

SIMD Implementation of Interpolation in Algebraic Soft-Decision Reed-Solomon Decoding

Laurier Boulianne and Warren J. Gross

Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
Email: {lboulian,wjgross}@macs.ece.mcgill.ca

Abstract—The Koetter-Vardy algorithm is an algebraic soft-decision decoding algorithm for Reed-Solomon codes. Software implementations of the Koetter-Vardy algorithm are considered as part of a redecoding architecture that augments a hardware hard-decision decoder with soft-decision decoding software on an embedded processor. In this paper we investigate the implementation of the interpolation step of the Koetter-Vardy algorithm on SIMD processor architectures. A parallelization of the algorithm is given using the K 'th order Horner's rule for parallel polynomial evaluation. The SIMD algorithm has a running time 2.5 to 4 times faster than a serial implementation on a DSP processor. To gain further speedup we propose a merged-SIMD architecture that calculates the Hasse derivative in parallel with the polynomial updates.

I. INTRODUCTION

Reed-Solomon (RS) codes are a popular class of error-correcting codes found in many communications and recording systems. While classical RS decoders operate on hard inputs, using *soft* reliability information at the input of a Reed-Solomon decoder can provide up to 2 dB of gain over classical hard-decision decoders on AWGN channels. This soft-information can come directly from the channel or from an inner decoder in a concatenated coding scheme.

The recent introduction by Koetter and Vardy of an algebraic soft-decision decoding algorithm [1] based on the Guruswami-Sudan (GS) list decoding algorithm [2] has reignited interest in the soft decoding of Reed-Solomon codes. Recent work [3–8] has begun to focus on VLSI implementations of the Koetter-Vardy (KV) algorithm. In VLSI implementation, the main bottleneck is the complexity of bivariate polynomial interpolation.

In this paper we consider the problem of the efficient software implementation of the interpolation step of the KV algorithm. Motivated by the complexity of the KV algorithm as compared to classical hard-decision decoders, we use an approach that only uses soft-decision decoding when needed resulting in a low-complexity architecture. The throughput requirements of the soft-decision processor are therefore relaxed, opening up the possibility of software implementations. SIMD implementations of the interpolation algorithm are explored on a processor with a fixed number of Galois field arithmetic units. We then propose a SIMD architecture tailored to the specifics of the interpolation algorithm.

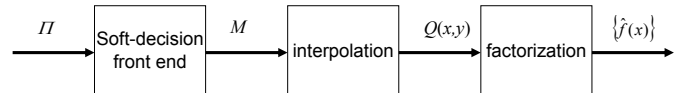


Fig. 1. The Koetter-Vardy algebraic soft-decision algorithm.

This paper is organized as follows. Section II gives a brief introduction to the soft-decision Koetter-Vardy decoding algorithm. In Section III we describe a hardware-software approach to implementing the KV algorithm. Section IV describes SIMD implementations of the interpolation algorithm on a DSP processor and proposes the “merged” architecture for SIMD interpolation. Section V offers conclusions.

II. ALGEBRAIC SOFT-DECISION DECODING

The KV algebraic soft-decision decoding algorithm is based on the list decoder of Guruswami and Sudan [2] with the addition of a *soft-decision front-end* [1]. This algorithm takes an approach based on the interpolation of a bivariate polynomial.

A Reed-Solomon code over the finite field with q elements, $\text{GF}(q)$, encodes messages of k symbols drawn from $\text{GF}(q)$ into n -symbol codewords. If the k symbols of the message are f_0, f_1, \dots, f_{k-1} , then we can construct a *message polynomial* as $f(x) = \sum_{i=0}^{k-1} f_i x^i$. The Reed-Solomon encoder calculates a codeword C as the evaluation map

$$C = (f(1), f(\alpha), f(\alpha^2), \dots, f(\alpha^{q-2})). \quad (1)$$

The evaluation map encoding suggests an interpolation-based decoding algorithm. Although the decoding algorithm will make use of the evaluation map representation of RS codes, the encoder can be implemented in practice using the standard shift register encoder.

The input to the algorithm is a $q \times n$ *reliability matrix* of probabilities for each of the q possible transmitted symbols at each of the n code positions, where $n = q - 1$.

