Square-Rich Fixed Point Polynomial Evaluation on FPGAs

Simin Xu
Xilinx Asia Pacific
Singapore
siminx@xilinx.com

Suhaib A. Fahmy
School of Computer Engineering
Nanyang Technological University, Singapore
sfahmy@ntu.edu.sg

Ian V. McLoughlin
School of Information Science and Technology
University of Science and Technology of China
ivm@ustc.edu.cn

ABSTRACT

Polynomial evaluation is important across a wide range of application domains, so significant work has been done on accelerating its computation. The conventional algorithm, referred to as Horner's rule, involves the least number of steps but can lead to increased latency due to serial computation. Parallel evaluation algorithms such as Estrin's method have shorter latency than Horner's rule, but achieve this at the expense of large hardware overhead. This paper presents an efficient polynomial evaluation algorithm, which reforms the evaluation process to include an increased number of squaring steps. By using a squarer design that is more efficient than general multiplication, this can result in polynomial evaluation with a 57.9% latency reduction over Horner's rule and 14.6% over Estrin's method, while consuming less area than Horner's rule, when implemented on a Xilinx Virtex 6 FPGA. When applied in fixed point function evaluation, where precision requirements limit the rounding of operands, it still achieves a 52.4% performance gain compared to Horner's rule with only a 4% area overhead in evaluating 5th degree polynomials.

Categories and Subject Descriptors

B.2.4 [**Arithmetic and Logic Structures**]: High-Speed Arithmetic—*Algorithms*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*Computations on polynomials*

Keywords

Fixed point; field programmable gate arrays; polynomial evaluation.

1. INTRODUCTION

Polynomials are commonly used in high-performance DSP applications to approximate the computation of functions, or to model systems parametrically. They are found inside many basic digital circuits including high-precision elementary functional evaluation circuits [1] and advanced digital filters [2].

Simple polynomials can be evaluated using lookup tables. However, when the degree and wordlength increases, lookup tables become infeasible and thus computation is more appropriate for polynomial evaluation.Work on software and hardware approaches for speeding up polynomial evaluation is numerous. The simplest scheme for computing polynomials is Horner's rule, which is an inherently serial process. Parallel schemes for software implementation, such as Estrin's method [3], have been shown to offer a significant speed improvement. With more resources available, polynomial evaluation can be accelerated significantly. For example, Estrin's method takes $2\lceil log_2(k+1)\rceil$ iterations with $\lceil k/2\rceil$ processing units to evaluate a k^{th} order polynomial. More recently, fully parallel hardware architectures have been more commonly investigated [4].

Field programmable gate arrays (FPGAs) offer an ideal architecture for such systems due to their fine-grained customisability in terms of datapath wordlength and pipelining. Numerous FPGA-based polynomial evaluation architectures have been presented [5, 6], and various methods have been proposed to speed up polynomial evaluation [7, 8, 9, 10, 11].

In this paper, polynomial evaluation methods are first reviewed, followed by the proposal of a novel evaluation algorithm that takes advantage of the reduced complexity of squaring compared to general multiplication. The new method is then evaluated against both Horner's rule and Estrin's method. The novel algorithm has two variations, suited to different implementations, called the Square-Rich method and Modified Square-Rich method. While this approach can be applied to both floating and fixed point evaluation, we discuss the latter in this paper.

2. BACKGROUND

2.1 Polynomial Evaluation

The general format for k^{th} degree polynomial is,

$$f(x) = \sum_{i=0}^{k} a_i x^i \tag{1}$$

The fixed point number x is the input of the polynomial with a set of coefficients a_i. These coefficients are defined by the application, and are computed in various ways. We assume that the coefficients do not change frequently, although they can be updated from time to time, as is the case in most

FPGA'14, February 26–28, 2014, Monterey, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2671-1/14/02 ...$15.00. http://dx.doi.org/10.1145/2554688.2554779.

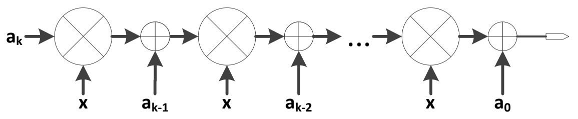

Figure 1: Architecture of polynomial evaluator using Horner's rule.

systems. In other words, we consider a system in which the computation *using* fixed a_i is the limiting factor, rather than the computation *of* a_i values themselves.

Polynomial evaluation procedures have been the subject of investigation since the 1950s. Apart from directly computing the polynomial, which is only practical for low degree polynomials [12], a few prominent methods have emerged.

2.1.1 Horner's Rule

Horner's rule is a basic and widely used method for computing polynomials, and is used in numerous complex applications [13, 14, 15, 16]. It works by transforming the polynomial into a series of multiply-add operations. Considering the polynomial equation stated in (1), Horner's rule re-writes the formula as:

$$f(x) = \{...((a_k \cdot x + a_{k-1}) \cdot x + a_{k-2}) \cdot x + ... + a_0\} \quad (2)$$

Horner's rule has been proven in [17] and [18] to use the minimal number of iterations to evaluate a particular polynomial, i.e. it is optimal in terms of the number of computational steps. Furthermore, it also has a regular structure which is easily implemented, and these factors have lead to its widespread adoption.

In a hardware context, where a custom pipelined datapath can be built, the required amount of hardware resources for polynomial evaluation increases linearly as the degree of the polynomial increases. k multiply-add computations are needed (generally k multipliers and k adders) in a parallel implementation. Figure 1 shows the structure of a polynomial evaluator using Horner's rule.

While the structure is simple, Horner's rule suffers from a long latency due to the serial arrangement of operations. If we denote multiplier latency as T_{mul}, squarer latency as T_{sq}, and adder latency as T_{add}, the latency for k^{th} order polynomial is $k \cdot T_{mul} + k \cdot T_{add}$, which increases linearly with the degree of the polynomial.

2.1.2 Parallel Methods

For applications with low latency requirements, such as communications or cryptography, parallel evaluation is desirable. Dorn [19] proposed a parallel scheme for Horner's Rule. Tree structure approaches have also been presented [20, 21, 22] for ultra-high degree polynomial evaluation ($k > 20$). Among parallel schemes, Estrin's method has been preferred due to its short latency [23].

