Implementation and Evaluation of Raptor Codes on Embedded Systems

Todor Mladenov, *Student Member, IEEE,* Saeid Nooshabadi, *Senior Member, IEEE,* and Kiseon Kim, *Senior Member, IEEE,*
.

Abstract—Raptor codes have been proven very suitable for mobile broadcast and multicast multimedia content delivery, and yet their computational complexity has not been investigated in the context of embedded systems. At the heart of Raptor codes are the matrix inversion and vector decoder operations. This paper analyzes the performance, energy profile and resource implication of two matrix inversion and decoding algorithms; Gassuian elimination (GE) and 3rd Generation Partnership Group (3GPP) standard (SA), for the Raptor decoder on a system on a chip (SoC) platform with a soft-core embedded processor. We investigate the effect of the cache size, memory type and mapping on the performance of the two algorithms under consideration. We show that with an appropriate data to memory mapping a speed up factor of 5.77 can be obtained for GE with respect to SA. This paper also proposes a dedicated peripheral hardware block that achieves 5.90 times better performance compared with the software, requiring an energy consumption that is lower by a factor of 5.5, when the symbol size and the data path word-length is small (32 bits). We show that with parallel processing in hardware, using the wider word-lengths, and employing bigger symbol sizes T, we can improve the performance, while reducing the energy consumption. Extending the hardware word-length and symbol size T to 128 bits will results in a performance improvement factor of 6.73 in favor of the hardware; while energy consumption reduces by a factor of 3.8!

Index Terms—Raptor Codes, decoder, sparse matrix, hardware/software co-design, system on a chip, embedded system.

1 Introduction

RAPTOR codes have drawn significant attention since their introduction in [1]. Being a member of the Fountain codes family [2], [3], Raptor codes are rateless, thus providing as much parity data as needed. They can be viewed as a powerful extension of Luby transform (LT) codes [4], [5] providing linear time encoding (regardless of the quantity of repair data generated), linear time decoding (independent of the amount of loss), very close to the ideal code performance under any channel condition, and the ability to efficiently support a large range of file sizes.

Multimedia on mobile devices requires secure delivery of various sized data with minimum negotiation overhead [6], [7] and minimum computational complexity. Here is where Raptor codes have come quite useful outperforming the already well known coding schemes. The 3rd Generation Partnership Group (3GPP) multimedia broadcast/multicast services (MBMS) [8] and digital video broadcasting - handheld (DVB-H) [9] are two recent standards that have included systematic Raptor codes in the application layer forward error correction (FEC) [10].

A good algebraic insight into the operation of Raptor codes can be found in [5] together with some guidelines for the implementation of Raptor encoders and decoders to achieve good coding performance, as specified in 3GPP. Two techniques, first one to determine a minimum set of source symbols to be requested for reliable delivery, and the second one to find a sufficient number of consecutive repair symbols are proposed in [7].

Implementation and performance evaluation of Raptor codes in comparison with LT code for multimedia applications, are discussed in [11]. The performance of Raptor codes on arbitrary binary input memoryless symmetric channel is investigated in [12]. The design problems of Raptor codes for binary input additive white Gaussian noise (AWGN) channel are studied in [13]. Improved decoding algorithms for Raptor codes are proposed for transmission over the fading [14] and Gaussian channels [15]. These two works introduce the notion of "incremental decoding" for Raptor codes. The idea is further investigated in [16], [17] and [18]. The later work proposes a modification of the 3GPP decoding algorithm and demonstrates an improvement. There are other low complexity coding techniques that reduce the complexity by avoiding the process of Gaussian elimination (GE) [19]. However, this comes at the cost of violating the systematic properly of Raptor codes, a highly desirable property in the multimedia application.

Although Raptor codes are growing as a preferred multimedia delivery scheme, experimental data relating to their implementations are reported from simulation on a workstation platform where the decoding time is assumed to be a minute fraction of

• *T. Mladenov, and K. Kim are with the Department of Information and Communications, Gwangju Institute of Science and Technology, Gwangju, Republic of Korea, E-mail: {todor,kskim}@gist.ac.kr.*
S. Nooshabadi is with the Department of Electrical and Computer Engineering, Michigan Technological University, Houghton, MI, E-mail: saeid@mtu.edu

the transmission time. As far as we are aware, their implementations and computational complexity evaluation on embedded systems for mobile platforms have not yet been investigated. The decoding time on an embedded system, as we will show in this paper, can be as high or even higher than the transmission time for most multimedia applications. This paper looks at the implementation of Raptor codes on an embedded system platform, where resources in terms of the computational cost and power dissipation are limited. We investigate the operation of matrix inversion combined with the symbol vector decoding, the most demanding part of the Raptor decoder, by implementing it using two algorithms; the well known GE and the efficient matrix inversion algorithm (SA) proposed in 3GPP [8], and DVB-H [9] standards. We evaluate the relative performance of these two algorithms in terms of decoding time, power, energy and hardware resources, on an embedded processor platform.

The effects of different memory types for the matrix storage, the cache size, and data to memory mapping, on the performance behavior of the Raptor codes are demonstrated for the software versions of both algorithms. Based on the results obtained, a dedicated matrix inversion cum symbol vector decoder hardware accelerator is proposed to build a hardware/software co-designed Raptor decoder. Its decoding time, hardware resources, power and energy consumptions are analyzed and compared with the pure software implementation. We also design hardware enhancements for GE and SA based on their algorithmic structures. Finally, using the profiling data, suitability assessments are made for the implementations of GE and SA on an embedded system.

We believe the analysis of implementation tradesoffs, and the proposed design techniques to reduce the computational complexity of Raptor codes on embedded systems, for practical multimedia applications such as common intermediate format (CIF), Quarter CIF (QCIF), standard definition TV (SDTV), and other DVB-H, and 3GPP MSBS services, are important for future multimedia signal processing terminals. The results presented in this paper will assist the researchers and developers of multimedia signal processing devices in designing high performance and low power systems. The techniques we propose in this paper are applicable to any application, requiring Raptor decoding over the binary erasure channel (BEC). The representative embedded system platform, chosen for this work, is a 100 MHz NIOS II soft-core processor, with customizable instruction set, running on a field programmable gate array (FPGA) device[1]. This FPGA device is housed on a development board [2]. Fig. 1 depicts the high level block diagram of the embedded system on a chip (SoC) platform for the implementation of Raptor codes.

This paper is organized as follows. Section 2, briefly explains the operation of the Raptor codes. Section 3 describes the details of GE and SA, the two chosen algorithms for the matrix inversion. Sections 4 and 5, respectively, present the software and hardware implementation performance results in terms of execution time, power and energy, and hardware resources. While the previous sections deal with decoder performance for a small symbol size of 4 bytes, Section 6 analyzes the decoder performance for the larger symbol sizes. Section 7 concludes the paper.

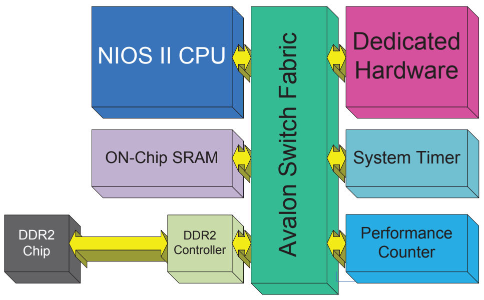


Fig. 1. Hardware/software NIOS II embedded system.

2 Raptor Codes

Raptor codes, being a class of the Fountain codes, have the ability to generate as many redundant encoding symbols as needed on-the-fly. The decoder can recover the source symbols from a set of slightly more encoded symbols, a performance very close to the ideal BEC code. After the initial introduction of Raptor codes in [1], a fully specified encoder has been adopted in [8] and [9] as part of forward error correction (FEC) scheme for data delivery services.

A Fountain code can be viewed as a regular linear block code [20], [21], which makes it possible to be represented by a generator matrix. This approach facilitates explaining the mechanism of operation of Raptor codes with an emphasis on the two algorithms that we investigate in this paper, without divulging the theoretical details of the codes.

Here, we only provide a brief description of the Raptor coding/decoding operations. For a detailed treatment of the Raptor codes, published material in [6] – [9], should be consulted.

2.1 Systematic Raptor Encoding

A block diagram of *Systematic Raptor Encoder/Decoder* is shown in Fig. 2. The encoding process is summarized in two main blocks, namely *Code Constrains*

1. An *EP3SL150F1152* Altera Stratix III FPGA, with 145, 920 logical elements, 6, 543, 360 total memory bits, 5, 630, 976 embedded RAM bits, and 48 DSP blocks, each containing 4 18×18-bit embedded multipliers.

2. Altera Stratix III development board augmented with 32 MB of 400 MHz double data rate synchronous dynamic random access memory (DDR2 SDRAM), and 4 MB of 250 MHz quad data rate static random access memory (QDRII SRAM).

Processor and *LT Encoder* [8], [9]. They both are represented by their equivalent generator matrices.

