Reducing Leakage Power of JPEG Image on Asymmetric SRAM

Yu-Hsun Lin, Xuan-Yi Lin and Yeh-Ching Chung
Department of Computer Science
National Tsing Hua University, Hsinchu 30013, Taiwan R.O.C.
lyman@sslab.cs.nthu.edu.tw, {xylin, ychung}@cs.nthu.edu.tw

Abstract

Leakage power becomes a key challenge and occupies an increasing portion of the total power consumption in nano-scale circuit design. There are many novel cache designs to reduce the leakage power based on the characteristics of programs. One of them is Asymmetric SRAM that can reduce leakage power on cache while storing bit "0". In this paper, we propose two algorithms, value-position-switch algorithm and code-bit-switch algorithm, to make the JPEG image bias on bit "0" based on Asymmetric SRAM. The value-position-switch algorithm and code-bit-switch algorithm can reduce the amount of bit "1" in Huffman coded data up to 7.33% and 25.20%, respectively. The overheads of instruction count, cycle count and power consumption for these two algorithms are negligible (< 0.3%). To the best of our knowledge, this paper is the first study to reduce leakage power in application-level by utilizing the feature of Asymmetric SRAM.

1. Introduction

Low power design has become an important issue from high performance systems to the embedded systems. The static power due to leakage power has become a major contributor in the total power dissipation when the semiconductor technology scales down. The leakage current is exponentially dependent on the value of threshold voltage (V_t). When the value V_t is reduced, the leakage power will increase exponentially [3, 6].

There are many research works to reduce leakage power in circuit-level and gate-level. The dual-V_t design is an effective method to reduce leakage power in circuit-level [11, 12]. The input vector control (IVC) method in gate-level is based on the fact that the leakage power for logic gates is dependent on the input vectors [3, 6]. The IVC method computes the minimum leakage input vectors for the logic gates in standby state [9, 10]. There are also researches to optimize the leakage power from the aspect of compiler [7, 14-16]. The compiler-based strategies introduce the data flow analysis and control flow analysis techniques to figure out the behavior of the programs. Compiler can insert special instructions to turn off the unused function units according to the behavior of programs.

Since the leakage power is proportional to the amount of transistors, memory structures such as caches would dissipate a lot of leakage power. There are emerging researches to reduce leakage power by novel cache design [4, 8]. The ideas of these researches are based on the characteristics of programs. Due to the access pattern of programs is centered on a small subset of the cache lines within a fixed period of time, the Drowsy Cache [8] reduces leakage power by turning the cold cache lines into low power drowsy mode. Another characteristic of ordinary programs is the asymmetric distribution of the bit values. The content of data cache and instruction cache has a strong bias on bit "0" for ordinary programs in standby state [4]. The Asymmetric SRAM design [4] is based on this fact to reduce the leakage power while storing bit "0".

Overall, the leakage power can be reduced in several aspects: circuit-level, gate-level, compiler-level and novel cache designs. Among them, the software design employing the feature of the novel cache design has not been much discussed. The software with good cache locality can reduce leakage power on Drowsy Cache [8]. But the research of software design to utilize the feature of Asymmetric SRAM design [4] is near to none. To the best of our knowledge, this paper is the first study to make software design utilize the feature of Asymmetric SRAM [4]. The characteristic of multimedia applications is the huge amount of compressed data which occupies a large portion in the cache. Take JPEG for example, the leakage power will be reduced when the image data is bias on bit "0". We integrate two zero-biased algorithms into the JPEG coding to reduce the amount of bit "1" in JPEG image data.

In multimedia coding, an encoder will encode the raw data into the compressed data to reduce the code size. The decoder will use the compressed data as input and decode it. Our method is to modify the encoder with negligible overhead and therefore the zero-biased compressed data can dissipate less leakage power on Asymmetric SRAM. We propose value-position-switch ("VPS" for short) algorithm and code-bit-switch ("CBS" for short) algorithm to make the compressed data bias on bit "0". We integrate the proposed VPS algorithm and CBS algorithm into Huffman coding to make the compressed data bias on bit "0". We evaluate the effectiveness and overhead of VPS algorithm and CBS algorithm in JPEG encoder and use the images from USC SIPI image database [2]. The experimental results show that the VPS algorithm can reduce up to 7.33% amount of bit "1" and the CBS algorithm can reduce up to 25.20% amount of bit "1" in Huffman coded data. To JPEG encoder, it introduces negligible (<0.3%) overhead in instruction count, cycle count and power consumption.

The rest of this paper is organized as follows. Section 2 briefly describes the Asymmetric SRAM [4]. Section 3 has an overview of the Huffman coding in JPEG. Section 4 shows the detail of VPS algorithm. Section 5 shows the detail of CBS algorithm.