Estrin's method has since been adopted in many applications [5, 24] for fast evaluation of polynomials. It works by reforming (1) as follows:
If k is even:

$$f(x) = a_k \cdot x^k + \sum_{i=0}^{(k-2)/2} (a_{2i+1}x + a_{2i}) \cdot x^{2i}, \quad (3)$$

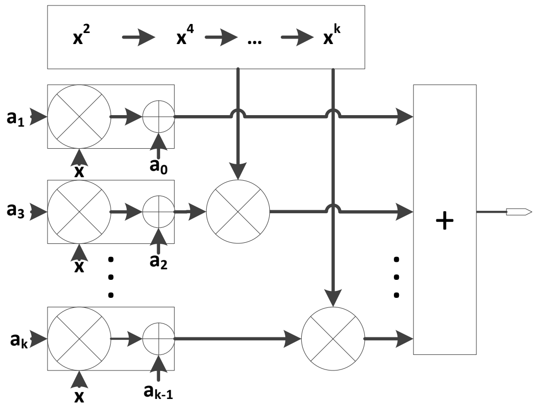

Figure 2: Architecture of polynomial evaluator using Estrin's method.

else if k is odd:

$$f(x) = \sum_{i=0}^{(k-1)/2} (a_{2i+1}x + a_{2i}) \cdot x^{2i} \quad (4)$$

Each sub expression can be computed in parallel and the results summed together at the end. The worst case latency is determined by the exponential function x^k and it can be computed in a total of $\lceil log_2(k) \rceil$ steps, which consists of a series of multiplication and/or squaring operations. This is followed by the multiplication of its coefficient and the final additions. Estrin's method is shown pictorially in Figure 2.

The overall latency for polynomials up to $k = 6$ is summarized in Table 1. Generally, the latency increases at a slower rate than Horner's Rule with increasing degree, and hence, for higher degree polynomials the latency is shorter.

Degree	T_{mul}	T_{sq}	T_{add}
2	1	1	1
3	2	0	2
4	1	2	1
5	1	2	1
6	2	2	1

Table 1: Latency for evaluating k^{th} degree polynomials using Estrin's method.

This performance is enabled by increased hardware cost. Although the same number of additions is required as for Horner's rule, as many as $\lfloor k/2 \rfloor$ more multiplications/squares are required in Estrin's method to compute each even degree of x^k and this leads to $\lfloor log_2(k) \rfloor$ more squarers and $\lfloor k/2 \rfloor - \lfloor log_2(k) \rfloor$ more multipliers needed in hardware. Overall implementation costs are summarized for polynomials up to $k = 6$ in Table 2.

2.2 FPGA Computation

Modern FPGAs include hard DSP blocks that enable fast and area efficient multiplication and addition. In the Xilinx Virtex 6, and all subsequent 7 Series FPGAs from Xilinx, the DSP48E1 blocks support 25×18 bit signed multiplication, followed by a programmable ALU in the datap-

Degree	*Mults*	*Squarers*	*Adders*
2	2	1	2
3	3	1	3
4	4	2	4
5	5	2	5
6	7	2	6

Table 2: Hardware cost for evaluating k^{th} degree polynomials using Estrin's method.

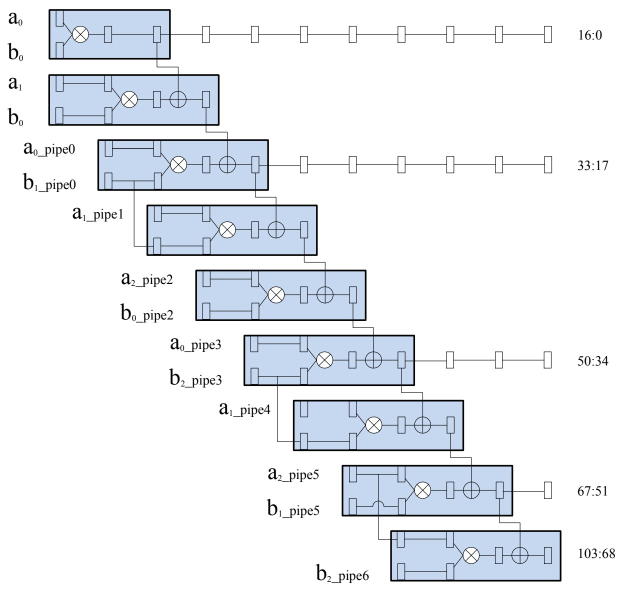

Figure 3: Pipelined 52-bit multiplier built using DSP blocks.

ath. These blocks can also be efficiently cascaded to support wider arithmetic. The DSP48E1 primitives also support dynamic modification of their datapath functions [25], which can be leveraged for iterative implementations of fundamental computational blocks [26]. For polynomial evaluation, the basic functions of multiplication and squaring can be efficiently built using DSP blocks. In order to obtain maximum performance, it is important to consider the structure of the DSP block when building a computational datapath [27].

Fast, wide multipliers are built by cascading together chains of DSP blocks using dedicated hard wires that do not add significantly to the routing delay. This allows for wide operations to be computed at near full frequency, as long as all the stages are pipelined. A 52-bit multiplication using three DSP blocks is shown in Figure 3. Each shaded block is a single DSP block, with the cascade chains between the blocks shown. The only extra resources needed are registers to maintain alignment in the datapath between subsequent DSP block stages. The pipeline stages before DSP blocks are indicated by the names of the input bus for simplicity while the alignment of pipeline stages within and after DSP blocks are illustrated using small rectangles.

While a squarer can be implemented using multiplication with its inputs tied together, it is also possible to build wide squarers more efficiently. The number of DSP blocks re-

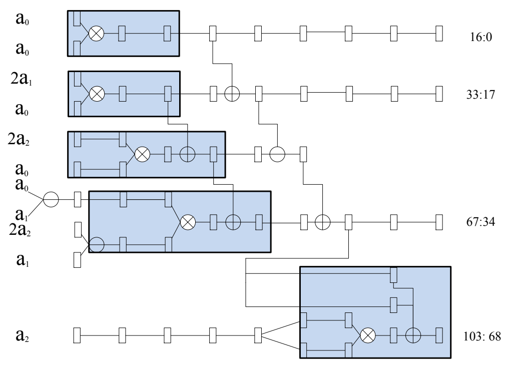

Figure 4: Pipelined 52-bit squarer using DSP blocks.

quired for a squarer grows more slowly than for a general multiplier, as operand wordlength increases [28]. Hence, where it is possible to use a squarer instead of a multiplier, efficiency gains are possible in terms of both area and performance. In our previous work [28], an efficient squarer was proposed which consumes up to 50% less hardware resources than an equivalent width multiplier. It can use 1 fewer DSP block than the method in [29] at a cost of only 127 additional LUTs. The architecture for a 52-bit squarer is shown in Figure 4.

