Reducing Reconfiguration Overhead for Reconfigurable Multi-Mode Filters Through Polynomial-Time Optimization and Joint Filter Design

Amir Hossein Gholamipour[‡], Fadi Kurdahi[‡], Ahmed Eltawil[‡] and Mazen A.R. Saghir*

[‡] Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
{ amirgh, kurdahi, aeltawil }@uci.edu

* Electrical and Computer Engineering
Texas A& M University at Qatar
Education City, Doha, Qatar
mazen.saghir@qatar.tamu.edu

CECS Technical Report 11-06
August 2nd, 2011

Abstract

FIR filters are vastly used in multi-mode systems where the behavior of the system changes based on user inputs or changes in the operational environment. FIR filters used for each mode of operation have different sets of parameters (coefficient sets). Partially reconfigurable FPGA platforms are shown to be viable choices to implement multi-mode filters. In our previous work [1] we proposed a clustering-based technique to design a multi-mode filter with minimal area and manageable reconfiguration delay. In this work we theoretically extend the previous work to propose an optimal polynomial time algorithm to optimize the structure of the filter. Using the optimal solution we can decrease the area by 17.2% compared to clustering approach. The results of our experiments show that our proposed heuristics give solutions within 1% of the optimal solution.

1 Introduction

An increasing number of digital systems, from wireless devices to multi-media terminals, are characterized by their multi-mode operation. This refers to the ability of a system to modify its characteristics or behavior based on user inputs or changes in the operational environment. For example, in a WiMax terminal, the characteristics of the MAC and PHY can be changed based on the current status of the wireless channel. In spectrum sensing, the matched-filter used to find idle frequency bands is changing periodically to accommodate the characteristics of the protocols operating in the corresponding band. Finally, in pattern matching applications, the pattern being searched can be modified by changing the dimensions and coefficient values of an image template. Multi-mode systems vastly incorporate Finite Impulse Response (FIR) filters as one of the main components in their design. This is due to FIR filters' inherent stability and linear phase. Therefore, one of the challenges of designing multi-mode systems is to design multi-mode FIR filters where the characteristics of the filter (value and number of coefficients) are changing depending on the mode of operation. The structure of an *n*-tap FIR filter consists of *n* coefficients and is shown in Figure 1. The Z^{-1} boxes represent delay elements.

Due to large data rate processing demand of applications incorporating multi-mode filters, software-programmable processors are not viable options to implement a multi-mode filter. ASICs on the other hand lack the flexibility to conform to varying characteristics of the design. FPGAs provide a third alternative for implementing multi-mode filters. Their short design cycles and reconfiguarbility give them an advantage over

ASICs, their inherent parallelism can be used to support high data rates, and they can be easily customized to support irregular datapaths with variable bit widths. Various implementation strategies can be targeted for FPGAs. Each of these implementation strategies has their pros and cons. *Generic design* to accommodate all varying parameters of a filter is large and slow, on the other hand *Space-multiplexing* (shown in Figure 2.a) to implement all optimized structures on the same chip is not scalable [1]. Support for dynamic partial reconfiguration (DPR) in Xilinx FPGAs enables *Time-multiplexing* (shown in Figure 2.b). This is achieved by modifying a system's architecture by loading or swapping logic blocks in user-defined reconfigurable regions while the FPGA device is operational. Time-multiplexing however, is possible at the cost of non-negligible reconfiguration overhead for the system.

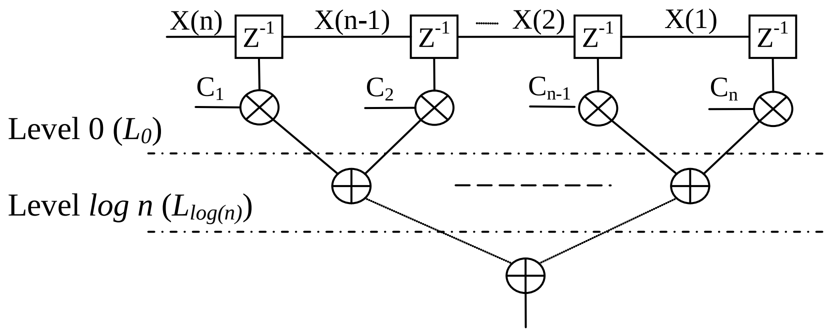
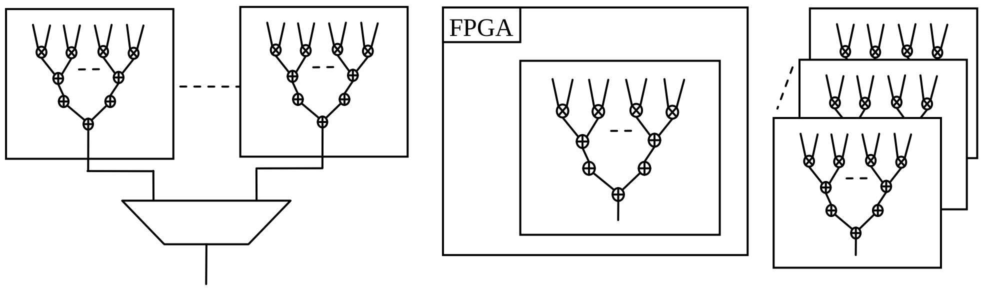


Figure 1. Direct-form structure of an n-tap FIR filter

Each of the abovementioned FPGA-based solutions exhibits a different characteristic in terms of area and reconfiguration overhead. The problem, is that the solution space is coarse with limited number of choices. Furthermore, each design alternative fully optimizes one design feature at the full cost of the other feature. From system-level perspective it is desirable to find solutions that lie between these extreme cases.



(a) **(b)**

Figure 2. Different implementations of an *n*-tap FIR filter

In our previous work in [1] we introduced the concept of *level of implementation* to partially optimize filter structures. This approach trades-off area and reconfiguration overhead at a finer granularity which is well-suited for designing systems that incorporate filters as part of their implementation. A clustering-based technique was proposed in [1] to optimize such filter structures. We further introduced *joint optimization* (aka **Joint Filter Design** or **JFD**) to exploit similarity across filters in a given sequence to provide another dimension of flexibility in fine-tuning filter structures based on system-level requirements. The result of JFD is a set of clusters (modules) that build each filter. Subsets of these clusters are shared among more than one design. This potentially reduces reconfiguration overhead. As we will show in this work, while clustering technique is effective in reducing the size of the circuit, it is not optimal. In this work we introduce an approach which minimizes the reconfigurable section of a filter. Furthermore we prove that this approach has polynomial run-time.

The above discussion presumes that the filter is mapped onto the FPGA's fine-grain fabric. However, today FPGAs widely embed hard DSP cores for DSP-intensive applications like filtering. Thus an issue that needs to be addressed at physical design level is to incorporate DSP blocks in the design of filters. JFD utilizes logic resources of

FPGA and is capable of fully or partially incorporating DSP blocks on FPGA to design filters. There are clearly significant tradeoffs between these two categories of resources. When implementing a filter structure, using just DSP blocks however, one must be aware that it limits the portability of filter designs. This in large is due to variance in availability of such resources across different FPGA devices and vendors. In our other work in [30] we studied incorporating DSP blocks fully or partially in design of filter. In that work we showed that running DSP blocks at high frequencies consumes non-negligible dynamic power. Filters implemented using logic resources on the other hand, are highly optimized in terms of area, delay and power consumption, but will require dynamic reconfiguration from off-chip memory. In [30] we proposed a comprehensive approach to incorporate both resources based on running frequencies of the circuit, input rate of the filter and availability of resources.

In Section 2 we present the related work in this area. In Section 3 we present our framework to optimize a single structure. We propose an optimal solution and show that the time complexity is polynomial. In Section 4 we propose a solution to joint-optimize a sequence of filters. Finally, in Section 5 we present the results of our experiments.

2 Related Work

The contribution of this work is to define a design space for multi-mode filter design and implementation. The design space shows the trade-off between the size of the filters and the corresponding reconfiguration time overhead for changing the filter implementation. In this work we propose algorithms to minimize the size of the reconfigurable section of the filters. In this regards the works related to this work fall in to two main categories. Works related to filter design and optimization, and works related to partial reconfiguration design and optimization on FPGAs.

There is a large body of research on FIR filters architecture. In general FIR filter structures can be categorized in three different implementation classes: Direct form, Transposed (or Inverted) form and Distributed Arithmetic (DA) [13] and [11]. For each of these implementation classes the filter can be implemented as Bit-Serial or Bit-Parallel. Bit-serial arithmetic filters [10] are used especially when clock frequency of the circuit is multiple times higher than the sampling frequency. Bit-serial implementations are small in size and have been used in early FPGAs mainly due to I/O and area constraints [12] and [14]. On the other hand bit-parallel implementation has the high throughput at the cost of large implementation size. Digit-serial implementation [15] has been introduced to enhance the design space between the two extreme implementations. In this implementation data words are divided into digits, having a digit size N, which are processed in one clock cycle. Digit-serial format offers a flexible trade-off for throughput versus size between bit-serial and bit-parallel approaches.