2. Asymmetric SRAM

The Asymmetric SRAM is an emerging architectural design to attack leakage power [4]. The Asymmetric SRAM designs asymmetric SRAM cell, which consumes less leakage power when storing bit "0" [4]. There are different types for asymmetric SRAM cell that consider different design trade-off among performance degradation, stability loss, area overheads and the level of leakage power reduction [4]. The different asymmetric SRAM cells have their own weight on bit "0" and bit "1" for leakage power consumption, but the fact that the bit "0" dissipates less leakage power than bit "1" is the same. In this way, this asymmetric feature can be used to build a simple leakage power model for multimedia coding. The compressed data of multimedia application occupies a large portion in the memory system and contributes a lot of leakage power.

3. Huffman Coding in JPEG Encoding

Entropy coding is the last phase for JPEG encoding, and it controls the final result of compressed data. Huffman coding is one of the widely used algorithms in entropy coding that assigns shorter Huffman code for more frequent Huffman value. The Huffman coding gathers the statistics for the appearance frequency of the Huffman values as first, and generates the Huffman codes according to the frequency information. After that, the Huffman coding starts to encode the Huffman values by the corresponding Huffman codes.

The original Huffman coding does not take into consideration the amount of bit "1" and bit "0". We consider this situation and integrate our algorithms into the original Huffman coding. Figure 1 shows the integration of the proposed algorithms and original Huffman coding. The VPS algorithm and CBS algorithm are to make the Huffman coded data bias on bit "0".

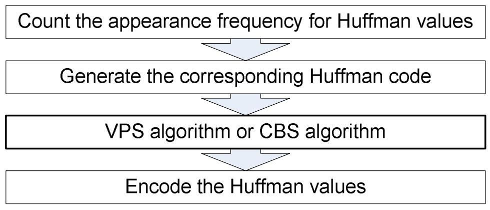

Count the appearance frequency for Huffman values
Generate the corresponding Huffman code
VPS algorithm or CBS algorithm
Encode the Huffman values

Figure 1. VPS algorithm or CBS algorithm integrates into original Huffman coding

4. The Value-Position-Switch (VPS) Algorithm

The VPS algorithm can make Huffman coded data bias on bit "0" by switching the position of Huffman value in the Huffman value list. To describe VPS algorithm, we define the following variables:

- *Huff_Value_List* is an array of Huffman value which the Huffman values correspond to the Huffman codes one by one.
- *Huff_Value_List_Copy* is an array of Huffman values which is a copy of *Huff_Value_List*.
- *High_Freq_Value_Index_Array* is an array that records the index of *Huff_Value_List*. The frequency of Huffman values would be in descending fashion when we traverse the *Huff_Value_List* via the index in the *High_Freq_Value_Index_Array* sequentially.
- *Less_1s_Value_Index_Array* is an array that records the index of *Huff_Value_List*. The amount of bit "1" in corresponding Huffman code would be in ascending fashion when we traverse the *Huff_Value_List* via the index in the *Less_1s_Value_Index_Array* sequentially.

```
Value-Position-Switch Algorithm
Input: Huffman value list (Huff_Value_List) with same-length Huffman code
Output: Switched Huffman value list (Huff_Value_List)
Value_Position_Switch(Huff_Value_List)
1. Copy the values in Huff_Value_List into Huff_Value_List_Copy;
2. Assign the sorted index (by the frequency of occurrence, in descending fashion) of Huff_Value_List into High_Freq_Value_Index_Array;
3. Assign the sorted index (by the amount of bit "1" in corresponding Huffman code, in ascending fashion) of Huff_Value_List into Less_1s_Value_Index_Array;
4. for i in 0..length(Less_1s_Value_Index_Array)-1 do
     Huff_Value_List[Less_1s_Value_Index_Array [i]]
       =Huff_Value_List_Copy[High_Freq_Value_Index_Array[i]];
   endfor
return Huff_Value_List;
```

Figure 2. Value-Position-Switch algorithm

The VPS algorithm is described in Figure 2. The algorithm retains the tree structure and switches the values of the tree nodes according to the frequency. In Figure 2, the *High_Freq_Value_Index_Array* stores the index of Huffman values with the frequency from high to low. The *Less_1s_Value_Index_Array* stores the index of Huffman values with the amount of bit "1" in corresponding Huffman code from small to large. The step 4 in VPS algorithm shows the switching process of Huffman values from the original positions to the specified positions by two index arrays. The purpose of VPS algorithm is to make more frequently Huffman value have fewer amount of bit "1" in the corresponding Huffman code. The Huffman coded data would be bias on bit "0" according to the improvement in Huffman coding. Since the total code size of compressed data is one of the major concerns of multimedia applications, the VPS algorithm only handles the Huffman values that have the same-length Huffman codes. In this way, we can keep the code size of Huffman coded data unchanged when we switch the positions of Huffman values.

