Tunable Bus Encoder for Off-Chip Data Buses

Abstract

Off-Chip buses constitute a significant portion of the total system power in embedded systems. Past research has focused on encoding contiguous bit positions in data values to reduce the transition activity in the off-chip data buses. In this paper, we propose ***TU****nable* ***B****us* ***E****ncoding (****TUBE****) scheme to reduce the power consumption in the data buses, which exploits repetition in contiguous as well as non-contiguous bit positions in order to encode data values. We also solve the problem of keeping just one control signal for our codec design.*

We compare our results with some of the already existing best schemes such as Frequent Value encoding (FVE) and FV-MSB-LSB encoding schemes. We find that our scheme achieves an improvement of 21% on an average and up to 28% on some benchmarks over the FVE scheme and up to 84% over unencoded data. In comparison to FV-MSB-LSB encoding scheme, our scheme improves the energy savings by 10% on an average and up to 21% for some media applications at the expense of minimal 0.45% performance overhead. We present a hardware design of our codec and provide a detailed analysis of the hardware overhead in terms of area, delay and energy consumption. We find that our codec can be easily implemented in an on-chip memory controller with small area requirement of 0.0521 mm^2.

1. Introduction

Off-chip buses are associated with high capacitance values and hence, while transmitting bus values, they consume a significant portion of the total embedded system power. The energy consumed by the off-chip bus is in direct proportion to the total number of transitions that take place in the bus. Bus encoding schemes are techniques that reduce the bus power consumption by encoding and decoding data prior and subsequent to transmission.

Bus encoding schemes exploit properties of bus values in order to encode data. It has been shown that a significant portion of the data values consists of a few values that have a tendency to repeat every few cycles. By storing recently encountered values, subsequent occurrences of the same value can be encoded. For a wide range of commonly used embedded system applications, we measure the number of repeating occurrences of entire data values and their portions. We find that even while executing a single program, different bit prefixes tend to repeat during different portions of the program. Hence, we present an encoder design capable of encoding repeating values of different width to achieve better switching reduction. In particular, we present a data bus encoding scheme that capitalizes on the following three properties of data streams in order to encode data: 1) partial/entire data value locality in temporal domain 2) hot bits and 3) silent bits. We will explain each of these terms in the subsequent paragraphs.

For most of the applications, the transition activity in the off-chip bus wires is non-uniform: some of the bus wires tend to toggle more than the rest. We refer to the set of bits that toggle the least as *silent* bits. Since silent bits remain steady for longer periods of time, the probability of silent bits having the same values in different data values is high. The complementary bit positions of the silent bits incur a lot of transitions and we refer to these bit positions as *hot* bits. Figure 1 shows the silent and hot bit positions in a portion of the data stream. The silent bit positions in the figure are shown in bold. The silent bits incur fewer transitions than hot bits. Hot bits and silent bits are complementary to each other. By encoding both silent and hot bits, the codec can efficiently encode data.

In this paper, we present a TUnable Bus Encoder (TUBE) that can encode contiguous (*prefixes and full data value*) and non-contiguous bit positions (*silent and hot bits*) of data values. 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. TUBE also has non-contiguous segments that encode data by identifying the *silent bits* and *hot bits* in data values.

We explore the design space and evaluate the performance of different TUBE codec configurations. We run a set of experiments by executing MediaBench[8], NetBench[13] and SPECINT2000 [16] benchmarks and we find that our scheme achieves an average energy reduction of 66% over unencoded data, which is a 21% improvement over FVE [20]. For some media applications, we achieve as much as 84% reduction in energy over unencoded data. We analyze the impact of the codec on the overall performance of the system and we find that TUBE provides significant energy savings with an overhead of as little as 0.45% of the total execution time. We also present a detailed analysis of our hardware design and quantify its overhead.

The rest of this paper is organized as follows. We describe our encoding scheme and its design issues in section 2. We illustrate our experimental framework in section 3. In section 4, we present the results obtained using our encoding scheme. We discuss the related work in section 5 and in section 6, we conclude.