Distributed Arithmetic (DA) filters [11] are the other type of implementation which uses memory (mainly ROMs) to store all possible combinations of the coefficients and addresses the memory using the input bits. DAs can be implemented in bit-serial, bit-parallel or digit-serial. For today FPGAs parallel DA filters are extensively used because of the level of parallelism the FPGAs can provide and because of the suitability of Look-Up Tables (LUTs) for implementing DAs [16] and [13].

Extensive research has been dedicated to optimizing the structure of FIR filters. Simple optimizing techniques like coefficient symmetry, negative symmetry or constant multiplication are widely exploited for filters [13], [9] and [11]. Complex optimization techniques are mostly applied to the multiplier block of filters. In [17] the authors propose an algorithm based on common sub-expression elimination to reduce the size of coefficient specific multipliers for FIR filters. The idea is to avoid replicating a common sub-circuit across coefficients. Applying the technique effectively reduces the area as well as power consumption of the filter while maintaining the throughput. In [18] a greedy optimization technique is proposed to minimize the area of linear digital systems using combination of common sub-expression elimination and modification of multiplier coefficients.

Other works, target designing partially reconfigurable applications or optimizing these designs for different purposes including reconfiguration time overhead, bit-stream storage size, total design size, etc. For filters, partial run-time reconfigurability of FPGAs can be exploited to change the structure of a filter in order to implement another filter. In [19] the authors present a reconfigurable filter which is implemented on Xilinx Virtex 4 devices. The implementation uses a reconfigurable Multiply-Accumulate (rMAC) unit which can be replicated to implement filters of different sizes. In [23] a programmable FIR digital filter using Canonic Signed-Digit (CSD) coefficients is presented.

To optimize reconfiguration time overhead, [20] suggests that by carefully floorplanning the reconfigurable designs and considering the sequence of reconfiguration we can effectively reduce reconfiguration time overhead. The authors in [7] introduce Automatic Target Recognition (ATR) and suggest a heuristic to cluster the template mask filters to fit the filter on the FPGAs and minimize reconfiguration time overhead for changing the filters. Authors in [22] present a framework that measures the reconfiguration time overhead on the board while reconfiguring using ICAP port for Xilinx FPGAs.

While reconfiguration time overhead is a major concern for FPGAs, for designs with many reconfiguration scenarios and reconfigurable modules the size of the bit-stream that we can store becomes critical. In [21] the authors propose a solution to save bit-stream storage space for FPGAs. In their approach different FIR filters' bit-streams are parameterized based on the coefficients.

3 FIR Filter Design

A generic FIR filter as a regular structure of multipliers and adders can be optimized in regards to the amount of resources and size of implementation. This in specific is true for the filters with constant set of coefficients. FIR filters and FIR-like structures with constant coefficients are widely used in a multitude of wireless and multi-media applications and applications which need heavy matrix multiplication processing. The well-known techniques to reduce the size of an FIR filter structure are: 1) to replace generic multipliers with coefficient specific multipliers, 2) to apply factoring for inputs that are multiplied to the same coefficients and 3) to apply optimal adder tree generation to minimize the size of adder tree.

A large body of research has been dedicated to the techniques using which the size of the constant multipliers can be reduced like [9], [17] and [18]. Optimization techniques 2 and 3 are shown in Figure 3. As can be observed in Figure 3.a we can exploit factoring to save on the number of multipliers while using smaller adders. The two structures shown in Figure 3.b and Figure 3.c clearly show how optimally generating the adder tree can minimize the size of the adders in the filter structure.

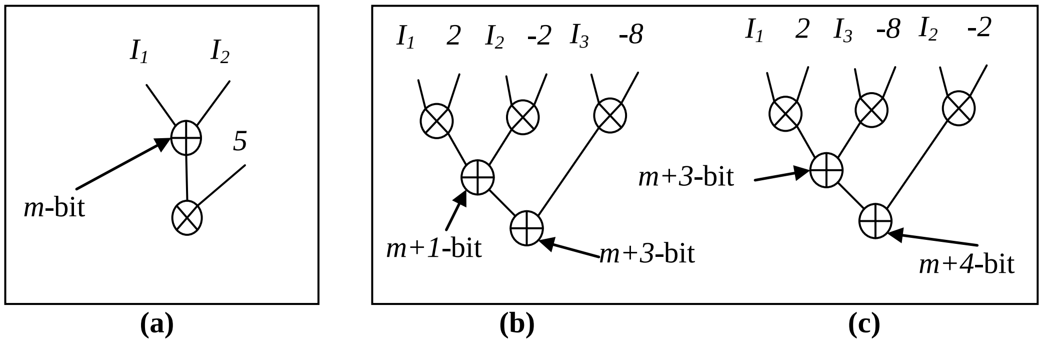


Figure 3. Adder size optimization considering the coefficients

In this work, we propose to split a filter structure to two sections: *reconfigurable* and *static* section. To realize that, we define **Levels** of an FIR filter as the rows of resources (adders or multipliers) shown in Figure 1. Accordingly **Level *i* Implementation (*LI_i*)** of the filter is to cut the structure shown in Figure 1, in to two sections between rows *i* and *i + 1*. The section above the cut line is the *reconfigurable section* of the filter and the structure below the cut is the *static region* of the filter. To change the mode of operation we need to reconfigure the reconfigurable section of the filter while static section is the same across all modes of operation. To reduce the reconfiguration overhead we need to minimize the size of the reconfigurable section of the filter for each mode of operation. We introduce RSAM problem as follows:

Reconfigurable Section Area Minimization (RSAM) problem is to use the optimization techniques introduced earlier to optimize the size of the reconfigurable section of a FIR filter for a given set of constant coefficients and a given level of implementation.

In [1] we proposed a clustering-based solution to this problem. As we will show later in this section, while the proposed clustering algorithm is a powerful technique in reducing the size of the reconfigurable section, it is not optimal. We propose an optimal algorithm which runs in polynomial time. We first define a few terms which will be frequently used throughout this paper.

A **Full Binary Tree** is a tree in which every node other than the leaves has exactly two children.

A **Perfect Binary Tree** [2] is a *Full Binary Tree* in which all leaves are at the same depth.

3.1 Filter Optimization

In this section we first propose a polynomial time algorithm to fully optimize a filter structure. While this is a special case of RSAM problem (highest level of implementation), the solution to RSAM can greatly benefit from the theorems and theoretical background laid to solve this problem. Throughout this work and theoretical analysis of optimization algorithms we assume that the size of adders and coefficient-specific multipliers are linearly proportional to the bit-width of the inputs. For different coefficients, corresponding multipliers' growth rate might be different.

To generate a fully optimized filter the algorithm proposed in [1] clusters all the inputs that are going to be multiplied with the same coefficient together before multiplying them. Figure 4 shows two different implementations for a 37-tap filter. The inputs and coefficients of the filter are 4-bits wide each and for the sake of simplicity it is assumed that the 37 coefficients represent three different values i.e -3 (17 inputs), 5 (17 inputs) and 4 (3 inputs). The numbers written next to the edges show the bit-width of the wires. Figure 4.a shows the implementation method used in [1]. In this implementation all the inputs that are being multiplied to the same coefficient are first added together. The triangles represent perfect binary trees of adders to add the number of inputs (power of 2) written in the triangles.

The two filter structures in Figure 4.a and Figure 4.b implement the exact same filter however as can be seen, there are key differences in the number and size of multipliers as well as the size of adders. The adders and multipliers with different sizes across the two designs are shaded in the figure. The cost of multipliers used in the structure shown in Figure 4.a, is smaller than the multiplier cost of the structure in Figure 4.b. On the other hand, the cost of adders in Figure 4.b is smaller than the adders in the structure of Figure 4.a. Therefore depending on the net size of the corresponding adders and multipliers each design might be smaller than the other one. In this particular case the design of Figure 4.b turns out to be smaller.

