CYCLE EFFICIENT BIT RATE MATCHING FOR LTE-A WITH INSRUCTIONS SUPPORT

Jui-Chieh Lin and *Yu Hen Hu, Fellow, IEEE*

Department of Electrical and Computer Engineering, University of Wisconsin – Madison

ABSTRACT

Efficient software implementation of Long Term Evolution Advanced (LTE-A) wireless standard over word-based micro-processor architecture is investigated. The very high data rate of LTE-A requires more sophisticated and high throughput bit level algorithms. Due to data format mis-match, traditional word-based microprocessors face great challenge implementing these complex bit-manipulations. In this work, the implementation of bit-level interleaving operation using perfect shuffling word-level instruction is studied. In particular, an efficient implementation of bit-rate matching algorithm is demonstrated on the Texas Instruments c6416 CCS cycle accurate simulator with order of magnitude performance enhancement.

***Index Terms*—Long Term Evolution, vector processing, software defined radio, bit rate matching**

1. INTRODUCTION

Wireless standards have continuously proliferated in daily life, and their data rate has been increasing incredibly. Looking forward to the future mobile devices, one obstacle remains is handling the increasing variety and complexity concurrently with limited on-device resources.

At one end, the mobile device must be flexible and support the proliferating wireless standards; to meet the flexibility requirement, researchers have proposed a programmable communication processor architecture which is to perform all standards with a fixed underlying device [1]. At the other end, however, such architecture alone may not meet the stringent throughput requirement.

To achieve both flexibility and high-throughput, current trend appears that more mobile devices are the integration of general purpose processor and a set of programmable hardware accelerators [2]. Depending on the applications supported, there may be a few ASICs and/or programmable application specified integrated processor (ASIP), coupled with one or a few processing cores. For examples, Texas Instruments' Keystone multicore architecture [2], aiming multi-standard communication scenario, employs two TI DSPs, a few decoders and FFT coprocessors. After off-loading the decoding and FFT operations, the majority of computing power is consumed by bit-level operation. The bit-level operations were reported to be as high as 50% for 2x2 LTE [3] and 75% for Wifi transmitter on a TI-C64 DSP [5]. Furthermore, these bit-operations may result in failure to comply with standard [4]. The common shortage of bit-level processing power for various communication standards inspires an efficient hardware support for bit-level operations.

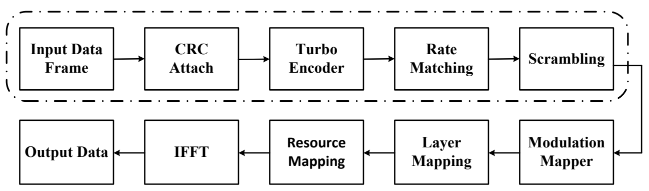


Fig. 1 LTE Downlink Block Diagram.

The low processing efficiency for bit-level algorithms is because the processor is operating at the granularity of instruction word. An instruction word is generally 32-bit or 64 bit wide. Such format mismatch frequently results in inefficient execution on a word-level processor. For example, a naïve version of the bit operations that treat each bit as a *char* in C is 70X slower than the implementation specially optimized for bit-level operation [4].

The bit-level implementation for Wifi has been discussed in [5][6][7]. Recent research also presents optimization for LTE-A along [8]. Yet efficient implementation of the bit-level operation techniques which can support both LTE and Wifi are not discussed. To achieve high flexibly and support both the LTE-A and Wifi, this work proposes the efficient software bit-operations with supports from a few bit-manipulating instructions. We constrained the instructions to be those demonstrated efficient in supporting Wifi bit-level operations. Thus the hardware cost for the multi-standard bit-level accelerator can be minimized.

The rest parts of this paper are illustrated as follow: Section 2 introduces the bit-level operations in LTE-A, and related works that address the problem. Section 3 presents the implementation technique of proposed bit-rate matching algorithm. The simulation results and a brief conclusion are presented in Section 4.