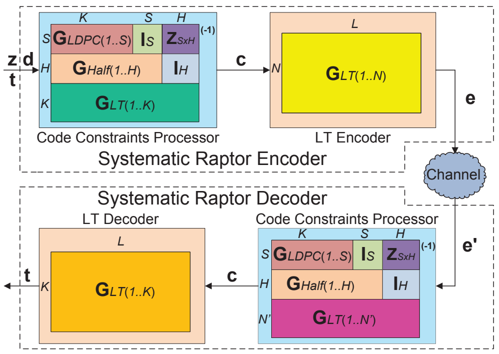

z d
t
K S H
S G LDPC(1..S) I S Z SxH (-1)
H G Half(1..H) I H
K G LT(1..K)
c
L
N G LT(1..N)
e
Code Constraints Processor
LT Encoder
Systematic Raptor Encoder
Channel
Systematic Raptor Decoder
LT Decoder
Code Constraints Processor
L
K G LT(1..K)
t
c
K S H
S G LDPC(1..S) I S Z SxH (-1)
H G Half(1..H) I H
N' G LT(1..N')
e'

Fig. 2. Block diagram of the systematic Raptor codes.

2.1.1 Code Constraints Processor

Let $\mathbf{t}$ denote K source symbols that are to be encoded. The size of each source symbol T varies from 1 to 1024 bytes. Then $\mathbf{d}$, at the input of the Raptor encoder, contains ($L = K + H + S$) symbols, (with H half symbols–named such due to the $\lceil H/2 \rceil$ weight of each column in the generating submatrix G_{Half} in Fig. 2– and S parity symbols) [8], [9], and is defined as:

$$\mathbf{d}_{[0:L-1]} = [\mathbf{z}^T \quad \mathbf{t}^T]^T \tag{1}$$

where $\mathbf{z}_{[0:H+S-1]}$ is a vector of zeros. The parameter S is the smallest prime integer such that

$$S \geq \lceil (0.01 \times K) \rceil + X \tag{2}$$

with $X(X-1) \geq 2K$. Similarly, the parameter H is the smallest integer such that

$$\binom{H}{\lceil H/2 \rceil} = \frac{H!}{2(H/2)!} \geq K + S \tag{3}$$

Code Constrains Processor multiplies $\mathbf{d}$ with the inverse of the pre-coding matrix $\mathbf{A}$ to produce the intermediate symbols $\mathbf{c}$ as:

$$\mathbf{c}_{[0:L-1]} = \mathbf{A}^{-1}_{L\times L} \cdot \mathbf{d}_{[0:L-1]} \tag{4}$$

with

$$\mathbf{A}_{L\times L} = \left[\begin{array}{c|c|c} \mathbf{G}_{LDPC} & \mathbf{I}_S & \mathbf{Z} \\ \hline \multicolumn{2}{c|}{\mathbf{G}_{Half}} & \mathbf{I}_H \\ \hline \multicolumn{3}{c}{\mathbf{G}_{LT}} \end{array}\right] \tag{5}$$

where submatrices $\mathbf{I}_S$ and $\mathbf{I}_H$ are identity matrices, and $\mathbf{Z}$ is a zero submatrix of dimension $S \times H$; $\mathbf{G}_{LDPC}$ is a $S \times K$ low density parity check (LDPC) generator submatrix defined as:

$$G_{LDPC} \cdot [c[0], ..., c[K-1]]^T = (c[K], ..., c[K+S-1])^T \tag{6}$$

$\mathbf{G}_{Half}$ is a $H \times (K + S)$ generator matrix of the Half symbols, defined as:

$$\begin{aligned} \mathbf{G}_{Half} \cdot [c[0], ..., c[S+K-1]]^T \\ = [c[K+S], ..., c[K+S+H-1]]^T \end{aligned} \tag{7}$$

$\mathbf{G}_{LT}$ is a $K \times L$ LT generator submatrix included in matrix $\mathbf{A}$ for the first K symbols to render Raptor codes systematic [6] – [8]:

$$\mathbf{G}_{LT} \cdot [c[0], ..., c[L-1]]^T = [t[0], ..., t[K-1]]^T \tag{8}$$

2.1.2 LT Encoder

With source vector $\mathbf{t}$ with K symbols pre-processed by *Code Constraints Processor*, *LT Encoder* can generate any number of encoded symbols $\mathbf{e}$ according to:

$$\mathbf{G}_{LT} \cdot \mathbf{c} = \mathbf{e}_{[0:N-1]} \tag{9}$$

where $\mathbf{G}_{LT}$ is an $N \times L$ LT generator matrix, with $N \geq K$. The value of N is selected to be sufficiently larger than K to compensate for the possible loss of encoded symbols in the channel and, therefor, to make matrix $\mathbf{A}$ invertible at the decoder side. Although LT itself is a non-systematic code, the overall Raptor code is systematic:

$$e[i] = d[H + S + i], \quad \forall_{i=0,...,K-1} \tag{10}$$

That is because $\mathbf{G}_{LT}$ for symbols ($i = 0$ to $K-1$) is included in the pre-processing matrix $\mathbf{A}$, therefore, making the resulting overall Raptor code systematic.

For a fixed value of K, submatrices $\mathbf{G}_{LDPC}$, $\mathbf{G}_{Half}$ and $\mathbf{G}_{LT}(0..K-1)$ are pre generated once and stored in memory. The structure of $\mathbf{G}_{LT}$ matrix shows how the encoded output symbols $\mathbf{e}$ are generated from the intermediate $\mathbf{c}$ symbols. The "1" values on the g^{th} row of matrix $\mathbf{G}_{LT}$ identify the intermediate symbols that are XORed to generate the encoded symbol $e[g]$.

Before the encoded symbols are sent to the channel, they are grouped and augmented with their corresponding encoding symbol identity, (ESI)–a number assigned consecutively to each produced encoded symbol–the details of which are omitted from the diagram in Fig. 2 in the interest of simplicity.

2.2 Systematic Raptor Decoding

The decoding process of Raptor codes exchanges the positions of *Code Constraints Processor* and *LT Encoder* (to be used as *LT Decoder*), as illustrated in Fig. 2 with $\mathbf{G}_{LT}$ LT generator matrices appropriately sized. The input vector $\mathbf{e}'$ containing N' ($K \leq N' \leq N$) encoded symbols (which may be nonconsecutive) is padded with $S + H$ zeroes to size it to ($M = N' + S + H$). Starting with $N' = K$ the value of N' is iteratively incremented to make the matrix $\mathbf{A}$ invertible. The difference ($N' - K$) is equal to or greater than the number of received encoded symbols lost in the channel.

The decoding is performed according to:

$$\mathbf{c}_{[0:L-1]} = [\mathbf{z}^T \quad \mathbf{e}'^T] \cdot \mathbf{A}^{-1}_{M\times L} \tag{11}$$

$$\mathbf{t}_{[0:K-1]} = \mathbf{G}_{LT} \cdot \mathbf{c}_{[0:L-1]} \tag{12}$$

where $\mathbf{G}_{LT}$ is a $K \times L$ LT generator matrix.

At the decoder side the submatrix $\mathbf{G}_{LT}(1..N')$ is first built from the input data. The ESI of the n^{th} received encoded symbol is used to generate the n^{th} row of the submatrix $\mathbf{G}_{LT}(1..N')$ through the LT encoding process.

3 Matrix Inversion Algorithms

The code profiling of Raptor codes on the 100 MHz NIOS II processor with zero data cache, DDR2 SDRAM for data, and each matrix element occupying a 32-bit memory word, shown in Table 1, highlights the fact that the inversion of the preprocessor matrix $\mathbf{A}$ (combined with symbol vector decoder) is the most time critical part of the system, contributing to up to 92% of the decoding time. Hence, we have concentrated our efforts on the optimization of inversion cum vector decoder algorithm in the Raptor decoder.

TABLE 1
Raptor Codes Profiling for $K = 1024$ and $T = 4$ Bytes.

	GE	SA
Task	Time(%)	Time(%)
Initializing Raptor Code	0.57	0.04
Generating Matrix $\mathbf{A}$	3.84	1.81
Generating Matrix $\mathbf{G}_{LT}$	0.03	0.01
Inverting Matrix $\mathbf{A}$	91.93	91.28
Other	3.63	6.87
Total	100.00	100.00

The most common matrix inversion algorithm is GE [22]. The pseudo code for a Galois Field-2 (GF(2)) GE algorithm where backward substitution and elimination are performed together, is shown in Algorithm I. The main operations involved in this algorithm are "row exchange" and "row XOR" (Exclusive-OR). While "row exchange" can be reduced to a simple pointer exchange operation, the "row XOR" has to be done one element at a time.

Algorithm I: The Gaussian Elimination algorithm (GE) for matrix inversion over GF(2)

```
Require: $A \in \{0,1\}^{n \times m}$
1: for $i = 0 : n-1$ do
2:     $j = i$;
3:     while $a_{ji} == 0$ do
4:         $j = j + 1$;
5:     if $i \neq j$ then
6:         row_exchange($\vec{a}_i$,$\vec{a}_j$);
7:     for $(k = 0 : n-1) \& (k \neq i)$ do
8:         if $a_{ki} == 1$ then
9:             for $l = 0 : m-1$ do
10:                $a_{kl} = a_{kl} \bigoplus a_{il}$
```

The specifications for 3GPP and DVB-H standards [8], [9] recommend SA as a more efficient technique for matrix inversion. A version of SA technique with the best reported performance [8], [18] is presented in Algorithm II. The operation of SA is as follows.

