# On The Performance of Code Block Segmentation for LTE-Advanced: An In-Depth Analysis

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 our previous work, we presented a brief analysis of the performance of the code block segmentation procedure adopted by the 3GPP LTE Advanced Standard as part of its physical layer channel coding scheme. Here, an in-depth analysis of its performance is offered together with a new approach to the LTE-Advanced code block segmentation. Code block segmentation is a generic procedure applied before turbo encoding whose function is to fragment a large transport block into smaller code blocks, reducing memory requirements. Analysis showed that only 39% of all transport blocks need segmentation. Results based on two different architectures, one focused on a DSP and the other on an FPGA, provided a speedup of 80 times over the original procedure with a total resources count of 166 slices, maximum frequency of 392 MHz and with very low latency**.

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

## I. INTRODUCTION

The computational complexity presented by algorithms adopted in recent communication standards constitutes a huge challenge to designers aiming to achieve the high data rates specified by these standards. These highly complex algorithms have been pushing hardware architectures to its limits, demanding more and more processing power. Algorithms specially developed for LTE-Advanced [1], which is one of the 4G wireless standards for broadband access, are testing designers' capabilities of devising systems that make the most of every resource available on 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 such applications. Of those, due mainly to its (re)configurability and parallel nature, FPGAs and ASICs are a feasible choice for such strict performance requirements and costume features, compared to other solutions. On the other hand, GPPs and DSPs are interesting solutions for projects that demand faster development cycles and code reuse.

In our previous paper [2], we proposed an FPGA-based hardware architecture for one of the five generic procedures used in the channel coding of the LTE-Advanced physical (PHY) layer, namely the code block segmentation generic procedure. The LTE technology can reach data rate peaks of one gigabit per second in downlink [1]. This high throughput demands intensive computation and a well-designed architecture to support such data rates. Currently, there are some solutions for LTE-Advanced PHY available on the market [3-6].

The code block segmentation procedure was not randomly chosen. This generic procedure does not perform as a stream-based computing like other channel coding procedures proposed by the LTE-Advanced PHY. Due to its sequential nature, it does not benefit from the parallelism offered by technologies such as FPGAs or ASICs. Nevertheless, this fact does not restrict designers from developing LTE PHYs on such technologies.

Still, a large number of projects, especially in the wireless communication industry, are conducted on GPPs or DSPs architectures. To make evident the benefits of the architecture proposed, we also presented an implementation of the procedure on a DSP architecture [7]. Preliminary results showed a speedup of 80 times the original code block segmentation procedure running on a commercial DSP.

In this paper, we aim to establish an in-depth analysis of the performance of the code block segmentation from a hardware and software perspectives, offering analysis not yet covered by our previous works as well as presenting novel results of an FPGA-based architecture, which surpassed past results.

This paper is organized as follows. First, details of the block segmentation procedure for LTE-Advanced are given in Section II. In Section III, some optimizations for the code block segmentation procedure are presented. In Section IV, we discuss a new approach to code block segmentation, based on the behavior of its input parameter. In Section V, we present the hardware impact of such procedure, offering results collected from its implementation on a Xilinx Virtex-6 FPGA [8]. Section VI discusses and presents the results of such procedure on the ADSP-BF533 Blackfin DSP processor from Analog Devices [9], analyzing its performance and code size. Finally, we conclude in Section VII.

## II. LTE-ADVANCED BLOCK SEGMENTATION

The LTE code block segmentation is a generic procedure defined by 3GPP TS 36.212 "Multiplexing and Channel Coding" Standard [10]. This procedure breaks an input transport block (TB) plus CRC (Cyclic Redundancy Check) of arbitrary size (B) greater than 6144 bits, into a series of smaller code blocks (CB), whose sizes (K) are selected from a

978-1-4799-2927-6/13 $26.00 © 2013 IEEE
DOI 10.1109/SBAC-PAD.2013.25

predefined set of values, before turbo encoding. This is done so that the memory requirements of the encoder may be reduced.