2. BIT RATE MATCHING IN LTE-ADVANCE

The bit-rate matching algorithm in the LTE-A system adjusts the code rate corresponding to the supported transmission rates. The block diagram is shown in Fig. 2. The blocks inside the dotted line is the rate bit matching blocks; the bit-rate matching block receives input from a rate 1/3 Turbo encoder, and the output is sent to a *virtual buffer* [9].

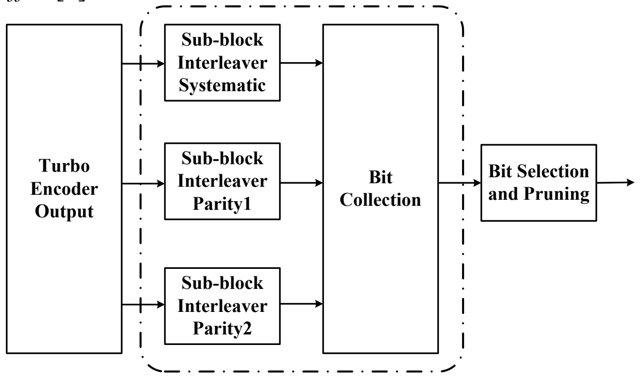

Turbo Encoder Output
Sub-block Interleaver Systematic
Sub-block Interleaver Parity1
Sub-block Interleaver Parity2
Bit Collection
Bit Selection and Pruning

Fig. 2. Rate Matching for Convolutional Coded Channels

Three bit streams are generated from the turbo encoder, *systematic*, *parity*-1 and *parity*-2. The *systematic* bit-stream is the original data coming from the MAC layer and the parity streams are redundant parity-bits added to combat channel noise.

These streams are then fed into the sub-block interleavers. Each of the sub-block interleaver can be essentially considered as a matrix transpose; the column number of matrix, $C_{subblock}$, equals 32, the corresponding row number noted as $R_{subblock}$, and the total bit number in the matrix is denoted by K_{Π}. After the matrix transpose, an inter-column permutation (ICP) is performed to further randomize the data. The inter-column permutation adopts the following pattern:
< 0, 16, 8, 24, 4, 20, 12, 28, 2, 18, 10, 26, 6, 22, 14, 30, 1, 17, 9, 25, 5, 21, 13, 29, 3, 19, 11, 27, 7, 23, 15, 31 >

The *parity*-2 stream has a slightly different formula:

$$\pi(k) = \left(P\left(\left\lfloor \frac{k}{R_{subblock}} \right\rfloor \right) + C_{subblock} \times \left(k \bmod R_{subblock}\right) + 1 \right) \bmod K_{\Pi} ,$$

in which k is the sub-block interleaver's output index. Thus for index k, the corresponding bit is originally the $\pi(k)$-th bit in the input.

After the interleaving process, the outputs of the three interleavers are collected in a *virtual buffer*. The outputs of the *systematic* interleaver go directly to the buffer. Such buffer determines the portion of bits to be transmitted. *Parity*-1 and *parity*-2 interleavers' output interlace with bits from the other parity interleaver. The reason for such bit re-arrangement is to provide equal probability of odd and even positioned bits from the encoder. The rate matching process is shown in Fig. 3.

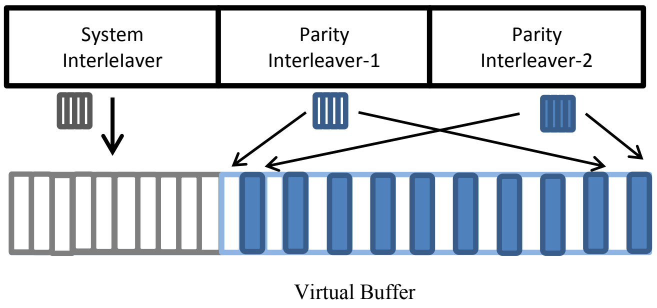
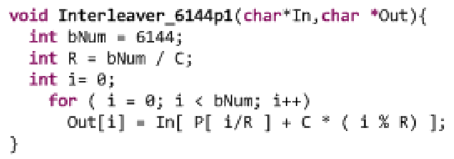


