FV Encoding for Low-Power Data I/O*

Jun Yang Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

ABSTRACT

The power consumed by I/O pins of a CPU is significant due to high capacitances associated with the pins. While highly effective techniques for reducing address bus switching exist [1], similarly effective techniques for data bus have not been developed. We have discovered a characteristic of values transmitted over the data bus according to which a small number of distinct values, called **frequent values**, account for 58-68% of transmissions over the external data bus. To exploit this characteristic we have developed a method for dynamic identification of frequent values and their use in encoding data values using **FV** (frequent value) **encoding** scheme. Our experiments show that FV encoding of 32 frequent values yields an average reduction of 42.7% (with on-chip data cache) and 67.63% (without on-chip data cache) in data bus switching activity for SPEC95 benchmarks.

1. INTRODUCTION

In CMOS circuits most power is dissipated as dynamic power for charging and discharging of internal node capacitances. Thus, researchers have investigated techniques for minimizing the number of transitions inside the circuits. The capacitances at I/O pins are orders of magnitude higher than internal capacitances. Thus, the power dissipated by an IC at these I/O pins is even greater than that dissipated at internal capacitances. Therefore techniques for minimizing switching at external address and data buses, even at the expense of a slight increase in switching at internal capacitances, are quite useful [1, 2, 3].

Many of the encoding schemes, such as the bus-invert coding [2], are general and can be applied to both address and data buses. General techniques can only provide modest reductions in switching activity. To obtain greater reductions we must identify special characteristics for information transmitted over address and data buses. Using such a specialized approach that exploits memory reference locality the method described in [1] reduces the address bus activity by as much as 66%.

*Supported by DARPA PAC/C Award. F29601-00-1-0183 and NSF grants CCR-0105355, CCR-0096122, EIA-9806525, and EIA-0080123 to the University of Arizona.

The goal of this work was to develop a technique for data buses that is as effective as the technique in [1] for address buses. Till now an effective specialized approach for a CPU's external data buses has been illusive. This is because no suitable characteristic for values transmitted over a data bus has been found. Unlike memory references that exhibit locality, the data values do not exhibit similar locality. In fact the values transmitted over data bus may vary widely across the range of representable values. We have recently discovered a characteristic of data values sent over a data bus that can be employed to develop an effective encoding scheme. Recently we have shown that a small number of distinct values, frequent values, occupy majority of the data locations in memory for a wide range of application programs [4]. Thus, these values are transmitted very frequently over the data bus.

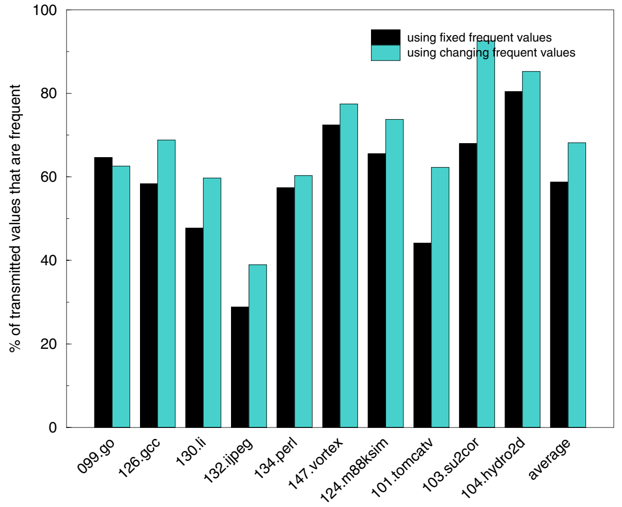

Figure 1: Data bus traffic due to frequent values.

In Figure 1 we show the percentage of total data bus traffic that is the result of transferring frequent values for SPEC95 benchmarks. The two sets of data correspond to situations where the set of frequent values, 32 in all, was kept *fixed* and allowed to *change* over the lifetime of the program. As we can see, on an average 58% (68%) of the traffic is created by fixed (changing) frequent values.

