On Determining Polynomial Evaluation Structures for FPGA based Custom Computing Machines

Mathew Wojko
Dept. of Electrical & Computer Engineering
The University of Newcastle
Callaghan, NSW 2308, Australia
mwojko@ee.newcastle.edu.au

Hossam ElGindy
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
hossam@cs.newcastle.edu.au

Abstract. In this paper we identify and present techniques for determining and constructing parallel evaluation structures for polynomials on Custom Computing Machines (CCMs). Since CCMs achieve high throughput rates by the effective use of pipelining, our primary interest is of the construction of resource efficient structures for any given polynomial $P(x)$. Specifically, we consider CCMs whose configurable logic resource exists as a set of Field Programmable Gate Arrays (FPGAs). We target our polynomial structure implementations for FPGAs and show how to construct them conscious of area requirements. Compared to known evaluation techniques for traditionally known parallel processing paradigms, our results are justified for providing low area parallel evaluation structures on CCMs.

1 Introduction

Custom Computing Machines provide a hardware/software programmable platform mixing processing strengths to obtain overall speedup for application execution. Generally, a core processor resourced with re-usable hardware utilises this to its advantage, configuring it as necessary to assist in execution of the task at hand. The configurable hardware is fast in comparison, and if used appropriately can provide large speedups over conventional processor based computing engines. Most common devices used for the configurable logic resource are FPGAs. FPGAs provide a reusable hardware resource that can be programmed, and reprogrammed during operation for the task at hand. Figure 1 presents the basic architectural components of a CCM.

There exist many CCM implementations, most subscribing to the basic architecture presented. Of the more famous CCMs, such implementations include Splash, DecPerle-1, Pam, Space and the Virtual Computer. These architectures gain their computational advantage by providing high performance throughput within the configurable hardware resources. Based on the systolic nature of

Proceedings of the Fourth Australasian Computer Architecture Conference, Auckland, New Zealand, January 18–21 1999.

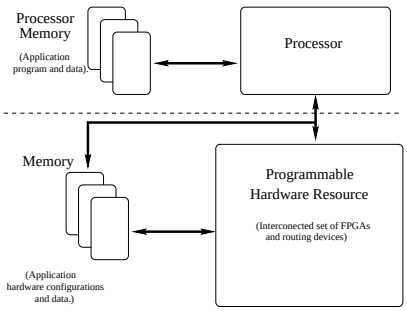


Fig. 1. Basic Architecture of a Custom Computing Machine.

these architectures, and the fact that parallelism is exploited within the hardware resource provided by the FPGAs there exist many applications that benefit with performance speedups when implemented. Such applications include signal processing techniques, image identification, data encryption/decryption, text searching and neural networks.

In implementing any application on a CCM, a developer must choose what part of the application to implement in hardware, and be aware of its relevant hardware AT costs. In general the main aim is to identify parts of the application application that perform better in hardware and generally operate on large amounts of data, allowing fast pipelining and high throughput to take place. For applications with arithmetic functions containing polynomial expressions, a high throughput hardware evaluator for the polynomial can easily remove the burden of a processor having to calculate the result in many cycles. Polynomials can be disguised in many forms other than their general form. For example, many functions can be approximated by Taylor or power series which are themselves polynomials. If the polynomial operates on large sets of data, the high throughput of the hardware evaluator can provide substantial benefit. However since multiply and square operations are costly for FPGAs, the polynomial evaluator must be conservative on its use of these operations while not sacrificing the throughput rate for its hardware implementation.

2 Background

Parallel evaluation of polynomials has been studied for many years. Of this work, most of it has been presented for parallel computation models where it is assumed that a single processing element is capable of performing an add, multiply or square operation within a unit of computation time. Additionally, it is assumed that a processing element counts as a unit of computing resource. Research within this domain has provided results under both resource constrained and unconstrained parallelism. For an nth degree polynomial, it has been shown with an arbitrary large number of processors that $2logn$ computational steps are required by the binary splitting technique [Estrin, 1960]. With k processors, Dorn [Dorn, 1962] presented a technique requiring $\frac{2n}{k} + 2logk$ steps. Further work has shown that at least $\lceil logn \rceil + 1$ steps are required to evaluate an nth

degree polynomial [Munro and Patterson, 1971] and that if $T_k(n)$ is the number of steps required to evaluate under k-parallelism then a lower bound of $T_\infty(n) \geq logn + O((logn)^{\frac{1}{2}})$ exists [Maruyama, 1973].