2. TUBE Design

In this section, we present TUBE design and explain various design parameters, codec algorithm, and hardware design. TUBE uses tables at the encoder and decoder ends. The table consists of various *segments,* where each *segment* extracts a predefined set of bit positions from the data values. Each *segment* consists of a finite set of segment entries and each segment entry, in turn, comprises of a code field and a data field. The code field in the segment entry contains an *M-hot code*. M-hot code is defined as 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). The data field of the segment entry stores selected bit positions of the incoming data value. For a segment storing codes of width k-bits, containing up to M-hot codes, the maximum number of allowable segment entries is given by

$$\sum_{i=1}^{M} C_i^K = \sum_{i=1}^{M} \frac{K!}{(K-i)!*i!}$$

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's table contents

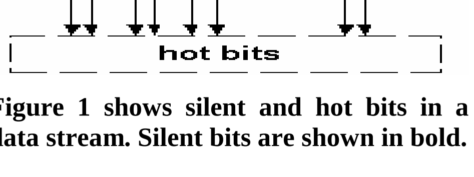

Figure 1 shows silent and hot bits in a data stream. Silent bits are shown in bold.

are exact replicas of each other For subsequent occurrences of the repeating bit pattern, the codec sends an *M-hot code* for the repeating portion. 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 corresponds to a code. As used in other techniques [20], 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.

2.1 TUBE Segments

TUBE uses two kinds of segments in order to encode data- contiguous segment and non-contiguous segment. Contiguous segments exploit repetition in contiguous bit positions across different data values while non-contiguous segments exploit repetition in non-contiguous bit positions. Contiguous segments include full-value segment, most significant bits (MSB) segments, and least significant bits (LSB) segments, whereas non-contiguous segments include hot bits segment and silent bit segments. By profiling the applications in advance, TUBE's non-contiguous segment is loaded with a mask for the silent bits and hot bits. However, masks can be changed at runtime in our hardware design using programmable CAM and complete re-initialization of the codec. 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.

We find the number of simultaneous MSB and least significant bits LSB hits to be significantly higher than the hits in LSB segment 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 a hit in the silent segment also. 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. Likewise, all the lower segments use the same code-width.

2.2 Table Codes

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. When using codes of width 'w', all bit patterns of width w or greater can be encoded by the codec. If the code-width were to be greater than the number of bits being encoded, enough bus wires will not be available to transmit the unencoded bit positions. Hence, in order to ensure integrity of the encoded data values, a segment's code-width should always be smaller than the width of the segment's data field.

2.3 Maintaining Table Entries

The codec associates a three-bit timestamp with the table entries and evicts stale entries using Least Recently Used (LRU) replacement policy. We picked LRU as it can be easily implemented with minimal hardware. During execution, the TUBE codec dynamically maps the incoming data value's bit-pattern to one of the available codes stored in the table.

2.4 Control Signals

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 MSB hit with an LSB hit and a silent segment hit with a hot segment hit. Hence, during a 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, 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.

2.5 Algorithms

Figure 2 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 where 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. This selected code is designated as the upper code. The encoder also searches its lower segments to see if the bit positions were encountered in the recent past.

```
TUBE Encoder Algorithm

For each incoming value
Do
   encode_signal = 1
   lookup value in segments
   if hit in full data segment then
         send code for the hit location
   else
         if hit in other upper segments then
               if hit in lower segments then
                     current_bus_value = upper_code OR lower_code
                     internal_control_signal = 1
                     /* MSBLSB or Silent-hot hit */
               else
                     current_bus_value = upper_code OR lower_data
                     internal_control_signal = 0
               end if
         else
               send data unencoded
               encode_signal = 0
         end if
   end if
Done
```

Figure 2 shows the TUBE encoder algorithm.

```
TUBE Decoder Algorithm

For each incoming value
Do
        If encode_signal == 0 then
                interpret data as-is
        else
                search upper segments
                decoded_data = upper_hit(data) OR lower (bus_value)
                if internal control signal == 1 then
                        decoded_data = upper_hit(data) OR lower_hit(data)
                end if
        end if
Done
```

Figure 3 shows the TUBE Decoder Algorithm.

During a hit in the lower segment, the hit code (lower_code) forms the lower order bits of the encoded bus value. 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. During a lower segment miss, the lower order bits of the incoming value constitute the lower portion of the encoded bus value. 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.

Figure 3 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.