The squarer design also benefits in terms of latency. After pipelining the circuit to meet the maximum frequency of the DSP blocks, a 52 bit squarer has a pipeline 23% shorter than an equivalent multiplier. For a 64 bit squarer, this advantage increases to 31%. Note that in order to achieve maximum DSP block frequency, it is necessary to add an additional register stage any time a DSP block output is passed to LUTs (for implemented small adders for example). This has not been taken into account in [29] and [13], but is done by default in this work.

3. SQUARE-RICH METHOD

In this section, we present the novel polynomial evaluation algorithm. It involves transformation of the general form of a polynomial into a "square rich" format and so we name it the "Square-Rich" (SR) method. The main benefit of the algorithm is to achieve faster evaluation with minimum hardware overhead. Although the total number of operations is more than for Horner's rule, the hardware implementation can be more efficient than the "optimum method in theory", thanks to the efficiency gains achieved by using squarers instead of multipliers. The latency of this method will be shown to be close to that of Estrin's method but using fewer hardware resources. Although this paper only presents results for polynomials up to the 6^{th} degree, the approach can be applied to polynomials of arbitrary degree.

3.1 Algorithm

The SR method is based on the following hypothesis, illustrated by a 2^{nd} degree polynomial example:

Hypothesis 1.

To evaluate the 2^{nd} degree polynomial,

$$f(x) = a_2x^2 + a_1x + a_0 \tag{5}$$

it is advantageous to reform the equation as,

$$f(x) = a_2 \cdot ((x + m_2)^2 + n_2) \tag{6}$$

Where m_2 and n_2 are defined as,

$$m_2 = \frac{a_1}{2a_2} \tag{7}$$

$$n_2 = \frac{a_0}{a_2} - \frac{a_1^2}{4a_2^2} \tag{8}$$

The new format completes the square and converts the original polynomial into its vertex form. In (6), m_2 and n_2 are coefficients derived from a_1 and a_0, thus they can be considered precomputed values. It can be seen that four steps are needed to compute (6): one addition, one square followed by another addition, and a final multiplication. The same number of steps is needed using Horner's rule but with two multipliers and two adders. The latency is thus $T_{mul} + T_{sq} + 2T_{add}$ for the proposed method, compared to $2T_{mul} + 2T_{add}$ for Horner's rule. If $T_{sq} < T_{mul}$, this represents a saving. The SR method also results in reduced hardware cost since the resources required for a squarer are fewer than for a multiplier.

To apply Hypothesis 1 to the general k^{th} degree polynomial in (1), we define an integer number j, which is the binary logarithm of k:

$$j = \lfloor log_2 k \rfloor \tag{9}$$

and define

$$p_t(x) = (a_{k-t}x^{k-t} + a_{k-j-t}x^{k-j-t} + a_{k-2j-t}x^{k-2j-t}), \tag{10}$$

where t ranges from $[0,\ j-1]$. Then the equation can be divided into the following groups:

$$f(x) = \sum_{t=0}^{j-1} p_t(x) + f'(x), \tag{11}$$

where $f'(x)$ only exists if $k - 3j \geq 0$ and is

$$f'(x) = \sum_{i=0}^{k-3j} a_i x^i. \tag{12}$$

In $p_t(x)$, a common divisor of $a_{k-t}x^{k-2j-t}$ can be extracted so that the equation becomes

$$p_t(x) = a_{k-t}x^{k-2j-t}(x^{2j} + \frac{a_{k-j-t}}{a_{k-t}}x^j + \frac{a_{k-2j-t}}{a_{k-t}}) \tag{13}$$

Completing the square in (13), it becomes

$$p_t(x) = a_{k-t}x^{k-2j-t}[(x^j + m_{k-t})^2 + n_{k-t}], \tag{14}$$

where

$$m_{k-t} = \frac{a_{k-j-t}}{2a_{k-t}} \tag{15}$$

$$n_{k-t} = \frac{a_{k-2j-t}}{a_{k-t}} - \frac{a_{k-j-t}^2}{4a_{k-t}^2} \tag{16}$$

Equation (11) with (14) are the general forms of the SR method for evaluating polynomials of arbitrary degree. The same iteration process from (10) to (14) is applied on $f'(x)$ with degree k' and for $f'(x)$, the degree k' is now $k-3j$. This continues until the lowest degree of polynomial has been computed.

The latency required for the SR method applied in 2th to 6th degree polynomial is shown in Table 3. It is clear that the SR method has shorter latency than Horner's rule for parallel evaluations of higher degree terms and a gain from an increased number of square operations instead of multiplications shown in Hypothesis 1. The latency is no longer limited by the computation of x^k as the highest power exponential that must be computed is only x^j. Therefore, the SR method can at least equal, or potentially improve upon the performance of Estrin's method. This will be discussed in more detail when FPGA implementation results are presented in Section 5.

deg	T_{mul}	T_{sq}	T_{add}
2	1	1	2
3	1	1	3
4	1	2	3
5	1	2	3
6	1	2	4

Table 3: Latency for evaluating k^{th} degree polynomials using SR method.

The hardware resource requirements for the SR method are shown in Table 4 and a diagram of the architecture is shown in Figure 5. The hardware requirements are smaller than for Estrin's method since fewer multipliers and fewer squarers are used. Compared to the Horner's rule, which needs the minimum number of computational steps, the SR method can have reduced hardware overhead as a result of the more efficient squarer (compared to multiplication). This comparison, in the context of FPGA implementation, is presented in Section 5.

deg	*Multiplier*	*Squarer*	*Adder*
2	1	1	2
3	2	1	3
4	3	2	4
5	3	3	5
6	4	3	6

Table 4: Hardware cost for evaluating k^{th} degree polynomials using SR method.