TABLE I. CODE BLOCK SIZES 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 | | |

The maximum CB size of 6144 was selected due to the fact that coding performance of the turbo code has diminishing return after block sizes greater than 8192 bits (< 0.05 dB coding gain) [11]. Table I presents the predefined code block sizes selected for the LTE 3GPP Std

If segmentation is applied, an additional CRC-24B sequence of length L = 24 bits is attached to each new CB [12, 13]. If B size is smaller than the total segmented CB sizes, filler bits (nulls) are padded at the beginning of the first CB segment.

The parameter Z is equal to 6144 and represents the maximum code block size. The transport block size plus CRC size is called 'B' and is the only input parameter to the code block segmentation procedure, assuming, theoretically, any positive integer value.

As mentioned, L is the length of the CRC sequence, and if segmented, the original TB plus CRC-24A will have an additional CRC-24B (24 bits) attached to each CB. Parameter C is the number of code block segments (CBs) found for a single input TB. B' is the transport block size with all additional CRCs included. Finally, F represents the number of filler bits padded to the beginning of the first code block.

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), transport blocks with sizes less than 6144 bits do not need segmentation, since a single code block is able to store the entire TB. On the other hand, if the TB size is greater than 6144 bits, segmentation is applied and the transport block is fragmented into C code blocks. A new CRC sequence is also appended to each new CB.