2.6 Design Parameters

Code width: Code width of the TUBE table determines the maximum number entries stored in the table. While using one-hot and two-hot codes of width k, the maximum number of table entries should be less than ((k-1) + (k*(k-1))/2). The code width also determines the width of the data portion to be encoded. Using codes of width k, data portions (m) of width k or higher can be encoded. If not, the unencoded portion (32 – m) cannot be sent using the remaining wires (32-k). For our experiments, we considered code widths of 14, 16 and 18 bits for the upper segment. We used code-widths of 10, 12 and 14 bits for the lower segments. We restricted the size of the upper and lower segments 120 and 45 entries respectively.

Segment size: For a given upper and lower code width, we simulated eight different codec configurations by varying the segment size and the width of the data portion captured in the segment. For each of these configurations, we had 5 upper segments and 3 lower segments. The upper segment consisted of a maximum of 120 entries distributed between 32-bit, 24-bit and 16-bit contiguous segments and two silent segments of width 16 and 18 bits respectively. Likewise, the lower segment consisted of one 12-bit LSB segment, a 14-bit hot segment and 16-bit hot segment.. Our best configuration had 32, 20, 28,20 and 20 entry tables in the upper segments and 20, 10 and 15 entries in the lower segment. The energy reduction achieved by this configuration will be presented in the following sections. We will present the result for this configuration only due to space constraints.

Choice of mask bits: We adopted a two-stage tuning approach. During an initial run of the benchmarks, we loaded the codec with application-specific masks based on the results from our profiling tool. Then we identified the silent and hot bits that are common across a given class of applications. Thus for each benchmark suite, we loaded the codec with silent and hot bits that were best suited for all the applications in that benchmark suite. Based on the profiled results, we fixed the masks for a class of applications. Figure 4 shows the silent bit position for different SPECINT applications. As shown in the graph, for the first, sixth, 30th and many other silent bit positions are the same across most of the SPECINT applications. We found that a similar correlation existed in applications from the MediaBench suite[8]. Likewise, applications from the NetBench suite[13] also had common set of silent bits. Hence, we fixed the silent/hot positions for each class of applications. We performed experiments to determine the changes in silent and hot bit positions due to different data set and we find that top 20 silent bits almost remain the same with only one or two different silent bit positions.

2.7 Hardware Design

In this section, we present the 2-stage pipelined tube codec design. We use content addressable memories (CAMs) to store and search the contiguous and non-contiguous data bus values of different widths. We use two CAMs for a segment to store the data field in one CAM and the corresponding code field in the other CAM. We refer to the first CAM as data-CAM and second CAM as 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 give priority to segment hit with larger bit width. It generates three bit multiplexer control signals to select the appropriate 32-bit data bus value from the previous stage. The selection logic block is also responsible to generate the encode signal and select appropriate value for the internal control signal. Our hardware design is symmetric in nature to handle both encoding and decoding operations..

The encoder operation in our hardware design is shown in Figure 5. In this figure, we have shown hardware design for a 4-segment tube codec, where three are three segments of contiguous bits (widths are 32,