3.2 Coefficients

The SR method needs a new set of coefficients, which are a one-off derivation from the original polynomial coefficients. For an application such as an adaptive filter, this may result in an additional overhead every time the polynomial adapts, however the generation process can be performed when deriving the polynomial. Even when the coefficients are computed on the fly, compared to the actual polynomial evaluation process, this overhead is small and tends to be negligible as the number of evaluations performed using each new coefficient set increases. In this paper, the coefficients are pre-computed and stored in block RAMs at design time. This is considered to be a typical real-world scenario; the polynomial is generated during system setup or programming rather than on the fly.

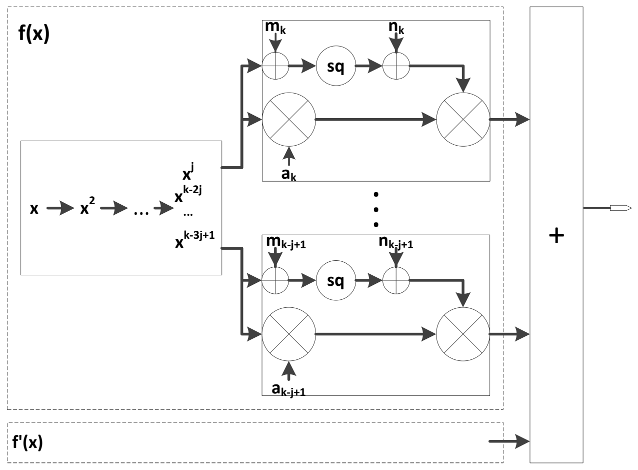

Figure 5: Architecture of polynomial evaluator using SR method.

4. ERROR ANALYSIS

In real applications, like function approximation, the complexity and latency of polynomial evaluation is often traded-off against accuracy. A certain tolerance of error, including evaluation error and other errors may be allowed and the designer can implement faithful rounding for coefficients and perform truncation in intermediate computations while controlling the total error. In this paper, as we are only interested in the process of polynomial evaluation, we assume that all other error factors involved are the same among all the evaluation algorithms except for evaluation error caused by rounding. The error contributed by the coefficient generation process in previous sections is negligible and we assume that the new coefficients themselves are computed without additional errors being introduced.

Two types of wordlength optimization for fixed-point implementation in FPGA will be discussed in this paper and the error analysis will performed differently in the individual contexts.

4.1 Fixed Point Implementation

First, we consider a general fixed point implementation where each computation will only truncate its result to the same wordlength as the input operands, which is common practice in many signal processing systems. In this case, as the error is mainly associated with the multiplications, different evaluation algorithms will have different evaluation error, due to the differing structures. Generally, as Horner's rule requires the least number of computational steps, it is more accurate than any other evaluation scheme in this context. Take a 3^{rd} degree polynomial as example. The total evaluation error using Horner's rule can be calculated from

$$\epsilon_{total} f(x) = \sum_{i=0}^{2} \epsilon_{eval}(q_i \cdot x + a_i) \cdot x^i \tag{17}$$

In (17), q_i is the multiply-add result for the $i+1$th term except for the highest degree, which is a_k. Each multiply-add evaluation error only depends on the previous multiply-add result. Assuming x is in the range of $[0, 1]$, for higher degree terms, each multiply-add evaluation error is multiplied by the exponential of x and thus it is less significant in the overall error than the lower degree terms.

For Estrin's method, as more multiplication steps (including squares) are required, it naturally has larger error than Horner's rule in this context. For the same 3^{rd} degree polynomial, the total evaluation error has four terms rather than three and it can be represented as,

$$\begin{aligned}\epsilon_{total} f(x) = {} & \epsilon_{eval}(a_1 \cdot x + a_0) \\ & + (a_3 x + a_2) \cdot \epsilon_{trunc}(x^2) \\ & + x^2 \cdot \epsilon_{eval}(a_3 \cdot x + a_2) \\ & + \epsilon_{trunc}((a_3 x + a_2) \cdot x^2)\end{aligned} \tag{18}$$

The first and third error term refer to the evaluation errors of the multiply-add computation and they are comparable with Horner's rule. However, the other two error components may contribute to a larger total error than Horner's rule, depending on the value of the coefficients. The second error term $\epsilon_{trunc}(x^2)$, which refers to the truncation error of the square and is comparable with $\epsilon_{eval}(q_1 \cdot x + a_1)$ in (17), is multiplied by $a_3 x + a_2$ which can be larger than x. The last error term $\epsilon_{trunc}((a_3 x + a_2) \cdot x^2)$, which refers to the truncation error of the multiplication indicated by $\cdot$, is due to the additional multiplication that Horner's rule does not require. For higher degree polynomials, Estrin's method is worse in terms of total evaluation error with the current wordlength optimization. There will be more error components for Estrin's method than Horner's rule due to a larger number of operations and the error from the exponential computation is significant as well, however, we do not detail them here due to space constraints.

In contrast, the SR method has smaller error than Estrin's method due to fewer multiplication operations. For the same 3^{rd} degree polynomial, the total evaluation error can be represented as,

$$\begin{aligned}\epsilon_{total} f(x) = {} & \epsilon_{eval}(a_0 + p_0) \\ & + a_3 x \cdot \epsilon_{eval}((x + m_3)^2 + n_3) \\ & + ((x + m_3)^2 + n_3) \cdot \epsilon_{trunc}(a_3 \cdot x_3)\end{aligned} \tag{19}$$

The first error term $\epsilon_{eval}(a_0 + p_0)$ refers to the evaluation error of the final multiplication in p_0 and the addition with a_0, which is the same amount as the first error term in Estrin's method (18). The second error term refers to the evaluation error for computing the square and the addition of n_3 while the last error term refers to the truncation error of $a_3 \cdot x_3$. Depending on the value of the coefficients, these two terms have similar values to the second and third error terms in (18). Therefore, the SR method is better in terms of evaluation error than Estrin's method as it has three comprable error components instead of four. For higher degree polynomials, as the SR method has much fewer multiplication operations and does not require a large degree exponentiation of x, it can be more accurate than Estrin's method generally.