Figure. 3 shows an example for applying VPS algorithm. Here we only demonstrate the three Huffman values (V1, V2 and V3) in the figure that correspond to Huffman codes (leaves) with same length (11, 00 and 01). The frequency relation for the three Huffman values in descending order is: V1, V2 and V3. The array elements are <1, 2, 3> for *High_Freq_Value_Index_Array*, and <2, 3, 1> for *Less_1s_Value_Index_Array*. The right-hand side of Figure 3 demonstrates the relation between Huffman values and Huffman codes. The arrow depicts the corresponding Huffman value for the Huffman code after applying the step 4 in VPS algorithm. The original mapping of Huffman code to Huffman value is 11 → V1, 00 → V2, and 01 → V3, respectively. After applying VPS algorithm, the new mapping is 11 → V3, 00 → V1, and 01 → V2, respectively. The Huffman values are switched to the positions corresponding to the mapped Huffman codes, and the Huffman tree structure is unchanged.

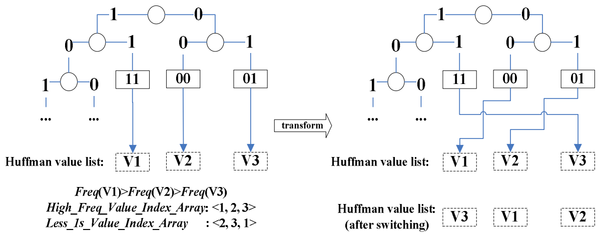

1
0
0
1
0
1
1
0
11
00
01
...
...
Huffman value list:
V1
V2
V3
transform
Freq(V1)>*Freq*(V2)>*Freq*(V3)
High_Freq_Value_Index_Array: <1, 2, 3>
Less_1s_Value_Index_Array : <2, 3, 1>
Huffman value list: (after switching)
V3
V1
V2

Figure 3. An example for Value-Position-Switch algorithm

5. The Code-Bit-Switch (CBS) Algorithm

The CBS algorithm switches the code bits on Huffman tree branches without modifying the tree structure to make the compressed data bias on bit "0". To describe CBS algorithm, we have the following definitions and an example for Huffman tree in Fig. 4.

- *Huff_Tree* is the root of Huffman tree in Figure 4.
- *Tree_Level* is the depth of the node corresponding to the root of Huffman tree.
- *Max_Switching_Tree_Level* is the upper bound for the depth when we apply CBS algorithm. We can control the traversing tree level of CBS algorithm by this variable. Default value is the maximum depth of the Huffman tree.
- *Tree_Node* is an internal node or an external node (leaf) of the Huffman tree. The leaf stands for the Huffman code.
- *Tree_Node.right_branch* is the right branch of *Tree_Node*.
- *Tree_Node.left_branch* is the left branch of *Tree_Node*.
- *Tree_Node.right_branch_code* is the code bit on the right branch of the *Tree_Node*.
- *Tree_Node.left_branch_code* is the code bit on the left branch of the *Tree_Node*.
- *Freq*(*leaf*) is the frequency of the *Tree_Node* leaf.
- *Branch_Freq*(*Tree_Node.right_branch* or *Tree_Node.left_branch*) computes the amount of the frequency of tree leaves in the right sub-tree or left sub-tree for the given *Tree_Node*.
- *Switch_Code*(*Tree_Node*) switches the position of code bit between left branch and right branch of *Tree_Node*.
- *Tree_Node.switch_flag* is a variable that records switching information for the *Tree_Node*. The variable *Tree_Node.switch_flag* equals 1 when the *Switch_Code*(*Tree_Node*) is performed. Default value is 0.