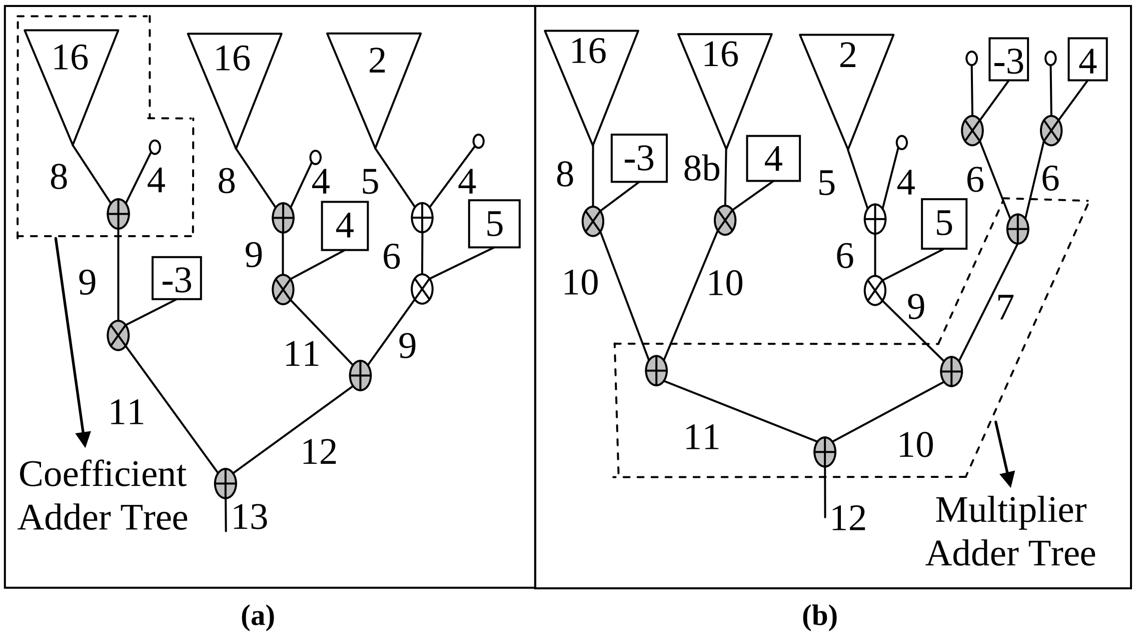


Figure 4. Building a fully optimized FIR filter

Figure 4 though given as an example, makes very good points on the characteristics of the optimal design. For the rest of this section we will highlight these characteristics and will justify them while proposing a solution to fully optimize the design of a filter (highest level of implementation).

Definition: Coefficient Adder Tree (CAT) (shown in Figure 4.a) is a binary tree to add the inputs that are being multiplied to the same coefficient. The inputs to CATs are the inputs to the filter and the outputs are inputs to the coefficient specific multipliers.

Definition: Multiplier Adder Tree (MAT) (highlighted in Figure 4.b) is a binary tree to add the outputs of the multipliers. MAT is not necessarily a balanced tree. The inputs to MAT are the outputs of the multipliers and the output of MAT is the input to the static section (in case of fully optimized filter, the only output of MAT is the output of the filter).

We have made two important observations from Figure 4.a and Figure 4.b, which are as follows:

1. In optimal filter implementation, for every coefficient, the CATs include perfect binary sub-trees[1]. The size of the sub-trees can be derived from the binary representation of the number of inputs to the CAT. We will elaborate on this later.
2. The optimal filter implementation is determined by the *optimal configuration of CATs*. The configuration of CATs is the set of perfect binary sub-trees that are included in the CAT implementation.

Lemma 1: The number of adders for implementing a filter is constant.

Proof: This is derived from the filter structure. Since the structure is represented as a binary tree and the adders represent the joint nodes of the tree, for n inputs we have $n - 1$ adders. However the number of multipliers varies in different implementations. For n inputs and k different coefficients ($k \leq n$), the number of multipliers varies between k and n.

Theorem 1: A perfect binary tree has the lowest cost to implement 2^k inputs of the same bit width.

[1] Throughout this paper, when we use the term “sub-tree”, we mean a perfect binary sub-tree unless otherwise is stated

Proof: To add two inputs, the size of the adder is equal to the max size (bit-width) of the two inputs. The output bit-width of the adder is equal to the max size of the two inputs plus one. Thus an adder's logic resources are fully utilized if both inputs are of the same size. We define **Utilization Ratio (UR)** of an adder as the cost of logic required to implement the adder considering the bit-width requirements of the inputs, over the logic cost required to implement a full adder as large as the larger input. UR of an adder depends not only on the input bit-widths but also the adder implementation algorithm. As an example for a Ripple-Carry Adder the size of an adder which adds a 5-bit and a 3-bit number is 80% the size of a full 5-bit adder (size of a Full Adder is considered to be twice as large as a Half Adder), so the UR for this adder is 0.8.

To implement an adder tree to add 2^k numbers we start from the inputs which are all of the same size. We add them together two by two. All of the inputs of the adders are again of the same size (input size plus one). We continue this. Using k level of adders we can add all the inputs (see Figure 1). The UR for every adder in this structure is 1, which means that the adder tree has minimum possible cost. The structure mentioned is indeed the structure for a perfect binary tree.

Theorem 2: The minimum cost adder tree to add n (not necessarily equal to 2^k) inputs of the same size w is obtained by a structure of perfect binary sub-trees. The size of the sub-trees is obtained from the binary representation of number n.

Proof: Figure 5.a shows the implementation of an adder tree for 22 inputs based on Theorem 2. Adder trees inside the triangles are implemented based on Theorem 1. To prove Theorem 2 we assume that there is another implementation that gives the optimal cost. We prove that the size of this implementation can be reduced by showing that some of the adders can be replaced by smaller adders.

Sort the adders in the supposedly optimal structure based on their sizes. If there are two under-utilized adders (an adder with UR less than 1) where the size of one of the inputs of the two adders is *w*, replace the smaller adder with a *w*-bit adder if the two adders are data independent (see Figure 5.b). If the adders are data dependent (the output of one directly or indirectly goes to the input of another one) then we can replace the larger adder (the dependent adder) with a *w*-bit adder as shown in Figure 5.c. If there are no such two under-utilized adders then we do the same for *(w + 1)*-bit adders and so on.

It can be shown that this process builds the binary trees based on the scheme of Theorem 2 and shown in Figure 5.a. The number of sub-trees of this adder tree is at most $(1 + \lfloor \log n \rfloor)$ where *n* is the number of inputs.

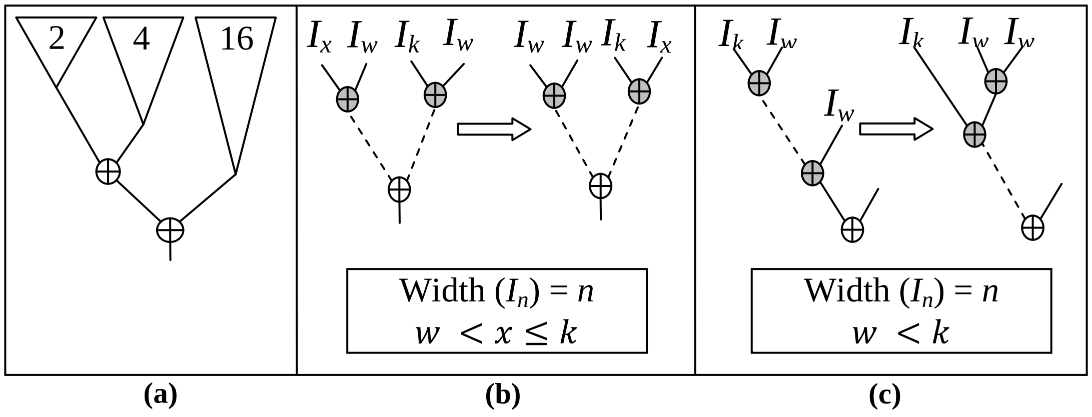


Figure 5. Implementation of adder trees for different input sizes

Theorem 2 gives a formal proof for observation 1.

Theorems 1 and 2 illustrate how to build CATs optimally. Once CATs are built for different coefficients the outputs of the adder trees are multiplied with the coefficients.

The outputs of the multipliers are added together using MAT which produces the output of a fully optimized filter or the output of the reconfigurable section. Unlike the adder trees in Theorems 1 and 2, the inputs to the MAT are not of the same size. In the following we propose an algorithm to optimally build a MAT.

3.2 Optimal Multiplier Adder Tree (MAT) Generation

The inputs to MAT are the outputs of multipliers. We assume that there are n inputs of different bit-widths to the MAT and there are k outputs ($k < n$ and for a fully optimized filter $k = 1$) which are the inputs to the static section. The problem is how to optimally (in area) generate this adder tree.

Our proposed solution is as follows: Insert the size (bit-width) of the n inputs to a list and sort this list in ascending order. At every iteration of the algorithm, check the first three elements in the sorted list. If any two of the first three elements are equal, insert an adder to add the corresponding inputs. The size of the output of the adder is inserted back to the list. We need to maintain the list ordered after each adder insertion. If no two of the first three elements are equal in size insert an adder to add the first two elements. The size of the adder is equal to the size of the larger element. After every adder insertion the size of the list is reduced by one. We continue until k elements are left in the list.

3.2.1 Proof of optimality

The proof of optimality is very similar to the proof of Theorem 2. We can assume that some other implementation gives the optimal solution. We need to iteratively check the smallest three inputs. If two of the inputs are equal in size and (in the assumed optimal implementation) are added to some other inputs, based on the scheme of Figure 5.b and Figure 5.c we can replace the adder with a smaller adder or an equal-sized adder (if the inputs are added to other equal sized or smaller input sizes). If none of the smallest three inputs are equal in size, we do the same check for the two smallest input sizes. Thus at each iteration, we are either replacing an adder with a smaller one which is contradictory to the assumption that the adder tree is optimal or we keep the same size adders which shows that our solution is also optimal.