Fig. 3. Bit Level representation of Rate Matching

An example naïve implementation of the sub-block interleaver parity-1 is showed in Fig. 4. Similar implementation can be used for the other sub-block interleavers and the other rate matching operations. Each element in *char* array stores only one bit and iteration of the *for*-loop calculates the index and retrieves one single bit to the output. In the *for*-loop, the naïve implementation uses several instructions per bit and results in inefficient implementation.



```
void Interleaver_6144p1(char*In,char *Out){
  int bNum = 6144;
  int R = bNum / C;
  int i= 0;
    for ( i = 0; i < bNum; i++)
      Out[i] = In[ P[ i/R ] + C * ( i % R) ];
}
```

Fig. 4. Naïve Example of Sub-block Interleaver Parity 1

3. BIT RATE MATCHING IMPLEMENTATION

In this section, we show our optimized bit-level operations in LTE-A. We adopt the code blocks size, K = 1024bit mode to address the algorithm. The method utilizes the regularity of the rate-matching algorithm with an operation called *perfect-shuffling*, which is implemented with the instructions supported by the experimenting platform.

3.1 Perfect Shuffling Instruction

The *perfect-shuffle* operation splits a set into multiple groups. However, with the platform in hand, we have only two-way perfect-shuffle instructions. Therefore, we demonstrated only the two-way perfect shuffling.

The *Perfect-shuffle* operation, **_shfl**, in Tic6416 [10] places the lower half words in even positioned bits and the higher half word in odd positioned bits. An exact *inverse perfect-shuffle* operation is called **_deal** in the intrinsic. Such instruction gathers the even-positioned bits into the lower half-word and the odd-positioned bits into the higher half-word. The formulas of a 2-way perfect shuffling for a 2N bits input, In[k], where $0 \le k \le 2N$-1 are:

$$\text{Out}[2k] = \text{In}[k],\ \text{if } 0 \le k \le N\text{-}1$$
$$\text{Out}[2(k\text{-}N)+1] = \text{In}[k],\ \text{if } N \le k \le 2N\text{-}1$$

Similarly, the *inverse perfect-shuffle* can be formulated, within the range of $0 \le k \le 2N$-1, as:

$$\text{Out}[k/2] = \text{In}[k],\ \text{if } (k \bmod 2) = 0$$
$$\text{Out}[(k\text{-}1)/2+N] = \text{In}[k],\ \text{if } (k \bmod 2) = 1$$

Illustration of the **_shfl** and **_deal** are shown in Fig. 5.

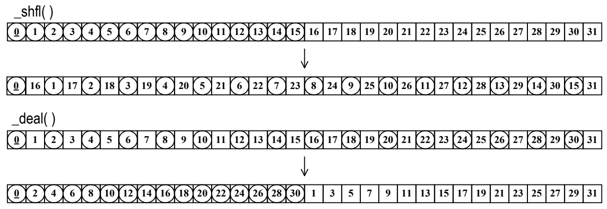


Figure 5. Illustration of perfect-shuffle and inverse perfect-

Since the provided perfect instructions in our experiment platform, TI C6416, do not support inter-word perfect shuffling, a packing instruction to extract the gathered half-word is required. The **_packl2** and **_pack2** instructions which collect the lower half-word, which has all the even bits, and the higher half-word, which has the odd bits, and put them into the output word correspondingly. The operations of **_packl2** and **_pack2** with the perfect shuffling instructions are shown in Fig. 6. If we let W_A contain the bit indexed by < 0, 1, 2 … 31> and W_B contain the bit indexed by < 32, 33, 34… 63>, the output W_C contains the bit indexed by < 0, 2, 4 … 62> and W_D contains those indexed by < 1, 3, 5 … 63>. W_C and W_D then contain the bit indexed that has remainder 0 and 1 modulate by 2 respectively.