Algorithm II: The efficient matrix inversion algorithm (SA) over GF(2) proposed in [8], [9].

```
Require: $A \in \{0,1\}^{n \times m}$
Algorithm II: $i = 0; u = 0;$
1: Phase I
2: while $i + u < m$ do
3:     $r = 1$;
4:     while $r \leq m$ do
5:         for $s = i : n-1$ do
6:             if (count_ones&store_pos($s$) == $r$) then
7:                 break;
8:         if (row_found) then
9:             break;
10:        else
11:            $r = r + 1$;
12:    if $(i \neq s)$ then
13:        row_exchange($\vec{a}_i$,$\vec{a}_s$);
14:    if $(a_{ii} == 0)$ then
15:        col_exchange($i$,$ones\_col[0]$);
16:    for $h = 1 : r-1$ do
17:        col_exchange($(m-u-1)$ , ones_col[$h$]);
18:        $u = u + 1$;
19:    for $(k = (i+1) : n-1)$ do
20:        if $a_{ki} == 1$ then
21:            for $l = 0 : m-1$ do
22:                $a_{kl} = a_{kl} \bigoplus a_{il}$
23:    $i = i + 1$;
24: Phase II
25: Gaussian Elimination
    on submatrix $S = [i+1 : n-1][i+1 : m-1]$
26: Phase III
27: for $(k = 0 : i)$ do
28:     for $(s = i+1 : m-1)$ do
29:         if $a_{ks} == 1$ then
30:             $a_{ks} = a_{ks} \bigoplus a_{ss}$
```

In *Phase I* matrix $\mathbf{A}$ is reduced to the following form:

$$\mathbf{A}_{Phase_I(M \times L)} = \left[\begin{array}{c|c} \dfrac{\mathbf{I}_i}{\mathbf{Z}_{(M-i) \times i}} & \mathbf{U}_{M \times u} \end{array} \right] \quad (13)$$

This reduction is performed, iteratively, by first relocating the rows containing the minimum number (≥ 1) of "1s" to the top, and then moving the first column having "1" in this row to the beginning at column location i, and the remaining columns with "1" to the end of the row at column locations $m-u-1$. Note that i and u are initialized to 0. While, in each row, i increments only once per row, u can increment multiple times.

In each iteration of the algorithm one row from the top is excluded from the consideration. Further, the "count of 1s" within a row is confined to columns i to $(m-u-1)$. *Phase I* is completed when $(L = i + u)$.

In *Phase II* submatrix $\mathbf{U}$ is partitioned into lower and upper submatrices $\mathbf{U}'_{i \times u}$ and $\mathbf{U}''_{M-i \times u}$, respectively. The lower matrix $\mathbf{U}''_{M-i \times u}$ is transformed into the identity matrix $\mathbf{I}_u$ through the normal Gaussian elimination technique. The $(M-L)$ rows that are left below $\mathbf{I}_u$ are discarded. The form of the matrix produced at the end of *Phase II* is:

$$A_{Phase_{II}} = \begin{bmatrix} \mathbf{I}_i & \mathbf{U}_{i \times u} \\ \mathbf{Z}_{u \times i} & \mathbf{I}_u \end{bmatrix} \quad (14)$$

In *Phase III* the upper matrix $\mathbf{U'}$ is zeroed by XORing of each of its individual rows with the sufficient number of rows from the lower matrix $\mathbf{I}_u$.

$$A_{Phase_{III}} = \begin{bmatrix} \mathbf{I}_i & \mathbf{Z}_{i \times u} \\ \mathbf{Z}_{u \times i} & \mathbf{I}_u \end{bmatrix} \quad (15)$$

In the following two sections, we will analyze and compare the performance of Algorithms I and II on an embedded system platform with severe resource constraints. We limit the source symbol size to $T = 4$ bytes. We will show that implementation on a software/hardware SoC platform with more design degrees of freedom, allows for the GE code, with a proper data format to, surprisingly, perform better than the optimized SA algorithm recommended in [8] and [9] in terms of execution time performance, energy dissipation and logic requirement.

4 Software Implementation

This section describes the implementation details of the Raptor codes for $T = 4$ bytes on the NIOS II processor of Fig. 1, running at 100 MHz clock speed.

4.1 Memory Mapping

Two types of data to memory mapping for matrix $\mathbf{A}$ have been investigated. In the first mapping scheme, denoted as "WORD", each 1-bit matrix element is assigned to a 32-bit memory word. In the second mapping scheme, denoted as "PACKED WORD", 32 matrix elements are packed together into a single 32-bit memory word, therefore, significantly (at best 32 times) reducing the size of the required memory.

Furthermore, the pre-coding matrix $\mathbf{A}$ is widely known to be a sparse matrix. The sparsity of $\mathbf{A}$ grows linearly with the code block size K as shown in Fig. 3. Therefore, the sparse matrix representation in a compressed form through the use of link lists becomes an attractive choice. Fig. 4 shows the memory requirement for the storage of matrix $\mathbf{A}$ as a function of K. It is clear that the memory requirement for the initial storage of $\mathbf{A}$ for the sparse matrix in compressed form is the least for all values of K above 800.

However, during the process of inversion of matrix $\mathbf{A}$ and decoding of the symbol vector the intermediate operating matrix grows in size. The memory usage for three data representations for three values of K are shown in Table 2. It can be seen that for the SA scheme, for K values of 512, 1024 and 2048 the size of the memory grows by a factors of 1.20, 1.35, and 1.88, respectively. The corresponding memory growth factors for the GE scheme are 6.62, 9.02 and 14.37, which are substantially larger. It is also seen that due to the growth in the size of the linked list, the advantage in the memory utilization of compressed sparse matrix over the "PACKED WORD" is significantly lost, even for the larger values of K for the SA scheme.

It should also be noted that the memory requirement for the sparse matrix representation in the compressed form in Table 2 are the lower limits. For an embedded system without an operating system with an utility to do efficient memory compaction [23] to combat the memory leakage, there will even be a significantly larger memory requirement, when the linked list representing the compressed sparse matrix is progressively altered. For example, for the SA algorithm for $K = 1024$ the actual memory requirement due to memory leakage is more than 320.39KB, a factor of 2.53 larger than the initial requirement, and twice as much what is needed to accommodate the natural growth in the link list. Unfortunately the size of memory leakage grows exponentially with K. However, there is no memory leakage problem in "WORD" and "PACKED WORD" data formats even without memory compaction.

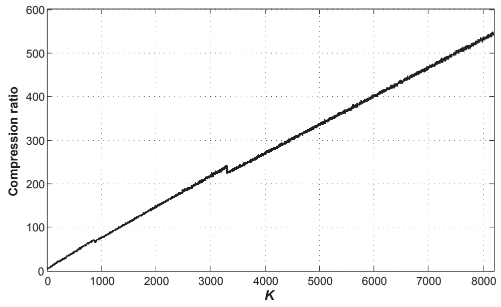


Fig. 3. Sparsity of the *Code Constraints Processor* matrix.

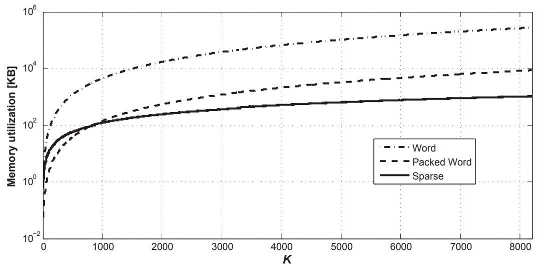


Fig. 4. Memory usage for Sparse, "PACKED WORD" and "WORD" memory schemes for the *Code Constraints Processor* matrix.

TABLE 2
Memory Utilization by Sparse, Word and Packed Word Schemes for the *Code Constraints Processor* Matrix.

Memory (KB)		***K*=512**	***K*=1024**	***K*=2048**
GE and SA WORD		1246.97	4692.25	18,073.44
GE and SA PACKED WORD		39.72	149.84	571.36
SA SPARSE	**Initial**	60.50	126.83	252.02
	Maximum	74.73	170.70	473.98
GE SPARSE	**Initial**	60.50	126.83	252.02
	Maximum	400.605	1,143.98	3,621.74

4.2 Software Performance Analysis

The execution time performances of the two matrix inversion cum symbol vector decoding Algorithms I (GE) and II (SA) are analyzed and compared when implemented, as software modules, on the NIOS II processor running at the clock speed of 100 MHz.

4.2.1 Effect of Memory Mapping

Fig. 5 shows the performance of SA and GE algorithms, implemented under the various memory mapping schemes for various values of K. The target memory for the storage of matrix $\mathbf{A}$ is the external 400

MHz DDR2, with zero data cache. The execution time grows exponentially with K. It is also seen that the "PACKED WORD" SA and GE schemes, respectively, perform worst and best across all K values. For $K = 1024$, for the "PACKED WORD" data format, the performance of GE is about 70 times better than SA.

The reason for the vastly superior performance of GE over SA in the "PACKED WORD" memory mapping scheme can be explained as follows. The SA algorithm uses "column exchange" (Algorithm II, *Phase I*, lines 15 and 17) in order to reduce the amount of "row exchange" and time consuming "row XOR" operations. It performs well under the "WORD" memory mapping but lags significantly behind under the "PACKED WORD" memory mapping. This is due to the fact that the significant increase in the number of memory accesses needed to extract the necessary rows for the "count of 1s" and "column exchange" operations through bit extraction and masking far outweighs the positive effect of the reduction in the number of "row XOR" operations. Moreover, columns are moved to the beginning or the end of the matrix, which prevents the usage of the same memory word. It should be noted that while "row exchange" operation can be reduced to simple pointer exchange operation, "columns exchange" operation requires memory exchanges one element at the time.