3.2.2 Time-complexity analysis

We can sort the elements of the list in *O(n log(n))*. At each iteration we need to check the first three elements of the list which can be done in *O(1)* and inserting the size of the adder output back to the list which takes *O(n)* to maintain the list sorted. The number of iterations is *O(n – k),* thus the worst case time complexity of the proposed algorithm is *O(n (n – k)).*

3.2.3 Run-time Improvement

Inserting the result of adding two elements of the list though in the worst case takes *O(n)* can be done in a more efficient way which results in making the algorithm faster. This can be done by introducing a side list which works similar to a FIFO. In this improved version of the proposed algorithm whenever we are adding two elements, the result of the adder is inserted to the back of the side list. At each iteration we check the first three elements of the main list and the first three elements of the side list. We find the smallest three elements from these 6 elements. If two of the smallest three elements are equal in size, we insert an adder to add them and insert the size of the result to the back of the side list. Otherwise we insert an adder to add the two smallest elements and insert the size of the output to the back of the side list. After each iteration, the size of the inserted adder is equal to or larger than the size of the adder inserted in the previous iteration of the algorithm. Thus the side list always stays sorted.

While introduction of the side list does not affect the optimality of the algorithm, it reduces the worst case execution time from *O(n (n – k))* to *O(n - k)* after sorting the elements in the main list. The reason is that, at each iteration we need to sort a list of 6 elements which takes *O(1)* to select the smallest three elements. Also inserting the result to the side list can be done in *O(1)*. Thus the time complexity of the algorithm is *O(n log(n))* for sorting the main list and *O(n - k)* for finding the optimal solution.

3.2.4 Circuit delay analysis

Figure 6.a and Figure 6.b represent two MAT implementations for a set of inputs. The numbers associated with the edges, represent the bit-width of the corresponding connection. The two MAT implementations are both optimal in size, but as can be observed, have different circuit delays. To enhance our algorithm to build a structure with better delay characteristics, we need to add a parameter for each entry in the sorted list. This **Level** parameter as we call it, gives a notion of the number of adders in a row (similar to critical path delay calculation) used to generate that entry. So for all the entries the level parameter is initially set to 0. To improve the critical path delay of the adder tree from all possible choices for optimally inserting adders, we choose the entries that have lower level number. This way we reduce the number of adders in a row which leads to critical path delay reduction while the size of the design is still optimal.

Theorem 3, states an important conclusion drawn from the proposed algorithm for building an optimal MAT.

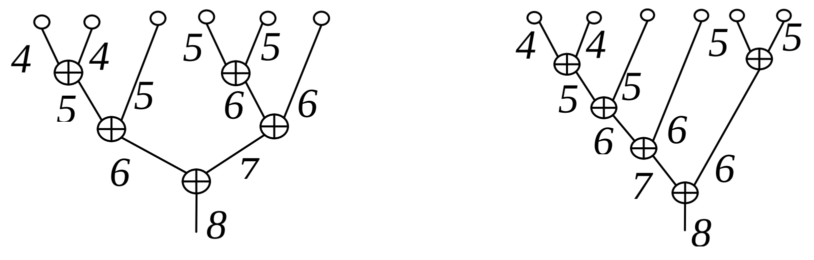
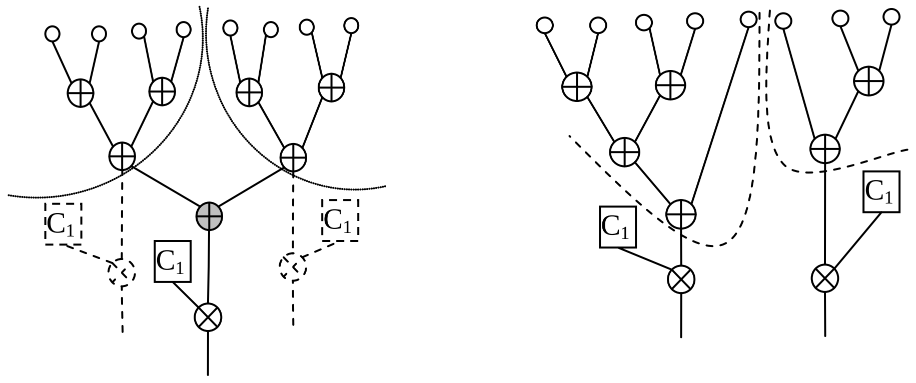


Figure 6. Optimal adder tree for *n* inputs of different sizes

Theorem 3: In a fully optimized filter structure, the sub-trees of the CATs will not be split in to smaller sub-trees.

Splitting a sub-tree is to divide it in to two or more, smaller sub-trees and inserting multipliers to multiply the output of each sub-tree independently. This theorem states that in the optimal implementation of the filter the sub-trees cannot be split in to smaller sub-trees as this split increases the cost of the implementation. A CAT might be broken in to its constituting sub-trees. For example in Figure 5.a, the sub-tree of size 16 will not be split in to smaller sub-trees in the optimal implementation. However it is possible that the sub-tree of size 2 forms an independent CAT.

Proof: Splitting a sub-tree eliminates one of the adders in the sub-tree. As shown in Figure 7.a, if the sub-tree is split in to half then the largest adder which is shaded in the figure, is eliminated at the cost of an additional multiplier (see Figure 7.a). The outputs of the multipliers are of the same size, so based on the algorithm proposed to build the MAT, the outputs of the multipliers are added together. This means that the cost of the extra adder is later going to be paid as part of MAT. Furthermore, the adder needed to add the outputs of the multipliers is at least as large as the eliminated adder (depending on the coefficient it is possibly larger). Thus splitting the sub-tree at least keeps the same cost of implementation and quite possibly increases the cost. The same argument can be made for any other way to split a sub-tree such as the one shown in Figure 7.b. This means that splitting a sub-tree in any form saves an adder which later should be inserted to the MAT (based on Lemma 1 the total number of adders is constant.). However the adder inserted to the MAT will be larger than the saved adder. Besides an additional multiplier is also inserted to the structure which also increases the cost of implementation.



(a) **(b)**

Figure 7. Splitting a sub-tree in to 2 sub-trees

3.3 Filter Structure Optimization Algorithm

Theorems 2 and 3 are formal confirmations of Observation 2 we made earlier in this section. Theorem 2, mentions that in the optimal implementation of a filter the inputs are grouped to sub-trees and Theorem 3 mentions that these sub-trees cannot be split in the optimal implementation. Considering the algorithm proposed earlier to optimally generate a MAT, the optimal implementation of a filter is dependent on optimally determining the configuration of CATs. As defined earlier, configuration of a CAT is to determine which sub-trees, of the inputs of the corresponding coefficient, are added together before being multiplied to the coefficient. To optimally design a filter structure we cannot find the optimal CAT configurations for coefficients, independently from each other. In the example shown in Figure 4, the optimal CATs configurations for coefficient (-3) were different depending on the number of inputs multiplied to coefficient (4).

We propose to exhaustively search all different CATs configurations for all coefficients and determine which configuration leads to optimal implementation. As will be shown in this section, this exhaustive search does not affect the polynomial run-time of our algorithm.

3.3.1 Search Algorithm

We assume that there are n inputs to a filter. The set of coefficients include k unique values. n_1 of the inputs are multiplied with coefficient C_1, n_2 are multiplied to coefficient C_2 and so on. The number of sub-trees of a CAT of size n is at most $(1 + \lfloor \log n \rfloor)$. So for C_1 we have at most $(1 + \lfloor \log n_1 \rfloor)$ sub-trees, for C_2 this number is at most $(1 + \lfloor \log n_2 \rfloor)$ and so on. From the algorithm proposed for building MAT we know that to build a MAT of m inputs and 1 output (fully optimized filter structure) it takes $O(m \log(m))$ to sort the inputs and $O(m)$ to build the MAT. However to search all possible CAT configurations we do not need to build the MAT each time from scratch. Instead we can intelligently move through the design space so that the MAT can be sorted in $O(m)$ compared to the previous MAT. We later explain how we can search the design space to achieve updating the sorted list in $O(m)$ time. Thus it takes $O(m)$ to find the optimal solution for a configuration of CATs.

The total number of possible CAT configurations for each coefficient is the number of possible partitioning on the set of the sub-trees for that coefficient. For a set of n elements the total number of possible partitioning is the nth Bell number [3]. The equation for the *n*th Bell number B_n is $B_n = \sum_{k=0}^{n-1}\binom{n-1}{k}B_k$. B_n for large *n* grows rapidly. Now assuming that we have $(1 + \lfloor\log n_1\rfloor)$ sub-trees for C_1 and $(1 + \lfloor\log n_2\rfloor)$ sub-trees for C_2, etc, we can calculate X_i the total number of CAT configurations for C_i as follows: $X_i = \sum_{k=0}^{\lfloor\log n_i\rfloor}\binom{\lfloor\log n_i\rfloor}{k}B_k$. Thus the size of the design space would be equal to $X_1 \times X_2 \times \ldots \times X_k$. As mentioned earlier we can design the optimal MAT for a configuration of CATs in *O(m)*, where m is the number of CATs. In our problem, $m \leq (1 + \lfloor\log n_1\rfloor) + \cdots + (1 + \lfloor\log n_k\rfloor)$. Thus in the worst case it takes $O(\log[n_1 \times n_2 \times \ldots \times n_k])^2$ to find the optimal solution for each configuration. This makes the total run-time of the search algorithm to be $O(X_1 \times X_2 \times \ldots \times X_k \times \log[n_1 \times n_2 \times \ldots \times n_k])$. The size of the search space in this case is huge but we can efficiently reduce it.