A. The Decoding Algorithm

A bivariate polynomial over $\text{GF}(q)$ is defined as:

$$P(x, y) = \sum_{u=0}^{d_x} \sum_{v=0}^{d_y} p_{u,v} x^u y^v, \quad (2)$$

where the coefficients $p_{u,v} \in \text{GF}(q)$. Define the (w_x, w_y) -weighted degree of $P(x, y)$ as the maximum value of $uw_x + vw_y$ such that $p_{u,v} \neq 0$.

Given a reliability matrix Π , the decoding algorithm for an (n, k) RS code consists of three main steps as illustrated in Figure 1:

- 1) **Soft-Decision front end:** Choose a real-valued parameter $\phi > 0$. Calculate an integer $q \times n$ *multiplicity matrix* proportional to Π with entries $m_{i,j} = \lfloor \phi \pi_{i,j} \rfloor$. The maximum multiplicity in M is $m_{\max} = \lfloor \phi \rfloor$.
- 2) **Interpolation:** Each index (i, j) of M represents an interpolation point (x_j, y_i) , where $x_j = \alpha^j, j = 0, \dots, q-2$, and $y_0 = 0, y_i = \alpha^{i-1}, i = 1, \dots, q-1$. Perform a weighted interpolation through the points represented by the non-zero entries of M to find the bivariate polynomial of minimal $(1, k-1)$ -weighted degree $Q(x, y)$ which passes through the interpolation points with at least the given multiplicity at each point.
- 3) **Factorization:** Factor the interpolation polynomial $Q(x, y)$ to determine the list of factors of the form $y - f(x)$ with $\deg f(x) < k$. Output the list of candidate messages $\{\hat{f}(x)\}$. Usually, the number of entries on this list is small, most often just one.

The soft-decision step described above [9] is a low-complexity implementation that calculates an equivalent multiplicity matrix as the original algorithm given in [1]. The parameter m_{\max} plays a central role in the decoding. Larger values of m_{\max} provide a greater error-correcting capability at the expense of increased computational complexity.

B. Reduced-Complexity Decoding

The algorithm given above is, in its native form, not well-suited to practical implementation due to both the large number of constraints to be satisfied in the interpolation step and the required memory storage. In this paper, we focus on a simplified version of the algorithm based on the *reencoding* and coordinate basis transformations [3, 8, 10, 11] that reduce the number of iterations and memory requirements to practical levels. The resulting simplified soft-decision interpolation algorithm is run on $O(n-k)$ interpolation points instead of $O(n)$.

III. HARDWARE-SOFTWARE ARCHITECTURE

Recently, there has been much interest in practical implementations of the KV algorithm. Software implementations have been developed taking advantage of reencoding and coordinate transformation that are capable of decoding rates on the order of 1 Mbps for the RS(255,239) code with $m_{\max} = 4$. For the much higher data rates required in high-speed networking and magnetic recording, high-throughput VLSI architectures have been proposed [4–6]. These architectures achieve high decoding rates with massive parallelism and require substantial hardware resources to implement.

In this paper, we take a different approach. If we consider that the RS decoder is usually run at moderate to low frame

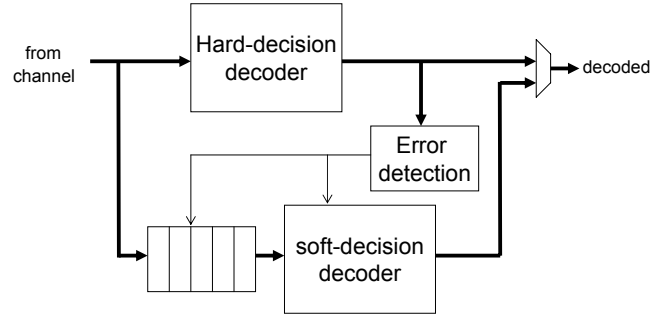


Fig. 2. The redecoding architecture.

error rates, most frames are decoded correctly by a hard-decision decoder. Even at the relatively high error rate of 10^{-3} , only 1 in 1000 frames fail hard-decision decoding and is a candidate for soft-decision decoding. Therefore, it may make sense to not run a massively parallel, high-power soft-decision decoder on each received frame. The idea of a two-stage *redecoding* procedure with a first-pass hard-decision decoding, and second-pass soft-decision decoding was proposed and studied in [9, 12]. A soft-decision decoder based on redecoding is shown in Figure 2. Input frames, are decoded with a hard-decision decoder in a first pass. Most frames will be decoded correctly by this decoder. The decoded codeword is then passed through an error detector which determines if the frame was likely correctly decoded. This detector can be a CRC [12] which is convenient if it is already performed as part of the communications standard. Other error detectors that can be used in a redecoding architecture have been proposed [9].