These results illustrate the area–time tradeoff that exists for polynomial evaluation. For an nth degree polynomial, a single processor requires $2n$ steps to evaluate the result (by Horners Rule, refer to [Borodin and Munro, 1975]), k processors require $\frac{2n}{k} + 2logk$ steps, and an arbitrarily large number of processors require $logn + O((logn)^{\frac{1}{2}})$ steps. However for a CCM, implementing these techniques in hardware will not guarantee the same area–time tradeoff, particularly if pipelining is used. Additionally, the primitive operations of squaring, multiplication and addition which form the basis of computation, all require different amounts of hardware resource and have different time characteristics. If we consider the cost of area required by a binary adder implemented on an FPGA to be $C_a = n$, where n is the input bit size, we can provide relative relations for binary multiplication, binary squaring and multiplication by a constant coefficient, as $C_m = 2n^2$, $C_s = n^2$ and $C_c = \frac{1}{2}n^2$ respectively. From previous studies of binary multipliers on FPGAs [Wojko and ElGindy, 1997], we see a parallel array multiplier to consist of n n-bit adders, and the parallel add vector based multiplier to require similar hardware resource. Coupling this wth the logic required to produce the partial products (which requires exactly n^2 AND operations) we derive the hardware cost of an n-bit multiplier to be $C_m = 2n^2$. To compute the square, it has been shown how to reduce the number of bits representing the partial products by half by efficient computation [Smith, 1989]. As a result the hardware cost of an n-bit binary square is $C_s = n^2$. For a constant coefficient multiplier, which computes the result cx, the value c is stored in the hardware implementation. Implementations using common Look-Up Table (LUT) based FPGAs require about a quarter of the hardware resource needed to implement two input binary multipliers of the same bit size, giving $C_c = \frac{1}{2}n^2$.

For the remainder of this paper we are interested in the problem of how to construct polynomial evaluators using the primitive operations mentioned above where each has an associated hardware cost, such that the overall cost of the polynomial evaluator is minimised.

3 Common Polynomial Evaluation Techniques

To evaluate any given polynomial $P(x)$ of degree n using the configurable logic resource of a CCM, we must provide a hardware implementation which calculates the polynomial output $P(x)$ given an input x. In this section, show the use of polynomial sub-factorisation to provide a means of reducing the hardware requirements of the evaluator. In this comparison we assume that the polynomial coefficients are fixed for the determination of the evaluation implementation, allowing some of these inputs to be placed in hardware. It may be possible to change these coeficients during run-time by the use of run-time configuration by using self configurable multiplier techniques [Wojko and ElGindy, 1998]. The three evaluation techniques we investigate include Horner's Rule, an all power term calculation technique, and the binary splitting technique. An example top-level layout for all three techniques where $n = 8$ is presented below in Figures

2(a), 2(b) and 2(c). A description and hardware cost analysis of all three techniques follows.

3.1 Horner's Rule

Horner's Rule is the polynomial evaluation technique which suggests writing $P(x) = \sum a_i x^i$ as:

$$P(x) = (\dots(a_n x + a_{n-1})x + a_{n-2})\dots)x + a_0 \tag{1}$$

Clearly, for an n^{th} degree polynomial we require n multiplications and n additions. Ostrowski found this to be an optimal method for evaluating polynomials using the least number of multiply and add operations [Borodin and Munro, 1975]. To implement this technique within an FPGA hardware resource, we realise that the first term evaluated, $a_n x$ is a constant value a_n multiplied by the input x. We can thus evaluate this term by use of a constant coefficient multiplier, where the value a_n is placed in hardware. Thus for an n^{th} degree polynomial evaluator we require one constant coefficient multiplier, $n-1$ multipliers and n adders. This results in a hardware cost of $C_c + (n-1)C_m + nC_a$.