Theorem 4 states an important characteristic of optimal CAT configurations which reduces the design space.

Theorem 4: In an optimal CAT configuration all the possible sub-trees, smaller than the largest sub-tree and larger than the smallest sub-tree of the CAT will be part of the CAT.

Proof: Based on Theorem 4, for the example shown in Figure 5.a, the possible CAT configurations in the optimal filter design are {{2}, {4}, {16}}, {{2, 4}, {16}}, {{2}, {4, 16}}, {2, 4, 16}, while the total number of partitioning also includes the partitioning {{4}, {2, 16}}. This theorem states that the latter partitioning cannot be a valid CAT configuration in the optimal implementation.

To prove the theorem we assume that we have a CAT configuration as part of the optimal implementation where some of the sub-trees larger than the smallest sub-tree in

[2] $\log n_1 + \log n_2 + \cdots + \log n_k = \log[n_1 \times n_2 \times \ldots \times n_k]$

the configuration and smaller than the largest sub-tree are missing i.e they are part of another CAT. We choose the *largest missing sub-tree (LMST)*. Consider all the *smaller sub-trees (SST)* present in the CAT configuration. We swap the LMST and SST in their corresponding CATs configurations. Replacing the SST with LMST would not change the output size of the CAT, because it is dominated by the largest sub-tree in that CAT configuration. However replacing the LMST with the SST might decrease the output size of the other CAT. Reduction in the output size of the CATs will lead to reduction in the cost of the MAT. Therefore overall cost of the CAT configurations after the swap would be the same before the swap or would be less. Thus the CATs configuration before the swap cannot be part of the optimal design. This proves Theorem 4.

As mentioned earlier the result of Theorem 4 is very important as it reduces the number of possible partitions that need to be examined to find the optimal result. To count the actual number of possible partitions we need to find all different partitioning sets of size 1, 2, … i.e we partition the set of sub-trees into 1 set, 2 sets, etc. This problem is very similar to the problem of finding all the possible solutions to the following equation: $Y_1 + Y_2 + \cdots + Y_r = n$, where $Y_1, Y_2, \ldots, Y_r \in \mathbb{N}$ and $Y_1, Y_2, \ldots, Y_r \geq 1$

The number of possible answers for this problem is $\binom{n-1}{r-1}$.

In the context of our problem *n* represents the total number of sub-trees for a coefficient and *r* is the number of CATs we are generating. If for example $Y_1 = 3$, then it means the three largest sub-trees for a coefficient are clustered together in one CAT. In this problem we need to find all possible solutions for $r = 1, 2, \ldots, n$. Thus the total number of possible partitions would be: $\binom{n-1}{0} + \binom{n-1}{1} + \cdots + \binom{n-1}{n-1} = 2^{n-1}$

For each coefficient C_i we have at most $(1 + \lfloor \log n_i \rfloor)$ sub-trees. Thus all number of possible CAT configurations for coefficient C_i is $X_i = 2^{\lfloor \log n_i \rfloor} \cong n_i$. Therefore based on the timing analysis mentioned earlier the worst case time complexity for the search algorithm would be $O(n_1 \times n_2 \times ... \times n_k \times \log[n_1 \times n_2 \times ... \times n_k])$. Since $n_1 + n_2 + \cdots + n_k = n$ and $k << n$, the time complexity of the algorithm becomes polynomial.

3.3.1.1 *Intelligent Search Strategy*

To intelligently search the design space and find the optimal MAT in linear time for the number of inputs we need to maintain the quality of the input list as being sorted or in a position that the list can be sorted in linear time. If we make just one change or constant number of changes to the configuration of one or constant number of CATs we can resort the input list in linear time from the already sorted list for the new inputs or the changed inputs to the MAT. We focus on searching the design space where we make just one change at a time and sort the list and build the optimal MAT for the sorted list. We use an implementation similar to the idea of Gray codes to represent binary numbers. The advantage of Grady code is that every two consecutive numbers are different by just one bit. Figure 8 shows an example for traversing the design space of partitioning the sub-trees for a coefficient. As shown in the example the coefficient has 4 sub-trees which are represented by numbers 1, 2, 3 and 4 which also implies the order of the sub-trees sizes. In this example, the sequence 1/2/3/4 means that the 4 sub-trees are partitioned into 4 different CATs while for example the sequence 1 2 3/4 means that sub-trees 1, 2 and 3 are partitioned into 1 CAT while sub-tree 4 is partitioned into a different CAT.

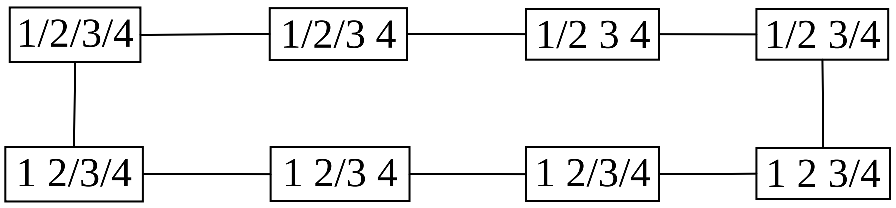


Figure 8. Searching the design space for CAT partitioning

3.4 Reconfigurable Section Area Minimization (RSAM) Problem

Up to this point we showed how to optimally generate a fully optimized filter (a special case of RSAM problem) for a given set of coefficients. However we have not yet proposed a solution to Reconfigurable Section Area Minimization (RSAM) problem which is to minimize the area of the reconfigurable section of the filter for a given level *i* of implementation and a set of coefficients. By specifying a level of implementation we determine the number of inputs to the static section of the filter as well as the bit-width of the inputs. In [1] we studied this problem as a clustering problem. In the solution discussed in [1] we proposed to cluster the inputs to generate as many clusters as we have inputs to the static section of the filter. As well the output bit-width of the cluster cannot exceed the size of the input to the static section. In this work we follow the same idea but will discuss it in the context of the theoretical analysis discussed earlier in this paper. The solution is optimal and can be obtained in polynomial time.

3.4.1 Proposed Solution

We assume that there are k different coefficients $\{C_1, C_2, \dots, C_k\}$ in the set of coefficients of the filter. n_1 of the inputs are multiplied with C_1, n_2 of the inputs are multiplied with C_2, and so on. We further assume that the number of inputs to the static section of the filter cut at level i (level i implementation), is S_i and the width of the inputs is W_i. This means that the static section of the filter includes $S_i - 1$ adders structured as a binary tree.

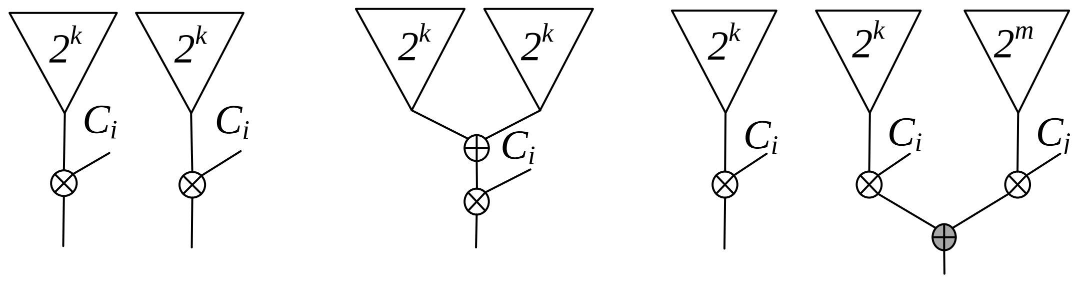
We define the **Multiplier-Adder Comparative (MAC)** cost of a coefficient to be the difference in hardware cost of merging two equal-sized sub-trees of the coefficient versus implementing them independently. In Figure 9 the MAC cost of the coefficient is the cost of the structure shown in Figure 9.b minus the hardware cost of the structure shown in Figure 9.a.

Theorem 5: If merging two arbitrary equal-sized perfect binary sub-trees of a coefficient C_i reduces the cost of implementation i.e $MAC_{C_i} \leq 0$, then the largest sub-tree which meets the bit-width constraint is part of the minimal sized reconfigurable section of the filter.

Proof: If the two sub-trees in Figure 9.a are direct inputs to the static section, then unless merging them would violate the bit-width constraints, it reduces the cost to merge them first and then give them as an input to the static section (see Figure 9.b). If one of the sub-trees is an input to the static section and the other one is first added to the output of another sub-tree (see Figure 9.c) then the additional adder (highlighted in Figure 9.c) is larger than or equal to the adder we would insert to add the two sub-trees of the same coefficient (adder inserted in Figure 9.b).