Words that fail the error-detection test (or have a decoding failure indicated by the hard-decision decoder) are then redecoded by a soft-decision decoder. The hard decision decoder resumes decoding the incoming frames while the soft-decision decoder decodes the failed frame in parallel. In most practical cases, the arrival rate of frames to the soft-decision decoder is very low (for example 10^{-3} or lower) and therefore, the soft-decision decoder has a relatively long time to decode a frame. This means that the decoding rate and hence the complexity of the soft-decision decoder can be dramatically reduced, to the point where a software implementation on an embedded processor might be feasible. A queue is required at the input to the soft-decision decoder to hold frames that fail the hard decoding while the soft decoder is busy, however, on an AWGN channel, the length of this queue is at most 4 or 5 frames of soft-information in practice [9]. We should note that the redecoding architecture is restricted to applications that do not need to guarantee a decoding latency for a given frame and that can tolerate out-of-order frame delivery to the upper layers. In these applications, redecoding is probably the preferred method of introducing soft-decision decoding with a reasonable amount of additional complexity.

Proc.	Arch.	GFMULTs	IP/OP (bits)	IP/OP (elems)
NIOS-II	RISC	N/A	$2 \times 32 / 32$	$8 / 4$
Microblaze	RISC	N/A	$8 \times 32 / 8 \times 32$	$32 / 32$
LEON2	RISC	N/A	$2 \times 64 / 64$	$16 / 8$
C6416	VLIW	2×4	$2 \times (2 \times 32/32)$	$2 \times (8/4)$

TABLE I
EXAMPLES OF EMBEDDED PROCESSORS.

IV. SIMD IMPLEMENTATION OF INTERPOLATION

In this section we focus on software implementation of the interpolation algorithm which is the step with the greatest complexity and can represent up to 60% of the running time of the KV algorithm [8].

A. The Interpolation Algorithm

The input to the algorithm is the set of L triples $(x_j, y_j, m_j), j = 0, \dots, L-1$, where x_j and y_j are elements of $\text{GF}(q)$ and m_j is a positive integer. On each iteration, $d_y + 1$ bivariate polynomials are updated, where d_y represents the maximal y -degree of the polynomials. The algorithm is shown in Algorithm 1 assuming reencoding and coordinate transformation. The algorithm is shown for binary extension fields ($\text{GF}(q = 2^\beta)$) where Galois field subtraction is equivalent to Galois field addition. The heart, and most computationally complex parts of the algorithm are the three polynomial operations, HDXY, POLY1XY, and POLY2XY, shown in boxes. Define the (μ, ν) 'th Hasse derivative of a bivariate polynomial $P(x, y)$ as

$$P^{[\mu, \nu]}(x, y) = \sum_{u \geq \mu} \sum_{v \geq \nu} \binom{u}{\mu} \binom{v}{\nu} p_{u,v} x^{u-\mu} y^{v-\nu}. \quad (3)$$

The rest of the algorithm consists of loop control, index updating and other simple control structures.

B. Processor Architecture

The polynomial operations are over Galois fields of the form $\text{GF}(2^\beta)$. In these fields, Galois field elements are represented as bit vectors with an element of $\text{GF}(256)$ being one byte. We assume the field $\text{GF}(256)$ in this paper but the results easily scale to other fields. Representations of elements using the standard or composite [13] bases implement Galois field addition (GFADD) with an XOR operation performed by the processor's ALU. Multiplication and inversion are more involved operations and therefore embedded software implementations of Reed-Solomon decoders usually implement them with lookup tables. Since Galois field multiplication (GFMULT) is one of the basic operations in the interpolation algorithm, it makes sense to augment the processor with one or many single-cycle hardware GFMULT units (inversion is more sparsely used and it therefore can be implemented using a lookup table).

We consider simple embedded processors with a RISC or VLIW architecture. Assume that the processor has a coprocessor interface or other similar low-latency bus that gives direct access to the register file. Examples of processor architectures

Algorithm 1 Interpolation algorithm. The polynomial operations are shown in boxes.