The number of segments (or blocks) 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 size 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'
```

There is only one input parameter to the block segmentation procedure, namely, the transport block size B, and there are five output parameters from this procedure, the sizes of two code block segments ($K_+$ and $K_-$, where $K_- < K_+$), the number of segments of each size ($C_+$ and $C_-$) and the number of filler bits F.

With these parameters, the remain of the code block segmentation can be performed, i.e., filling these CB segments, starting with $K_-$ and moving up to the $K_+$ blocks, with the original un-fragmented data bits from the TB, appending filler bits at the beginning of the first CB, if needed, and attaching a new CRC sequence at the end of each CB (see [10] for more details on how to fragment data and append CRC). After that, the CB is ready for encoding.

## III. OPTIMIZATING THE PROCEDURE

From a code optimization perspective, there are some techniques that can be applied to reduce the complexity of the code block segmentation procedure and improve its performance, namely, strength reduction, code motion, copy propagation and dead code removal.

Taking a closer look at the listing (a) and (b), the first technique that can be used is strength reduction. In the computation of C there is a division by constant that can be automatically converted to a multiply by the inverse constant.

The constant P = 1 / (Z - L) = 1 / 6120, although eliminates the need of a floating-point division, still implies a floating-point multiply operation. By analyzing this constant, it is possible to see that it represents a repeating binary number with 12 leading zeros before the binary point. If scaled properly, this floating-point number can be easily represented as a fixed-point representation.

Of course, this conversion will introduce a quantization error, but if scaled by $2^{27}$, not only this multiply operation will

become an integer one (assuming floor rounding), but will also lead to an error of 9 x $10^{-12}$ to the full precision representation, which is lower than a single precision floating-point quantization error [14]. This conversion avoids the use of a time-consuming function call to floating-point emulation in fixed-point architectures or the need to employ long latency floating-point instructions.

A second benefit is the reduction of the codeword length, which for a full-precision fixed-point format would be 66 bits. This way, only 15 bits are used, leaving a simple short int type to be enough to represent and store this constant.

To obtain the final division by (Z-L), a shift back to the right of 27 bits is made after multiplying B by P. The ceiling operation is performed by masking the lower 27 bits of the result and checking if it is different from zero. If true, one is added to the final result.

Another opportunity to apply strength reduction is given by the multiplication by L, listed in (a). It can be rewritten as a shift and add operation by powers of two, since L can be decomposed in (8 + 16). This will save a few clock cycles.

Regarding code motion, it is possible to observe that if B is lower than or equal to Z, C will always be 1, if not true, C will always be greater than or equal to 2. Therefore, the branch condition depicted by listing (b) can be removed by moving the YES branch of C == 1 to the conditional branch where B > Z is evaluated.

This would eliminate a pipeline stall depending on the branch prediction technique employed by the processor running the procedure. The code motion makes evident that the segmentation procedure is only executed if B is greater than Z. In the next section this point will be reinforced and the optimizations presented here will be furtherer improved.

Regarding (b), two optimizations are possible. The first is a direct consequence of the two code block sizes, $K_+$ and $K_-$, being adjacent blocks, which can be inferred by listing (b). This fact is no coincidence and it was created to minimize the number of filler bits, which reduces the burden of the decoder due to padding, and to limit the worst-performing segment, since turbo encoders improve their coding performance as the code block sizes increases [11].

Knowing that $K_+$ and $K_-$ are adjacent eliminates the need to search for the $K_-$ block size, thus removing an unnecessary function call and a search procedure. The value of $K_-$ is retrieved by accessing the address of $K_+$ minus one.

The second, since $K_+$ and $K_-$ are adjacent block sizes and all block sizes are spaced by multiples of 8 bits, $\Delta_K$ is always a multiple of eight. From Table I, it is possible to verify that $\Delta_K$ can only assume one of these values: 8, 16, 32 or 64; consequently making the $\Delta_K$ division a $\log_2(\Delta_K)$ shift (by 3, 4, 5 or 6 bit positions, respectively) to the right, where $\log_2(\Delta_K)$ is known at compile time.

The last optimization proposed is to apply copy propagation to the expression C * K used on the search for $K_+$. At the end of the search, $K = K_+$, thus producing $C * K_+$, which is used once again (it is a copy) during the computation of $C_-$. This saves a multiplication operation from the original procedure. Better yet, if the search condition is changed to $K_+ * C - B'$, we also eliminate a subtraction operation. This copy propagation can also be used to rewrite the filler bits expression [28]:

Listing (c). Optimized filler bits computing.

$$F = [K_+ * C - B'] - [C_- << \log_2(\Delta_K)]$$

Listing (c) presents an efficient way to compute the filler bits in case segmentation is applied. By rewriting the expression, it is possible to notice that both $K_+ * C - B'$ and $\log_2(\Delta_K)$ were already computed in previous iterations, therefore, copy propagation may be applied and only a subtraction operation needs to be performed to compute the number of filler bits.