3.2 Power Term Calculation Technique

For an n^{th} degree polynomial, the power term calculation technique first computes all powers x^i, $i = 1 \dots n$, and then multiplies each power result x^i by its corresponding coefficient a_i to calculate each polynomial term $a_i x^i$, and then sums the result of each to provide the polynomial result $P(x)$. The hardware implementation cost for a polynomial evaluator of degree n is as follows. To calculate all powers x^i we encounter the following cost: If n odd, then $\lfloor \frac{n}{2} \rfloor$ multiply, and $\lfloor \frac{n}{2} \rfloor$ square primitives are required. Otherwise $\frac{n}{2}$ squares and $\frac{n}{2}-1$ multiplies are required. To multiply each power result x^i by its coefficient a_i, n constant coefficient multipliers are required. Then adding all terms in parallel requires n adders. From this we see associated hardware cost for evaluating a polynomial $P(x)$ of degree n as:

(i) $\lfloor \frac{n}{2} \rfloor C_s + \lfloor \frac{n}{2} \rfloor C_m + nC_c + nC_a$ if n odd, or
(ii) $\frac{n}{2} C_s + (\frac{n}{2} - 1)C_m + nC_c + nC_a$ if n even.

3.3 Binary Splitting Technique

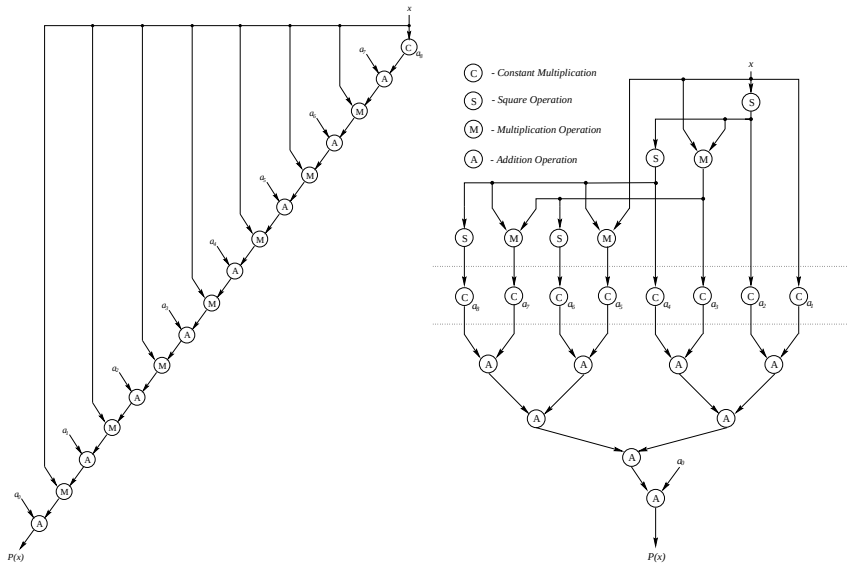
For the evaluation of an n^{th} degree polynomial $P(x)$ the binary splitting technique [Estrin, 1960] recursively computes