INPUT: A set of triples $\{(x_j, y_j, m_j)\}, j = 0 \dots L-1$

INIT:

$G \leftarrow \{g_0 = 1, g_1 = y, g_2 = y^2, \dots, g_{d_y} = y^{d_y}\}$

$D \leftarrow \{d_0 = 0, d_1 = -1, d_2 = -2, \dots, d_{d_y} = -d_y\}$

for each point (x_j, y_j) with multiplicity m_j **do**

for $\mu = 0$ to $m_j - 1$ **do**

for $\nu = 0$ to $m_j - 1 - \mu$ **do**

for $i = 0, \dots, d_y$ **do**

HDXY : $h_i = g_i^{[\mu, \nu]}(x_j, y_j)$

end for

$\Delta = \{i : h_i \neq 0\}$

$\delta = \text{argmin}\{d_i\}$

$i \in \Delta$

for $i = 0, \dots, d_y$ **do**

if $i \neq \delta$ **then**

POLY1XY : $g_i(x, y) \leftarrow h_\delta \cdot g_i(x, y) + h_i \cdot g_\delta(x, y)$

end if

end for

POLY2XY : $g_\delta(x, y) \leftarrow (x + x_j)g_\delta(x, y)$

$d_\delta \leftarrow d_\delta + 1$

end for

end for

end for

$\delta = \text{argmin}_i \{d_i\}, i = 0, \dots, d_y$

OUTPUT: $P(x, y) = g_\delta(x, y)$

that might be suitable for this sort of architecture are given in Table I. Indicated are the processor name, architecture and the number of built in GFMULT units. The number of input and output bits to the Galois field hardware is indicated. These numbers are also stated as the number of GF elements this represents, assuming that a GF element is a byte is taken from $\text{GF}(256)$. We have chosen these processors as examples because they either already have built-in GFMULT units (TI C6416 DSP) or, have low-latency coprocessor interfaces. FPGAs can provide a systems level solution for the redecoding architecture with the hard-decoder implemented in custom logic and the software implemented on a soft-core processor. The Altera NIOS-II and Xilinx Microblaze are FPGA soft-core processors that feature low-latency interfaces to custom hardware units that can be implemented in the FPGA fabric. The LEON2 processor is an open-source VHDL model of a processor that can be implemented in an FPGA or ASIC technology and features a 64-bit coprocessor interface. The TI C6416 DSP has two independent 32-bit datapaths, each featuring 4 GFMULTs.

C. SIMD Implementation of Interpolation on a DSP

This section describes a SIMD implementation of the interpolation algorithm on a DSP processor. The TI TMS320C6416 (“C6416”) DSP is suited for this application because it comes with built-in Galois field multipliers for fields up to GF(256). This 720 MHz processor features two independent 32-bit datapaths, each with a 32-bit multiplier that can perform 4 GFMULTs on 4 Galois field elements packed into 32-bit integers. This can be described in terms of a parallelization factor N . An N -GFMULT architecture takes $2N$ -element vectors as input, performs N parallel GFMULTs producing a single N -element vector as output. If N is the number of bytes in a processor word, then the processor ALU can be used to perform N GFADDS on the vector. Since the C6416 has two independent 32-bit datapaths, this is a $2 \times N$ -GFMULT architecture, with $N = 4$.

We now consider the mapping of the three polynomial operations to this architecture. A bivariate polynomial can be expressed as

$$W(x, y) = \sum_{j=0}^{d_y} w_j(x) y^j, \quad (4)$$

which can be considered as a univariate polynomial in y whose coefficients are univariate polynomials in x . POLY1XY can be expressed as:

$$\begin{aligned} W(x, y) &\leftarrow aW(x, y) + bV(x, y) \\ &= \sum_{j=0}^{d_y} (aw_j(x) + bv_j(x)) y^j, \end{aligned} \quad (5)$$

