On The Performance of Code Block Segmentation for LTE-Advanced

Karlo G. Lenzi, Felipe A. P. Figueiredo, José A. B. Filho and Fabricio L. Figueiredo
DRC – Convergent Networks Department
CPqD – Research and Development Center on Telecommunication
Campinas, SP - Brazil
{klenzi, felipep, jbianco, fabricio}@cpqd.com.br

Abstract— **In this paper we present a new approach to code block segmentation used on the 3GPP Standard LTE-Advanced channel coding physical layer. Code block segmentation is a generic procedure commonly applied before turbo encoding whose sole function is to fragment a large transport block into smaller code blocks, reducing memory requirements of the turbo code interleaver. The main result of this paper is a speedup of 80 times over the original procedure defined by the 3GPP Std. when implemented in a DSP architecture.**

Keywords— LTE; code block segmentation; code optimization; architecture design; DSP.

I. INTRODUCTION

New applications are driving computational complexity to its limits, demanding each time more from hardware architectures. Algorithms specially developed for LTE-Advanced [1] (a 4G wireless standard for broadband access), Software-Defined Radio (SDR) and High Performance Computing (HPC), among others, defy designers to make the most of every resource available for a given architecture. In this context, General Purpose Processors (GPP), Graphic Processing Units (GPU), Digital Signal Processors (DSP), Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) are striving to support current requirements of modern applications.

In this paper, we revisit the optimizations proposed in [2] for the code block segmentation, a channel coding generic procedure of the LTE-Advanced, which can reach data rate peaks of one gigabits per second in downlink. The results presented in [2] focus on hardware design. We now present a new approach to this procedure mapping it to a commercial DSP, where analysis of instructions count, memory size and clock cycles are made.

This paper is organized as follows. First, details of the block segmentation procedure for LTE-Advanced are given in Section II. In Section III, we present a new approach to block segmentation. Section IV discusses and presents the results of such optimized procedure on the ADSP-BF533 Blackfin DSP processor from Analog Devices [3]. Finally, we conclude in Section V.

II. LTE BLOCK SEGMENTATION

The LTE code block segmentation is a generic procedure defined by 3GPP TS 36.212 "Multiplexing and Channel Coding" Standard [4]. This procedure breaks an input transport block (TB) and its cyclic redundancy check (CRC) of an arbitrary size B greater than 6144 bits, into a series of smaller code blocks (CB), whose sizes K are selected from a predefined set of values, as presented by Table I, before turbo encoding [5].

TABLE I. CODE BLOCK SIZES (K) FOR TRANSPORT BLOCK SEGMENTATION.

40	48	56	64	72	80	88	96	104	112
120	128	136	144	152	160	168	176	184	192
200	208	216	224	232	240	248	256	264	272
280	288	296	304	312	320	328	336	344	352
360	368	376	384	392	400	408	416	424	432
440	448	456	464	472	480	488	496	504	512
528	544	560	576	592	608	624	640	656	672
688	704	720	736	752	768	784	800	816	832
848	864	880	896	912	928	944	960	976	992
1008	1024	1056	1088	1120	1152	1184	1216	1248	1280
1312	1344	1376	1408	1440	1472	1504	1536	1568	1600
1632	1664	1696	1728	1760	1792	1824	1856	1888	1920
1952	1984	2016	2048	2112	2176	2240	2304	2368	2432
2496	2560	2624	2688	2752	2816	2880	2944	3008	3072
3136	3200	3264	3328	3392	3456	3520	3584	3648	3712
3776	3840	3904	3968	4032	4096	4160	4224	4288	4352
4416	4480	4544	4608	4672	4736	4800	4864	4928	4992
5056	5120	5184	5248	5312	5376	5440	5504	5568	5632
5696	5760	5824	5888	5952	6016	6080	6144		

If segmentation is applied, an additional CRC sequence of length L = 24 bits is attached to each new CB [6, 7]. If B size is smaller than the total segmented CB sizes, F filler bits (nulls) are padded to the beginning of the first CB segment. The core processing of this procedure follows the pseudo-code presented in (a) and (b), extracted from [4]. The parameter Z is equal to 6144 bits and represents the maximum code block size. The sum of the TB plus CRC size is equal to B and is the only input parameter to the code block segmentation procedure.

Parameter C is the number of CB segments found for a single input TB. B' is the new transport block size with all additional CRCs included.