$$P(x) = q(x)x^{\lfloor n/2 \rfloor + 1} + r(x), \tag{2}$$

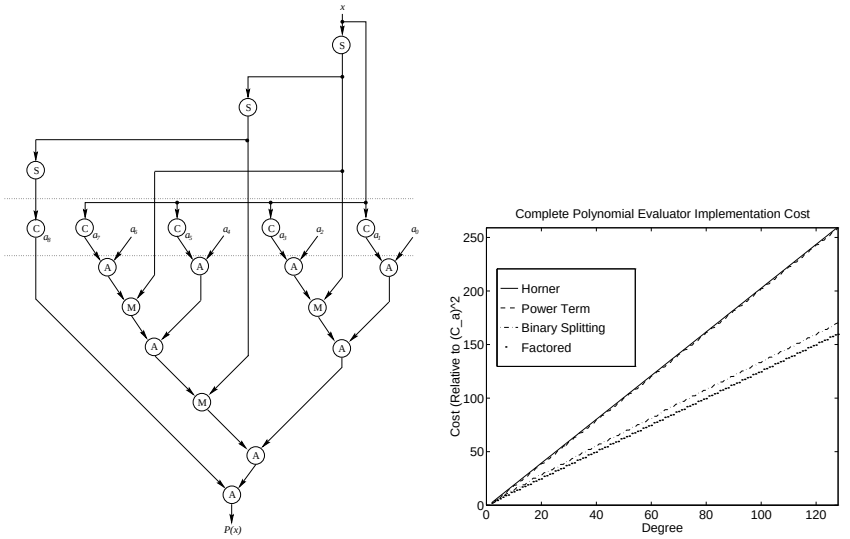
where

$$q(x) = \sum_{i=0}^{\lceil n/2-1 \rceil} a_{i+\lceil n/2 \rceil + 1} x^i \quad \text{and} \quad r(x) = \sum_{i=0}^{\lfloor n/2 \rfloor} a_i x^i. \tag{3}$$

To implement the binary splitting technique to evaluate a polynomial of degree n, we require the following operators: $log_2 n$ square operators are required to calculate the power results which to multiply $q(x)$ at each split. The multiplication of $q(x)$ by the power results occurs $\lfloor \frac{n-1}{2} \rfloor$ times during the recursive computation for $P(x)$. As a result, $\lfloor \frac{n-1}{2} \rfloor$ multiply operators are required. At the bottom level of recusion, $\lfloor \frac{n+1}{2} \rfloor$ constant multiplications are required, and overall n add operations are used. Thus, the hardware implementation cost of the binary splitting polynomial evaluation technique is $log_2 n \cdot C_s + \lfloor \frac{n-1}{2} \rfloor C_m + \lfloor \frac{n+1}{2} \rfloor C_c + nC_a$.



(a) (b)



(c) (d)

Fig. 2. Top-level layout of three different complete polynomial evaluation techniques. From (a) to (c), Horner's Rule, the Power Term calculation technique, and the Binary Splitting technique and (d) an implementation comparison between them.

3.4 Evaluation by Factorisation

For the three methods reviewed above, implementation results are presented in Figure 2(d). Of these three the binary splitting technique provides lowest cost implementations. This reduction in cost is acheived by splitting the polynomial into sub-polynomials for which common power operators are factored out. As an example, a sub-polynomial of the form $a_i x^m + a_j x^n$ acheived from a split can have a common power factored out giving $(a_i x^{m-n} + a_j)x^n$. The result, is the reduction of a constant multiplier within the implementation since the coefficient a_j is added to the term $a_i x^{m-n}$ before it is multiplied by x_n. From this, we view the factorisation of polynomials as a means for deriving minimal cost implementations of polynomial evaluators where no manipulations on input coefficient values are performed. Where the results from the binary splitting technique are obtained from a structured factorisation method, the lowest curve labelled factored, provides better results where factorisations were tailored specifically for each polynomial case. It follows, that given any polynomial $P(x)$ we must obtain a factorisation for that polynomial such that minimal hardware implementations for the evaluation of $P(x)$ are acheived.

4 Constructing Power Term Evaluators

A first step to establishing the evaluation of a given polynomial $P(x)$ is the evaluation of power terms. Given a factorisation for a polynomial $P(x)$, we are supplied a set of power terms that must be evaluated such that the factorisation itself can be evaluated. In this section we investigate the problem of best implementing structures computing any requested set of power terms T. We define the set $R \in \mathbb{N}$ to be the set of all indices of power terms in T, and n as $max\{R\}$. In previous work, *best* refers to the least number of multiplications required to evaluate the power functions T. Known techniques reviewed by Knuth [Knuth, 1969] include the binary method of exponentiation and factor method. However these techniques do not provide the optimal result for the fewest number of required multiplications in all cases, and generally do not combine well in eliminating overlapping primitive operations for the evaluation of multiple power terms. On the other hand, addition chains are viewed as giving the most economical method to compute T. Since exponents are additive under multiplication, addition chains provide a representation of the primitive operations and intermediary values required for the evaluation of a given set of power exponents R. An addition chain is defined as follows.