It is also seen from Fig. 5 that in the "WORD" format GE performs marginally better than SA for all K values. However, this is not the case for the compressed sparse representation, where SA performs much better than GE for all K values. For the compressed sparse form for $K = 1024$, SA performs about 6 times better than GE. This confirms the superior performance of SA over GE, as reported in the 3GPP [8] and DVB-H [9] standard recommendations.

From Fig. 5 it is seen that two best schemes in order of performance are "PACKED WORD" GE, and compressed sparse SA, with GE scheme performing 5.77, 3.48 and 1.37 times better for $K = 128, 1024$ and 8190, respectively.

It should be noted that with the best software implementation of Raptor codes in Fig. 5 ("PACKED WORD" GE), successful decoding can be only achieved for low bit rate applications, such as 192 kbps QCIF, corresponding to K values less than 512. This is an obvious limitation of implementation on an embedded platform. This motivates us to look for alternative solutions in form of dedicated hardware accelerators in Section 5.

Due to superior performance of "PACKED WORD" GE and compressed sparse SA, the remainder of this paper limits the discussion to relative comparison of these two schemes. Since the "PACKED WORD" SA substantially benefits from hardware implementation, as will be discussed in the next section, we also include its results in the comparisons when appropriate.

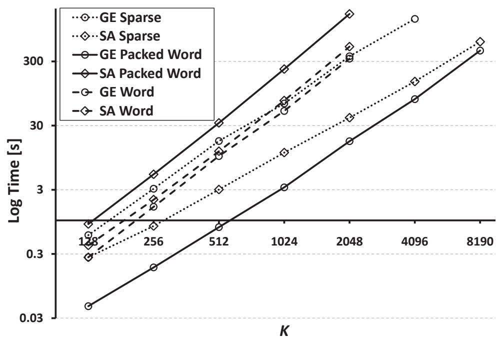


Fig. 5. Software benchmark for GE and SA under Sparse, "PACKED WORD" and "WORD" memory schemes for the *Code Constraints Processor* matrix for the various K values for $T = 4$ bytes, at 100 MHz processor speed for DDR2 memory.

4.2.2 Effect of Data Cache

To see the effect of memory bandwidth on the performance, we investigated the effect of data cache on the performance of selected algorithms ("PACKED WORD" GE/SA and compressed sparse SA) for different memory types; external DDR2 (with an 8-bit interface) and internal ONCHIP_SRAM (with 32-bit interface). The cache organization is direct mapped, write back, and with the line size of 32 bytes. The experimentation was carried out for $K = 1024$ which leads to $\mathbf{A}$ matrix with $S = 59$, $H = 13$, adding up to $L = 1096$ columns. We also assume $M = N = L$.

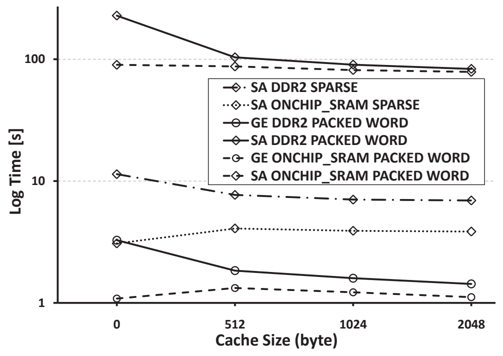


Fig. 6. Effect of cache size in *Code Constraints Processor* matrix inversion and symbol vector decoding in Software for $K = 1024$ and $T = 4$ bytes, at 100 MHz processor speed.

The comparative performance of two algorithms is summarized in Fig. 6 for $K = 1024$. The effect of the cache is significant for the case of DDR2. There is a big improvement in performance when the cache is increased to 512 bytes. The rate of improvement in the performance is much reduced when the cache size is increased to more than 512 bytes. This is specially true for the "PACKED WORD" GE and SA schemes.

For ONCHIP_SRAM the effect of cache size is either negligible ("PACKED WORD" SA), or even slightly negative ("PACKED WORD" GE and compressed sparse SA). This is because the speeds of ONCHIP_SRAM and the cache are very close to each other. In fact for the 512-bit cache size it has an appreciable negative (slowing) effect for "PACKED

WORD" GE and compressed sparse SA schemes. That is because the cache size is not enough to hold the required amount of data, resulting in frequent cache updates. The higher rate of cache conflict, and lower cache penalty for the ONCHIP_SRAM compared with much larger penalty incurred in the case of DDR2, adversely affects the performance for ONCHIP_SRAM in comparison with DDR2 [24], [25].

Comparing the "PACKED WORD" GE and compressed sparse SA, it can be seen that the former performs better by factors of 3.48 and 4.84, for cache sizes of 0 and 2048 bytes for DDR2. The increase in the performance gap with the cache size indicates the non localized memory access pattern that is associated with the compressed sparse SA data format, where the cache does not help.

4.3 Power and Area

Table 3 shows the power and energy measurements for the matrix inversion and decoding operation, using the SA and GE algorithms, implemented in DDR2, and ONCHIP_SRAM for no cache and 1024 bytes of cache cases. The power and energy are calculated for the case of $K = 1024$.

The total power includes the core dynamic and static powers, and the I/O power of the FPGA that is programmed with soft-core NIOS II processor and executes the codes for SA and GE algorithms. It also includes the power from the external memory chips. The total powers are computed by accurately measuring the current flowing through various operational modules inside the FPGA and memory components on board. The measurement of current is performed through the measurement of voltage across mΩ resistors, on many test points, provided on the development board. The static component of power dissipation was accurately estimated using the Quartus power analyzer tool, and accounts for 50 to 60$\%$ of the power dissipation.

As seen from Table 3, power dissipation does not vary significantly between the two different algorithms and memory mapping schemes, as long as the same memory type is used. This is due to large component of static power dissipation, a common feature of modern CMOS deep submicron technology. From the energy point of view, this feature favors the schemes with the smaller execution time, as energy is the product of time and power. There is a difference between the distinct memory types. The inclusion of cache results in negligible amount of additional power dissipation, although providing significant performance improvement for the DDR2 memory.

The GE algorithm under the "PACKED WORD" memory mapping scheme shows an overwhelming better energy dissipation compared to the other schemes for all memory types considered. The energy rating for "PACKED WORD" GE scheme is 3.45 and 69.75 times better than the compressed sparse and "PACKED WORD" SA schemes for the DDR2 memory, and zero cache size. The corresponding values for the ONCHIP_SRAM memory are 2.84 and 82.80. Also, as expected, as the memory accesses do not go off-chip, the energy rating for all three selected schemes for the ONCHIP_SRAM memory is better than DDR2, by a large margin.

From Table 3 it is also seen that utilization of cache memory marginally improves the energy consumption of all schemes, for the DDR2 memory, a counter intuitive result. This is primarily due to reduction in the execution time with the inclusion of the cache. However, in the case of ONCHIP_SRAM memory the cache is ineffective in reducing the energy metric. In fact there is a small increase in the energy consumption, because the cache degrades the performance for the ONCHIP_SRAM memory while marginally increasing the power dissipation. This is specially true in the case of compressed sparse SA where cache significantly hurts the system performance.

Furthermore, Table 3 gives the hardware resource requirements in terms of logic elements and the memory bits used by the implementations on FPGA for the different memory types. As seen, the addition of cache results in a significant increase in the amount of logic elements to implement the cache controller circuitry.

TABLE 3
Power, Energy and Hardware Resources for the Software Implementation for $K = 1024$ and $T = 4$ Bytes at 100 MHz Processor Speed.

		No Cache		Cache 1024 Bytes	
		DDR2	ONCHIP SRAM	DDR2	ONCHIP SRAM
PACKED WORD GE	Power [mW]	$1,301$	$1,917$	$1,309$	$2,083$
Sparse SA	Power [mW]	$1,287$	$1,917$	$1,312$	$2,069$
PACKED WORD SA	Power [mW]	$1,302$	$1,915$	$1,313$	$2,122$
PACKED WORD GE	Energy [J]	4.255	2.081	2.087	2.546
Sparse SA	Energy [J]	14.661	5.901	9.245	8.084
PACKED WORD SA	Energy [J]	296.797	172.311	118.494	172.913
Hardware	Logic Elements	$4,793$	$5,045$	$6,082$	$6,388$
Resources	Memory Bits	$59,296$	$2,859,296$	$68,352$	$2,868,352$

In summary:

- The "PACKED WORD" mapping has the lowest memory footprint.
- The "PACKED WORD" GE Performs better than compressed sparse SA for all K values.
- Due to non localized memory access pattern associated with compressed sparse SA, the inclusion of data cache is not as effective in improving its performance, as it is for "PACKED WORD" GE.
- "PACKED WORD" GE has a much better energy rating.

5 Hardware Implementation

In Section 3 it was shown that the matrix inversion cum symbol vector decoder in *Code Constraints Processor* in Fig. 2 is the most complex part of the Raptor decoder. To reduce this computational bottleneck, in what follows, a dedicated hardware block for matrix inversion and vector decoding is proposed (*Dedicated Hardware* in Fig. 1), and its performance, power and the required hardware resources are presented and analyzed. As it was shown, the "WORD" memory mapping scheme is not an efficient implementation for the embedded systems. Further, the hardware design of a dynamic link list required for the implementation of compressed sparse SA scheme is too complex for an efficient hardware realization. Therefor, only the "PACKED WORD" schemes are investigated in the sequel. We also again limit the source symbol size $T = 4$ bytes. The dedicated hardware block is supervised by the NIOS II soft-core processor through the Avalon switch fabric in Fig. 1.