where $a, b \in \text{GF}(q)$. Therefore, POLY1XY can be implemented with $d_y + 1$ POLY1X operations on the component univariate polynomials. The mapping of these univariate POLY1X operations is shown in Figure 3. Each data path is used to compute 4 coefficients of $aw_j(x)$ and $bv_j(x)$. The results are added using a 32-bit XOR. To approximate the speedup over a serial architecture with a single GFMULT, we can count the number of GF operations required. If l is the number of terms in the univariate polynomial, a serial architecture would require $3l$ operations while the C6416 DSP with only one GF multiplier can do it in $2l$ operations since it can perform additions and multiplications in parallel. Some typical values of l are given in Table II. The $2 \times N$ -GFMULT architecture requires $\lceil \frac{l}{N} \rceil$ operations. The approximate speedup is therefore $\frac{2l}{\lceil \frac{l}{N} \rceil}$ for the C6416 DSP and $\frac{3l}{\lceil \frac{l}{N} \rceil}$ for a serial case. The results of an implementation on the C6416 DSP are shown in Table III. The number of clock cycles corresponding to all calls of the POLY1XY operation are both reported for a serial implementation and a SIMD implementation. Actual speedup over a serial implementation range from 4.6 to 6 times. The theoretical model is an approximation used to obtain a first-order approximation to the speedup. It does not model cycle-level effects. As well, the parameter l is the *maximum* possible polynomial length, and the actual polynomials can be shorter. We see that the actual results demonstrate a dependence on

n	k	m_{\max}	max. length (bytes)
255	239	4	35
255	239	8	69
255	239	16	137
255	223	12	199

TABLE II

MAXIMUM LENGTH OF THE COMPONENT UNIVARIATE POLYNOMIALS.

the maximum multiplicity, and hence on the length of the polynomials, l . This can be explained by overhead calculations which reduce the efficiency of the parallelization when applied to shorter polynomials, such as with $m = 4$.

Similarly, POLY2XY can be reduced to a series of POLY2X operations as

$$\begin{aligned} W(x, y) &\leftarrow W(x, y)(x + a) \\ &= \sum_{j=0}^{d_y} (aw_j(x) + xw_j(x)) y^j. \end{aligned} \quad (6)$$

Note that $xw(x)$ is simply a shift of $w(x)$ by one coefficient so it requires no GF computation. On a serial architecture, the multiplication takes l cycles, the addition also takes l operations and the shift is free if the individual bytes are addressable. On the other hand, for the SIMD case, an unaligned memory access is required to perform the shift. For analytical purposes, it will also be considered free, however this is not necessarily true in practice. Therefore, the expected speedups of $\frac{2l}{\lceil \frac{l}{2N} \rceil}$ for the serial case and $\frac{l}{\lceil \frac{l}{2N} \rceil}$ for the single GFMULT C6416 DSP case are optimistic. The results of the implementation of POLY2XY are shown in Table III. The speedup ranges from 2.7 to 4.5 times. As for POLY1X, the implementation overhead reduces the speedups for shorter polynomials.

The HDXY operation can also be decomposed into univariate HDX operations as:

$$\begin{aligned} P^{[\mu, \nu]}(a, b) &= \sum_u \sum_v \binom{u}{\mu} \binom{v}{\nu} p_{u,v} a^{u-\mu} b^{v-\nu} \\ &= \sum_v \binom{v}{\nu} W_v^{[\mu]}(a) b^{v-\nu}, \end{aligned} \quad (7)$$

where,

$$W_v^{[\mu]}(a) = \sum_u \binom{u}{\mu} w_{u,v} a^{u-\mu}. \quad (8)$$

This univariate Hasse derivative evaluation has the flavor of a polynomial evaluation. We will first consider the mapping of polynomial evaluation in general and then apply these results to calculating the univariate Hasse Derivative, HDX. One solution for polynomial evaluation using a SIMD architecture was presented in [14] that stores l precomputed powers of a and then multiplies these powers of a with the coefficients w_u . This technique is appropriate if $l \sim N$, which is not true in our case. Instead, we consider an efficient N -way decomposition of the evaluation algorithm that does not require the precomputation of l values.

On a serial architecture, the classical way of efficiently computing a polynomial evaluation is Horner's rule [15]. The evaluation of $W(a) = w_0 + w_1a + w_2a^2 + \dots + w_{l-1}a^{l-1}$ can be expressed as $w_0 + a(w_1 + a(w_2 + \dots a(w_{l-1})))$, which results in an iterative algorithm requiring one GFMULT by a and one GFADD in each iteration. The K 'th order Horner's rule decomposition of $W(x)$ suitable for processing on K parallel arithmetic units was proposed by Dorn in [16]:

$$\begin{aligned} W(a) &= v_{K-1}a^{K-1} + \dots + v_1a + v_0, \\ v_j &= w_j, j = l-1, \dots, l-K, \\ v_j &= w_j + (a^K)v_{j+K}, j = l-K-1, \dots, 0. \end{aligned} \quad (9)$$