Listing (a). Determining the number of code blocks C.

```
if B ≤ Z
   L = 0
   C = 1
   B' = B
else
   L = 24
   C = ceil [B / (Z – L)]
   B' = B + C * L
end if
```

From pseudo-code (a), when B is less or equal to Z, the input TB do not need segmentation, since a single CB is able to store it entirely. On the other hand, if B is greater than Z, segmentation is applied and the TB is fragmented into C code blocks. A new CRC sequence is also appended to each new CB. The number of segments C is determined from the current TB size, the maximum block size Z and the CRC length. A ceiling operation is applied to guarantee that the number of CBs is sufficient to store the entire TB. The pseudo-code presented in (b) shows how to select a value from Table I for the segmented code block sizes K_+ and K_-.

Listing (b). Selection of K_+ and K_-

```
K+ = minimum K in Table I such that C * K ≥ B'
if C = 1
   C+ = 1, K- = 0, C- = 0
else if C > 1
   K- = maximum K in Table I such that K < K+
   ΔK = K+ – K-
   C- = floor [(C * K+ – B') / ΔK]
   C+ = C – C-
end if
F = K+ * C+ + K- * C- – B'
```

III. A New Approach To Code Block Segmentation

The code block segmentation procedure as defined in [4] may assume any input size B. The values of K_+ and K_-; C_+ and C_-; and F are derived from B following the steps listed in pseudo-codes (a) and (b). Assuming that in [2] all possible code optimizations of pseudo-codes (a) and (b) were exhausted, the only way to further improve the performance of such procedure would be to have some information about the behavior of its input parameter B. Looking into 3GPP Std. 36.212, where all procedures related to channel coding are defined, there is no information that could help us on that task.

As known, LTE-Advanced uses adaptive modulation and coding to make efficient use of its wireless resources [1]. As the channel degrades, more robust coding and modulation schemes are employed which consequently reduces the bit rate due to lower order modulations and increase of channel coding redundancy. It is possible to compute the total link capacity based on the modulation and coding scheme used as well as on the number of physical resource blocks available for a given user. These two parameters limit the possible transport block sizes, since they control the channel capacity for transmitting data.

As defined in 3GPP Std. 36.213 "Physical Layer Procedures" [8], there are 32 modulation and coding schemes, all of them indexed by the I_{MCS} (modulation and coding scheme) parameter, that when associated with the modulation order will then refer to a I_{TBS} (transport block size) index, where only 27 valid values are permitted and the remaining ones are reserved, with physical resource blocks (N_{PRB}) ranging from 1 to 110. The crossing of both parameters, I_{TBS} and N_{PRB}, is depicted in [8], section 7.1.7.2.1.

This table of size 27 x 110, which for being too large will not be presented here, gives us all possible TB sizes that a LTE-Advanced PHY can receive according to its configuration. This information can be used to further improve the block segmentation procedure. Analyzing this data, it is possible to see that, depending on the values of I_{TBS} and N_{PRB}, the same TB size is retrieved. Therefore, to simplify the analysis, we filtered all possible TB sizes that were unique and added 24 bits to it (since CRC will be attached to the TB before segmentation is performed). Table II summarizes theses results.

TABLE II. ALLOWED B SIZES FOR BLOCK SEGMENTATION.

40	360	704	1312	2432	4608	9552	19872	40600
48	368	720	1344	2496	4800	9936	20640	42392
56	400	736	1376	2560	4992	10320	21408	43840
64	416	768	1408	2624	5184	10704	22176	45376
80	432	800	1440	2688	5376	11088	22944	46912
96	448	832	1504	2752	5568	11472	23712	48960
112	464	864	1568	2816	5760	11856	24520	51048
128	480	896	1632	2880	6016	12240	25480	52776
144	496	928	1696	3008	6224	12600	26440	55080
160	512	960	1760	3136	6480	12984	27400	57360
168	528	992	1824	3264	6736	13560	28360	59280
176	544	1024	1888	3392	6992	14136	29320	61688
200	560	1056	1952	3520	7248	14712	30600	63800
232	576	1088	2016	3648	7504	15288	31728	66616
248	592	1120	2048	3776	7760	15864	32880	68832
280	608	1152	2112	3904	8016	16440	34032	71136
304	624	1184	2176	4032	8272	17016	35184	73736
312	640	1216	2240	4160	8528	17592	36720	75400
320	656	1248	2304	4288	8784	18360	37912	
352	672	1280	2368	4416	9168	19104	39256	