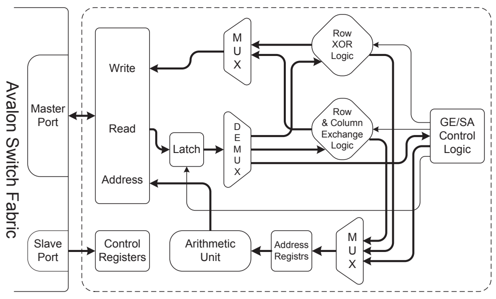


Fig. 7. Hardware accelerator block diagram.

Fig 7 shows the block diagram of the hardware accelerator block. The Avalon switch fabric uses a slave port to set the *Control Registers* that initialize the hardware accelerator. During the initialization the size and initial addresses for matrix and vectors are set. *Control Registers* also control operation of the hardware like initiating *START* and *STOP* commands. The Hardware accelerator block uses an Avalon master port to access the whole memory mapped address space of the NIOS II processor, and send interrupts to NIOS II and receive interrupts from other peripheral devices. For a more flexible design *Row & Column Exchange Logic*, and *Row XOR Logic* units are designed to be self contained units, that interface with the *GE/SA Control Logic* finite state machine. They share a common address generation path through the *Arithmetic Unit* that contains two 32 bit adders and one 16 bit multiplier. The *Latch* circuit only exists in the SA version of the hardware accelerator.

5.1 Memory Interface Enhancement for SA

Consider the "count of 1s" procedure in SA (Algorithm II, *Phase I*, lines 4 to 11). After a memory read access the Avalon switch fabric does not retain the data word just read, but goes into the "high impedance" state until the next read cycle. Since we use the "PACKED WORD" memory mapping we need a mechanism to retain the read data to avoid multiple memory accesses and bit masking to acquire all the bits in a row. In what follows we present the design of a hardware that allows the retention of data read from memory.

5.1.1 Performance Improvement with Enhancement

The proposed hardware block in Fig. 7 for the SA algorithm employs a special circuit that latches the data word read from the memory into the *Latch* until the next word is needed. The subsequent memory accesses for the matrix elements are made to this register, as long as the required entries are in the current word. Fig. 8 depicts the performance of SA with and without the addition of this circuit for all memory types for several values of K. For the purpose of comparison we have also included the plots for the software versions of compressed sparse SA.

It is interesting to see that the simple memory interface enhancement for SA improves the performance by factors of 10.51, and 1.91 for DDR2, and ONCHIP_SRAM, respectively. It can be seen that the enhancement is most effective when DDR2 is used, as accesses to the external DDR2 are much slower [26].

5.1.2 Performance Comparisons with the Software

It should be noted that the performance improvement is solely due to the "count of 1s" operation in *Phase I* of Algorithm II for SA where memory accesses are sequential. However, there is no performance improvement in *Phases II* and *III* of this algorithm, and Algorithm I for GE, where "row exchange", "column exchange" and 'row XOR" operations involve search for matrix elements from non sequential addresses.

In Fig. 8, comparing the plots with enhancement with the corresponding plots for the software version of compressed sparse SA the following trends emerge. For DDR2 the performance of "PACKED WORD" SA in hardware with enhancement falls below the compressed sparse in software for a K value larger than about $6,000$. The corresponding point for ONCHIP_SRAM is about $2,000$. This highlights the domain of validity of each SA scheme. Note that these cross over points are extrapolated and not identified in Fig. 8. That is because on the the chosen platform, larger values of K do not fit in ONCHIP_SRAM.

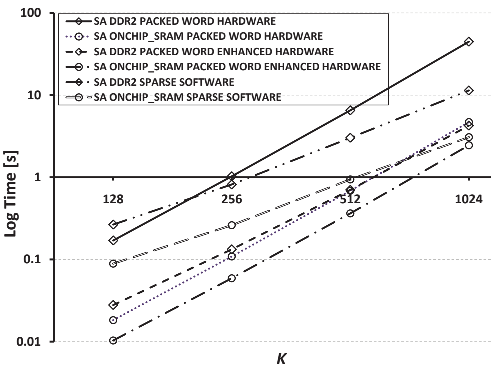


Fig. 8. Performance of the hardware–enhanced Phase I of SA algorithm for various values of K for $T = 4$ bytes, at 100 MHz logic speed.

5.2 Effect of logic clock rate

Fig. 9 shows the performance of GE and SA algorithms under the "PACKED WORD" memory mapping scheme for three different logic clock rates, for the dedicated hardware on the FPGA fabric, and the DDR2 clock rate of 400 MHz for $K = 1024$ and $T = 4$ bytes. Also, included in the plot, for the purpose of comparison, is performance of the software versions of "PACKED WORD" GE scheme (the best performing software scheme), for several processor speeds.

5.2.1 Hardware Performance Comparisons

As can be seen from Fig. 9, "PACKED WORD" SA algorithm benefits far more that the GE in migrating from the software to the hardware platform. This is mainly due to the speed up in the "count of 1s" procedure from the simple enhancement feature discussed before. For example at 100 MHz logic speed, and for "PACKED WORD" schemes, GE performs better than SA by a factor of 2.51 for DDR2. Recall that the similar ratio in the software implementation of two "PACKED WORD" schemes was 69.69. It is worth noting that the similar ratio for the software versions of "PACKED WORD" GE over compressed sparse SA was 3.48.

The enhancement in the "count of 1s" procedure has another effect. From Fig. 9 it is also seen that the increase in the logic speed from 50 to 100 MHz is far more effective in improving the performance of "PACKED WORD" SA. This is due to the fact that the number of slow memory accesses are significantly reduced due to the enhancement, causing the computation to be more dependent on the speed of the logic.

In brief, the SA algorithm benefits most from moving to hardware as a "PACKED WORD" mapping. It also benefits from a faster logic, as it implements the "count of 1s" more efficiently and requires less memory accesses. However, for the chosen values of $K = 1024$ and $T = 4$, it still falls short of the performance of "PACKED WORD" GE.

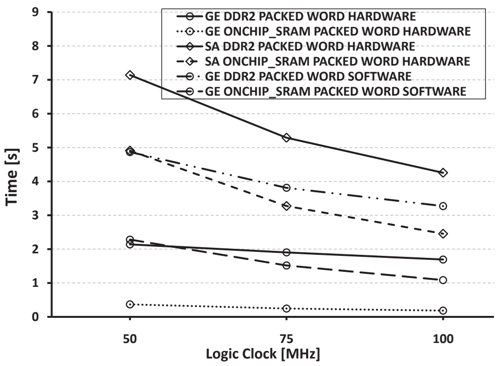


Fig. 9. Matrix inversion and vector decoding in hardware for various logic clock speeds for $K = 1024$ and $T = 4$ bytes.

5.2.2 Performance Comparisons with the Software

Comparing the hardware results with those from the software implementation in Section 4, we can see that at the logic speed of 100 MHz the dedicated hardware block inverts the *Code Constraints Processor* matrix and decodes the symbol vector with the "PACKED WORD" GE scheme 1.93 and 5.90 times faster for DDR2, and ONCHIP_SRAM, respectively, than the equivalent software implementations with no cache.

Fig. 9 also reveals another interesting feature. Comparing the performance of two versions of "PACKED WORD" GE in software for ONCHIP_SRAM, and in hardware for on DDR2 shows that given a choice between going to a faster hardware or to a faster memory, the faster memory wins for all logic/processor speeds more than 50 MHz. That shows that increasing the speed of the processor through the dedicated hardware is less effective if not matched with a higher speed memory–that is the performance of the hardware is memory bound. Comparing compressed sparse SA in software and "PACKED WORD" SA in hardware, the similar improvements in favor of the hardware version are only 2.68, and 1.25.

We have so far demonstrated that migration to a dedicated hardware improves the performance significantly. However, as seen before, we can also improve the performance of software by employing cache. With the inclusion of 1024 bytes of data cache, dedicated hardware versions of the "PACKED WORD" GE on DDR2 and ONCHIP_SRAM perform, respectively, 0.94 and 6.65, times faster than their corresponding software versions. So for the DDR2 memory "PACKED WORD" GE benefits more from the inclusion of the cache than a dedicated hardware. However the 6% improvement in favor of software comes at substantial cost of cache. The system with the dedicated hardware only requires 263 (for DDR2) and 157 (for ONCHIP_SRAM) more logical elements than the system with 1024 bytes of cache. On the other hand, 1024 bytes of cache require 9056 bits of memory, far outweighing the cost of extra logic elements.

For the case of compressed sparse SA software and "PACKED WORD" SA hardware, the similar improvements in favor of the hardware implementation are 1.66, and 1.59. So again we observe that the inclusion of cache works in favor of compressed sparse SA software for DDR2, while still not performing better than the hardware. However, from the case of ONCHIP_SRAM the inclusion of cache degrades the performance of compressed sparse SA software as discussed before.

The performance of the hardware can significantly improve if it is augmented with a similar size cache, something that has not been done in this work.