Since it is assumed that the size of the adders and multipliers is linearly proportional to the size of the input, Theorem 5 is true for all input sizes.

The solution to RSAM is very similar to the solution we proposed earlier to find the optimal implementation of a filter. Based on Theorems 2 and 3, the constituents of CATs in the optimal implementation of a filter are perfect binary sub-trees. The solution to RSAM also includes clustering among complete graphs with a power of 2, number of nodes. Figure 10 shows the graph representation for a sample RSAM problem.



(a) **(b)** **(c)**

$$MAC_{C_i} = Cost(b) - Cost(a)$$

Figure 9. Merging sub-trees of the same size

Each *node N* represents inputs that are multiplied to a coefficient. As can be observed in Figure 10.a, each node has two fields. The coefficient and also the sub-tree size (number of inputs) for that coefficient. Initially each input is represented using one node, thus the sub-tree size for the node is equal to 1. A collection of nodes constructs *hyper-nodes (HN)* as shown in Figure 10.a. Each hyper-node includes a list of coefficients as well as the sub-tree size associated with each coefficient. Initially each hyper-node holds complete graphs of nodes. All the nodes in the hypernode include the same coefficient. The size of the complete graphs is derived from binary representation of the number of inputs for each coefficient (Theorems 2 and 3). Figure 10.b shows 17 nodes and two hyper-nodes (one of size 1 and the other of size 16) for coefficient *C = -3* from the example of Figure 4.

The nodes can only be merged with other nodes inside the same hyper-node. We call a node **Incomplete (IC)** if it can still be merged with other nodes inside the same hyper-node. Accordingly we call a node, **Complete-Unassigned (CU)** if the node is complete (it cannot be merged further) but the output bit-width is less than the constraint W_i. IC and CU nodes are shown in Figure 10.b. Hyper-nodes can be merged together only if the nodes they cover are CU nodes. When merging the hyper-nodes, for the sub-trees that are multiplied to the same coefficient, we first add the result of the sub-trees and then multiply them to the corresponding coefficients.

At stage 0 of our algorithm, we determine the coefficients C_i for which $MAC_{C_i} \leq 0$. As mentioned in Theorem 5, for these coefficients we merge the nodes inside the hyper-nodes unless the output bit-width of the nodes violates the constraint W_i when we assign the output of the IC nodes to the inputs of the static section. For the rest of the coefficients ($MAC_{C_i} > 0$) we need to find the lowest cost merging of the nodes. The options include merging the IC nodes or merging the CU nodes. To optimally merge the hyper-nodes we need to apply the optimal MAT generation algorithm. As well we use the search algorithm we proposed earlier to determine the optimal hyper-node merging strategy (to determine the optimal CAT configuration for each coefficient). We continue this until we generate as many as S_i outputs from the reconfigurable section.

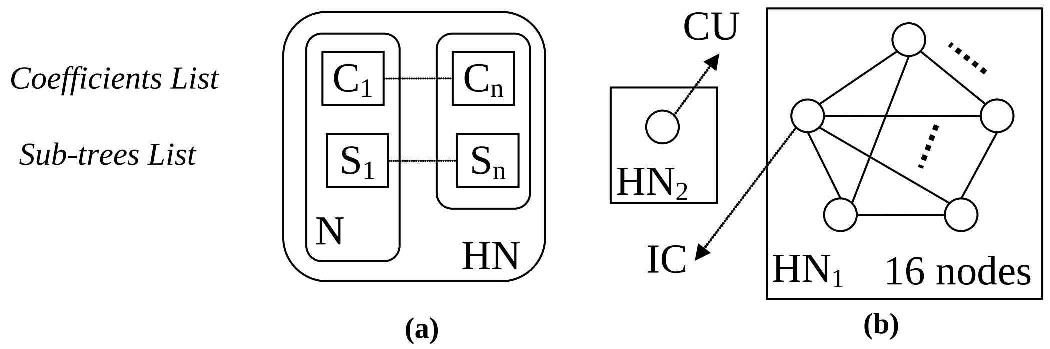
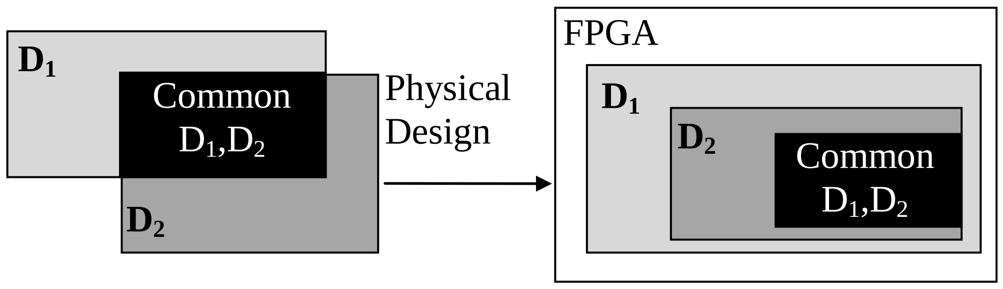


Figure 10. Graph representation of the filter to solve RSAM problem

Using the same analysis for full optimization of the filters, it can be shown that the solution to RSAM is optimal with polynomial worst case run-time.

4 Joint Optimization Across Designs

Reconfiguration time overhead for reconfiguring one design for another, can be accurately measured after the floor-planning and placement when the exact area of the circuit on the FPGA is determined. This overhead is directly proportional to the size of the area being reconfigured ([6] and [22]). Thus to reduce the overhead it is important to reduce the size of the circuit that is reconfigured. However in the context of filter design problem discussed in this paper, this does not necessarily mean that we need to minimize the size of the reconfigurable section of the individual designs to reduce the overhead.



$$\textit{Reconfiguration cost} = \max(D_1, D_2) - \textit{Common } D_1, D_2$$

Figure 11. Reconfiguration scheme of two designs

In the previous works including [24] and [26] *module reuse* has been mentioned as an effective way to reduce the reconfiguration overhead. As shown in Figure 11 designs D_1 and D_2 have a common module. If the common module of the two designs is placed at the same location then, as can be observed in Figure 11, the reconfiguration overhead can be effectively reduced. The reduction in overhead is proportional to the area of the common module.

This prompts that in the problem of filter design, for a sequence of filters, we need to find similarity in filters structures and exploit that similarity to reduce the reconfiguration overhead while reconfiguring the filters. Our approach in this work is to design the individual filters to become similar in structure instead of explicitly extract the similarity in structure.

While this approach provides better opportunities to find similarity in filters structures, it may result in larger implementations. The two structures shown in Figure 12.a and Figure 12.c can be represented in functionality by the structure shown in Figure 12.b. However, as can be observed this structure is larger than both structures.

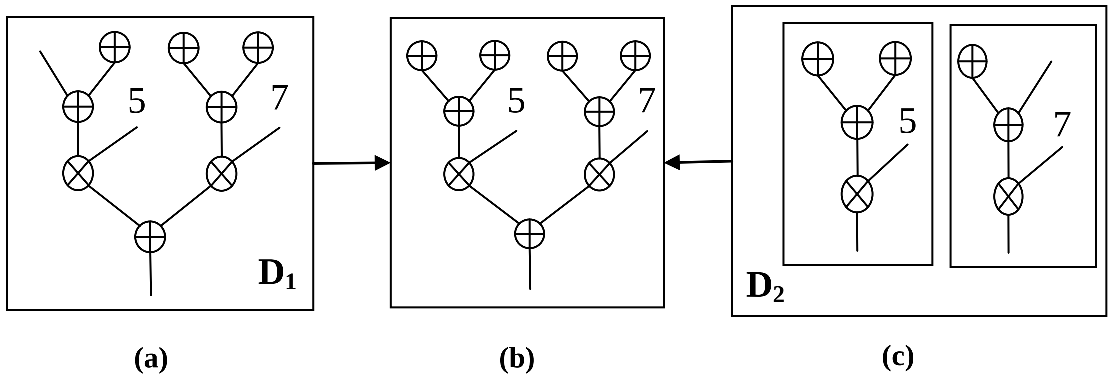


Figure 12. Common structure which enforces similarity

To have control over the abovementioned tradeoff between the size of the individual designs and the total reconfiguration overhead, we propose minimizing a linear cost function of area and total reconfiguration overhead for a sequence of filters.

We define the problem of **Joint Filter Design (JFD)** as follows:

A known sequence of designs $D_1, \ldots, D_n$ is given. For a given level of implementation, a structure should be designed for each filter to minimize a linear cost function of total reconfiguration time and the area of the design.