The remainder of the paper is organized as follows. In section 2 we present the FV encoding scheme in detail and also describe how the frequent values are identified by our scheme. In section 3 we describe the result of experiments including experimental comparison of our method with existing techniques.

2. FREQUENT VALUE ENCODING

Now we present the design of our encoder and decoder used to reduce the switching activity on the data bus. Our overall approach is as follows. The frequent values are transmitted over the bus in encoded form while the infrequent values are transmitted in their original unencoded form. The set of frequent values are kept in a table implemented as a CAM at both the encoder and decoder. This table is searched and if the value to be transmitted is found in it, then the value is regarded as a frequent value which is then transmitted in encoded form. In order to ensure that the decoder can determine whether the transmitted value is in encoded form or not, additional *control* signal must be sent from the encoder to the decoder in some situations. As we describe later in this section, our method for maintaining frequent values is such that the contents of the frequent value tables at both the encoder and decoder are identical to each other.

In the remainder of the section we first describe our encoding scheme in detail. Then we describe our method for finding frequent values, that is, we describe how the contents of the frequent value tables are maintained.

Encoding-decoding scheme. Our method for encoding frequent values has the flavor of one-hot encoding with one important difference. Our encoding scheme overcomes the major drawback of one-hot encoding in that it does not need exponential number of wires (2^n, where n is the number of bits representing the value) to transfer data. Instead, it uses the same number of wires as the data bus width (we assume that this number is 32) and maintains the desired property of low switching activity.

We are able to achieve the above goal as follows. The "hot" wire generated from the encoder is not used to represent the true decimal value being transfered but rather it indicates in which entry of the frequent value table in the encoder or decoder that frequent value can be found. In other words if the i^{th} entry is found to hold the same value as the one to be transmitted, the i^{th} output wire is set as 1 and all the remaining wires are set as 0. Thus a **code** with only one wire at high voltage is formed and sent over the data bus, completing the coding process.

When the decoder receives the code from the bus, it reads out the value from the i^{th} entry indicated by the code. We will show later how our method for maintaining the contents of the tables at the encoder and decoder ensures that the contents of the two tables are identical and thus the value is correctly decoded.

The infrequent values are transmitted in unencoded form. If a value to be transmitted is an infrequent value it cannot be found in the encoder CAM. Thus, the encoder does not attempt to generate a code. Instead, it simply passes the original value onto the data bus. When the decoder receives the value and finds more than one hot wires in it, it concludes that the transmitted value is not encoded.

It is possible that a value being transmitted in unencoded form contains a single high bit and all of its remaining bits are zeros. We ensure that the decoder does not incorrectly decode this value by sending a single bit control signal from the encoder telling the decoder to skip decoding. Our experimental results also include the switching overhead from sending the control signal.

With continuously occurring frequent values, FV encoding will send out codes one after another with "hot" bits at different position, resulting in 2 switching transitions for each pair of frequent values. We can further reduce the switching down to 1 using an approach used in [1, 3]. By XORing current code with the previous value sent over the data bus as shown in Figure 2 such a reduction is achieved. Moreover now even if a frequent value follows an infrequent value, reduction is switching can be substantial since frequent values contain mostly zero bits.

$$Send_n = Send_{n-1} \oplus Code_n$$
$$DCode_n = Send_n \oplus Send_{n-1}$$

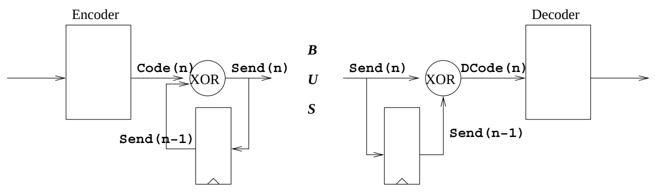

Figure 2: FV encoding setup.

Identifying frequent values. Having described our encoding scheme, let's now discuss how we fill and update the encoder and decoder tables with data values. There are two ways of doing this that we consider in this paper:

1. A *fixed set* of values known in advance to initialize both encoder and decoder can be used. The set of values can be obtained through ranking of the frequency of values that appeared in a previous run of the program.
2. A *changing set* of frequent values can be maintained as the program runs. Thus, the contents of the frequent value tables adapt to changes in the frequent values for different parts of execution.