5.3 Effect of DDR2 SDRAM clock rate

Fig. 10 shows the effect of the DDR2 speed on the two "PACKED WORD" schemes for $K = 1024$ and $T = 4$ bytes. The data is plotted for three different logic clocks - 50, 75 and 100 MHz. Also, included in the plot, for the purpose of comparison, is performance of the software version of "PACKED WORD" GE scheme at 100 MHz processor speed, for several DDR2 speeds.

5.3.1 Hardware Performance Comparisons

From the plots of Fig. 10 it is seen that while both schemes benefit from a faster DDR2 SDRAM, specially at the lower speeds, the effect of memory speed is more noticeable on the operation of the "PACKED WORD" SA scheme. The fact that SA scheme benefits more from a faster logic is also highlighted in Fig. 10.

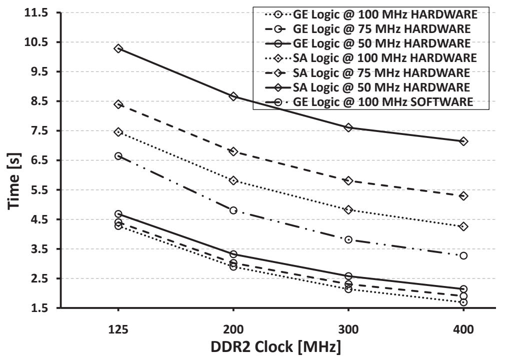


Fig. 10. Effect of DDR2 SDRAM memory speed on "PACKED WORD" GE and SA, for several logic clock rates.

5.3.2 Performance Comparison with the Software

Note that even the software version of "PACKED WORD" GE performs faster than the corresponding hardware version of SA.

5.4 Power and Area

Table 4 shows the power, energy and hardware resource requirements for the FPGA implementation of NIOS II soft-core processor with the proposed dedicated hardware acceleration block for the matrix inversion and vector decoding, running at 100 MHz.

The dedicated hardware resource requirement for the 32-bit symbol size is minimal, requiring little more than $7,084$ logic elements. It also needs $59,296$ (DDR2) and $2,859,296$ (ONCHIP_SRAM) memory bits.

The NIOS II processor is placed in the idle state when the control is passed to the dedicated hardware accelerator. In spite of the addition of a new block the power dissipation is less for all cases, when compared to the corresponding software implementations.

5.4.1 Hardware Power Comparisons

Power consumption is lowest for DDR2 and highest for ONCHIP_SRAM. From the data in Table 4 it is seen that the energy required for the matrix inversion and vector decoding using the "PACKED WORD" schemes, is less for GE than for SA. Saving factors for the case of $T = 4$ bytes (32-bit word size) are 2.46, and 13.14 for DDR2, and ONCHIP_SRAM, respectively.

TABLE 4
Power, Energy and Hardware Resources for the Hardware Implementation for $K = 1024$, $T = 4, 8, 16$ Bytes for 100 MHz Logic Speed.

		T= 4 bytes		*T*= 8 bytes		*T*= 16 bytes	
		DDR2	ONCHIP SRAM	DDR2	ONCHIP SRAM	DDR2	ONCHIP SRAM
PACKED WORD GE	Power [mW]	$1,279$	$2,041$	$1,296$	$2,069$	$1,318$	$2,062$
PACKED WORD SA	Power [mW]	$1,250$	$2,008$	$1,256$	$2,035$	$1,280$	$2,035$
PACKED WORD GE	Energy [J]	2.165	0.375	1.5	0.245	1.137	0.173
PACKED WORD SA	Energy [J]	5.32	4.929	4.513	4.894	4.166	4.843
Hardware	Logic Elements	$7,084$	$7,774$	$7,839$	$8,815$	$9,031$	$9,940$
Resources	Memory Bits	$59,296$	$2,859,296$	$68,608$	$2,868,608$	$87,264$	$2,887,264$

5.4.2 Power Comparison with the Software

There is also little difference in the power dissipation between the hardware and software implementations with or without cache. The dedicated hardware block for "PACKED WORD" GE needs several times less energy to invert the same matrix and decode the symbol vector compared to the pure software implementation without a cache, and 32-bit symbol size. The saving factors are 1.97, and 5.54 for data stored in DDR2, and ONCHIP_SRAM, respectively.

Comparing the power dissipation between the software compressed sparse, and hardware "PACKED WORD" versions of SA, the saving factors in favor of the hardware version are 2.76 and 1.20, indicating that for $K = 1024$ it is advantageous from the performance and energy dissipation point of view to move the highly efficient software version of compressed sparse SA to its corresponding "PACKED WORD" version in the dedicated hardware.

The energy profiles with the inclusion of a 1024-byte cache track their corresponding performance ratios discussed before to within 10%.

In summary:

- The "PACKED WORD" SA algorithm benefits most in migrating from software to hardware, due to memory interface enhancement feature in the count of 1s" procedure. However, "PACKED WORD" version GE stills win over its SA version even in hardware.
- Both "PACKED WORD" schemes benefit similarly from faster DDR2 memory. However, "PACKED WORD" SA benefits much more from a faster logic speed.
- For the "PACKED WORD" GE a choice of software with cache performs slightly better than migrating to hardware. However, this comes at the substantial cost of cache hardware.
- "PACKED WORD" GE needs several times less energy to invert the same matrix and decode the symbol vector compared to the pure software implementation.

6 Analysis for Larger Symbol Size

In the forgoing sections we had used a symbol size of $T = 4$ bytes. In this section we show the effect of increasing the size of T. Note that, as said before, in the 32-bit NIOS II system the memory accesses are through an $8 - bit$ bus in burst mode, and a 32-bit bus, respectively, for the external DDR2 and ONCHIP_SRAM. On the other hand, in the dedicated hardware, internally the data path is as wide as the size of the symbol T. However, the memory access to the DDR2 is still through an $8 - bit$ bus, while for ONCHIP_SRAM the memory access is through a bus as wide as the size of T.

6.1 Software Implementation

Fig. 11 and Fig. 12 show the performance of the "PACKED WORD" GE and the compressed sparse SA schemes for several values of K and T for DDR2 and ONCHIP_SRAM, respectively. It is seen that for small values of $T = 4$ and 16 the performance of the "PACKED WORD" GE is better than the compressed sparse SA schemes. For the larger value of $T = 128$ the performance plots turn in favor of compressed sparse SA. The performance cross over points (not shown in Fig. 11 and Fig. 12) are about $T = 80$ and $T = 120$ for $K = 128$ DDR2 and ONCHIP_SRAM, respectively. For $K = 1024$ the overspending cross over points are reduced to $T = 50$ and $T = 38$.

Not easily apparent from Fig. 11 and Fig. 12 is that for each value of K the execution time is a linear function of the symbol size T. Further, the rate of increase with T exponentially grows with K, with GE having a much higher growth rate.

The set of observations made in the last two paragraphs can be explained as follows. For the compressed sparse SA scheme, the higher complexity of matrix inversion operation is amortized for all values of T, initially less significant part of decoding time, but loosing its significance as T grows. This is where we observe a cross over in the performance plots of GE and SA, indicating that the execution time of GE grows at a higher rate with T than that of SA. The higher growth rate with T for the "PACKED WORD" GE compared with SA is due to the fact that the reduced complexity of the matrix inversion is heavily paid for with the substantial increase in the number of operations on the symbol vector. Inversely, the number of row operations on the symbol vector is substantially reduced for the SA scheme, therefore, resulting in much slower rate of increase in the execution time with T for SA with respect to GE. Further, since the number of row operations grows exponentially with K, we expect a higher rate of increase in the execution time with T as K grows.

Further, both the exponential rate of growth with K and linear growth with T are much higher for DDR2 than for ONCHIP_SRAM, as should be expected.

6.2 Hardware Implementation

Avalon interconnect easily allows for devices with different word sizes to operate together. The on-chip

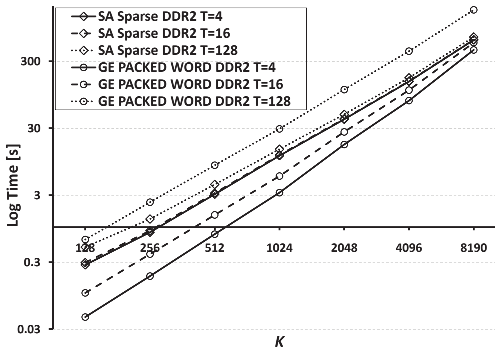


Fig. 11. Effect of T on the performance of the "PACKED WORD" GE and the compressed spared SA, for DDR2 SDRAM memory.

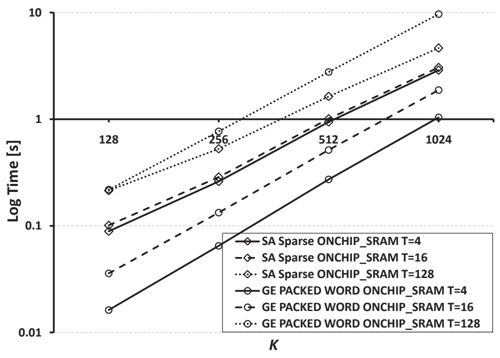


Fig. 12. Effect of T on the performance of the "PACKED WORD" GE and the compressed spared SA, for ONCHIP_SRAM memory.