With these optimizations, code block segmentation can be implemented only with integer operations without any need to support fixed or floating-point types. Another improvement is that no division operation is applied.

Although these optimizations can significantly reduce computational complexity, it is the algorithm used in the code block size procedure that will greatly impact the performance of the code segmentation procedure. For that reason, we assume that the intended search procedure for code block segmentation is the binary search, although any other algorithm could be used.

## IV. Code Block Segmentation Revisited

To further improve the performance of the code block segmentation procedure some information about its input was needed, since the output of this procedure depends solely upon the value of B. Looking exclusively into 3GPP Std. 36.212, where all procedures related to channel coding are defined, no information about the behavior of B is found.

If we look into the API L2/L1 definition for LTE eNodeB [15], we find that the transport block size passes through a 16-bit register that represents the size of the transport block in bytes, which let us conclude that the allowed range (from minimum to maximum) of B in bits is $[8 + L; 8(2^{16} - 1) + L]$ (where L is the length of the CRC that is attached to the TB). Therefore, all B values are multiple of 8 bits, which could lead to some improvement based on the fact that the least 3 significant bits of B are always zero (especially if we are designing our own custom hardware). Depending on the type of interface between L2/L1, extra information could be used for improving this procedure, but the improvements would be very small.

Another alternative to examine the behavior of B is 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 transport block sizes, and thus, some assumptions may be made about the possible values of B.

We can find in the 3GPP Std. 36.213 "Physical Layer Procedures" [16], all possible TB sizes generated from the crossing between modulation and coding scheme, presenting us the information that we needed about the behavior of B. This table is depicted in section 7.1.7.2.1 of the 36.213 Std.

The original table has a total of 27 x 110 entries containing predefined transport block sizes. Table II presents only the unique values from that table. A total of 178 unique values are found and from those, only 70 TB sizes are greater than 6144 bits. This means that only 39% of the TB sizes defined by the standard will actually demand segmentation. Another interesting result is that 80 TB sizes, from Table I, are defined but not used by LTE-Advanced during operation.

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 | |

When simulating the behavior of the code block segmentation procedure for all TB sizes that requires segmentation, we observed that F and $C_-$ will always be equal to zero. Consequently, $C_+ = C$, which completely eliminates the need to search for $K_-$ and compute $\Delta_K$ in any situation; and since $F = 0$, it means that if $B \leq Z$, no segmentation is applied and $K_+ = B$, which eliminates the need to search for $K_+$ in this case. The code block segmentation procedure only needs to compute C and search for $K_+$ when $B > Z$. Having this small set of values, searching for $K_+$ and computing C can be pre-computed and stored in a small LUT (look-up table).

## V. AN FPGA-BASED CODE BLOCK SEGMENTATION

Based on the observations presented in section VI, an optimized hardware architecture can be designed for the LTE-Advanced code block segmentation.

As mentioned, only 178 transport block sizes are actually used by the LTE-Advanced standard. After segmentation, a transport block will be fragmented into code blocks of sizes found in Table I. From this set, only 70 transport block sizes need segmentation, making possible to compute all output parameters during compile or synthesis time.

In order to design the architecture, it is important to know the data width of C, $K_+$ and B. To do that, we just need to define the largest value possible of each one. From Table II, we find that the largest B value is 75400, thus resulting in a data width of 17 bits. From Table I, we find that the largest K value is 6144, leading to a data width of 13 bits. At last, the maximum C value can be inferred from B, which returns 13 and consequently, a data width of 4 bits.

With such a reduced set of possibilities, the architecture for LTE-Advanced code block segmentation could be implemented as a large multiplexer or a LUT, depending on the implementation style.

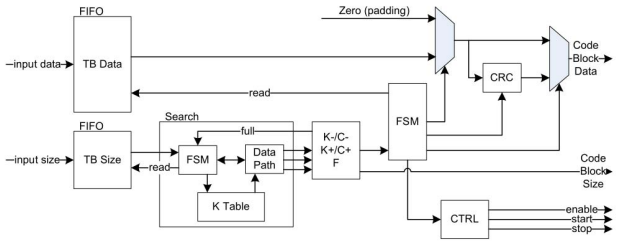


Figure 1. Code block segmentation hardware architecture.

Since the multiplexing is based on the value of B, a 17-bit comparator would be needed. This comparator can be reduced if we scale B by 256. After scaling, we would still have unique values for all B sizes that need segmentation, but reducing the comparator to only 9-bits.

A shift of 8 bits to the left will reduce the size of the comparator and will limit the range of the input block size from 23 to 294, instead of 6224 to 75400, retrieved from Table II for code blocks greater than 6144. Values below that will be reduced to zero, but will always fall under the condition where C is equal to 1 and $K_+$ is equal to the input block size B. That way a $K_+$ / C (13 + 4) bit 71 : 1 multiplexer with a 9-bit comparator is enough to implement the whole segmentation procedure.

The first architecture proposed is depicted by Fig. 1, extracted from [2]. Here, a full code block segmentation hardware architecture, based on the optimizations suggested on section III, is presented. The main results of this implementation were a total resource utilization of 190 slices at a maximum frequency of 351 MHz and a total latency of 110 cycles in a Virtex-6 FPGA device.

As we can observe, the main drawback of such architecture is the latency. There are two reasons for that. One is that in this architecture there are two search procedures, one for $K_+$ and one for $K_-$. The second is that moving data to/from memory in FPGA can easily drop the maximum frequency, since memory elements are placed in equally spaced columns while logic are placed around it. The distance between memory and logic usually generates long nets that limit the maximum frequency.