4.1 Partial Reconfiguration Scheme

In this work we follow Module-based partial reconfiguration scheme introduced in [5] and [6]. In this scheme the modules of the design are categorized as *static* modules and *reconfigurable* modules. Our description of static and reconfigurable sections of a filter fits this definition. Static modules are not reconfigured during the run-time of the system while reconfigurable modules are reconfigured in run-time. The communication between the static modules and reconfigurable modules is possible through specialized connections called “bus macros”. Our approach to solve the JFD problem enhances the approach introduced in [1] and proposes a solution in the context of the theoretical analysis explained earlier in this paper. Our proposed solution can be summarized in the following three steps:

1. Determine the occurrence frequency of each coefficient across the designs. For each coefficient form an *n*-tuple.
2. Partition each *n*-tuple, and for each partition find the optimum number of inputs that maximizes a profit function which is explained later in this section.
3. Jointly design the filters for the given inputs.

The idea is to find the similar filters and for those filters, to jointly design the structure. We need to do that for every coefficient. Figure 13 can make this clearer. The *n*-tuple shown in Figure 13 represents the occurrence frequency for a specific coefficient (an arbitrary coefficient C_1 in this example) for the 10 filters we need to jointly design. The order of the elements in the *n*-tuple, represents the given sequence for reconfiguring the filters i.e we first configure F_1 and then F_2 and so on. The value of the elements of the *n*-tuple represent the number of occurrences of that coefficient in the coefficient set of the corresponding design, as an example in the coefficient set of filter F_1 there are 13 occurrences of coefficient C_1.

By partitioning the *n*-tuple associated with each coefficient in Step 2, we are grouping the filter designs that have similar occurrence frequency of that coefficient. The filters in each partition have a common sub-structure for that coefficient i.e F_1, F_2 and F_3 that are partitioned together, will have a common substructure for coefficient C_1.

The next part of Step 2, is to find the optimum size of this common substructure for the filters that are grouped together. This optimum size is determined by a profit function which should be maximized. This profit function as will be explained later in this section is a function of area and reconfiguration overhead. In the example of Figure 13, for filters F_1, F_2 and F_3 the size of this common substructure for coefficient C_1 is determined to be 16.

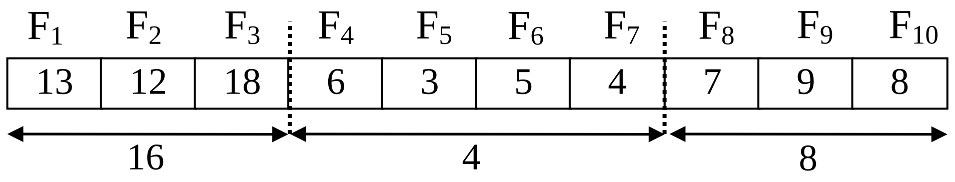


Figure 13. Partitioning the designs to similar sub-sequences

After determining the size of the common substructures which are shared among a subset of filters, we need to jointly design the filters. As mentioned earlier, the objective is to minimize a linear cost function of area and reconfiguration overhead. The three important steps in joint design of the filters are partitioning, determining the optimal common cluster size and designing the individual filters. In the following we will explain these three steps in more detail.

4.2 *Optimal Common Cluster Size*

Definition: For a given sequence of designs $F_1, F_2, \ldots, F_n$ find the **Optimal Common Cluster Size (OCCS)** that minimizes a linear function of the area of the individual designs and the total reconfiguration time for the designs.

We define **Reconfiguration Size Saving (RSS)** of a tree as the size of the common tree structure between the two consecutive designs. While reconfiguring the filters we avoid reconfiguring this common structure. The reconfiguration time that we save is proportional to the size of the structure we are avoiding to reconfigure. The **Overhead** of a tree for a design is defined as the difference in size of that tree and the size of the trees in the original implementation. In Figure 14 the number of inputs to the coefficient for the original filter is 13. The size of the chosen sub-tree is 16. The RSS is the part of the structure that is drawn in solid lines. On other hand the part of the structure in dotted lines shows the Overhead we need to pay to implement the adder tree to add 13 inputs by the chosen sub-tree. The additional inputs do not affect the functionality of the filter as the values to these inputs are all set to 0.

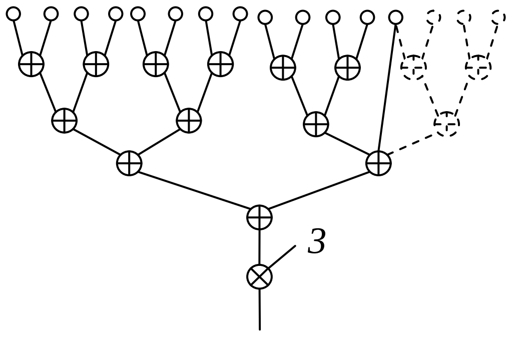


Figure 14. Overhead and reconfiguration saving of a sub-tree

The value for OCCS should maximize the following profit function:

$$Profit = RF \times RSS - (1 - RF) \times Overhead$$

RF (Reconfiguration Factor) is the coefficient that determines the importance of reconfiguration time versus the size of the individual filters. By appropriately tuning *RF* we can generate a spectrum of designs from highly optimized individual implementations (when *RF* = 0) to designs with negligible reconfiguration overhead (when *RF* = 1).

To determine the OCCS we need to search through different tree sizes. Assume $OPT_{CCS}(\mathcal{A}_1^n)$ is the optimal profit value we gain from finding the *OCCS*. The array $\mathcal{A}_1^n$, represents the elements of the *n*-tuples from Step 1. The following equation shows how $OPT_{CCS}(\mathcal{A}_1^n)$ is calculated.

$$OPT_{CCS}(\mathcal{A}_1^n) = Max_{K \leq 2^{\lceil \log(\max(A[i])) \rceil}}^{K \geq 2^{\lfloor \log(\min(A[i])) \rfloor}} (Profit(K) + OPT_{CCS}(\mathcal{A}_1^n - K))$$

For each possible value of *K* (*K* is a power of 2) we find the profit (based on the equation for profit function) of having a cluster of size *K* as part of the *OCCS*. Since we might have residuals left in the *n*-tuple, we need to repeat the process for the residual considering that value *K* is part of the tree of the common cluster ($OPT_{CCS}(\mathcal{A}_1^n - K)$ calculates the residual). The *OCCS* can be obtained by adding all the *K*s that result in the optimum value for the corresponding *n*-tuple. As well we have: $OPT_{CCS}(\mathcal{A}_1^n)_{\forall A[i] \leq 1} = 0$.

4.2.1 Time-complexity Analysis

Assume that the occurrence frequency list given as the input to the *OCCS* algorithm has *m* elements and that the value of the largest element is *r*. The time complexity to find *r* is *O(m)*. At each level of recursion we need to compare *O(log r)* (lower and upper boundary of *Max*) solutions. By applying branch and bound techniques to avoid suboptimal solutions we can show that by at most *O(log r)* recursions we can find the optimal solution. Thus in the worst case the algorithm finds the optimal solution in

$O(\log^2 r)$ steps. At each step we need to find the profit of having a common cluster of size K (calculate *Profit (K)*). The main part in calculating the profit function is to calculate the cost of the adder tree to determine the reconfiguration saving and also to determine the overhead for individual designs. The cost of an adder tree to add h inputs of equal size together, can be calculated in $O(\log h)$. In the worst case the value of h is equal to r. Thus the time complexity of the algorithm is $O(m + m(\log^3 r))$. If we have n filter masks (the maximum value of m) each of which has at most k inputs (the maximum value of r) the worst case time complexity would be $O(n(\log^3 k))$.

4.3 Partitioning

In the previous subsection we showed how to find the OCCS for a given n-tuple. Now we need to use that information to optimally partition the designs sequence into subsequences. The designs that are grouped together during partitioning, share a common cluster determined by the OCCS algorithm explained earlier. This common cluster is not reconfigured while reconfiguring the filter designs in the same partition.

We propose a solution which uses dynamic programming to solve the problem of partitioning. $OPT_{SP}(0,n)$ is defined as the optimal partitioning of the first n elements. By optimal it means that this partitioning maximizes overall profit function that was defined in Section 4.2. $OPT_{CCS}(\mathcal{A}_{k+1}^{n})$ is the profit of the Optimal CCS (OCCS) for the subsequence $F_{k+1}, \ldots, F_n$. We can define a recursive relation for $OPT_{SP}(0,n)$ as follows:

$$OPT_{SP}(0,n) = Max_{0\le k}^{k\le n-1}(OPT_{CCS}(\mathcal{A}_{k+1}^{n}) + OPT_{SP}(0,k)) \; OPT_{SP}(0, 0) = 0.$$

The above equation assumes that in the optimal partitioning of the first n elements, the last $n - k$ elements form a partition. Then we need to find the optimal partitioning of the first k elements. We need to find k at each level of recursion. From all possible values of k ($0 \leq k \leq n - 1$), the one that maximizes the profit of the partitioning is chosen. The corresponding values for k determine the subsequences for partitioning.