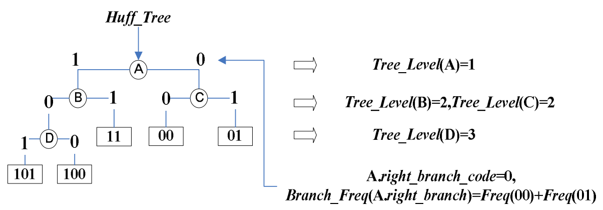


Figure 4. An example for Huffman tree

```
Code-Bit-Switch Algorithm
Input: Huffman tree (Huff_Tree), tree level (Current_Tree_Level, initial value=1)
Output: Switched Huffman tree (Huff_Tree)
Code_Bit_Switch(Huff_Tree, Current_Tree_Level)
1.  if (Huff_Tree==null || Current_Tree_Level > Max_Switching_Tree_Level) then
2.    return Huff_Tree;
3.  else
4.    Huff_Tree->switch_flag=0;
5.    if(Huff_Tree->left_branch_code==1) then
6.      Bit_1_Freq=Branch_Freq(Huff_Tree->left_branch);
7.      Bit_0_Freq=Branch_Freq(Huff_Tree->right_branch);
8.    else
9.      Bit_1_Freq=Branch_Freq(Huff_Tree->right_branch);
10.     Bit_0_Freq=Branch_Freq(Huff_Tree->left_branch);
11.   endif
12.
13.   if(Bit_1_Freq>Bit_0_Freq) then
14.     Switch_Code(Huff_Tree);
15.     Huff_Tree->switch_flag=1;
16.   endif
17.
18.   Current_Tree_Level++;
19.   Huff_Tree->left_child=Code_Bit_Switch(Huff_Tree->left_child, Current_Tree_Level);
20.   Huff_Tree->right_child=Code_Bit_Switch(Huff_Tree->right_child, Current_Tree_Level);
21. endif
return Huff_Tree;
```

Figure 5. The Code-Bit-Switch Algorithm

The CBS algorithm is shown in Figure 5. The CBS algorithm traverses the Huffman tree recursively and switches the code bits in a top-down fashion. The code bit on the branch appears at the *Tree_Level*-th position in the external nodes which are the leaves of the sub-tree connected by the branch. As a result, the frequency of the branch is the sum of frequency of the tree leaves. Figure 5 presents the switching process for each *Tree_Node n*. The two variables (*Bit_1_Freq* and *Bit_0_Freq*) store the value of *Branch_Freq*() according to the current code bit on the branch. The CBS algorithm switches the position of branch's code bit "0" and bit "1" by the comparison between *Bit_1_Freq* and *Bit_0_Freq*. The CBS algorithm makes the code bit "0" on the branch with higher frequency. The CBS algorithm also takes the code size into consideration. Since the CBS algorithm switches the branch's code bit of the tree node. The switching process only affects the content of code bits in each Huffman code. The switched Huffman tree is isomorphism with the original Huffman tree. The code size for each Huffman code remains unchanged. The total code size for the Huffman coded data is also unchanged. The Huffman table specification in JPEG coding only records the Huffman values and the number of Huffman codes with length *i* (*i*=1…16 in JPEG coding) [13]. The information for switching code bits on the branch is needed to be embedded into JPEGheader. We record the switching information (*Tree_Node.switch_flag*) in an array for a given switched Huffman tree. Figure 6 demonstrates an example for CBS algorithm and Table 1 shows the frequency information for each *Tree_Node*. Since the *Bit_1_Freq* is more than *Bit_0_Freq* in *Tree_Node* A and *Tree_Node* C, the switching process is performed on the two *Tree_Node*. The *Tree_Node.switch_flag* for each *Tree_Node* is shown in the right hand side of Figure 6. Table 2 presents Huffman codes for the external nodes in Figure 6. Table 2 shows the content of Huffman codes is changed without increasing code size.

Since the switching information (*Tree_Node.switch_flag*) is needed to be embedded into JPEG header, the number of *Tree_Node.switch_flag* should be as few as possible. The maximum number of *Tree_Node.switch_flag* is exponentially related to the variable *Max_Switching_Tree_Level*. We can count the maximum number of the *Tree_Node.switch_flag* for a given switched Huffman tree by following equation:

$$N=2^{Max\_Switching\_Tree\_Level}-1$$

N: The maximum number of *Tree_Node.switch_flag* to be record in JPEG header.

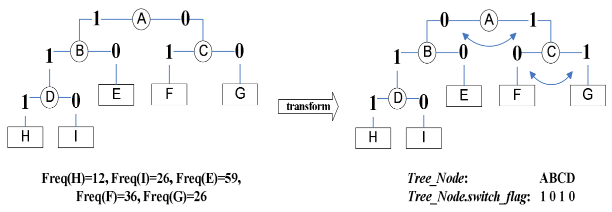
Since the code length of Huffman code in JPEG coding will up to 16 [13], the value of *Max_Switching_Tree_Level* will also up to 16. That is, the number of *Tree_Node.switch_flag* will be unacceptable (2^{16} -1) when applying CBS algorithm to the whole Huffman tree. Because the characteristic of Huffman coding is to assign shorter Huffman code for more frequent Huffman value, the value of *Branch_Freq*(*Tree_Node.right_branch* or *Tree_Node.left_branch*) will be small for a *Tree_Node* with high *Tree_Level*. In this way, we can limit the number of *Tree_Node.switch_flag* in an acceptable range by constraining *Max_Switching_Tree_Level* in small value and also keep the effectiveness of CBS algorithm. The effectiveness comparison for different *Max_Switching_Tree_Level* in CBS algorithm is shown in the section 6.3.