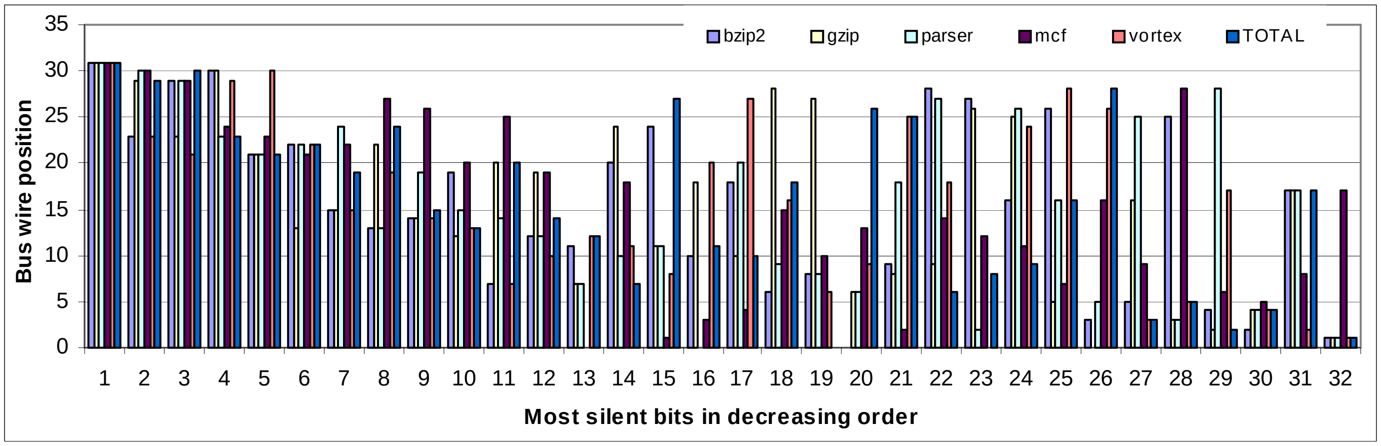

Figure 4 shows silent and hot bits for SPECINT2000 class of applications, where 31st bit is found to be most silent and 0th bit is found to be the hottest bit.

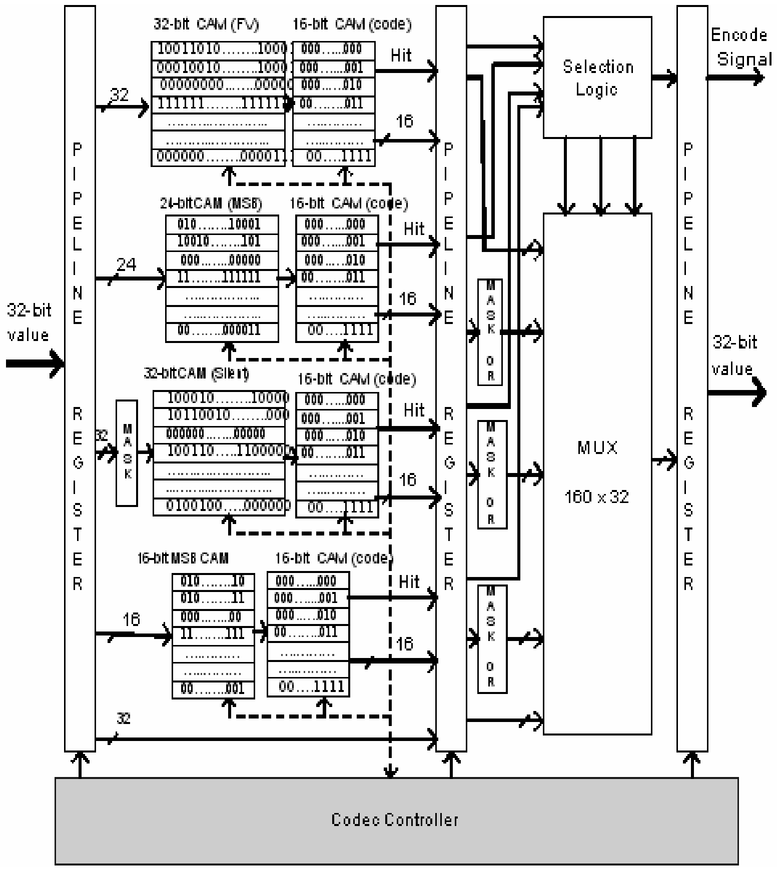

Figure 5 shows the operation of a TUBE encoder.

24, 16) and one silent segment of 24 bit width. The encoder operation takes two cycles to encode the incoming data bus value. But in most of the cases, we transfer multiple words due to a larger cache line and hence, there is a large scope of aggressively pipelining the encoding operation. In the first stage, data value is fed to the *32*-bit pipeline register and then fed to the data CAMs after applying the appropriate mask for each segment. On a match in data-CAM, the corresponding matchline is used to drive the wordline of the code-CAM. All the codes corresponding to segment hits, is read into the *second* pipeline register. All the hits are also fed to the second pipeline register to make appropriate encoding decision. Selection logic block in the second stage utilizes these hit signals from various segments to generate the 3-bit multiplexer control signals. The (160 x32) multiplexer selects 32-bits from the appropriate segment based on the 3-bit control signals. For example, on a 16-bit hit, the MASK-OR component generates the 16-bit lower values after applying 16-bit inverse mask and then it is ORed with *code* to get the final 32-bit value. On a silent segment hit, the silent mask is applied with the original data bus value to the get the hot bits. Then, these bits and silent bits are shifted and ORed together to get the 32-bit data value and fed to the (160 x 32) multiplexer. Finally, multiplexer selects the appropriate 32-bit data bus value and puts it in the data bus after correlation. On a valid encoding, select logic block asserts the encoding signal. We decode the incoming data bus value in a symmetric fashion using the same pipelined hardware design.