SRAM can be configured with wider word sizes. External memories can be connected in parallel to achieve the same result. Within the two algorithms under investigation, the most common operations are the "row exchange" and "row XOR". Both algorithms would strongly benefit from the wider word size memory accesses. This feature allows us to design codes with larger symbol sizes T. In our hardware implementation we also increase the word size that holds the rows of the *Code Constraints Processor* matrix. That means that the "row exchange" and "row XOR" operations can be performed in less number of steps as the word size increases.

6.2.1 Hardware Performance Comparisons

Fig. 13 shows the performance of "PACKED WORD" GE and SA for DDR2 for $K = 1024$ with three different symbol sizes - 4-byte (32-bit), 8-byte (64-bit) and 16-byte (128-bit). Performance of both schemes improve with the wider symbol size, with GE having a higher rate of improvement. At 100 MHz logic clock speed GE outperforms SA 2.51 times for 32-bit, 3.10 times for 64-bit, and 3.77 times for 128-bit symbol size implementations. The corresponding numbers for ONCHIP_SRAM, as shown in Fig. 14 are 13.34, 20.32 and 28.41. Also note that with the faster ONCHIP_SRAM memory the improvement factors are 4 to 7 times more than those obtained for DDR2.

Noting the trends in plots of Fig. 13 and Fig. 14 carefully, we see that the relative improvement in the performance with the symbol size rapidly saturates. This is more noticeable for the case of ONCHIP_SRAM and the "PACKED WORD" SA scheme. This observation can be explained as follows.

Looking at the plots in Fig. 14 for ONCHIP_SRAM, the slight improvement in the performance is independent of the symbol size, as all bits in the symbol are processed in parallel, irrespective of its size. The improvement comes from the faster "row exchange" and "row XOR" operations due to the longer word sizes. Since the number of row operations in "PACKED WORD" SA is less than that of GE, and primarily relies on the "column exchange" operation, it benefits less from longer word sizes. Even in the case of GE with longer word sizes, the performance bottleneck will shift from the "row exchange" and "row XOR" operations to the "column search" operations that do not benefit from the longer word sizes.

In the case of the plots in Fig. 13 for DDR2 while internally the hardware works on the longer word sizes for the matrix rows and larger symbol sizes, accesses to the memory are still sequential. Therefore, higher performance with the longer word and symbol sizes come from the burst mode nature of DDR2 accesses. So memory transfers for both matrix elements and symbols perform faster with the wider word and symbol sizes due to the burst nature of DDR2. Therefore the performance saturation in the case of DDR2 is more gradual.

The real power of hardware with the larger symbol size is that it comes at no extra execution time, as all bits in a symbol are processed in parallel. That is not the case for the software implementation schemes as we saw before. In effect, for the same $K = 1024$, we decode $2/\ 4$ times more data when we extend T from 4 bytes to $8/\ 16$ bytes. Further, noting that in the "PACKED WORD" implementation we also extend the word-length for the matrix inversion to the same size as T; for the GE scheme the execution time of 128-bit symbol size implementations performs 1.96 and 2.20 times faster than the 32-bit implementation, for DDR2 and ONCHIP_SRAM, respectively. This corresponds to a 2-fold performance improvement for a 4-fold increase in the amount of data decoded; an 8-fold effective increase in the decoding rate. This is in fact an impressive result! The similar performance improvements for the "PACKED WORD" SA scheme are 1.30 and 1.03.

6.2.2 Performance Comparisons with Software

Table 5 compares the performance of software versions of compressed SA and "PACKED WORD" GE, with the hardware versions of "PACKED WORD" SA and GE, across a wide range of symbol sizes ($T = 4$, 16, 128 bytes) for $K = 1024$ for DDR2.

As was observed before, while the execution time of the software schemes increase linearly with T the execution time of the hardware reduces to lower floor

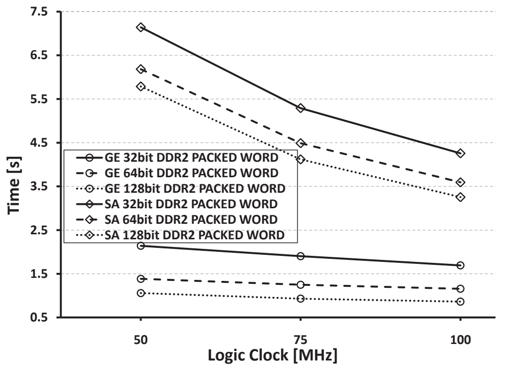


Fig. 13. Effect of symbol size on GE and SA for DDR2 SDRAM.

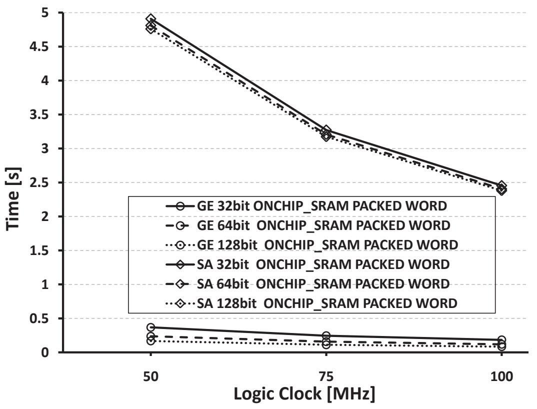


Fig. 14. Effect of symbol size on GE and SA for ONCHIP_SRAM.

around $T = 480$ bits and starts increasing at a slow rate. This results in the ratios of the executions times of software and hardware versions to grow first at a square rate and then taper off at a linear rate. For example the hardware version of "PACKED WORD" GE performs better than its software version, by factors of 1.94, 6.73 and 29.58, respectively, for the symbol sizes of $T = 32$ bits, $T = 128$ bits, and $T = 1024$ bits.

TABLE 5
Comparison Between Hardware and Software Performance for $K = 1024$, $T = 4, 16, 128$ Bytes for 100 MHz Logic Speed for DDR2 memory.

Time [s]	T= 4 bytes	T= 16 bytes	T= 128 bytes
Hardware PACKED WORD GE	1.692	0.862	0.992
Hardware PACKED WORD SA	4.256	3.254	3.255
Software PACKED WORD GE	3.275	5.797	29.34
Software Sparse SA	11.68	11.96	14.59

Similarly, hardware version of "PACKED WORD" SA performs better than its compressed sparse version in software, by factors of 2.79, 3.68 and 4.45, for the corresponding set of bits. In addition, the corresponding ratios of performance of the hardware version of "PACKED WORD" GE over the compressed sparse SA in software, are 7.01, 13.87 and 14.60. To illustrate the power of the parallel processing note that the corresponding ratios when comparing the software versions, "PACKED WORD" GE performs better than compressed sparse SA by factors of 3.62, 2.06 and 0.49.

The interesting observation to be made is that the performance ratios of the software version of compressed sparse SA and the hardware versions of “PACKED WORD” GE and SA grows with T. Therefore, there is no cross over point in the performance, as we observe in comparing the software versions of compressed sparse SA and “PACKED WORD” GE!

It is also worth mentioning that the reason for lower flooring (around $T = 480$ bits), and the subsequent increases in the execution times for the hardware versions of “PACKED WORD” GE and SA is that while the row operations are performed in parallel in the hardware, DDR2 accesses are still sequential burst mode. So, around $T = 480$ bits matrix inversion forms a small part of the total execution time, and the time required for the memory accesses will dominate.

6.3 Power and Area

6.3.1 Hardware Power and Area Comparisons

The dedicated hardware is designed to be easily scalable, provided proper memory alignment is possible. From the data in Table 4 presented before, it is seen that it takes only 755 (DDR2) and 1041 (ONCHIP_SRAM) more logic elements for the 64-bit symbol size, and 1947 (DDR2) and 2166 (ONCHIP_SRAM) more logic elements for the 128-bit word size, compared to the 32-bit word size.

Also from Table 4 we observe that there is only marginal increase in the power dissipation for wider symbol sizes. Note that for 32-bit, 64-bit and 128-bit symbol sizes, the energy required for the matrix inversion and vector decoding using the “PACKED WORD” schemes, is less for GE compared with SA. The energy ratios between these two“PACKED WORD” schemes increases with the symbol size. For example the relative saving factors in favor of GE for the case of 128-bit symbol size are 3.66, and 27.99 for DDR2, and ONCHIP_SRAM, respectively. These ratings are markedly better than the corresponding values of 2.46 and 13.14 for the 32-bit symbol size.

Comparing the energy consumption of 128-bit symbol sizes with that of the 32-bit “PACKED WORD” GE scheme, the saving factors are 1.90 and 2.17 for DDR2 and ONCHIP_SRAM, respectively; a 2-fold reduction in energy dissipation for an 8-fold increase in the effective decoding rate!

6.3.2 Power Comparison with the Software

Recall that for the 32-bit symbol size “PACKED WORD” GE, with no cache, the saving factors in favor of hardware were 1.97, and 5.54 for data stored in DDR2, and ONCHIP SRAM, respectively. For the 64-bit symbol size , the energy reduction factors compared to the 32-bit symbol size software implementation, are improved to 2.84 and 8.50 for DDR2, and ONCHIP_SRAM, respectively. For 128-bit the corresponding saving factors increase to 3.74 and 12.03 for “PACKED WORD” GE. It is interesting to see that both the energy saving factor and the effective decoding rate, progressively, improve for the “PACKED WORD” GE with the symbol size: an amazing efficiency of parallel hardware. Further, noting that for the software implementation energy is proportional to the decoding time, the energy saving factor of hardware over software, for 128-bit symbol size, for DDR2 is 3.8.