From the 27 x 110 original table, only 178 unique values of B were found. From those, only 70 values are above 6144 bits and therefore, will need to be segmented. By these numbers, it is possible to verify that 80 code block sizes from the 188 possible values presented in Table I are defined but not used by LTE-Advanced. With only 178 unique input sizes to the block segmentation procedure, we can simulate every input value and verify its outputs. By doing so, two relevant aspects of the LTE-Advanced standard regarding block segmentation can be observed: 1) For all unique input sizes, filler bits will always be

equal to zero; and 2) For all unique input sizes, C_- will always be equal to zero. Based on these, two points are observed: 1) $C_+ = C$ and, thus there is no need to perform any of the operations related to C_-, such as searching for K_- and computing Δ_K; and 2) since $F = 0$, it means that if $B \leq Z$, no segmentation is applied, $K_+ = B$, which eliminates the need to search for K_+ in that situation. In summary, the whole segmentation procedure resumes itself to computing C and searching for K_+ only if $B > Z$. In other words, besides searching for K_+ in case of segmentation, pseudo-code (b) is unnecessary (dead code). Further on, the searching procedure for K_+ and the computation of C with such reduced block size set can be implemented with a simple switch-case statement in software.

IV. Results & Discussions

In order to demonstrate the results of the suggested optimizations on a DSP architecture, we have chosen the ADSP-BF533 Blackfin processor from Analog Devices [3]. This DSP processor is a 32-bit fixed-point SIMD architecture with a 10-stage RISC pipeline and 148 KB of L1 SRAM memory, allowing the use of both instructions and data caches. The Blackfin processor can operate at a maximum frequency of 600 MHz, achieving a peak performance of 1200 MMACs.

All coding was done in C language. We used the VisualDSP++ 5.0 IDE from Analog Devices for compiling the code. For all implementations, we assumed default compiler options. No architecture-specific optimization was made. The choice of the processor is arbitrary since our goal is just to have a common architecture to establish the speedup of the code block segmentation procedure proposed here, avoiding as much as possible any technology-dependent optimizations. The choice of a different architecture will have little influence on the results because block segmentation is not a data hungry procedure, which will diminish improvements that rise from caching data. The only memory used has 188 entries of short ints, which gives a total memory space of 376 bytes, a quantity small enough to fit into any internal SRAM memory of modern DSP processors.

Pipeline performance will be determined by branch prediction accuracy, since the search procedure will consume most of the CPU time. Data dependencies and a small instruction count will limit instruction reordering, leaving only pipeline forwarding techniques to resolve these issues. A small difference may be observed if a floating-point architecture were chosen instead of the fixed-point one adopted here, but it would only affect two floating point instructions present in the pseudo-codes (a) and (b). Therefore, we believe that the results presented here will be a good approximation of the performance expected in other types of DSP or GPP architectures.

Let's start our performance evaluation with the original procedure listed in (a) and (b). Here, two points must be considered: 1) the direct transcription of pseudo-code (a) and (b) imply into two search procedures, one for the minimum size of K_+ and the other for the maximum size of K_-; and 2) there is no indication of how this search should be implemented by the standard, therefore we chose the simplest one, linear search and called it NAIVE implementation, by the reasons mentioned before. The new approach proposed in this paper is called OPT.

To evaluate the performance of the code block segmentation procedure for both implementations, all unique block sizes (TB plus CRC size) summarized by Table II are fed as input to the procedure. The number of clock cycles consumed by each input is recorded. The minimum, mean and maximum clock cycle count obtained for each implementation is presented in the sequence.

Starting with the NAIVE implementation, the coding of the original code block segmentation procedure is straightforward. Since we are using a fixed-point architecture, where floating-point operations are emulated, the computing of C and C_- will involve calling four floating-point functions, one for converting an integer type to float, a division by float, a rounding operation (ceil or floor) and one for converting a float back to an integer type. All these function calls, except the rounding one, are handled automatically by the compiler, as presented by (d), where C, B, Z and L are unsigned ints and Z and L are known at compilation time.

Listing (d). Code fragment for computing the number of code blocks C.

```
C = ceil((float) B / (Z - L));
```

The code block sizes presented in Table I are stored in a static memory region as an array of 188 unsigned short ints, occupying a total memory space of 376 bytes. This array will be used in the NAIVE implementation during the search procedures of K_+ and K_-. The number of clock cycles from NAIVE is presented in Table III.