However, the SR method is not always as accurate as Horner's rule. In fact, with the current optimization scheme, the SR method has the same number of evaluation error components for a 3^{rd} degree polynomial, but the total evaluation error could potentially be larger than Horner's rule, depending on the value of the coefficients. For higher degree polynomials, the SR method has more error components than Horner's rule due to the number of computations being higher and this difference increases as the degree increases. Therefore, a modified formulation, we call the "Modified Square-Rich" (MSR) method can be created and applied to reduce the total evaluation error. The new format is

$$p'_t(x) = x^{k-2j-t}(a_{k-t} \cdot (x^j + m_{k-t})^2 + n'_{k-t}), \tag{20}$$

where

$$n'_{k-t} = a_{k-2j-t} - \frac{a^2_{k-j-t}}{4a_{k-t}} \tag{21}$$

To implement the MSR method in hardware, the same amount of hardware resources are needed as for the SR method, as the total number of operations is the same. However, MSR has slighly increased latency, where one more multiplication must be serially computed for 3^{rd}, 5^{th} and 6^{th} degree polynomials, as summarized in Table 5. The structure of the MSR method is shown in Figure 6.

deg	T_{mul}	T_{sq}	T_{add}
2	1	1	2
3	2	1	3
4	1	2	3
5	2	2	3
6	2	2	4

Table 5: Latency for evaluating k^{th} degree polynomials using MSR method.

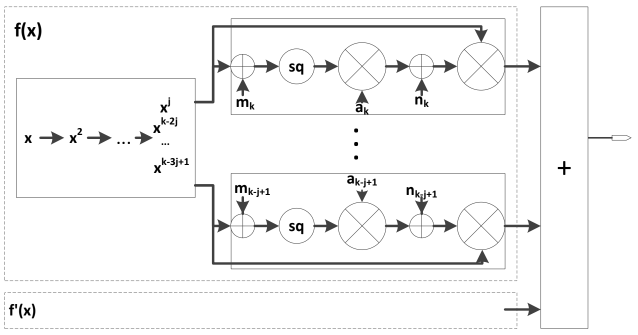

Figure 6: Architecture of polynomial evaluator using MSR method.

Although one more multiplication must be performed serially, the MSR method reduces the total evaluation error. Without the common factors to be taken out, the total evaluation error for the same 3^{rd} degree polynomial now becomes,

$$\begin{aligned}\epsilon_{total} f(x) = {} & \epsilon_{eval}(a_0 + p'_0) \\ & + a_3 x \cdot \epsilon_{trunc}((x + m'_3)^2) \\ & + x \cdot \epsilon_{eval}(a_3 \cdot (x + m'_3)^2 + n'_3)\end{aligned} \tag{22}$$

Similarly, the first error term refers to the evaluation error including final multiplication of p'_0 and the final addition. The second error term multiplies the same a_3x coefficient as (19) while the last error term has a much smaller coefficient which leads to smaller total error.

The MSR method shows its advantages further when a_{k-t} is small, which is usually the case in high degree polynomial applications. In this case, n_{k-t} tends to be large and the total evaluation error using the SR method will be large. In contrast, as the evaluation error is not related to the value of n_{k-t}, the MSR method retains its small evaluation error.

In summary, with the above general fixed point error analysis, Horner's rule remains the most accurate method. For lower degree polynomials, the SR method is more accurate than using Estrin's method and close to the accuracy of Horner's rule. However, for higher degree polynomials, the evaluation error from the SR method is much larger than for Horner's Rule and using the MSR method can reduce the gap. Section 5 presents the implementation results for the SR method and MSR method with general wordlength optimization along with results for Estrin's method and Horner's rule as reference designs.

4.2 Faithful Rounding in Specific Application

Another optimization is faithful rounding of coefficients and other operands, which is only possible in the context of a specific application, like fixed point function evaluation using polynomials. Typically, implementation in FPGAs is more flexible than architectures that enforce a fixed wordlength for each computation as each operand can be freely optimized according to requirements. Computational complexity can be reduced by reducing the wordlength of each operand bit by bit, as long as the total evaluation error is no worse than the error target. FloPoCo [13] includes an automated design generator with such optimization for a function evaluator using Horner's rule. For a fair comparison to the FloPoCo design, we apply a similar rounding strategy for function evaluation using Estrin's method, the SR method and the MSR method and our target is to achieve evaluation error no worse than the FloPoCo designs in each specific interval. We assume no overflow/underflow occurs in each computational step.

Gappa [1] is used to verify that the optimized designs are within the evaluation error bounds. Gappa is a tool that helps verify and formally prove properties of numerical programs dealing with floating-point or fixed-point arithmetic [30] and has been used to verify fixed point polynomial evaluations previously [5, 8, 13]. It computes the range of a given function based on the constraints of using interval arithmetic. In this case, each rounding trail of a particular algorithm will be verified by Gappa. If the verification shows the evaluation error for the SR method, MSR method or Estrin's method are not within the same bound as the FloPoCo design, rounding is modified for the next iteration and verified once more by Gappa. The verification scripts are not presented due to space constraints, though the optimized design details are presented in Section 6.

5. IMPLEMENTATION IN FPGA

In this section, a series of evaluators using the SR method and MSR method are built in Verilog and synthesized, placed and routed using Xilinx ISE 13.4 on a Virtex 6 XC6VLX240T-1 FPGA as found on the ML605 evaluation board. These evaluators are built for 52 and 64 bit arithmetic with polynomial degree ranges from 2 to 6 and equal input and output wordlengths. Each intermediate step truncates its result to 52 or 64 bits after the maximum wordlength has been computed. Evaluators using Horner's rule and Estrin's

[1]Version 0.16.4 gappa.gforge.inria.fr

method with the same wordlengths and degrees are built as references. Fixed point multiplications used in the evaluator are built using multipliers provided by Xilinx CoreGen and squarers are built using the method in [28]. Adders are synthesized to use carry chain resources in CLBs automatically. The designs are all pipelined and targeted to achieve the maximum frequency of DSP blocks for the targeted -1 speed grade device, which is 450MHz [31]. The same approach can be used on higher speed grade devices and result in higher frequencies. We have chosen the slowest speed grade to prove the baseline performance gain of the proposed methods, which can be applied across any FPGA containing the DSP48E1 primitive. DSP block pipelining is done by instantiating Xilinx primitives and turning on all the optional register stages to maximize performance. Adders with data flowing from or to DSP blocks are pipelined using flip flops or shift registers in CLBs.

