analysis has demonstrated that the proposed codec designs are both efficient and practical.

Bus value traces in shared multiprocessor systems often contain repetition in contiguous higher order bits and are highly conducive to VALVE-like encoding schemes. TUBE provides best energy savings when the application can be profiled in advance to identify the silent and hot bit positions. Our encoding techniques could be applied with equal effectiveness to other domains such as lossless image/video compression.

References