A reasonable amount of pipelining is needed between logic and memory to reduce these long nets to improve performance, but at a cost of higher latencies. Depending on the search algorithms used, the total latency can be worse.

The architecture proposed can be reused by the new code block segmentation approach. The only component that will change is the "Search" component. Here, the "FSM", "Datapath" and the "K Table", whose functions were well explained in [2], will be replaced by a multiplexer. Some minor modifications are made in the FSM next to the search component, which controls the filling of the code blocks. Since most of the code block segmentation procedure outputs are zero, the control logic can be simplified, taking in consideration only the C and $K_+$ values.

By replacing the contents of the search component by a single multiplexer, when implemented in a Virtex-6, this new architecture achieved 166 slices, 392 MHz and 9 cycles of latency. Since most of the architecture did not change, area and performance results are almost the same.

The main contribution of this architecture is the latency reduction of about eleven times when compared with our

previous results. Before, an iterative search was needed to find the correct code block size. This search took many cycles to conclude. Now the search resumes itself to a multiplexer. Table III contrast the results of both implementations on the same device, a Virtex-6 XC6VLX75T-2FF484 FPGA [8] with default implementation options.

As we can see, some area reduction was obtained with the new approach. This reduction is due to the simplification of the control logic on the FSM component next to the search component. The latency improvement is due to the fact that there are no search procedures in this architecture.

TABLE III. ORIGINAL PROCEDURE CLOCK CYCLES COUNT.

| *Implementation* | *Resources (Slices)* | *Frequency (MHz)* | *Latency (clock cycles)* |
|---|---|---|---|
| *OPTMIZED (Section III)* | *190* | *351* | *110* |
| *NEW APPROACH (Section IV)* | *166* | *392* | *9* |

## VI. A DSP-BASED CODE BLOCK SEGMENTATION

An ADSP-BF533 Blackfin processor from Analog Devices [9] was used to implement the code block segmentation engine. This processor is a 32-bit fixed-point SIMD architecture with a 10-stage RISC operating at a maximum frequency of 600 MHz.

Here the performance evaluation is focused on the procedure listed in (a) and (b), not including CRC attachment and CB data filling. At first, the direct transcription of pseudo-code (a) and (b) implies in two search procedures, one for the minimum size of $K_+$ and the other for the maximum size of $K_-$. No indication of which search algorithm should be used is given by the standard.

Since the original coding of the block segmentation procedure serves as a reference for performance evaluation of the optimizations suggested in Sections III and IV, the choice of including or not both searches, based on the findings presented in section III and how to implement it will greatly impact the total cycle count of the procedure.

That way, to make a fair comparison, we have chosen to implement two versions of the original procedure: one naive, where both searches are performed using a linear (brute force) search, and one where we assume that hardware/software designers have noticed from start that the second search is unnecessary in the code block procedure and adopted binary search as the search algorithm. These two implementations will be designated from now on as NAIVE and DIRECT, respectively.

We also propose two other versions, one based on the optimizations presented in Section III, and one based on the new approach proposed on Section IV. These implementations were named OPT1 and OPT2, respectively.

Finally, to evaluate the performance of the code block segmentation procedure for all proposed versions, all unique block sizes (TB size plus CRC size) allowed by the LTE-Advanced Std. and summarized by Table II (a total of 178 possible block sizes) are fed as input to the procedure. The number of clock cycles consumed by each block size input is registered. The minimum, mean and maximum clock cycle count obtained for each implementation is presented in the sequence.

Starting with the NAIVE and DIRECT implementations, the coding of the original code block segmentation procedure is straightforward. The only point open for discussion during coding is how the search procedure will be implemented.

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 operation, 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 int's 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 int's, occupying a total memory space of 376 bytes. This array will be used in the NAIVE, DIRECT and OPT1 implementations during the search procedures of $K_+$ and $K_-$. The results (number of clock cycles) from NAIVE and DIRECT are presented in Table IV.

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 answer is found. For those reasons, the mean clock cycle count for the NAIVE implementation is 5067 clock cycles.

TABLE IV. ORIGINAL PROCEDURE CLOCK CYCLES COUNT.

| *Implementation* | *Min (clock cycles)* | *Mean (clock cycles)* | *Max (clock cycles)* |
|---|---|---|---|
| *NAÏVE* | *87* | *5067* | *7854* |
| *DIRECT* | *106* | *728* | *1383* |

After removing the unnecessary search procedure and choosing a more efficient algorithm for the remaining one, the results of the DIRECT implementation showed improvement. The mean clock cycle count reduced to only 728 clock cycles, having its maximum count as almost twice that value.

Moving on to the optimized versions, (e) presents a fragment of the code written for implementation OPT1, where the computation of the number of code blocks is modified so that no floating-point operation is needed.

Listing (e). Code fragment for an optimized code block computing.