A comparison of the cost in DSB blocks for all the evaluators is shown in Table 6.

bits	*deg*	*MSR/SR*	*Estrin's*	*Horner's*
52	2	14	23	18
	3	23	32	27
	4	37	46	36
	5	42	55	45
	6	51	73	54
64	2	24	40	32
	3	40	56	48
	4	64	80	64
	5	72	96	80
	6	88	128	96

Table 6: DSP block usage for all evaluators.

Both the SR and MSR methods use same number of DSP blocks since the number of operations is the same. They require at least 3 fewer DSP blocks than Horner's Rule for a wordlength of 52 bits and 8 fewer blocks for a wordlength of 64 bits except for the 4th degree evaluators. Compared to the Estrin's method, they are much smaller with up to 22 and 40 fewer DSP blocks for wordlengths of 52 and 64 bit respectively. Hence, both novel methods are more efficient in terms of DSP usage.

The equivalent hardware cost for all the evaluators is shown in Figure 7. Cost is determined in terms of the equivalent number of LUTs, which we use as a metric to combine the DSP block count and LUT count. The equivalent number of LUTs for one DSP block is defined as the total number of LUTs in the device divided by the total number of DSP blocks. Hence, a circuit that uses an additional DSP block should save at least that number of LUTs for it to be considered an overall area saving. While this metric is not universally applicable, it serves as a useful proxy here. For the specific target FPGA, with 150720 LUTs and 768 DSP blocks, this ratio is 196 (it ranges from 160 to 240 for most general purpose Xilinx Virtex FPGAs).

For 52 and 64 bit polynomials, both novel methods use up to 20.9% fewer equivalent LUTs than Horner's rule for 2nd and 3rd degree polynomials and no more than 5% more for higher degrees. Considering that Horner's rule has the fewest operations, the novel methods are very efficient in terms of resource requirements, mainly as a result of the efficient squarer design. Compared to Estrin's method, the novel methods use up to 38.3% fewer equivalent LUTs for 2nd degree polynomials. The equivalent LUT count for 3rd degree polynomials is 18.6% less and up to 29.3% less for higher degrees. The hardware savings become more significant as the polynomial degree increases.

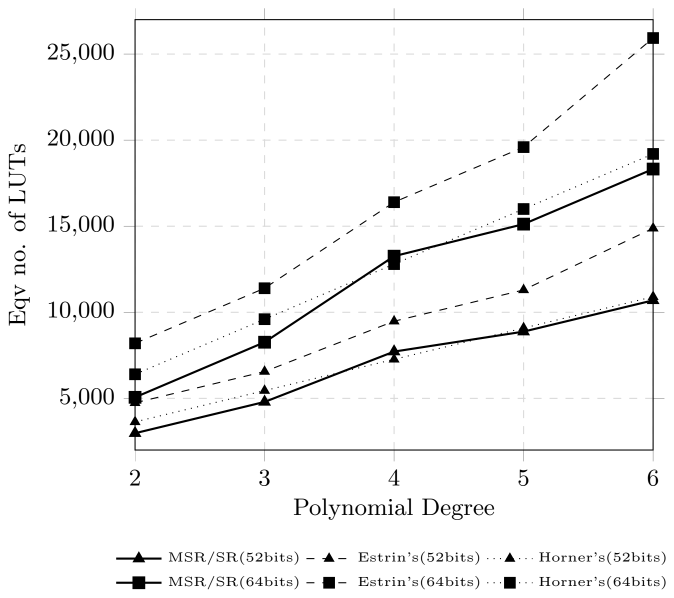

Figure 7: Equivalent hardware cost for all evaluators.

Although other FPGA devices have different equivalent LUT counts, the novel methods still shows an advantage in terms of overall hardware cost as savings are dominated by the use of fewer DSP blocks. The SX series DSP-rich FPGAs from Xilinx, with a different LUT to DSP block ratio, give a 3% reduction in area savings against Estrin's method and Horner's rule, for example.

Meanwhile, the SR and MSR methods also show a benefit in terms of latency compared to Horner's rule, as shown in Table 7. The evaluator using the SR method can achieve up to 54.4% shorter latency for a wordlength of 52 bits and up to 57.9% shorter latency for a wordlength of 64 bits. For the MSR method, the figures are 40.0% and 42.9% less than Horner's rule.

bits	*deg*	*SR*	*MSR*	*Estrin's*	*Horner's*
52	2	27	27	25	30
	3	29	42	30	45
	4	39	39	35	60
	5	39	52	35	75
	6	41	54	48	90
64	2	36	36	34	42
	3	38	57	42	63
	4	51	51	47	84
	5	51	70	47	105
	6	53	72	66	126

Table 7: Latency comparison for all evaluators with F_{max} all above 400MHz.

Comparing to Estrin's method, the SR method is as fast in terms of latency for both 52 and 64 bit wordlength polynomials below degree 4. For 4^{th} and 5^{th} degree polynomials, the SR method is four stages or 11.4% longer than Estrin's method, as it can only compute two groups of terms in parallel while the other is able to compute three at the same time. With a 14.6% shorter latency for 6^{th} degree polynomials, the SR method shows its benefit in higher degrees where Estrin's method is limited by the slow computation of x^6. Although the details of implementation are not presented here, the latency benefit for the SR method is sustained for polynomials with degree higher than 6 and below 15, as the worst case path for the SR method does not change while the worst case path for Estrin's method becomes longer.

On the other hand, the MSR method increases latency by 34.5% on average compared to the SR method for 3^{rd} 5^{th} and 6^{th} degree polynomials and therefore the latency is longer than Estrin's method (but still much shorter than Horner's). There is no penalty in latency for 2^{nd} and 4^{th} degree polynomials evaluated using the MSR method.

6. APPLICATION TO FUNCTION EVALUATION

Function evaluation is one of the most important applications for polynomials. In this section, we present function evaluators using the SR/MSR method and compare the performance with designs generated by the FloPoCo fixed point function evaluator module [13], which uses Horner's Rule, as well as function evaluators using Estrin's method that we have built.

The function evaluators built in this section are used to approximate the function $log_2(x+1)$ in the range of $x \in [0,1]$. $3^{rd}, 4^{th}, 5^{th}$ and 6^{th} degree polynomials are used in the designs with input/output precisions of 36, 52, and 64 bits. The same amount of range reduction is applied for all the evaluators, which divides the range of x into sub intervals to achieve higher precision. Coefficients generated from the approximation process for each interval are shared among all the methods to minimize approximation error.