Definition 1 *An addition chain for n is defined as an ascending sequence of integers $a_0, \ldots a_r$ such that $a_0 = 1$ and $a_r = n$ where $n, r \in \mathbb{N}$, with the property that for all $1 \leq k \leq r$, there exists $i, j \leq k$ such that $a_k = a_i + a_j$. The minimal length, r, for which an addition chain exists is denoted $l(n)$.*

There has been considerable work performed on calculating minimal length $l(n)$ addition chains for any input n. This work can be extended to calculating minimal length $l(R)$ addition chains A satisfying the set R, i.e. $R \subseteq A$. Knuth [Knuth, 1969] reports results and properties of addition chains, Bleichenbacher [Bleichenbacher and Flammenkamp, 1998] presents an efficient algorithm for computing the shortest length chains with impressive results, while other results

show lower bounds for the chain length [Schonhage, 1975], and the proof that the problem of the fewest additions for an arbitrary given sequence of integers is NP-complete [P. Downey and Sethi, 1981].

For all of these results, no distinction is made between the difference of square and multiplication primitives and their use. It is assumed that a square operation is counted as a multiply with two inputs the same. The objective is to provide the shortest sequence A containing the values R, i.e. $R \subseteq A$, having minimal length $l(R)$. However, if we define $c(R)$ as the minimal cost of an addition chain for R, it is true that all chains for R satisfying $l(R)$ are not guaranteed to satisfy $c(R)$. For example consider the two minimal length 4 addition chains, $1, 2, 3, 5$ and $1, 2, 4, 5$, with differing respective costs $2C_m + C_s$ and $C_m + 2C_s$.

4.1 The Minimal Cost Addition Chain Problem

Definition 2 *For a given set of integers R and cost C, the minimal cost addition chain problem is to find if an addition chain A exists where $R \subseteq A$ such that its associated cost $C_A < C$ where*

$$C_A = \textstyle\sum_{a \in A} w(a) \text{ where } w(a) = \begin{cases} C_M & \text{if } a = b + c,\ b \neq c,\ a, b \in A, \\ C_S & \text{if } a = 2b,\ b \in A, \\ 0 & \text{if } a = 1. \end{cases}$$

This defines the problem crucial to calculating the optimal hardware cost evaluation for the set of power terms T. This problem can be shown to be NP-complete since a specific case of it where the costs $C_m = C_s$ is presented in [P. Downey and Sethi, 1981]. We now provide an efficient search algorithm that provides minimal cost addition chains for any given R. The basic approach of our algorithm is to compute an addition chain of reasonably low cost z for R, such that $c(R) \leq z$ and then to show that no chain of cost less than z exists. To do this, we provide techniques giving close lower bounds for solution subspaces to determine if there solution need be evaluated, and store and make use of previously computed information of subsets of R (i.e. the results of computation of $c(R')$ where $R' \subset R$) to improve computational efficiency. The solution S to R and z is computed recursively by computing S' from R' and z'. If a solution S' exists then $S = S' \cup max\{R\}$, otherwise no solution exists is returned. R' is constructed from $R' = \{R \backslash max\{R\}\} \cup \{x, y\}$ where $maxR = x + y$, and $z' = z - C_s$ if $x = y$ or $z' = z - C_m$ otherwise. This recursive technique, otherwise referred to as back-tracking has previously been used to compute minimal length addition chains [Knuth, 1969; Bleichenbacher and Flammenkamp, 1998].

As mentioned previous, efficient searching of the solution space is achieved by the use of lower bound cost computations for a given request. If we have a request and know that the cost lower bound for a solution to that request is equal to, or more than the cost that we wish to better then we can ignore searching for a solution to that request (refer to line 6). The tighter the lower bound is, the more efficient our algorithm will be. We provide two theorems below which are used to calculate the lower bound cost for an addition chain to the value n, and the cost of operations required within a subsequence of an addition chain between two values k and r where $k > r$. For each case we omit the proof,