3. Experimental Setup

We modified the *sim-outorder* simulator in the Simplescalar toolset [4] 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 16 benchmarks from the MediaBench [8], NetBench [13] benchmark suites and four applications from SPECINT2000 [16].

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 – 1KB, 2KB, 4KB and 8KB. For the desktop-like architecture, we fixed the L1 and L2 cache sizes at 64KB and 512KB respectively. We used the desktop like architecture while simulating the SPECINT applications and simulated the remaining benchmarks using an embedded system like architecture. 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. Results and Analysis

In this section, we present our experimental results and analyze the effectiveness of our scheme in reducing the off-chip energy consumption. We begin by describing our bus power model and then estimate the energy consumed by the TUBE codec. We use this result to evaluate the energy reduction achieved using our encoding scheme. We also measure the area requirements of our codec and illustrate that ours is a feasible design in terms of energy, delay and area overheads.

4.1 Bus Power Model

We use a bus power model similar to the one discussed by Catthoor et. al [5]. In general estimating the energy used in the off-chip interconnects is difficult. We can approximate the capacitance for the bus using the formula:

$$C_{bus} = C_{metal} * \textit{Number of Bus lines}$$

In this expression C_{metal} is the capacitance of the metal interconnect for each bus line. Using the numbers given in [5], it is estimated to be 20pF. C_{bus} gives the effective capacitive load to be driven during a bus transaction. We calculated the total bus energy per cycle using the following formula:

$$E_{total} = E_{encoder} + \frac{T_r * C_L * V^2}{\#\ \text{of cycles}} + E_{decoder}$$

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*

4.2 Codec Delay and Energy Consumption

In this subsection, we analyze area overheads of our hardware design and quantify the delay and energy consumption. We use Cacti-3.0 [14] to model most of hardware components and some previously published results for some hardware components. We use 0.18 um CMOS technology and we scale various parameters accordingly. Our hardware design can be classified into some main components such as Content Addressable Memories (CAMs), pipelined registers, 160 x 32 MUX and selection logic. Now, we describe how we calculate the various performance metrics.

We take the results of CAM from this paper [10] and we find that CAM requires 45.5fJ/bit/search in 0.35 um technology. We estimate the area of a CAM cell from the results in [10] and it is found to be 11 um^2 in 0.18um technology. The delay for 32 x 32 CAM is estimated to be 5.6 ns in 0.18um technology. We estimate the maximum energy consumption, maximum delay of area of CAMs by adding the results of all the CAM segments in 0.18um technology and it is found to be 95.68 pJ, 5 ns, and 0.045 mm^2 respectively. We model the 32-bit pipelined register using Cacti and we find that the maximum delay, energy