In an architecture with K parallel GFMULTs and GFADDs, K of the v_j can be computed in parallel using the value of a^K . The problem is then reduced to a final evaluation a K -term polynomial which can be done using a related parallel decomposition by Estrin's rule [16]. Estrin's rule on the C6416 requires $2(\lfloor \log_2(2N) \rfloor + 1)$ operations [16]. Now we consider how to implement the Hasse Derivative given the parallelization of polynomial evaluation. The complete HDX algorithm requires the evaluation of binomial coefficients. Fortunately, the computation of the binomial coefficients can be avoided for GF(2^b), and only the parity is required. The parity of $\binom{u}{\mu}$ is computed using Lucas' rule and is odd if $u \& \mu = u$ [8]. The complete K 'th order Horner rule for HDX is:

- 1) Compute l values of the parity of the binomial coefficients: $p_i, i = 0, \dots, l-1$. Alternatively, these may be precomputed and stored.
- 2) Calculate $z(x)$ by multiplying the coefficients of w_i by the $p_i, i = 0, \dots, l-1$ using the K parallel GFMULTs.
- 3) Evaluate $z(a)$ using the K 'th order Horner's rule.
- 4) $w^{[\mu]}(a) = z(a)/a^\mu$. This implements the shift by μ (step 2 guarantees that the first μ coefficients of $z(x)$ are zero).

Assume the parity values have been precomputed. Then the multiplication to form $z(x)$ takes $\lceil \frac{l}{2N} \rceil$ operations. The calculation of a^N takes $\lceil \log_2 2N \rceil$ operations. The parallel portion of the evaluation takes $\lceil \frac{l}{2N} \rceil$ SIMD multiplications and additions, which can be done in parallel. The final evaluation using Estrin's rule takes $2(\lfloor \log_2 2N + 1 \rfloor)$ operations. Assume the division (multiplication by an inverse) by a^μ can be done in 2 operations using a lookup table and multiplication. Then, the improvement in the number of iterations over a serial processor is

$$S_{\text{HDX}} = \frac{3l + 2}{\lceil \log_2 2N \rceil + 2 \lceil \frac{l}{2N} \rceil + 2(\lfloor \log_2 2N \rfloor + 1) + 2} \quad (10)$$

Similarly, the improvement over a single GFMULT C6416 DSP is

$$S_{\text{HDX}} = \frac{2l + 2}{\lceil \log_2 2N \rceil + 2 \lceil \frac{l}{2N} \rceil + 2(\lfloor \log_2 2N \rfloor + 1) + 2} \quad (11)$$

The results of the implementation of HDXY in Table III indicate that our implementation only realizes speedups of

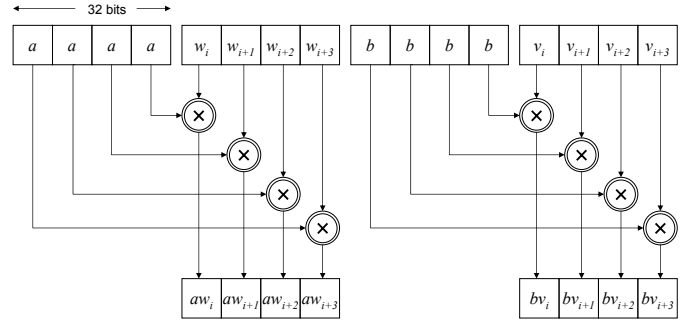


Fig. 3. The mapping of POLY1X on the C6416 DSP.

Op.	m_{\max}	cc (SISD)	cc (SIMD)	SU (pred.)	SU (actual)
POLY1XY	4	1.3 M	0.275 M	7.8	4.6
POLY1XY	8	37.8 M	6.3 M	7.7	6
POLY2XY	4	0.34 M	0.13 M	7.0	2.6
POLY2XY	8	5 M	1.1 M	7.7	4.5
HDX	4	0.42 M	0.38 M	3.1	1.1
HDX	8	7.4 M	5.6 M	4.5	1.3

TABLE III

PERFORMANCE OF POLYNOMIAL OPERATIONS MAPPED TO THE C6416 DSP

m_{\max}	cycles (SISD)	cycles (SIMD)	SU (actual)
4	2.07 M	0.835 M	2.5
8	50.5 M	13 M	3.9

TABLE IV

PERFORMANCE OF THE COMPLETE INTERPOLATION ALGORITHM MAPPED TO THE C6416 DSP

10% to 30% and does not reach the potential indicated by our theoretical model. The speedup is less, since in the serial processor, if the parity of a binomial coefficient is 0 then the Horner's rule update is completely skipped, whereas it is explicitly performed in the parallel version regardless of the parity value. The overhead seems to overwhelm the advantages of parallelizing the algorithm. In the next section we propose an architecture to help overcome these limitations.