Using fixed values to preset the encoder and decoder has the advantage that the coders do not have to change the contents of the table dynamically thus reducing the runtime overhead. However, it requires that values be known before hand and different program needs different values. The second method, on the other hand, does not need a priori information of data values and does not distinguish among different programs. With these features, we pay the price of identifying the frequent values on the fly.

In our previous work [4] we have always used a small number of fixed frequent values for encoding. This was because the application we considered in the past was that of storing compressed (encoded) data in data cache to improve on-chip cache performance. Frequent values cannot be varied in this case because every time the encoding changes, the contents of the cache must be flushed or reencoded. This operation is too costly and therefore not practical. However, the application that we are considering in this paper does not have similar constraints – we can easily exploit changing frequent values by changing the contents of the frequent value tables.

The changing frequent value scheme has the potential for giving better performance. This is because a value with high frequency in one span of time may not occur as frequently in another span of time during a program run. We conducted an experiment to determine whether the need for an adaptive scheme exists. In this experiment we divided the execution of a program into smaller time intervals and for each of these intervals we found the best 32 frequent values. We considered the commonality between this nearly ideal set of values and the values used in our changing value scheme (described later in this section) as well as the fixed set of frequent values. The plot in Figure 3 shows that the overlap between the changing set and the near ideal set is much greater (around 20 or higher for most of the time) than the overlap between the fixed set and the near ideal set (slightly less than 10). This plot is for the `su2cor` benchmark and represents a time period which corresponds to 25% of the program run over which three million values were transmitted over the data bus. We favor the dynamic encoding scheme but will also include experimental results for fixed value scheme in the experimental section. Next we will describe how we find the frequent values dynamically.

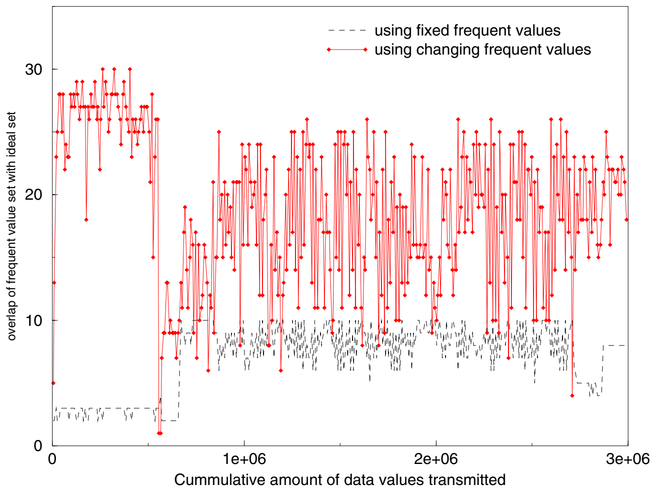

Figure 3: Changing set vs fixed set.

LRU policy. We use the LRU replacement policy for filling and updating both encoder and the decoder frequent value tables. To gain time ordering information, we use a *reference bit* and a n-bit timestamp for each value recorded in the coder. The reference bit is set when the value appears at the input. At regular intervals, the reference bit is shifted right into the high-order bit position of the n-bit timestamp causing all bits in the timestamp also to be shifted right and the lowest-order bit in the timestamp being discarded. This operation is performed for all entries in the two tables and at the same time all the reference bits are reset. Thus, the timestamp keeps the history of value occurrences for the last n time periods. The timestamp of 000 means this value did not appear during the last three time intervals, timestamp 100 means it was just seen in the last interval, and the timestamp 000 with reference bit set means it is encountered in the current time slot. When an entry is required and a value is to be evicted, the entry that is selected is the one with the smallest timestamp and clear reference bit. The new value is put in with a fresh reference bit and timestamp (all 0's) in this selected entry.