however these proofs have been found. By effective use of these lower bounds we are able to reduce the number of iterations or cases analysed by the algorithm. Figure 3 shows the number of cases analysed for computing optimal cost addition chain sequences for all values up to 8192. Compared to the bounded exponential number of solutions 2^n that result for any input n (where $n = max\{R\}$, we see a significant reduction in the search space size to obtain an optimal solution even though the addition chain problem is NP-complete.

```
1.    additionChainSearch(R, z)
2.    /* Preliminaries */
3.    if R = {1} then return S = {1}
4.    if z ≤ 0 then return 'No solution found'
5.    costLowerBound = calculateRequestLowerBound(R)
6.    if z < costLowerBound then return 'No solution found'
7.    if S = cacheRequestChain(R) exists then return S
8.    bestCost = z
9.
10.   /* Search all possible solutions */
11.   for i = n/2 downto 1 do
12.       R' = {R\{maxR}} ∪ {i, n − i}
13.       if i = (n − i) then operationCost = C_s
14.       else operationCost = C_m
15.       requestLowerBound = calculateRequestLowerBound(R')
16.       if (requestLowerBound + operationCost) < bestCost then
17.           z' = bestCost - operationCost
18.           T = additionChainSearch(R', z')
19.           if T found and calcChainCost(T) < z' then
20.               bestCost = calcChainCost + operationCost
21.               S = T ∪ {maxR}
22.       if bestCost = CostLowerBound then i = 0
23.
24.   if S exists then
25.       cacheStoreRequest(R, S)
26.       return S
27.   else return 'No solution found'
```

Theorem 1 *Given respective double and non-double costs C_s and C_m for an addition chain where $C_s < C_m$, a lower bound for the cost of any addition chain to n is $C_s\lfloor log_2 n \rfloor + C_m\lceil log_2 v(n) \rceil$ where $v(n)$ represents the number of binary ones in the binary representation of n.*

Theorem 2 *Given respective double and non-double costs C_s and C_m for an addition chain where $C_s < C_m$, a lower bound for the cost C_{rk} of any addition chain sub-sequence from r to k where $k > r$ is given by:*

(a). if $k \geq 2^t r$, then if k even $C_{rk} = tC_s$, otherwise $C_{rk} = (t-1)C_s + C_m$.

(b). if k odd and $k \geq 3 \cdot 2^t r$, then $C_{rk} = (t+1)C_s + 2C_m$.

4.2 Constructing Power Term Evaluation Structures

To construct a evaluator for a set of power terms T, we first obtain the optimal cost addition chain as per the methods described above. Once obtained, we construct the evaluator as follows.

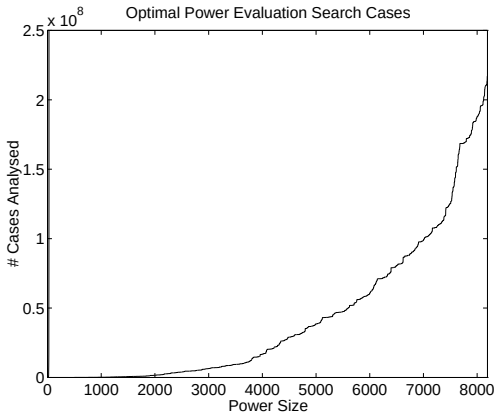


Fig. 3. Number of cases evaluated for n upto 8192.

Definition 3 *Given an addition chain A for $R \in \mathbb{N}$ defined as $a_0, \ldots a_r$ where $a_0 = 1$ and $a_r = max\{R\}$ for $r \in \mathbb{N}$, and $R \subseteq A$. For each a_k $(k = 1 \ldots r)$ construct the following:*

- *If $a_k = 2a_i$, then construct a square operator taking input from the output of the operator for a_i.*
- *If $a_k = a_i + a_j$, then construct a multiply operator taking input from the output of operators for a_i and a_j.*

The term a_0 denotes the input value x such that ever operator requiring the output of the operator from a_0 is supplied with the input value x. The output of operators for $a_i \in R$ denote the power term values required by R.

5 Constructing Polynomial Evaluators

As discussed previously, minimal hardware cost polynomial evaluators can be constructed by the effective use of factorisation. Factorisation can reduce the number of constant coefficient multiplications required and can also eliminate overlapping primitive operations from calculating each polynomial term result individually. Given that we know how to construct optimal evaluation structures for the set of power terms T, we must determine the factorisation that provides the set of power term factors from $P(x)$ such that when the factorisation is evaluated, the overall hardware cost for the evaluation of the polynomial is minimised. We define the evaluation cost of the polynomial as $C_P = C_F + C_T$, where C_F is the factorisation evaluation cost, and C_T is the addition chain hardware cost for the power terms T.