The corresponding improved saving factors for “PACKED WORD” SA over the compressed sparse SA in software are 2.76 and 1.20 for 32-bit, 3.25; 1.21 for 64-bit, and 3.52; and 1.22 for 128-bit symbol sizes, respectively, showing that the improvement in energy saving saturates with the symbol size for ONCHIP_SRAM.

In summary:

- In software “PACKED WORD” GE will start lagging compressed sparse SA for T values larger than 80.
- “PACKED WORD” GE implemented in hardware performs better than compressed sparse SA in software for all values of T.
- Hardware implementation allows for internal parallel processing using the wider than 32-bit word sizes, even though DDR2 memory access are still 8-bit wide sequential.
- Using wider word sizes improves the performance of both matrix inversion and vector decoder operations, resulting in impressive increase in the effective decoding rate.
- The impressive performance improvement with the wider word sizes, results in similar simultaneous improvement in energy efficiency.

7 Conclusion

This paper has evaluated the performance of Raptor decoder on a typical embedded system. The performance of two matrix inversion and vector decoding algorithms on software and hardware implementation platforms have been presented. The effect of storing data on different memory types have been analyzed for both algorithms under software and hardware implementations. Three memory mapping schemes have been investigated with clear benefits of the adoption of the “PACKED WORD” scheme demonstrated. It was found that contrary to the 3GPP recommendation, based on the profiling on a workstation platform, the Gaussian elimination performs significantly better than the alternative reportedly more efficient implementation, in terms of execution speed and energy saving when it is implemented in hardware. It also requires less logic elements and is much more suitable for hardware implementation. Under software implementation GE performs better then SA for smaller values of T. Nevertheless, the SA algorithm outperforms GE in software for values of T larger then $T = 128$. The effect of cache memory size on performance and energy dissipation was investigated. The addition of the cache, while significantly improves the performance, does not increase the energy dissipation. On the hardware platform the use of longer symbol sizes reduces the computation time, and energy consumption, while improving the decoding rate by a large factor.

Acknowledgments

This work was supported in part by The National IT Industry Promotion Agency of Korea.

References

[1] A. Shokrollahi, "Raptor codes," *IEEE Trans. Inf. Theory*, vol. 52, pp. 2551–2567, June 2006.

[2] D. J. C. Mackay, "Fountain codes," *IEE Proc. Commun.*, vol. 152, no. 6, pp. 1062–1068, Dec. 2005.

[3] J. Byers, M. Luby, and M. Mitzenmacher, "A digital fountain approach to asynchronous reliable multicast," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 8, pp. 1528–1540, Oct 2002.

[4] M. Luby, "LT codes," in *Proc., 43rd Annual IEEE Sym. on Foundations of Computer Science*, Nov. 2002, pp. 271–280.

[5] M. Luby, M. Watson, T. Gasiba, T. Stockhammer, and W. Xu, "Raptor codes for reliable download delivery in wireless broadcast systems," in *Proc., Third IEEE Consumer Communications and Networking Conf.*, vol. 1, Jan. 2006, pp. 192–197.

[6] M. Luby, T. Gasiba, T. Stockhammer, and M. Watson, "Reliable multimedia download delivery in cellular broadcast networks," *IEEE Trans. Broadcast.*, vol. 53, no. 1, pp. 235–246, Mar. 2007.

[7] T. Gasiba, T. Stockhammer, and W. Xu, "Reliable and efficient download delivery with Raptor codes," in *Proc., Fourth Int. Sym. on Turbo Codes & Related Topics, Munich*, Apr. 2006.

[8] *3GPP TS 26.346, Technical Specification Group Services and System Aspects; Multimedia Broadcast/Multicast Service (MBMS); Protocols and codecs*, 3GPP Technical Specification, Rev. V7.4.1, June 2007.

[9] *Digital Video Broadcasting (DVB); IP Datacast over DVB-H: Content Delivery Protocols*, ETSI Technical Specification, Rev. V1.2.1, 2006.

[10] N. Elarief and B. Bose, "Diversity combining ARQ over the m($\geq$ 2)-ary unidirectional channel," *IEEE Trans. on Comput.*, vol. 58, no. 8, pp. 1026–1034, Aug. 2009.

[11] P. Cataldi, M. P. Shatarski, M. Grangetto, and E. Magli, "Implementation and performance evaluation of LT and raptor codes for multimedia applications," in *Proc., IEEE Int. Conf. on Intelligent Information Hiding and Multimedia Signal Processing, (IIH-MSP)*, Dec. 2002.

[12] O. Etesami and A. Shokrollahi, "Raptor codes on binary memoryless symmetric channels," *IEEE Trans. Inf. Theory*, vol. 52, no. 5, pp. 2033–2051, 2006.

[13] Z. Cheng, J. Castura, and Y. Mao, "On the design of Raptor codes for binary-input Gaussian channels," *IEEE. Trans. Commun.*, vol. 57, no. 11, pp. 3269–3277, Nov. 2009.

[14] K. Hu, J. Castura, and Y. Mao, "Reduced-complexity decoding of Raptor codes over fading channels," in *Proc., IEEE Global Telecommunications Conf. (GLOBECOM)*, Nov. 2006, pp. 1–5.

[15] ——, "Performance-complexity tradeoffs of Raptor codes over Gaussian channels," *IEEE Commun. Lett.*, vol. 11, no. 4, pp. 343–345, Apr. 2007.

[16] J. Heo, S. Kim, J. Kim, and J. Kim, "Low complexity decoding for Raptor codes for hybrid-ARQ systems," *IEEE Trans. Consum. Electron.*, vol. 54, no. 2, pp. 390–395, may 2008.

[17] A. A. Hussein, A. Oka, and L. Lampe, "Decoding with early termination for Raptor codes," *IEEE Commun. Lett.*, vol. 12, no. 6, pp. 444–446, June 2008.

[18] S. Kim, S. Lee, and S.-Y. Chung, "An efficient algorithm for ML decoding of Raptor codes over the binary erasure channel," *IEEE Commun. Lett.*, vol. 12, pp. 578–580, Au. 2008.

[19] X. Yuan and L. Ping, "Quasi-systematic doped LT codes," *IEEE J. Sel. Areas Commun.*, vol. 27, no. 6, pp. 866 –875, Aug. 2009.

[20] S. Lin and D. J. Costello, *Error Control Coding*. Englewood Cliffs, N.J.: Prentice Hall, 2004.

[21] R. E. Blahut, *Algebraic Codes for Data Transmission*. The Edinburgh Building, Cambridge: Cambridge University Press, 2003.

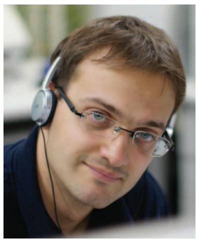
[22] W. H. Press, *Numerical Recipes: The Art of Scientific Computing*. The Edinburgh Building, Cambridge: Cambridge University Press, 2007.

[23] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein, "An efficient parallel heap compaction algorithm," *ACM SIGPLAN Not.*, vol. 39, no. 10, pp. 224–236, Oct. 2004.

[24] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufman Publishers, 2003.

[25] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, and H. De Man, "Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications," *IEEE Trans. on Comput.*, vol. 54, no. 1, pp. 76–81, Jan. 2005.

[26] F. Poletti, A. Poggiali, D. Bertozzi, L. Benini, P. Marchal, M. Loghi, and M. Poncino, "Energy-efficient multiprocessor systems-on-chip for embedded computing: Exploring programming models and their architectural support," *IEEE Trans. Comput.*, vol. 56, no. 5, pp. 606–621, May 2007.



Todor Mladenov (M06) received the B.E. degree in communications technologies from Technical University Sofia, Bulgaria, in 2005, and the M.S. degree in information and communications from Gwangju Institute of Science and Technology (GIST), Republic of Korea, in 2007. Currently, he is pursuing his Ph. D. with the Department of Information and Communications, Gwangju Institute of Science and Technology, Republic of Korea. His research interests include the design of low power, high speed application specific circuits and systems for multimedia, communications and information theory, channel coding and computing in its broad sense.

Saeid Nooshabadi (M01SM07) received the MTech and PhD degrees in electrical engineering from the India Institute of Technology, Delhi, India, in 1986 and 1992, respectively. Currently, he is the professor of Computer Systems Engineering with Department of Electrical and Computer Engineering, Michigan Technological University, Michigan. Prior to his current appointment he has held multiple academic and research positions. His last two appointments were with the Department of Information and Communications, Gwangju Institute of Science and Technology, Republic of Korea (2007 to 2010), and with the School of Electrical Engineering and Telecommunications, University of New South Wales, Sydney, Australia (2000 to 2007). His research interests include VLSI information processing and low-power embedded processors.

Kiseon Kim received the B.Eng. and M.Eng. degrees, in electronics engineering, from Seoul National University, Korea, in 1978 and 1980, and the Ph.D. degree in electrical engineering-systems from University of Southern California, Los Angeles, in 1987. From 1988 to 1991, he was with Schlumberger, Houston, Texas. From 1991 to 1994, he was with the Superconducting Super Collider Lab, Texas. He joined Gwangju Institute of Science and Technology (GIST), Korea, in 1994, where he is currently a professor. His current interests include wideband digital communications system design, sensor network design, analysis and implementation both at the physical layer and at the resource management layer.