Table 1. Frequency of the tree nodes

Tree_Node	*Branch_Freq* (*Tree_Node.left_branch*)	*Branch_Freq* (*Tree_Node.right_branch*)	*Bit_1_Freq*	*Bit_0_Freq*	*Tree_Node.switch_flag*
A	12+26+59	36+26	97	62	1
B	12+26	59	38	59	0
C	36	26	36	26	1
D	12	26	12	26	0

Table 2. Huffman Code for external tree nodes

External Node	Huffman code (original)	Huffman code (with CBS)
E	10	00
F	01	10
G	00	11
H	111	011
I	110	010



A
B
C
D
E
F
G
H
I
1
0
transform
Freq(H)=12, Freq(I)=26, Freq(E)=59,
Freq(F)=36, Freq(G)=26
Tree_Node: ABCD
Tree_Node.switch_flag: 1 0 1 0

Figure 6. An example for code-bit-switch algorithm

6. Experimental Results

We evaluate the reduction of bit "1" in Huffman coded data for VPS algorithm and CBS algorithm. We use the images from USC SIPI image database [2] as our test cases. There are 215 images in the USC SIPI database which are divided into 4 volumes according to their characteristics. There are various image sizes for the images in one volume, such as 256x256 pixels, 512x512 pixels or 1024x1024 pixels, and color images with 24 bits/pixel, black and white images with 8 bits/pixel. We integrate the VPS algorithm and CBS algorithm into the JPEG encoder developed by Independent JPEG Group (IJG) [1].

In order to meet the input image format, we convert the images in USC SIPI database from TIFF format to BMP format. We measure the overheads of VPS algorithm and CBS algorithm by Wattch simulator [5]. The JPEG encoder can produce custom Huffman codes for the Huffman values. Since the header information must follow the rules of multimedia specification, we only measure the reduction of bit "1" and the power consumption in Huffman coded data. The default value of *Max_Switching_Tree_Level* is 16 for CBS algorithm in the following sections. Section 6.1 shows the reduction of the Huffman coded data for the amount of bit "1" and leakage power consumption. Section 6.2 shows the overheads of JPEG encoder by integrating VPS algorithm and CBS algorithm. Section 6.3 presents the settings of *Max_Switching_Tree_Level* and the effectiveness of CBS algorithm.

6.1. Reduction in Huffman Coded Data

Figure 7 shows the reduction of bit "1" in common test images which are widely used in image processing and compression. The common test images are chosen from USC SIPI image database. For VPS algorithm, we can see the maximum reduction is 6.16% for Huffman coded data in Lena image. The minimum reduction is 0.36% in Airplane image. The average reduction for VPS algorithm is 2.83% in these common test images. For CBS algorithm, we can see the maximum reduction is 11.22% in Airplane image and Lena image. The minimum reduction is 3.04% in Elaine image. The average reduction for CBS algorithm is 8.66% among these common test images.

We also apply the VPS algorithm and CBS algorithm to all of the images in USC SIPI image database. Table 3 shows the results for the test images in USC SIPI image database. The values of maximum, minimum and average reduction in database are listed for each image volume. The file name in Max reduction and Min reduction columns stand for the file name in the database.

In Table 3, the maximum reduction value for VPS algorithm is 7.33% and the minimum reduction value is 0%. The image texmos2.s512 in Texture volume is the only one that has no reduction of bit "1" for VPS algorithm. The VPS algorithm reduces the amount of bit "1" for other 214 images in USC SIPI image database that shows the effectiveness of VPS algorithm. The original amount of bit "1" in image texmos2.s512 only occupies 4.77% proportion of the whole Huffman coded data. Since the purpose of VPS algorithm is to reduce the amount of bit "1", the quite low proportion of bit "1" (4.77%) in image texmos2.s512 leaves no room for VPS algorithm to further reduce. The maximum reduction value for CBS algorithm is 25.20% and the minimum reduction value is 2.13%. The CBS algorithm can reduce all the images in USC SIPI image database. The numerical file name 7.2.01 in Miscellaneous volume has maximum reduction value for CBS algorithm and contains 55.31% proportion of bit "1" in original Huffman coded data. The proportion (55.31%) of image 7.2.01

is the highest value in the USC SIPI image database and provides a lot of space for CBS algorithm to improve.