5.1 Optimal Factorisation

Computing the optimal hardware cost factorisation such that the overall hardware cost is minimised is a computationally intensive task. A first attempt may involve searching all possible factorisations and then computing the resulting

addition chain for each and identifying the best result. Obviously any feeble attempt at investigating the whole solution space will require an exponential amount of time to provide an optimal solution. We investigate the concept of addition chains extended to that of polynomial chains [Knuth, 1969]. We define a polynomial chain computing a polynomial $P(x) = a_{c_n} x^{p_n} + a_{c_{n-1}} x^{p_{n-1}} + \cdots + a_{c_2} x^{p_2} + a_{c_1} x^{p_1} + a_{c_0} x^{p_0}$ as a sequence of the form

$$x = \lambda_0, \quad \lambda_1, \quad \ldots, \quad \lambda_r = P(x) \tag{4}$$

where for $1 \leq i \leq r$,

$$\begin{array}{rll} \text{either} & \lambda_i = (\pm)\lambda_k \circ \lambda_j, & 0 \leq j, k < i \\ \text{or} & \lambda_i = a_j \circ \lambda_k, & 0 \leq k < i. \end{array} \tag{5}$$

The operation "$\circ$" denotes any of the operations "$+$", "$-$", or "$\times$", and α_j is defined as a *parameter* . This definition ensures polynomial evaluation by factorisation where no preliminary adaptation of coefficients is made (i.e. no evaluations of polynomials by factorisation such as $ax^2+bx+c = (x+d) \times (x+e)$ are allowed).

We define the cost of the polynomial chain as the sum of the individual costs of all primitive operations used the the construction of the chain. In considering an algorithm to construct optimal cost polynomial chains for any given polynomial, if we can provide accurate cost lower bounding for all possible factorisations for a given polynomial we can use efficient search techniques similar to those shown for the evaluation of power operators. Also, for this searching technique to be computationally time effective the lower bound must computed in polynomial time (as lower bounds were computed for addition chains). At this time there are no methods aware to the authors for calculating the cost lower bound in polynomial time for all possible factorisations leading to the evaluation of any given polynomial. It is critical for any algorithm developed to be able to evaluate the lower bounds for possible factorisations of a given polynomial before it will search a solution for that polynomial. If the cost lower bound for a polynomial evaluation by factorisation is less than costs of known solutions for the given polynomial, then solutions for the factorisations will be sought. Otherwise they will be skipped - reducing the solution search space.

In order to accurately determine the lower bound of an evaluation of any polynomial by factorisation, all factorisations must be considered whereby an addition chain lower bound is accurately computed for the required factor terms and the cost of the factorisation is added on. The least cost provides an accurate lower bound. But computationally, this is still exhaustive searching. Thus, we cannot enjoy the benefits of efficient solution search space pruning as experienced for the addition chain algorithm. Implementations of minimal cost polynomial chain determination algorithms have seen their running times observed as exponential. Regardless of the running times required, we present below in Figure 4 a comparison of implementations with and without factorisation for polynomials of maximum degree 16 and 32. Each graph was constructed by computing a sample set of k-term polynomials (where k is shown on the x-axis) from n, calculating the cost for each polynomial evaluation and then plotting the average cost from each sample set for each technique.

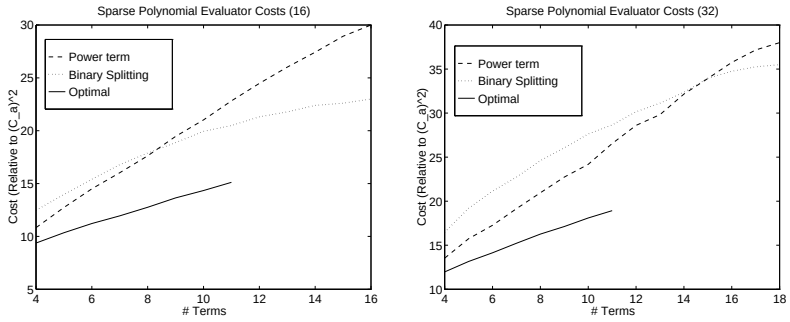


Fig. 4. Average implementation costs for k-sparse polynomials with maximum degree 16 and 32.