Keeping encoder/decoder consistent. It is extremely important to keep the sender side encoder and the receiver side decoder consistent all the time. We use the **same** replacement policy for both to assure they contain the same values. In more detail, if there are multiple entries that have the same timestamp, both the encoder and the decoder follow the same rule for picking up a victim, say the first victim they encounter during searching. By doing so, we guarantee both sides contain not only the same values but also the same indices for every value. The grounds for this to be true is that they have the same timestamp value and reference bit. This can be easily achieved by using the same time interval for updating the timestamp and the reference bit.

3. EXPERIMENTAL EVALUATION

We conducted experiments by executing a subset of the `SPEC95` benchmarks and measuring the switching activity on the external data bus. The results of the various experiments conducted are described next. In all of these experiments a one bit timestamp was used because we observed that increasing the size of the timestamp has very little impact on performance.

Switching activity reduction. We measured the reduction in switching activity for the two schemes: one in which the frequent values are kept fixed and the other in which they are allowed to dynamically change during program execution. These results were gathered for varying number of frequent values (2, 4, 8, 16 and 32). The results are given in Figure 4. From the data in Figure 4 we can draw three conclusions.

- The reduction in switching activity at the data bus is greatly reduced using FV encoding when the set of frequent values is allowed to change. An average reduction of 42.7% and a maximum reduction of 81.65% is observed when a changing set of 32 frequent values are used.
- We can see that the reductions in switching activity increases significantly with an increase in allowable frequent values. The average reduction increases from 15.2% for 2 changing frequent values to 42.7% for 32 changing frequent values.
- Finally the changing value scheme outperforms the fixed frequent value scheme by greater and greater degree as the number of allowable frequent values is increased. For 32 values the fixed value scheme provides only 29.06% reduction in contrast to 42.7% reduction in switching achieved by changing value scheme.

Performance without on-chip cache. In all the experiments described so far we assumed that there was an on-chip data cache present. We also repeated our experiments without an on-chip cache. This is because in many embedded and DSP processors from AT&T Microelectronics, Motorola, Zilog and Texas Instruments there is no on-chip cache. The results in Figure 5 show that in the absence of an on-chip cache the reductions in switching activity are

Program	2 freq. values		4 freq. values		8 freq. values		16 freq. values		32 freq. values	
	fixed	changing	fixed	changing	fixed	changing	fixed	changing	fixed	changing
099.go	24.12	18.49	25.23	20.67	27.45	25.61	34.81	33.15	38.02	41.51
126.gcc	13.88	16.53	14.26	18.54	14.71	23.12	18.60	31.29	24.14	39.35
130.li	10.09	11.46	10.29	13.71	12.82	18.64	17.26	27.85	24.40	40.58
132.ijpeg	-10.71	-8.46	-10.61	-4.93	-9.48	1.05	-8.12	5.16	-5.78	9.56
134.perl	14.13	16.55	23.17	18.75	26.39	23.17	30.55	30.70	38.32	42.54
147.vortex	8.36	12.05	11.04	14.71	14.16	19.19	17.23	24.94	21.90	31.17
124.m88ksim	22.04	25.57	27.19	30.47	29.00	35.29	32.23	42.67	37.93	49.03
101.tomcatv	4.30	4.33	9.51	9.13	12.06	17.38	15.82	28.37	20.57	43.57
103.su2cor	30.51	38.55	30.80	48.13	39.39	60.86	41.91	72.58	52.06	81.65
104.hydro2d	0.18	16.97	19.02	31.93	33.62	41.04	36.49	46.28	39.09	48.05
Average	11.69	15.20	15.99	20.11	20.01	26.54	23.68	34.30	29.06	42.70

Figure 4: Switching activity reduction in data bus.

even greater. The average reduction for 32 changing values increases from 42.7% to 67.63%.