We implemented evaluators with the SR and MSR methods as well as the reference designs targeting the same Xilinx Virtex 6 FPGA (XC6VLX240T-1) using ISE 13.4 with default settings. Note that the FloPoCo designs are pipelined by the tool using the default strategy. We additionally enable the DSP optional registers and add one register stage after the DSP blocks to achieve higher frequency. The other parallel evaluators evaluated are pipelined manually using a similar pipelining strategy.

Table 8 summarizes the hardware results for all evaluators. The latencies are in clock cycles, with the DSP blocks fully pipelined. After each optimization iteration, a new set of coefficients must be applied to both Estrin's and the SR method. For Estrin's method, it is a simple rounding for reducing complexity while for the SR method, the coefficients are re-generated and then rounded to the optimized wordlength. Coefficients are stored in BRAMs after being generated and optimized. Although final coefficients may not have exactly the same wordlengths, they all fit into same number of BRAM blocks. Therefore, the BRAM count is not included in the results.

With effective optimizations, all of the methods are able to reduce DSP block usage. Note that the wordlengths of the operands are sometimes either too large to fit into a smaller multiplier or too small to fully utilize the DSP blocks in a larger multiplier. Therefore, LUTs are used to complete the multiplication rather than wasting DSP resources. This is reflected in the LUT usage increases in all the evaluators.

The SR method is extremely efficient for the 3^{rd} degree polynomial evaluator with a wordlength of 36 bit. It is 24.2% smaller in terms of equivalent LUTs and 36.8% faster in terms of latency than the FloPoCo design. Compared to Estrin's method, where both evaluators have 12 pipeline stages, it is 35.2% smaller in terms of area. The performance gain and area savings are mainly as a result of the faster and smaller squarers used in place of multipliers.

It is also efficient for the 5^{th} degree polynomial evaluator with a wordlength of 52 bits, where the SR method is 52.4% faster in latency than FloPoCo at the cost of only 4% more equivalent LUTs. Compared to Estrin's method, which can also achieve the same latency gain for the same polynomial, the SR method saves two DSP blocks and 49 LUTs.

For the 4^{th} degree evaluators with wordlengths of 36 and 52 bits, the SR method has a 45.8% reduced latency compared to the FloPoCo design on average. However, the equivalent LUT overhead is larger, averaging 17.6%. Evaluators using Estrin's method for these two polynomials do not have significantly reduced latency compared to the SR method, but require 6.6% more hardware resources.

When the degree is as high as 6 and the precision requirement is up to 64 bits, the SR method needs 38.3% more hardware resources than the FloPoCo designs to maintain equal evaluation error. Meanwhile, due to the large multiplication, the latency gain reduces to 34.6%. Estrin's method can only achieve a similar latency gain with two more DSP blocks and 119 more LUTs.

On the other hand, the MSR method is 14.1% faster in terms of latency than FloPoCo while it uses only 8 more DSP blocks but less than half the number of LUTs compared to the FloPoCo 6^{th} degree evaluator. Although it has more computations and the latency is 12.8% longer than the SR method, it saves 17.2% in terms of resources due to smaller multipliers. Interestingly, the MSR method is also efficient for 5^{th} degree polynomial evaluation, where the MSR method is only 3 stages slower than SR method, but reduces the equivalent LUT usage by 15.4%. This translates into a 45.2% latency gain and 11.9% area saving compared to the FloPoCo design. As the MSR method has a smaller evaluation error by design, it can be used with fewer operand bits compared to the SR method, reducing area and minimising the latency overhead. Thus it is useful in high precision, large degree polynomial evaluation. For lower degrees, although it is smaller than the SR method, the MSR method has a longer latency.

After pipelining, all the parallel evaluators can achieve an operating frequency in the range of 375MHz to 385MHz, agains the FloPoCo designs which only achieve a frequency below 334MHz, representing more than a 12% throughput improvement. Further pipelining on top of the FloPoCo default pipeline strategy could increase the frequency, however, it would increase latency. Note that as the coefficient BRAM sizes are the same, the latency for the BRAM reads is identical for all the evaluators and so is not detailed here.

We have also implemented both Estrin's method and the SR method to approximate other functions, including $sin(x)$ and $\sqrt{1+x}$. For lower degree polynomials, the SR method

		FlopoCo					Estrin's Method				
bits	*deg*	*LUTs*	*DSPs*	*eq. LUTs*	*Latency*	*F (MHz)*	*LUTs*	*DSPs*	*eq. LUTs*	*Latency*	*F (MHz)*
36	3	314	6	1490	19	320	174	8	1642	12	392
36	4	280	7	1652	26	319	214	10	2174	13	395
52	4	665	14	3409	33	332	189	20	4109	18	384
52	5	901	18	4429	42	334	350	24	5054	20	378
64	6	1215	26	6311	52	322	615	44	9239	33	380
		Square-Rich					Modified Square-Rich				
bits	*deg*	*LUTs*	*DSPs*	*eq. LUTs*	*Latency*	*F (MHz)*	*LUTs*	*DSPs*	*eq. LUTs*	*Latency*	*F (MHz)*
36	3	149	5	1129	12	394	149	5	1129	17	390
36	4	190	9	1954	14	388	212	8	1780	19	385
52	4	259	19	3983	18	381	259	17	3591	21	380
52	5	301	22	4613	20	380	375	18	3903	23	375
64	6	496	42	8728	34	379	563	34	7227	39	376

Table 8: Performance and hardware cost for all evaluators when used to approximate $log_2(x+1)$.

demonstrates that, with faithful rounding, it is able to outperform both Estrin's method and Horner's rule similarly to the results shown in Table 8. However, to achieve 64 bit precision or higher where a 6^{th} degree polynomial is required, Estrin's method, the SR method and the MSR methods are all unable to match the evaluation error of Horner's rule. The solution to this problem would be to further increase the number of intervals, and reduce the range of each to compensate for the loss of precision due to evaluation error. The hardware overhead would be more BRAMs and the latency penalty would be negligible. However this refinement is beyond the scope of this paper.