Observing the curves in the graphs, we see the optimal factoring technique to provide substantial hardware cost benefit for any polynomial tested. While it was shown before that factoring can provide reduced cost implementations for complete polynomials, it is illustrated here that factorisation and its benefits apply equally well to k-element polynomials of maximum degree n where $n > k$. However, not all methods of factoring apply well to k-element polynomials. This can be seen by the associated curves for the binary splitting technique. Initially the technique performs poor, for the two graphs it is observed that it is not until $k > \frac{n}{2}$ that binary splitting performs better than the power term technique.

When comparing running times of the optimal factoring technique to the power term, and binary splitting evaluation techniques, a large difference was found. This accounts for the discontinuities in the optimal curves due to the amount of computation required to calculate the solution. From simulation of varying input, we observed the running time of the power term and binary splitting techniques to be within an acceptable time frame and the optimal technique not. We view acceptable time frames to be similar to those experienced when compiling programs for execution.

It follows that to provide techniques which give low cost factored evaluations of polynomials in acceptable running times, we must investigate alternative heuristic techniques for determining factorisations. It is not the scope of this paper to present these techniques, rather identify their need, as we have, hence proposing future research directions.

6 Conclusion

In this paper we investigated the implementation of parallel evaluation structures for polynomials on CCMs. The structures were defined to be comprised of primitive binary square, multiply, constant multiply and add operations, and were targeted to be area minimal while not sacrificing throughput due to the computational nature of the hardware resource of CCMs. It was identified that to provide minimal area evaluators for any given polynomial where no preliminary adaptation of coefficients was made, effective factorisation of the polynomial terms resulted in the reduction of primitive operations and their cost.

The problem of computing minimal cost addition chains where the doubling and non-doubling operations are individually weighted was addressed and efficient techniques were developed to provide optimal addition solutions A for a request set of terms R to be present within the chain. It was shown how this problem combined with factorisation provides a means for determining evaluators for a given polynomial $P(x)$. The technique of using polynomial chains was investigated for providing optimal cost solutions for evaluating any given polynomial, however this technique was shown to perform at an exponential rate based on the number of input terms. Additionally, on average, known heuristic techniques did not provide results as anticipated for sparse polynomial evaluators. As a result future work was proposed for providing heuristic techniques giving methods for determining polynomial factorisations which result in reduced cost hardware implementations for the structures evaluating the polynomials. It is a requirement that any technique considered must provide results in an acceptable time frame.

References

Bleichenbacher, D. and Flammenkamp, A. (1998). An efficient algorithm for computing shortest addition chains. *To appear in SIAM J. of Computing.*

Borodin, A. and Munro, I. (1975). *The Computational Complexity of Algebraic and Numeric Problems.* Theory of Computation Series. Elsevier Publishing Company.

Dorn, W. (1962). Generalisations of horner's rule for polynomial evaluation. *IBM J. Res. Develop.*, 6:239–245.

Estrin, G. (1960). Organisation of computer systems – the fixed plus variable structure computer. In *IEEE Transactions on Computers*, pages 33–40.

Knuth, D. (1969). *The Art of Computer Programming*, volume Vol 2 - Seminumerical Algorithms. Addison Wesley.

Maruyama, K. (1973). On the parallel evaluation of polynomials. *IEEE Transactions on Computers*, C-22(1):2–5.

Munro, I. and Patterson, M. (1971). Optimal algorithms for parallel polynomial evaluation. In *Proc. IEEE 12th Ann. Symp. Switching and Automata Theory*, pages 132–139.

P. Downey, B. Leong and Sethi, R. (1981). Computing sequences with addition chains. *SIAM Journal of Computing*, 10(3):638–646.

Schonhage, A. (1975). A lower bound for the length of addition chains. *Theoretical Computer Science*, 1:1–12.

Smith, S.G. (1989). Incremental computation of squares and sums of squares. *IEEE Transactions on Computers*, 38(9):1325–1328.

Wojko, M. and ElGindy, H. (1997). Comparative analysis of multiplication techniques for FPGA architectures. In Shardi, Tam, editor, *Proceedings of PART'97*, pages 446–457, Newcastle, New South Wales, Australia. Springer.

Wojko, M. and ElGindy, H. (1998). Self configurable binary multipliers for LUT addressable FPGAs. In *To be presented at PART'98*, Adelaide, South Australia.