TABLE III. ORIGINAL PROCEDURE CLOCK CYCLES COUNT.

Implementation	*Min*	*Mean*	*Max*
NAIVE	*87*	*5067*	*7854*

As can be seen, a naive implementation of the procedure is very time consuming. This is mainly because of the search algorithm adopted. In fact, in this implementation there are two searches: one for the minimum K_+ and one for the maximum K_-, and both of them use a linear search, where every memory index is tested until the correct value is found. For those reasons, the mean clock cycle count for the NAIVE implementation is 5067 clock cycles. Regarding implementation OPT, the choice of design was a simple switch-case statement with 71 entries as presented by (e).

Listing (e). Code fragment for code block segmentation.

```
switch (B >> 8) {
      case  24 : C = 2;  Kp = 3136; break;
      case  25 : C = 2;  Kp = 3264; break;
      ...
      default   : C = 1;  Kp = B;
}
```

Table IV presents the number of clock cycles to perform the LTE-Advanced optimized code block segmentation generic procedure. As we can see, the approach proposed has significant impact on the code performance. The mean number of clock cycles to compute the entire code block segmentation

procedure, drops from 5027 clock cycles to only 61 cycles, meaning a speedup of 82 times.

TABLE IV. OPTIMIZED PROCEDURE CLOCK CYCLES COUNT.

Implementation	*Min*	*Mean*	*Max*
OPT	*40*	*61*	*76*

Regarding program, constant data and stack memory sizes, Table V summarizes the results of the two implementations of this paper.

TABLE V. BLOCK SEGMENTATION PROGRAM AND DATA MEMORY.

Implementation	*Program (bytes)*	*Data (bytes)*	*Stack (bytes)*
NAIVE	*488*	*376*	*46*
OPT	*1224*	*0*	*12*

As we can see, when optimizing for speed, program size tends to increase. From the NAIVE implementation to the OPT there is an increase in program size of 2.5 times. On the other hand, the amount of memory for data and stack reduced by 3.8 times. Since OPT do not need to store any code block sizes, since it do not perform any search, no data constant memory is necessary.

V. CONCLUSION

In this paper we presented a new approach to the LTE-Advanced code block segmentation procedure. As presented, an speed up of 82 times over the straight implementation of the procedure as defined in the standard is achieved.

ACKNOWLEDGMENT

The authors thanks the support given to this work, developed as part of the RASFA project, financed by the Fundo de Desenvolvimento das Telecomunicações - FUNTTEL, from the Brazilian Department of Communication, through the partnership no. 01.09.0631.00 with FINEP – Financiadora de Estudos e Projetos.

REFERENCES

[1] 3rd Generation Partnership Project, Technical Specifications Series 36 for E-UTRA (Release 10), available online at http://www.3gpp.org.

[2] Karlo G. Lenzi, José A. Bianco F. and Felipe A. P. de Figueiredo, “Code Block Segmentation Hardware Architecture for LTE-Advanced”, IEEE Wireless Communications and Networking Conference (WCNC): PHY, April, 2013.

[3] Analog Devices, “Blackfin Embedded Processor: ADSP-BF531/ADSP-BF532/ADSP-BF533 Datasheet”, Januray, 2011.

[4] 3rd Generation Partnership Project, “3GPP TS 36.212 version 10.6.0 Release 10: Multiplexing and Channel Coding”, July, 2012.

[5] Motorola, "Code Block Segmentation for LTE Channel Coding", R1-071059, 3GPP TSG RAN WG1 #48, Feb. 2007; XP-050105053.

[6] R1-074473, Ericsson, ETRI, ITRI, LGE, Motorola, Nokia, Nokia Siemens Networks, Nortel, Qualcomm, Samsung, ZTE, “TB CRC Generator Polynomial,” 3GPP TSG RAN WG1#50b, Shanghai, China, Oct. 8 - 12, 2007.

[7] R1-074448, Qualcomm Europe, “Generator polynomial for transport block CRC,” 3GPP TSG RAN WG1#50b, Shanghai, China, Oct. 8 - 12, 2007.

[8] 3rd Generation Partnership Project, “3GPP TS 36.213 version 10.7.0 Release 10: Physical layer procedures”, October, 2012.