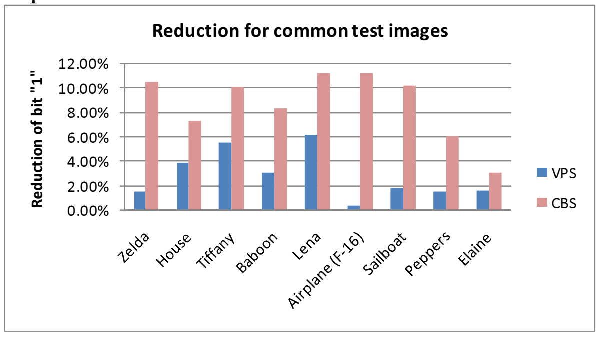


Figure 7. Reduction of bit "1" in common test images

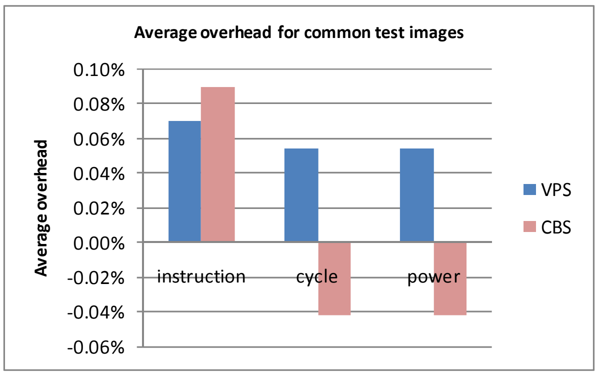


Figure 8. Average overhead for common test images

Table 3. Reduction of bit "1" in USC SIPI image database

Volume name in USC SIPI	VPS algorithm			CBS algorithm		
	Max reduction (File name)	Min reduction (File name)	Avg. reduction	Max reduction (File name)	Min reduction (File name)	Avg. reduction
Aerials (38 images)	4.67 % (2.1.09)	0.61 % (2.2.15)	2.88 %	12.03 % (2.2.12)	3.31% (2.1.12)	8.00 %
Miscellaneous (44 images)	6.42 % (4.1.04)	0.36 % (4.2.05)	2.44 %	**25.20 %** (7.2.01)	**2.13 %** (ruler.512)	8.82 %
Sequences (69 images)	**7.33 %** (6.1.13)	1.06 % (6.2.08)	3.21 %	13.25 % (6.1.01)	5.61 % (6.3.01)	8.96 %
Textures (64 images)	4.83 % (1.2.08)	**0 %** (texmos2.s512)	2.42 %	22.24 % (1.5.06)	3.09 % (1.4.07)	8.29 %

Table 4. Average Reduction of leakage power on Asymmetric SRAM for USC SIPI image database (LE version)

Volume name in USC SIPI	*P* (Original images)	*P* (Images with VPS)	Reduction	*P* (Images with CBS)	Reduction
Aerials (38 images)	71128.55	69308.69	2.56 %	65898.97	7.35 %
Miscellaneous (44 images)	15765.93	15441.76	2.06 %	14406.52	8.62 %
Sequences (69 images)	4730.54	4609.34	2.56 %	4373.55	7.55 %
Textures (64 images)	45736.42	44790.97	2.07 %	42517.94	7.04 %

Table 5. Maximum overhead in USC SIPI image database

Volume name in USC SIPI	VPS algorithm			CBS algorithm		
	Instruction	Cycle	Power	Instruction	Cycle	Power
Aerials (38 images)	0.043 %	0.037 %	0.037 %	0.059 %	-0.013 %	-0.013 %
Miscellaneous (44 images)	0.175 %	0.166 %	0.166 %	0.218 %	0.023 %	0.023 %
Sequences (69 images)	0.105 %	0.068 %	0.068 %	0.132 %	-0.076 %	-0.076 %
Textures (64 images)	0.033 %	0.050 %	0.050 %	0.043 %	-0.034 %	-0.034 %

Table 6. Comparison of different Max_Switching_Tree_Level for CBS algorithm in USC SIPI image database

Volume name in USC SIPI	CBS algorithm *Max_Switching_Tree_Level*=4			CBS algorithm *Max_Switching_Tree_Level*=16		
	Max reduction (File name)	Min reduction (File name)	Avg. reduction	Max reduction (File name)	Min reduction (File name)	Avg. reduction
Aerials (38 images)	11.01 % (2.2.12)	2.75 % (2.1.12)	6.91 %	12.03 % (2.2.12)	3.31 % (2.1.12)	8.00 %
Miscellaneous (44 images)	21.05 % (7.2.01)	1.66 % (elaine.512)	7.71 %	25.20 % (7.2.01)	2.13 % (ruler.512)	8.82 %
Sequences (69 images)	11.97 % (motion09.512)	4.77 % (6.3.01)	8.02 %	13.25 % (6.1.01)	5.61 % (6.3.01)	8.96 %
Textures (64 images)	20.92 % (1.5.06)	1.93 % (1.3.08)	7.19 %	22.24 % (1.5.06)	3.09 % (1.4.07)	8.29 %