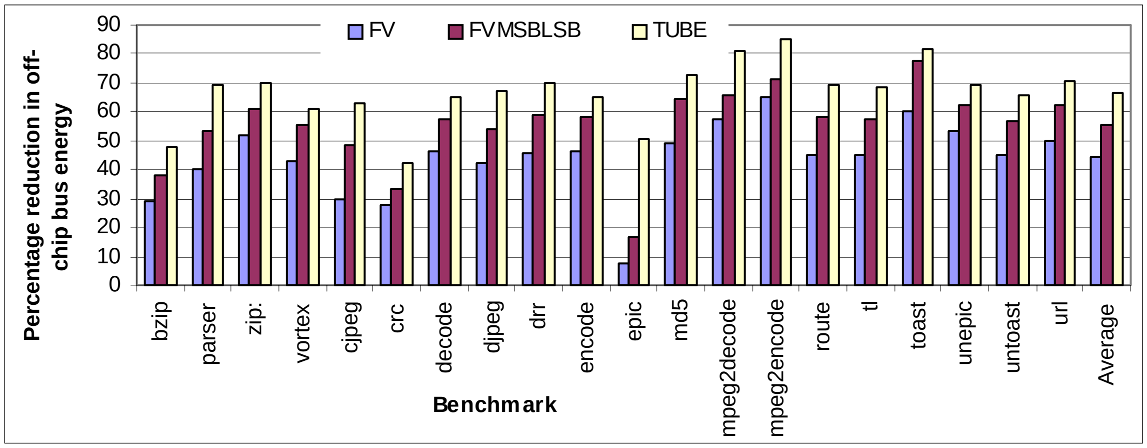

Figure 6 shows the percentage reduction in off-chip energy for different data bus encoding schemes. Cache size for SPECINT : L1=64KB, L2=512KB; Cache size for other applications : L1=2KB, no L2. On an average, TUBE provides an improvement of 21% over FVE and 10% over FV-MSB-LSB.

consumption, and area requirement is 0.173 ns, 0.443 pJ and 0.00164 mm^2 in 0.18um technology respectively. We model the 160 x 32 MUX using cacti and found that it requires 0.002120 mm^2 and maximum energy consumption and delay is found to be 0.78 ns and 4.3 pJ respectively. We use the *cadence* tools layout results for primitive gates and estimate the area, maximum energy consumption and delay of selection logic block to be 0.000121 mm^2, 0.354 pJ and 0.16 ns respectively.

The clock frequency of hardware design is determined by the CAM pipeline stage. We calculate the clock frequency based on the CAM delay and it is found to be ~175 MHz. It is sufficient for most of the embedded SRAM memories. It can be increased further by selecting delay-optimized CAMs. The energy consumption for the codec is found to be 103.12 pJ and the total area requirement of our codec design is found to be 0.0521 mm^2.

4.3 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 contemporary VLSI technology and 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. We instrumented the Simplescalar simulator to measure the performance penalty for a set of benchmarks and we assumed an off-chip memory latency of 70 cycles. On an average, TUBE incurs 0.29%, and 0.76%, performance penalty with codec delay of 2 cycles for MediaBench, and NetBench respectively.

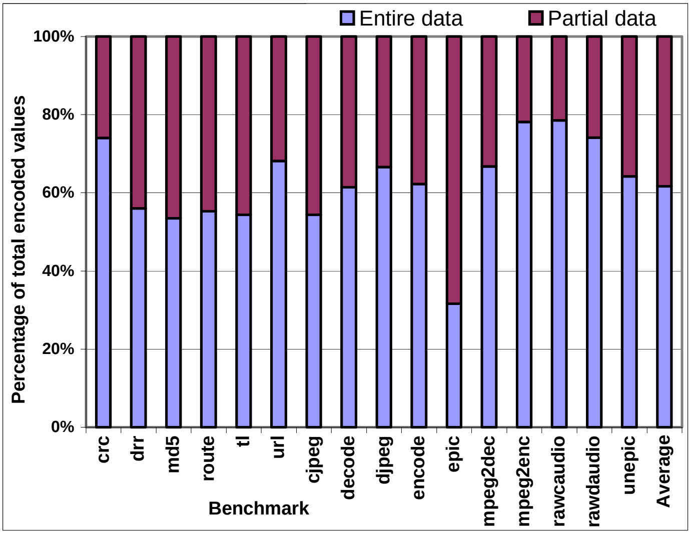

Figure 7 shows the percentage of Entire data Vs partial data amongst encoded bus values. On an average, TUBE manages to encode all the bit positions in nearly 61% of the encoded values.

4.4 Off-chip Bus Energy Savings

We evaluate our proposed scheme for the configuration described in section 3. Figure 6 shows the percentage reduction in energy while using a L1 cache of size 2KB. We evaluate the energy savings by taking care of energy consumption of hardware codec. We simulated FVE [20] and FV-MSB-LSB [17] schemes using the same set of tools as used in this paper. To evaluate energy consumption for FVE and FV-MSB-LSB we used parameter values mentioned in [20]] and [17] respectively.

For applications like *cjpeg*, TUBE outperforms FVE by 28%. *Epic* application has a lot of hits in the partial segments (MSB and silent) and hence yields significant energy improvement. We observe a similar energy saving trend in spec2000 applications, where *parser* provides an extra energy saving of 18% over FV-MSB-LSB scheme. However, for *crc* application, energy saving is not significantly high, which demonstrates less data value locality in *crc* program. On an average, TUBE provides an energy improvement of 21% of FVE and 10% over FV-MSB-LSB, and 64% over unencoded data.