```
mul = B * scaled_inv_Z_minus_L;
rem = mul & 0x7ffffff;
tmp = mul >> 27;
C   = rem ?  tmp + 1 : tmp;
```

Regarding implementation OPT2, the choice of design was a simple switch statement with 71 entries as presented by (f).

Listing (f). 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 V presents the results, in number of clock cycles, to perform the LTE-Advanced optimized code block segmentation generic procedure. As we can see, both optimizations have significant impact on the code performance. The mean number of clock cycles to compute the entire code block segmentation procedure, drops from 728 clock cycles to only 61 cycles, meaning a speedup of almost 12 times.

TABLE V. OPTIMIZED PROCEDURE CLOCK CYCLES COUNT.

| *Implementation* | *Min (clock cycles)* | *Mean (clock cycles)* | *Max (clock cycles)* |
|---|---|---|---|
| *OPT1* | *82* | *335* | *455* |
| *OPT2* | *40* | *61* | *76* |

To summarize the speedup achieved by the approaches proposed here, Table VI shows the crossing of all implementation results. Regarding program, constant data and stack memory sizes, Table VII summarizes the results of the four implementations realized for this paper.

TABLE VI. BLOCK SEGMENTATION SPEEDUP (BASED ON MEAN VALUES).

| *Implementation* | *NAIVE* | *DIRECT* | *OPT1* | *OPT2* |
|---|---|---|---|---|
| *NAÏVE* | *1.00* | *0.14* | *0.07* | *0.01* |
| *DIRECT* | *6.96* | *1.00* | *0.46* | *0.08* |
| *OPT1* | *15.13* | *2.17* | *1.00* | *0.18* |
| *OPT2* | *83.07* | *11.93* | *5.49* | *1.00* |

TABLE VII. CODE BLOCK SEGMENTATION PROGRAM AND DATA MEMORY.

| *Implementation* | *Program (bytes)* | *Data (bytes)* | *Stack (bytes)* |
|---|---|---|---|
| *NAIVE* | *488* | *376* | *46* |
| *DIRECT* | *482* | *376* | *58* |
| *OPT1* | *632* | *376* | *68* |
| *OPT2* | *1224* | *0* | *12* |

As we can see, when optimizing for speed, program size tends to increase. From the NAIVE version to the OPT2 one there is an increase in program size of almost 3 times. On the other hand, the amount of memory for data and stack reduced in the order of almost 4 times. Data memory size represents the constant table of code block sizes used in the first three versions, 188 unsigned short int's.

## VII. CONCLUSION

In this paper we presented an in-depth analysis of the LTE-Advanced code block segmentation procedure, presenting performance and area results from both hardware/software perspectives. For that to be accomplished, a commercial DSP architecture and an FPGA device were used. Having an improved performance of over 80 times when compared with the original procedure running on a DSP, this paper showed that actually, only 39% of all transport block sizes will need segmentation. Regarding the FPGA architecture, a reduction of eleven times of the total latency is obtained when compared with our previous work.

## 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 – Finaciadora de Estudos e Projetos.

## REFERENCES

[1] 3rd Generation Partnership Project, Technical Specifications Series 36 for E-UTRA (Release 10), available online on 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] Texas Instruments, "Enabling LTE development with TI's new multicore SoC architecture", Feb. 2010.

[4] Aricent, "LTE eNodeB PHY Framework", http://www.aricent.com/software/lte-enodeb-phy-framework.html, accessed on September 14, 2012.

[5] Picochip, "Developing LTE Small Cell with Picochip (MWC2011)", http://www.picochip.com, 2011.

[6] Ubiquisys, http://www.ubiquisys.com, accessed on September 14, 2012.

[7] Karlo G. Lenzi, José A. Bianco F., Felipe A. P. de Figueiredo and Fabricio L. Figueiredo, "On The Performance of Code Block Segmentation for LTE-Advanced", IEEE Application (ASAP), Jun, 2013.

[8] Xilinx, "Virtex-6 Family Overview Datasheet", Jan, 2012.

[9] Analog Devices, "Blackfin Embedded Processor: ADSP-BF531/ADSP-BF532/ADSP-BF533 Datasheet", Januray, 2011.

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

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

[12] R1-074848, Ericsson, "CRC Computation Method", 3GPP RAN1#51bis, Jeju, Korea, November 05-09, 2007.

[13] 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.

[14] B. Widrow and I. Kollár, "Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications", Cambridge University Press, Cambridge, 2008.

[15] FemtoForum API, "LTE eNB L1 API Definition v1.1", FemtoForum Technical Document, December, 2010.

[16] 3rd Generation Partnership Project, "3GPP TS 36.213 version 10.7.0 Release 10: Physical layer procedures", October, 2012.