We calculate the leakage power consumption for a given image by the following equation:

$$\boldsymbol{P} = n_0 \times w_0 + n_1 \times w_1$$

P: The total leakage power consumption of the Huffman coded data in the image.
n_0: The amount of bit "0" in the Huffman coded data.
n_1: The amount of bit "1" in the Huffman coded data.
w_0: The weight of leakage power consumption for storing bit "0".
w_1: The weight of leakage power consumption for storing bit "1".

Since there are different types for asymmetric SRAM cell, we choose leakage enhanced cell (LE) as our target SRAM cell which is focus on reducing leakage power [4]. For regular SRAM cell (RV), the weight of leakage power consumption for bit "0" and bit "1" is set as w_0 =100% and w_1 =100%, respectively. The weight for LE is w_0 = 1% and w_1 =14% in [4]. Table 4 shows the average leakage power consumption for each volume in USC SIPI image database. The reduction of leakage power consumption is slightly less than the reduction of bit "1" since the bit "0" still dissipates the leakage power on Asymmetric SRAM.

6.2. Overheads for JPEG Encoder

The average overheads for applying VPS algorithm and CBS algorithm in the common test images are illustrated in Figure 8. The overheads for instruction count, cycle count and power consumption of JPEG encoder are shown in the Figure 8. The value of instruction count increases in 0.1%. The overhead of power consumption is proportional to the overhead of cycle count. We can see the values of cycle count and power consumption for applying CBS algorithm are less than original. The reason for the less cycle count is the slight better cache locality for JPEG encoder after applying CBS algorithm. Although the instruction count will affect the value of cycle count, the cache locality has more influences in this situation. The overheads for applying VPS algorithm are all under 0.08%. As a result, the VPS algorithm and CBS algorithm can reduce the amount of bit "1" in Huffman coded data with negligible overheads.

Table 5 presents the overheads of applying VPS algorithm and CBS algorithm to all images in USC SIPI image database. We only list the maximum overheads in the table. The statistic data shows the overheads of VPS algorithm and CBS algorithm are negligible for all images in the database. The negative values of overheads by applying CBS algorithm are due to the slight improved cache locality. We can see that the VPS algorithm and CBS algorithm provide effectiveness (214 of 215 images works for VPS algorithm, all 215 images works for CBS algorithm) with negligible overheads (less than 0.3%) to reduce the amount of bit "1". And the zero-biased Huffman coded data can reduce the leakage power consumption on Asymmetric SRAM.

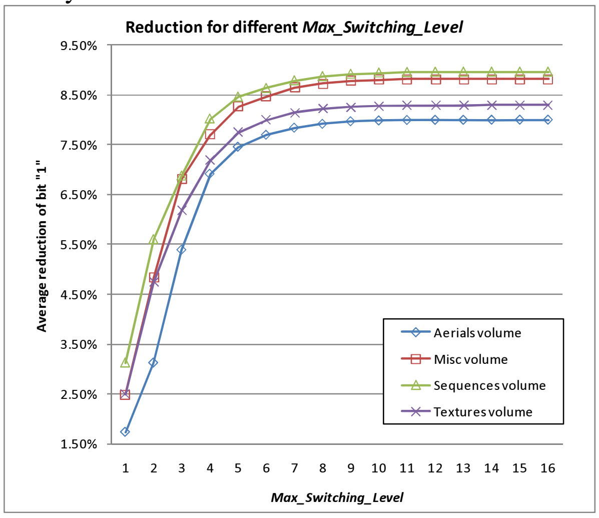


Figure 9. Average reduction for different *Max_Switching_Tree_Level*

6.3. Max_Switching_Tree_Level in CBS Algorithm

The number of *Tree_Node.switch_flag* will be unacceptable when applying CBS algorithm to the whole Huffman tree. Due to the property of Huffman coding, we can limit the *Tree_Node.switch_flag* in an acceptable range by constraining the value of *Max_Switching_Tree_Level* and without losing the effectiveness of CBS algorithm. Figure 9 demonstrates the average reduction of bit "1" for different *Max_Switching_Tree_Level* when applying CBS algorithm to the USC SIPI image database. We can see the average reduction for *Max_Switching_Tree_Level*=4 is close to the average reduction for *Max_Switching_Tree_Level*=16. In order to limit the number of *Tree_Node.switch_flag* in an acceptable range and without losing the effectiveness of CBS algorithm, we can choose *Max_Switching_Tree_Level*=4. In this way, the maximum number of *Tree_Node.switch_flag* is $2^4 - 1 = 15$ for a switched Huffman tree when *Max_Switching_Tree_Level*=4. Table 6 further compares the effectiveness between *Max_Switching_Tree_Level*=4 and *Max_Switching_Tree_Level*=16 in CBS algorithm for the database. The difference between the average reduction for *Max_Switching_Tree_Level*=4 and *Max_Switching_Tree_Level*=16 is only around 1%.