Figure 7 shows the percentage of cases during which the entire and partial data are encoded upon a table hit. As shown in the graph, on nearly 62% of the encoded cases, TUBE codec encodes either a 32-bit contiguous value or it encodes a MSBLSB hit or a silent hot hit. The remaining 39% of the cases consists of MSB or silent segment hits alone. Since the codec manages to encode entire data on 62% of all the encoded values, it yields significant energy savings.

4.4.1 Effect of various cache configurations

We execute our embedded applications for different cache sizes to study the impact of cache sizes on the overall energy savings. Table 1 shows the percentage reduction in energy for different cache configurations. We find that there is a slight decrease in the percentage of energy saving with the increasing cache size. The energy saving for 2 KB cache size over FVE is 21%, whereas it reduces to 20.4% for 8 KB cache. This finding can be attributed to the fact that there is a decrease in the off-chip data value locality with increasing cache sizes.

5. Related Work

Previously proposed bus encoding schemes have targeted both address and data streams. Address Bus encoding schemes have exploited

Table 1. Percentage average energy reduction for different cache configurations

Cache size	FVE	FV-MSB-LSB	TUBE
1KB	44.94	55.52	66.71
2KB	42.88	53.78	63.89
4KB	40.87	51.06	61.84
8KB	40.62	50.93	61.03

the sequential and strided nature of address streams in order to encode bus values. Data streams are often less regular and are hence, more difficult to encode. Frequent Value Encoding (FVE) [20] is a data bus encoding scheme capable of encoding entire data values. FV-MSB-LSB [17] stores the entire data value, Most Significant Bit (MSB) portions and Least significant Bit (LSB) portions of values in separate tables. The codec sends a one-hot code for subsequent occurrences of the data value or its MSB or LSB portions. While encoding MSB or LSB portions alone, the remaining portion of the data is sent unencoded.

Kaul et al.[11] propose a low latency spatial encoder circuits based on bus-invert coding. Basu et al. [1] proposed a value cache at both ends of the communication channel. During a hit, the index to the cache entry is sent instead of the whole word. Bus Expander [6] and Dynamic Base Register Caching (DBRC) [7] 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 [12] 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) [14] 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. However, these schemes fail to exploit locality in non-contiguous bit positions.

LV et al. [9] 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. Wen et. al [19] investigate the use of value prediction techniques to reduce transition activity on the data buses. Sector-based coding [1] is an address bus encoding technique that partitions the address space into a number of sectors. Each incoming value is then encoded with respect to the sector head.

TUBE differs from all of the aforementioned works in the following aspect. TUBE captures chunks of varying widths from data values. These chunks can consist of bits from contiguous and non-contiguous bit positions of the data value. For subsequent occurrences of these bit positions, TUBE sends a code instead of data. Control signals require the availability of a free pin on the chip and are hence, very expensive to provide. TUBE uses just one external control signal to indicate the presence of encoded values on the data bus.

6. Conclusion

We proposed and evaluated TUBE, a novel bus-encoding scheme for reducing the power consumption in off-chip data buses. TUBE is capable of exploiting repetition in contiguous as well as non-contiguous bit positions in order to encode data values. TUBE requires just one external control signal to encode data.

We performed simulations for 20 workloads covering the SPECINT2000 [16] and commonly used embedded system applications like the Mediabench [8] and Netbench [13]. On an average, TUBE architecture reduces the off-chip bus energy by 64%, which is a 21% improvement over FVE [20]. TUBE achieves as much as 84% reduction in energy over unencoded data for some embedded system applications.

We also estimated the area and performance overhead of the TUBE codec. We found that TUBE codec can be implemented with a minimal area of 0.0521 mm^2 and has a performance overhead as low as 0.45% of the total execution time. Hence, TUBE design is feasible in terms of energy, performance and delay overhead. TUBE can also be effectively extended to other domains like on-chip data value encoding, multiprocessor data trace encoding and trace compression.

References

[1] Y. Aghaghiri, M. Pedram, and F. Fallah. ”Reducing Transitions on Memory Buses Using Sector-based Encoding Technique”, In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pp 190–195, Monterey, California, August 2002.