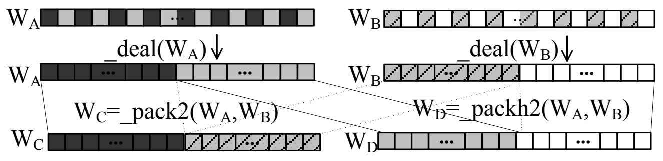
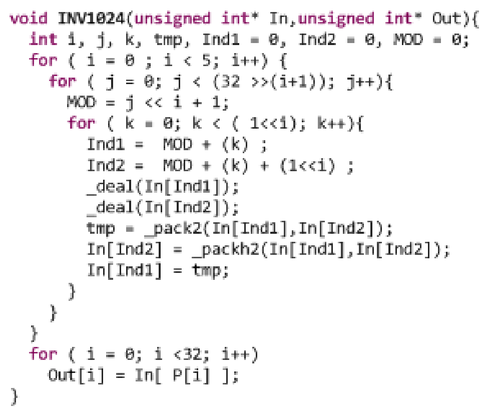


Figure 6. Inverse *Perfect-shuffle* and *Packs* on two 32-bit word.

3.2 Sub-Block Interleaver

Because the sub-block interleavers' column width is 32, one can adopt a five-level perfect shuffling and perform the sub-block interleavers [5]. We briefly show the C code implementation of the sub-block interleaver with *perfect-shuffling* intrinsic in Fig. 7.

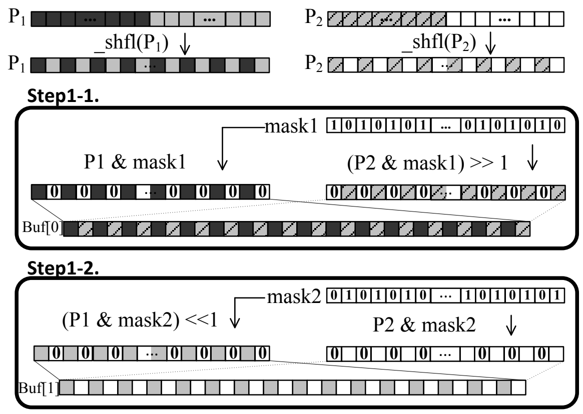


```
void INV1024(unsigned int* In,unsigned int* Out){
  int i, j, k, tmp, Ind1 = 0, Ind2 = 0, MOD = 0;
  for ( i = 0 ; i < 5; i++) {
    for ( j = 0; j < (32 >>(i+1)); j++){
      MOD = j << i + 1;
      for ( k = 0; k < ( 1<<i); k++){
        Ind1 =  MOD + (k) ;
        Ind2 =  MOD + (k) + (1<<i) ;
        _deal(In[Ind1]);
        _deal(In[Ind2]);
        tmp = _pack2(In[Ind1],In[Ind2]);
        In[Ind2] = _packh2(In[Ind1],In[Ind2]);
        In[Ind1] = tmp;
      }
    }
  }
  for ( i = 0; i <32; i++)
    Out[i] = In[ P[i] ];
}
```

Fig. 7. Perfect Shuffling Implementation of Sub-block Interleaver Parity 1

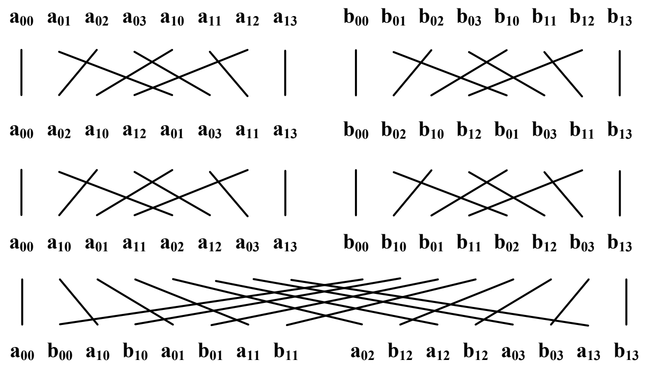
3.3 Integration Interlaever-P1&P2 with Bit Collection