7. CONCLUSION

In this paper, an efficient polynomial evaluation algorithm is presented. It can achieve a 57.9% latency reduction over Horner's rule or a 14.6% latency reduction over Estrin's method in general fixed point implementation without faithful rounding of the coefficients on a Xilinx Virtex 6 FPGA, with the help of an efficient squarer design. It can achieve hardware savings over Horner's rule implementations and over 38.3% area reduction compared to Estrin's method.

When the novel method is applied to a function evaluation application on FPGA, using the SR method can be 52.4% faster in latency than design generated by FloPoCo using Horner's rule with 4% equivalent LUT overhead. For higher precisions, although the latency of the SR method is still 34.6% shorter than the FloPoCo design at a cost of 38.3% more hardware resources, the MSR method is more efficient as the latency is only 12.8% longer than using SR method but with a 17.2% reduction in area overhead. Both the SR and MSR methods are much smaller than Estrin's method in terms of area and have lower latency than designs using Horner's rule. We aim to demonstrate more function evaluators, and larger polynomial degrees, before releasing the source code for our generator tool.

8. REFERENCES

[1] J.-M. Muller, *Elementary functions: algorithms and implementation.* Birkhauser Boston, Inc., 1997.

[2] L. Rabiner, J. McClellan, and T. Parks, "FIR digital filter design techniques using weighted Chebyshev approximation," *Proceedings of the IEEE*, vol. 63, no. 4, pp. 595–610, 1975.

[3] G. Estrin, "Organization of computer systems: the fixed plus variable structure computer," in *Proceedings of Joint IRE-AIEE-ACM Computer Conference*, 1960, pp. 33–40.

[4] J. Duprat and J.-M. Muller, "Hardwired polynomial evaluation," *J. Parallel Distrib. Comput.*, vol. 5, no. 3, pp. 291–309, 1988.

[5] A. Tisserand, "Hardware reciprocation using degree-3 polynomials but only 1 complete multiplication," in *Proceedings of Midwest Symposium on Circuits and Systems*, 2007, pp. 301–304.

[6] J. A. Pineiro, J. D. Bruguera, and J. M. Muller, "FPGA implementation of a faithful polynomial approximation for powering function computation," in *Proceedings of Euromicro Symposium on Digital Systems Design*, 2001, pp. 262–269.

[7] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *IEEE Trans. Comput.*, vol. 54, no. 12, pp. 1520–1531, 2005.

[8] A. Tisserand, "High-performance hardware operators for polynomial evaluation," *Int. J. High Perform. Syst. Archit.*, vol. 1, no. 1, pp. 14–23, 2007.

[9] N. Brisebarre, J. M. Muller, and A. Tisserand, "Sparse-coefficient polynomial approximations for hardware implementations," in *Conference Record of Asilomar Conference on Signals, Systems and Computers*, 2004, pp. 532–535.

[10] M. Wojko and H. ElGindy, "On determining polynomial evaluation structures for FPGA based custom computing machines," in *Proceedings of Australasian Computer Architecture Conference*, 1999, pp. 11–22.

[11] B. Rachid, S. Stephane, and T. Arnaud, "Function evaluation on FPGAs using on-line arithmetic polynomial approximation," in *Proceedings of IEEE North-East Workshop on Circuits and Systems*, 2006, pp. 21–24.

[12] F. Curticpean and J. Nittylahti, "Direct digital frequency synthesizers of high spectral purity based on quadratic approximation," in *Proceedings*

International Conference on Electronics, Circuits and Systems, 2002, pp. 1095–1098.

[13] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *Proceedings of IEEE International Conference on Application-specific Systems Architectures and Processors*, 2010, pp. 216–222.

[14] F. Haohuan, O. Mencer, and W. Luk, "Optimizing logarithmic arithmetic on FPGAs," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007, pp. 163–172.

[15] J. C. Bajard, L. Imbert, and G. A. Jullien, "Parallel montgomery multiplication in GF(2k) using trinomial residue arithmetic," in *Proceedings of IEEE Symposium on Computer Arithmetic*, 2005, pp. 164–171.

[16] D. De Caro and A. G. M. Strollo, "High-performance direct digital frequency synthesizers using piecewise-polynomial approximation," *IEEE Transactions on Circuits and Systems I*, vol. 52, no. 2, pp. 324–337, 2005.

[17] V. Y. Pan, "Methods of computing values of polynomials," *Russ. Math. Surv*, vol. 21, p. 105, 1966.

[18] S. Winograd, "On the number of multiplications required to compute certain functions," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 58, no. 5, pp. 1840–1842, 1967.

[19] W. S. Dorn, "Generalizations of Horner's rule for polynomial evaluation," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 239–245, 1962.

[20] K. Maruyama, "On the parallel evaluation of polynomials," *IEEE Transactions on Computers*, vol. 22, no. 1, pp. 2–5, 1973.

[21] I. Munro and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," *J. Comput. Syst. Sci.*, vol. 7, no. 2, pp. 189–198, 1973.

[22] Y. Muraoka, "Parallelism exposure and exploitation in programs," Ph.D. dissertation, 1971.

[23] M. Abbas and O. Gustafsson, "Computational and implementation complexity of polynomial evaluation schemes," in *Proceedings of NORCHIP*, 2011.

[24] M. Bodrato and A. Zanoni, "Long integers and polynomial evaluation with Estrin's scheme," in *Proceedings of International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2011, pp. 39–46.

[25] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "idea: A dsp block based fpga soft processor," in *Proceedings of the International Conference on Field Programmable Technology (FPT)*, Dec. 2012, pp. 151–158.

[26] F. Brosser, H. Y. Cheah, and S. A. Fahmy, "Iterative floating point computation using FPGA DSP blocks," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2013.

[27] B. Ronak and S. Fahmy, "Evaluating the efficiency of DSP block synthesis inference from flow graphs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 727–730.

[28] S. Xu, S. A. Fahmy, and I. V. Mcloughlin, "Efficient large integer squarers on FPGA," in *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2013, pp. 198–201.

[29] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Proceedings of Interenational Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2009, pp. 250–255.

[30] G. Melquiond. (2013) Gappa. [Online]. Available: http://gappa.gforge.inria.fr/

[31] Xilinx Inc, "Virtex-6 FPGA Data Sheet: DC and Switching Characteristics," Xilinx Inc, 2012.