7. Conclusions

In this paper, we propose the VPS algorithm and CBS algorithm to utilize the feature of Asymmetric SRAM to reduce leakage power. From the experimental results, we have the following remarks:

Remark 1: The VPS algorithm can reduce up to 7.33% amount of bit "1" in Huffman coded data. The average reduction of VPS algorithm is between 2.42% and 3.21% for USC SIPI image database. The VPS algorithm provides both effectiveness (214 of 215 images works) and negligible overhead (less than 0.2 %).

Remark 2: The CBS algorithm can reduce up to 25.20% amount of bit "1" in Huffman coded data. The average reduction of CBS algorithm is between 8.00% and 8.96% for USC SIPI image database. The CBS algorithm provides effectiveness (all 215 images works) with negligible overhead (less than 0.3 %).

Remark 3: The difference between average reduction for *Max_Switching_Tree_Level*=4 and *Max_Switching_Tree_Level*=16 is only around 1%. As a result, the value (*Max_Switching_Tree_Level*=4) for CBS algorithm can effectively reduce the amount of bit "1" with acceptable number ($2^4 - 1 = 15$) of *Tree_Node.switch_flag* for a switched Huffman tree in maximum.

8. Acknowledgement

The work of this paper is partially supported by MediaTek Inc., R.O.C. under Grant 98F2211EA and National Science Council, R.O.C. under contract NSC-97-2220-E-007-039.

9. References

[1] *Independent JPEG Group*. http://www.ijg.org/
[2] *The USC-SIPI Image Database*. http://sipi.usc.edu/database/index.html
[3] Agarwal, A., Mukhopadhyay, S., Raychowdhury, A., Roy, K. and Kim, C. H. Leakage Power Analysis and Reduction for Nanoscale Circuits. *IEEE Micro*, 26, 2 2006, 68-80.
[4] Azizi, N., Najm, F. N. and Moshovos, A. Low-leakage asymmetric-cell SRAM. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11, 4 2003, 701-715.
[5] Brooks, D., Tiwari, V. and Martonosi, M. Wattch: a framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28, 2 2000, 83-94.
[6] Butts, J. A. and Gurindar, S. S. A static power model for architects. In *Proceedings of the Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* (Monterey, California, United States, 2000). ACM.
[7] Chen, G., Li, F., Kandemir, M., Ozturk, O. and Demirkiran, I. Compiler-directed management of leakage power in software-managed memories. In *Proceedings of the Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on* (2006).
[8] Flautner, K., Kim, N. S., Martin., S., Blaauw, D. and Mudge, T. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on* (2002).
[9] Gao, F. and Hayes, J. P. Exact and Heuristic Approaches to Input Vector Control for Leakage Power Reduction. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25, 11 2006, 2564-2571.
[10] Jayakumar, N. and Khatri, S. P. An Algorithm to Minimize Leakage through Simultaneous Input Vector Control and Circuit Modification. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07* (2007).
[11] Karnik, T., Ye, Y., Tschanz, J., Wei, L., Burns, S., Govindarajulu, V., De, V. and Borkar, S. Total power optimization by simultaneous dual-Vt allocation and device sizing in high performance microprocessors. In *Proceedings of the Design Automation Conference, 2002. Proceedings. 39th* (2002).
[12] Ketkar, M. and Sapatnekar, S. S. Standby power optimization via transistor sizing and dual threshold voltage assignment. In *Proceedings of the Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on* (2002).
[13] Pennebaker, W. B. and Mitchell, J. L. *JPEG Still Image Data Compression Standard* Van Nostrand Reinhold, 1993.
[14] You, Y.-P., Huang, C.-W. and Lee, J. K. A sink-n-hoist framework for leakage power reduction. In *Proceedings of the Proceedings of the 5th ACM international conference on Embedded software. (EMSOFT'05)* (Jersey City, NJ, USA, 2005). ACM.
[15] You, Y.-P., Lee, C. and Lee, J. K. Compilers for leakage power reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 11, 1 2006, 147-164.
[16] Zhang, W., Kandemir, M., Vijaykrishnan, N., Irwin, M. J. and De, V. Compiler support for reducing leakage energy consumption. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2003. (DATE'03)* (2003).