Table IV gives the number of clock cycles for the serial and SIMD implementations of the complete interpolation algorithm. The overall speedup observed for the SIMD implementation over the serial implementation ranges from 2.5 to 3.9 times.

D. Merged Interpolation Architecture

The SIMD mapping of the interpolation algorithm above was a first step in exploring the potential of SIMD instructions for implementing the interpolation algorithm. We found that the POLY1X and POLY2X operations were easily mapped to a SIMD architecture, however the results for HDX were not as encouraging. In this section we propose a SIMD architecture that addresses the shortcomings of the architecture considered above. The *merged architecture* is shown in Figure 4. The first improvement is to merge the scalar-polynomial multiplication

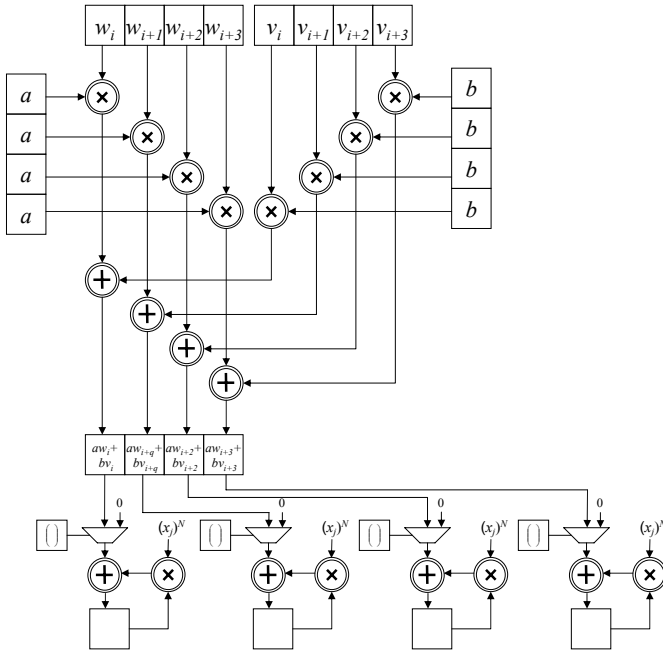


Fig. 4. The merged interpolation architecture.

and addition steps in POLY1X and POLY2X into a single clock cycle by adding N GFADDs after the GFMULTs. While this is already implemented on the C6416 DSP, it would result in a increase in the critical path for the other processors. For example, consider the GF(256) multiplier from [13] which has delay $T_{\text{AND}} + 5T_{\text{XOR}}$, where T_{AND} is the delay of an AND gate and T_{XOR} is the delay of an XOR gate. Considering an FPGA implementation where $T_{\text{AND}} = T_{\text{XOR}}$, this would simplify to 6 gate delays. Therefore, adding XOR gates to implement the GFADD stage would lengthen the critical path of the SIMD architecture by a factor of 1.2. This would be acceptable if the delay of this unit is not the critical path in the entire processor. Alternatively, the multiply-add structure could be pipelined, since there is no feedback required during a single POLY1XY or POLY2XY operation. Note that in architectures that only support 2 N -element inputs, internal registers to hold the scalars a and b are added and are set using an initialization instruction. This only has to be done once per POLY1XY or POLY2XY operation.

The other improvement is to merge the HDX operation for the *next* iteration with the POLY1X and POLY2X updates for the current iteration. This is accomplished by adding N GFMULTs and GFADDs to the SIMD architecture. The HDX unit performs the N -th order Horner's Rule concurrently with the POLY1X/POLY2X operations. The resulting N coefficient polynomial at the end of the update can be evaluated using the POLY1X/POLY2X stage. Further improvements are made by adding hardware to calculate the parity of the binomial coefficients, eliminating the expensive multiplication step. In this way, the latency of the HDX operation can be almost completely hidden (except for the final length- N evaluation).