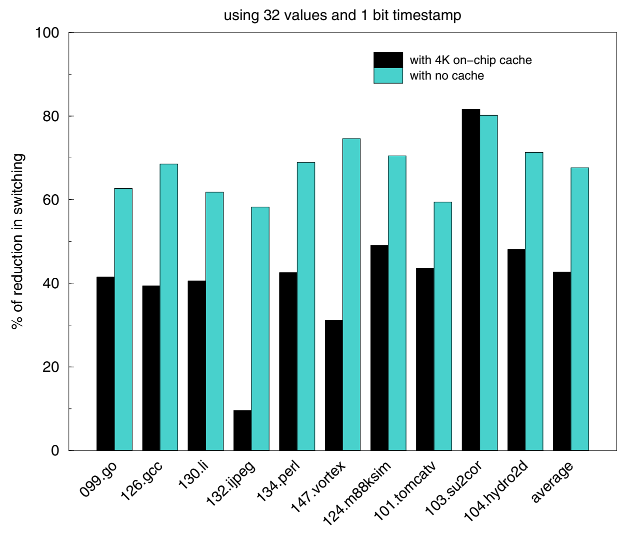

Figure 5: With on-chip cache vs without on-chip cache.

Comparison with other schemes. Other techniques that can be applied to data buses are those that have been designed to apply to general data streams. We compared the performance of our method with two existing techniques – *bus-invert* coding [2] and *adaptive* coding scheme in [3]. The results in Figure 6 show that on an average bus-invert scheme reduces switching by 10.44%. In contrast the FV encoding scheme with 32 changing (fixed) values reduces switching by 42.7% (29.06%). Thus, the changing (fixed) value FV encoding scheme outperforms the bus-invert coding method by nearly a factor of 4 (2). On an average, the reduction in switching using FV encoding is nearly six times of that achieved using the adaptive method.

4. CONCLUSIONS

In conclusion, in this paper we have demonstrated that by exploiting the characteristic of frequently transmitted values, we can design the FV encoding scheme which reduces the switching activity on an external data bus substantially.

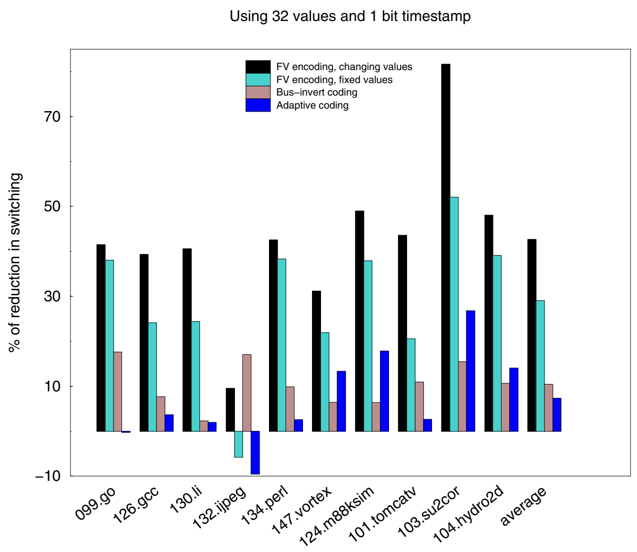

Figure 6: Comparison with bus-invert and adaptive coding.

The reductions are even greater for processors without on-chip caches. Furthermore we have demonstrated that the frequent values at any point during execution can be identified using a simple hardware mechanism. FV encoding greatly outperforms both bus-invert coding [2] and the adaptive coding in [3].

5. REFERENCES

[1] E. Musoll, T. lang, and J. Cortadella, "Exploiting locality of memory references to reduce the address bus energy," *ACM/IEEE ISLPED*, pages 202-207, Monterey, CA, August 1997.

[2] M.R. Stan and W.P. Burleson, "Bus-invert coding for low power I/O," *IEEE Trans. on VLSI Systems*, 3(1):49-58, March 1995.

[3] L. Benini, A. Macii, E. Macii, M. Poncino, and R. Scarsi, "Synthesis of low-overhead interfaces for power-efficient communication over wide buses," *ACM DAC*, New Orleans, Louisiana, 1999.

[4] Y.Zhang, J.Yang, and R.Gupta, "Frequent Value Locality and Value - Centric Data Cache Design, " *ASPLOS-IX*, pages 150-159, Cambridge, MA, November 2000.