The interlacing of parity-1 and parity-2 streams can be performed with *perfect shuffling* for each word from both streams; the implementation is shown in Fig. 8. Two bit-vector inputs, P_1 and P_2, are *perfect shuffled* and masked. The first half from P1 are *perfect shuffled* to the even indexed bits in Buf[0] (starts from 0); similarly, the first half of P2 is stored in the odd-indexed bits. The output stored in and Buf[1] is the interlaced bits from P_1 and P_2. Such implementation greatly reduces the computation cycles comparing to a naïve implementation.



However, since the interleaver output is the results of five stages of perfect-shuffling, we can conceptually omit one level of perfect shuffling from the interleaver design. To better present the implementation of interlacing, we consider a smaller yet illustrative example. Assume we are interlacing the transpose of two 2x4 matrixes [a_{00}, a_{01}, a_{02}, a_{03}, a_{10}, a_{11}, a_{12}, a_{13}] and [b_{00}, b_{01}, b_{02}, b_{03}, b_{10}, b_{11}, b_{12}, b_{13}], the corresponding outputs of transposed matrixes are: [a_{00}, a_{10}, a_{01}, a_{11}, a_{02}, a_{12}, a_{03}, a_{13}] and [b_{00}, b_{10}, b_{01}, b_{11}, b_{02}, b_{12}, b_{03}, b_{13}]. And the bit-interlacing result of these two matrices is: [a_{00}, b_{00}, a_{10}, b_{10}, a_{01}, b_{01}, a_{11}, b_{11}, a_{02}, b_{02}, a_{12}, b_{12}, a_{03}, b_{03},

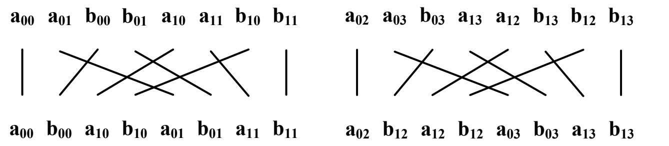
a_{13}, b_{13}]. The corresponding implementation in perfect-shuffling is shown in Fig. 9.



a00 a01 a02 a03 a10 a11 a12 a13 b00 b01 b02 b03 b10 b11 b12 b13
a00 a02 a10 a12 a01 a03 a11 a13 b00 b02 b10 b12 b01 b03 b11 b13
a00 a10 a01 a11 a02 a12 a03 a13 b00 b10 b01 b11 b02 b12 b03 b13
a00 b00 a10 b10 a01 b01 a11 b11 a02 b12 a12 b12 a03 b03 a13 b13

Figure 9. Perfect-Shuffling Implementation of Matrix Transpose and Bit-Interlacing.

We assume the instruction can combine the 2 bits of two words as *pack* instructions. So, we combine the lower-half of each word and the higher half of each word [a_{00}, a_{01}, b_{00}, b_{01}, a_{10}, a_{11}, b_{10}, b_{11}] and [a_{02}, a_{03}, b_{02}, b_{03}, a_{12}, a_{13}, b_{12}, b_{13}]:



a00 a01 b00 b01 a10 a11 b10 b11 a02 a03 b03 a13 a12 b13 b12 b13
a00 b00 a10 b10 a01 b01 a11 b11 a02 b12 a12 b12 a03 b03 a13 b13

After we feed the above matrix into the 1st interleaver with perfect-shuffle and concatenate the output, we obtain the expected interlacing bit-vectors.

Thus one level of perfect shuffling is saved.