V. CONCLUSIONS

In this paper we described the implementation of the interpolation step of the Koetter-Vardy algebraic soft-decision decoding algorithm on a SIMD DSP processor. We proposed a parallelization of the algorithm using the K 'th-order Horner's rule for polynomial evaluation. An overall speedup of 2 to 4 times was obtained using 8 Galois field multipliers. To overcome the inefficiencies in the implementation, a merged architecture was proposed that executes most of Hasse derivative calculation in parallel with the polynomial updates.

REFERENCES

- [1] Ralf Koetter and Alexander Vardy, "Algebraic soft-decision decoding of Reed-Solomon codes," *IEEE Transactions on Information Theory*, vol. 49, no. 11, pp. 2809–2825, November 2003.
- [2] Venkatesan Guruswami and Madhu Sudan, "Improved decoding of Reed-Solomon and algebraic-geometry codes," *IEEE Transactions on Information Theory*, vol. 45, no. 6, pp. 1757–1767, September 1999.
- [3] Warren J. Gross, Frank R. Kschischang, Ralf Koetter, and P. Glenn Gulak, "A VLSI architecture for interpolation in soft-decision list decoding of Reed-Solomon codes," in *Proceedings of the 2002 IEEE Workshop on Signal Processing Systems (SIPS'02)*, San Diego, CA, October 16–18 2002, pp. 39–44.
- [4] Arshad Ahmed, Naresh R. Shanbhag, and Ralf Koetter, "Systolic interpolation architectures for soft-decoding Reed-Solomon codes," in *Proceedings of the 2003 IEEE Workshop on Signal Processing Systems, SIPS'03*, August 2003, pp. 81–86.
- [5] Arshad Ahmed, Ralf Koetter, and Naresh R. Shanbhag, "VLSI architectures for soft-decision decoding of Reed-Solomon codes," in *Proceedings of the 2004 IEEE International Conference on Communications*, June 2004, vol. 5, pp. 2584–2590.
- [6] Warren J. Gross, Frank R. Kschischang, and P. Glenn Gulak, "An FPGA interpolation processor for soft-decision Reed-Solomon decoding," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, April 2004, pp. 310–311.
- [7] Xinmiao Zhang and Keshab K. Parhi, "Fast factorization architecture in soft-decision Reed-Solomon decoding," in *Proceedings of the 2004 IEEE Workshop on Signal Processing Systems (SIPS'04)*, October 2004, pp. 101–106.
- [8] Warren J. Gross, Frank R. Kschischang, Ralf Koetter, and P. Glenn Gulak, "Towards a VLSI architecture for interpolation-based soft-decision Reed-Solomon decoders," *Journal of VLSI Signal Processing*, vol. 39, no. 1–2, pp. 93–111, January 2005.
- [9] Warren J. Gross, Frank R. Kschischang, Ralf Koetter, and P. Glenn Gulak, "Applications of algebraic soft-decision decoding of Reed-Solomon codes," Accepted for publication in the *IEEE Transactions on Communications*.
- [10] Ralf Koetter, Jun Ma, Alexander Vardy, and Arshad Ahmed, "Efficient interpolation and factorization in algebraic soft-decision decoding of Reed-Solomon codes," in *Proceedings of ISIT 2003*, June 29–July 4 2003, p. 365.
- [11] Ralf Koetter and Alexander Vardy, "A complexity reducing transformation in algebraic list decoding of Reed-Solomon codes," in *Proceedings of ITW2003*, Paris, France, March 31 – April 4 2003.
- [12] Haitao Xia, Hongxin Song, and J. R. Cruz, "Retry mode soft Reed-Solomon decoding," *IEEE Transactions on Magnetics*, vol. 38, no. 5, pp. 2325–2327, September 2002.
- [13] Emina Soljanin and Rüdiger Urbanke, "An efficient architecture for implementation of a multiplier and inverter in GF(2⁸)," Tech. Rep. BL011217-960308-08TM, Bell Labs, 1996.
- [14] Wei-Ming Lim and M. Benaissa, "Design space exploration of a hardware-software co-designed GF(2^m) Galois field processor for forward error correction and cryptography," in *Proceedings of CODES+ISSS'03*, October 2003, pp. 53–58.
- [15] Donald E. Knuth, *The Art of Computer Programming*, vol. 2, Seminumerical Algorithms, Addison-Wesley, 1969.
- [16] W. S. Dorn, "Generalizations of horner's rule for polynomial evaluation," *IBM Journal of Research and Development*, vol. 6, pp. 239–245, April 1962.