[2] K. Basu, A. Choudhary, J. Pisharath and M. Kandemir, “Power protocol: Reducing Power Dissipation on Off-Chip Data Buses”, *35th Annl IEEE/ACM Symp. on Micro architecture (MICRO-35*) , Istanbul, Turkey, Nov. 2002.

[3] L. Benini, G. De Micheli, E. Macci, D. Scuito and C. Silvano, “Asymptotic zero-transition activity encoding for address buses in low-power microprocessor-bases systems”, *Great Lakes VLSI Symp.*, pp 77-82, Urbana IL, Mar. 1997.

[4] D. Burger and T. Austin, “The SimpleScalar Tool Set, Version 2.0, Technical Report”, University of Wisconsin-Madison, Computer Science Department, 1997

[5] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele and A. Vandecappelle, “ Exploration of Memory Organization for Embedded Multimedia System Design”, *Kluwer Academic Publishers”*, 1998.

[6] D. Citron and L. Rudolph, “Creating a Wider Bus using Caching Techniques”, *Proceedings of the first International Symposium on High Performance Computer Architecture*, pp 90-99, January 1995.

[7] M. Farrens and A. Park, “ Dynamic Base Register Caching: A technique for Reducing Address Bus width”, In Proceedings of 18th International Symposium on Computer Architecture (ISCA), pp 128-137, Toronto, Canada, May 1991.

[8] C. Lee, M. Potkonjak and W. Mangione-Smith, “MediaBench: a tool for evaluating and synthesizing multimedia and communications systems”, *Intl. Symp. on Microarchitecture*, pages 330-335, 1997.

[9] T. Lv, J. Henkel, H. Lekatsas and W. Wolf, “An Adaptive Dictionary Encoding Scheme for SOC Data Buses”, *Design Automation and Test in Europe,* Paris, France, Mar. 2002.

[10] I.Y.L. Hsiao, D.H. Wang, and C.W. Jen, “Power modeling and low-power design of content addressable memories”. *In Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 4, pages 926-929, 2001.

[11] H. Kaul, D. Sylvester, M. Anders and R. Krishnamurthy, “ Spatial Encoding Circuit Techniques for Peak Pwer Reduction of On-Chip High-performance Buses”, *ACM/IEEE Intl. Symp. on Low Power Electronic Design*”, Pages 194-199, Newport Beach, California, 2004.

[12] M. Mamidipaka, D. Hirschberg and N. Dutt, “Low Power Address Bus Encoding Using Self-organizing Lists”, *Intl. Symp. on Low Power Electronics and Design*, 2001, pp 188-193.

[13] G. Memik, W. H. Mangione Smith and W. Hu, “NetBench : A Benchmarking Suite for Network Processors”, *Intl. Conf. on Computer Aided Design (ICCAD)*, pp 39-42,San Jose, California, Nov. 2001

[14] E. Musoll, T. Lang and J. Cortadella, “Working Zone Encoding for Reducing the Energy in Microprocessor Address Buses”, *IEEE Trans. on VLSI Systems*, pp 568-572, Volume 6, Dec. 1998.

[15] P.Shivakumar and N. P. Jouppi, “Cacti 3.0: An Integrated Cache Timing, Power and Area Model," *Western Research Lab (WRL) Research Report 2001.*

[16] SPECINT2000, http://www.specbenh.org/cpu2000

[17] D. C. Suresh, B. Agrawal, J. Yang, W. Najjar and L. Bhuyan, “Power Efficient Encoding Techniques for Off-Chip Data Buses”, In the Proc. of *Compilers and Architecture and Synthesis for Embedded Systems (CASES),* San Jose, CA, Oct. 2003

[18] M. R Stan and W. P Burleson, “Bus-invert Coding for low power I/O”, *IEEE Trans. on Very Large Scale Integration (VLSI) systems*, pages 49-58, Volume 3, 1995

[19] V. Wen, M. Whitney, Y. Patel and J. D. Kubiatowicz, “Exploiting Prediction to Reduce Power on Buses”, *In Proc. of the 10th Intl. Symp. on High Performance Computer Architecture*, pages 2-13, Madrid, Spain, Feb 2004.

[20] J. Yang and R. Gupta, “FV-Encoding for Low Power Data I/O”, *ACM/IEEE Intl. Symp. on Low Power Electronic Design*”, Pages 84-87, 2001.