4. SIMULATION RESULTS AND CONCLUSION

We implemented the bit-rate matching algorithm in ANSI C and profiled the program with TI Code Composer Studio v5 [11]. The experimental platform is TI-C6416 DSP operating at 600MHz; the results in clock cycle are compared with the results of naïve *char* array type implementation; our program is 9 times faster due to the efficiency of bit-level parallelism. The cycle numbers for both implementations are shown in Table I. Note in Table I, the naïve implementation of 6144 mode has much worse performance may results from the temporary storage exceeds the cache size. TI bit rate co-processor (BCP) [12] works around with the memory shortage by calculating one of the interlaver out of the co-processor. However, we keep it here to show the performance loss due to the large storage.

To the best of our knowledge, there are very few works targeting efficient LTE-A rate matching implementation. Although TI has regarded the necessity of bit-level operation accelerator and proposed the BCP that supports LTE-A, their BCP's design was not disclosed and the blocks remain as black-box to users [12].

Table I. Selected Results of Bit Matching Algorithms

	Naïve Rate Matching	This work Rate Matching			
Code Block Size	Total (Cycle)	Inv. S1 (Cycle)	Integrated Inv. P1+P2+CB (Cycle)	Total (Cycle)	Comparison (X)
1024	60430	2116	3064	5180	11.66
2048	110606	5797	7885	13682	8.08
4096	137229	11591	15767	27358	5.54
6144	477199	17385	23649	41034	34.33

This work extracts the common bit-patterns from the operations in communications standards and executes them with *perfect-shuffling* instruction and its inverse. The compiled codes show that other than *perfect-shuffling* and *pack* instructions, the assembly contains only ADD, SUB, Branch, Load-word and Store-word instructions. Since no additional complex support is required, the proposed bit-rate matching algorithm can be easily integrated into a bit-operation processor.

We believe that such bit-operation accelerator is essential to device supporting multiple communication standards. However, to achieve this goal, regularity of the bit-level operations needs to be extracted from more communication standards. So far, we have performed the LTE and Wifi standard with similar instruction set supports; further research is required to achieve an ultimate goal of bit-level operations support for all communication standards.

5. REFERENCES

[1] J. Mitola III, *Cognitive Radio Architecture*, John Wiley & Sons, Inc., 2006.

[2] Texas Instruments, "Code Composer Studio Development Tools v3.3 Getting Started Guide," *SPRU509H*, May 2008.

[3] Z. Lin, "Delivering Performance, Efficiency and Differentiation with TI's Multistandard Base Station SoCs" *white paper*, 2011.

[4] J.-C. Lin, S. J. Chen and Y. H. Hu, "Cycle Efficient Linear Feedback Shift Register for Software Defined Radio", *IEEE Transaction on Computers*, Apr. 2012.

[5] J. Lin et al., "Perfect Shuffling for Cycle Efficient Puncturer and Interleaver for Software Defined Radio," *ISCAS'*10, Jun. 2010.

[6] J.-C Lin et al., "Parallel Implementation of Convolution Encoder for Software Defined Radio on DSP Architecture," *SAMOS,* Jul. 2009.

[7] J.-C Lin et al., "Cycle Efficient Scrambler Implementation for Software Defined Radio," *ICASSP*, Mar. 2010.

[8] J.-Fu Cheng et al. "Analysis of circular buffer rate matching for LTE turbo code," pp. 1-5, *VTC*, 2008.

[9] "LTE; Physical Channels and Modulation", *3GPP TS 36.211 v. 10.5.0*, Jul. 2012.

[10] Texas Instruments, "TMS320C66x DSP, CPU and Instruction Set," *SPRUGH7*, Nov. 2010.

[11] Texas Insrtuments, "Code Composer Studio v5 Users Guide," *available online, processors.wiki.ti.com/index.php/Download_CCS.*

[12] Texas Instruments, "KeyStone Architecture Bit Rate Coprocessor (BCP)," *SPRUGZ1*, Aug. 2011.