4.3.1 Time-complexity Analysis

To find the optimal partitioning, we need to consider calculating the OCCS for all subsequences for a sequence of n elements. There are $O(n^2)$ of these subsequences (n choices for the start index of the subsequence and n for the end index). Because of the recursive nature of the proposed dynamic programming approach a straightforward implementation would calculate the value of OCCS for the same subsequence over and over again. To avoid that, we use *memoization* [4] to calculate and store the value of OCCS for each subsequence just once. With this approach, there $O(n^2)$ times that we need to call our OPT_{CCS} subroutine. In an instance of JFD problem where we have a sequence of n filters with at most k inputs, the worst case time complexity for our partitioning algorithm is $O(n^3(\log^3 k))$.

4.4 Joint Filter Design

For a given level of implementation i, there are S_i inputs from the reconfigurable section of each filter F_j to the static section. The size of the inputs to the static section is W_i which is regarded as the constraint on the size of the outputs of the reconfigurable section. After partitioning the sequence of the filter designs for each coefficient and determining the OCCS for each partition, we need to design the filter structures.

In the graph representation of JFD problem, the nodes in addition to information regarding the associated coefficient and the tree size (shown in Figure 10.a) also include the list of the designs that share the node. Accordingly hyper-nodes also include the list of the designs sharing the hyper-node (shown in Figure 15). Initially each node represents one input. The list of the designs sharing the node is derived after applying partitioning algorithm, explained earlier. As well hyper-nodes initially contain complete graphs of the nodes that are shared among the same list of designs.

Similar to the solution to RSAM and based on Theorem 5, for the coefficients C_i where $MAC_{Ci} \leq 0$, we merge the nodes to generate a CU node, unless they violate the bit-width constraint W_i, when we assign the output of these IC nodes to the static section. For other coefficients we need to determine whether it is beneficial to merge the IC nodes or to merge the hyper-nodes covering CU nodes. The main difference between the proposed algorithms for JFD problem and RSAM problem is in how we merge the hyper-nodes. In the proposed algorithm for RSAM, to merge the hyper-nodes we need to apply optimal MAT generation algorithm. However in JFD problem the challenge in merging the hyper-nodes is that the hyper-nodes do not belong to the same design anymore.

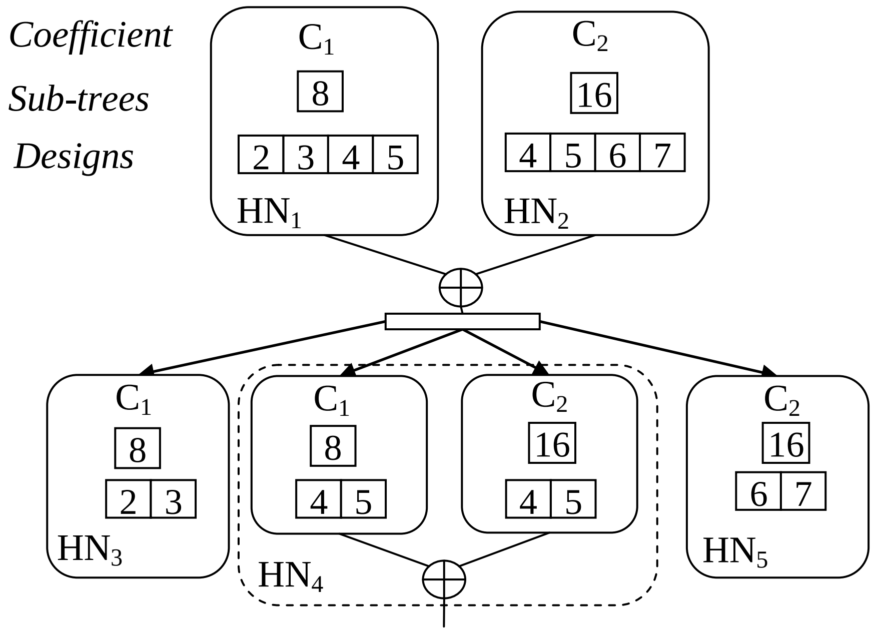


Figure 15. Adding two clusters with unequal design lists

Figure 15 shows an example where two hyper-nodes HN_1 and HN_2 are added (merged) together. Since the two hyper-nodes are shared among different design lists, the result of merging the two hyper-nodes can be used just for the intersection of the designs sharing the two hyper-nodes which is designs F_4 and F_5. Merging the two hyper-nodes, results in an overhead to reconfigure the sub-tree structure for coefficient C_1 from F_3 to F_4 and to reconfigure the sub-tree structure for coefficient C_2 from F_5 to F_6. In designing the MAT this problem has not been considered. Thus a new approach should be taken to design the MAT for individual filters to minimize the cost function which includes both design size and also reconfiguration cost. We call this new approach **Joint Multiplier Adder Tree** (**JMAT**) design.

4.4.1 Joint Multiplier Adder Tree (JMAT) design

We assume that for *k* different filter designs, there are a total of *n* inputs (outputs of CU nodes) that should be added together in a way that for filter F_1, there are n_1 inputs and m_1 outputs, for filter F_2, there are n_2 inputs and m_2 outputs and etc. Some of the inputs are shared among more than one design furthermore the bit-widths of the inputs are different. We need to design the minimum cost adder trees for each filter to add the given list of inputs and generate the outputs.

To design the JMAT we propose a solution based on *k*-clustering. There are initially *n* hyper-nodes that represent the *n* inputs to the adders. Each hyper-node holds information about the list of the designs that share the input. Only those hyper-nodes that share at least one design in their corresponding design list can be merged together. The weight of the edges connecting the nodes is the cost of adding the output of the two hyper-nodes (merging the hyper-nodes). The cost of merging two hyper-nodes is defined as follows:

$$\alpha \times Cost_{Adder} + \beta \times Cost_{Reconf}$$

$Cost_{Adder}$ is defined as the size of the adder being inserted to add the structure of the two hyper-nodes. $Cost_{Reconf}$ is the added/reduced reconfiguration cost between every two consecutive designs in the designs list of the two hyper-nodes. We study two different cases to determine the weight of the edge connecting the two hyper-nodes:

1. The designs lists for the two hyper-nodes are not equal but have an intersection: This is the case shown in Figure 15. In the example shown in Figure 15 the cost of inserting the adder for designs F_2, F_3, F_6 and F_7 is 0. However the area cost of the adder inserted in HN_4 increases the area of designs F_4 and F_5. This additional adder also counts as a saving in reconfiguration cost from F_4 to F_5. As explained earlier merging HN_1 and HN_2 incurs reconfiguration cost from F_3 to F_4 and from F_5 to F_6. As mentioned earlier in Section 4, the reconfiguration cost from F_3 to F_4 and from F_5 to F_6 is assumed to be $max(HN_3, HN_4) + max(HN_4, HN_5) = 2 \times Cost_{HN_4}$. Thus for the example shown in Figure 15 the cost of the edge connecting HN_1 and HN_2 is defined as follows:

$$(\alpha \times 2 - \beta) \times Cost_{Adder} + \beta \times 2 \times Cost_{HN_4}$$

2. The designs lists for the two hyper-nodes are equal: In this case the two hyper-nodes can be added together without incurring reconfiguration cost. Because of adding an additional adder the size of each design in the designs list increases by the area of the adder. Since the merged node increases the similarity across the

consecutive designs we assume the cost of the additional adder as reconfiguration saving among the designs. Assuming that there are k designs in the design list, the cost would be calculated as follows:

$$\alpha \times k \times Cost_{Adder} - \beta \times (k-1) \times Cost_{Adder}$$

Once the weight of all the edges is determined we cluster the hyper-nodes with lowest cost together until we get the required number of outputs for each design. For the edges that have the same weight, priority is given to the two hyper-nodes that have the same output sizes, otherwise a pair of hyper-nodes is randomly selected.

The proposed algorithm for JMAT is the general case of optimally building MAT. We get the same result of the proposed algorithms for MAT by running the proposed algorithm for JMAT if we set $\beta = 0$ and number of designs be equal to exactly 1.

5 Experimental Results

Correlation operation is a common function across a range of streaming applications. Throughout our experiments we target three important applications which make extensive use of FIR filters. These applications include "Preamble Detection" unit of 802.16e standard, correlation used for "Template Matching" in image processing and "Match-filter" used in spectrum sensing. We briefly explain these applications and present the result of our filter design techniques for these applications.

5.1 Preamble Detection

At the receiver end of a wireless system, correlation is often used to detect a certain pattern disguised in the received signal. In CDMA systems for example, to recover the information belonging to each user from the received signal, the signal is correlated against the Pseudo Noise (PN) sequence used to encode the corresponding users data. In 802.16e correlation operation is performed for timing synchronization. A preamble sequence, known to the mobile station, is sent from the base station and by correlating the received signal against the known sequence and comparing the result to a predetermined threshold timing acquisition is declared. This is done by simply filtering the received signal through an FIR filter with impulse response of *conj(p(N-n))* where *p(n)* is the preamble sequence and *N* is its length. Figure 16 shows the structure of a preamble detection unit [8]. As part of experiments to quantify the performance of our proposed algorithms, 6 sample preambles from 802.16e are arbitrarily chosen. Inputs and coefficients are assumed to be 4 bits wide.

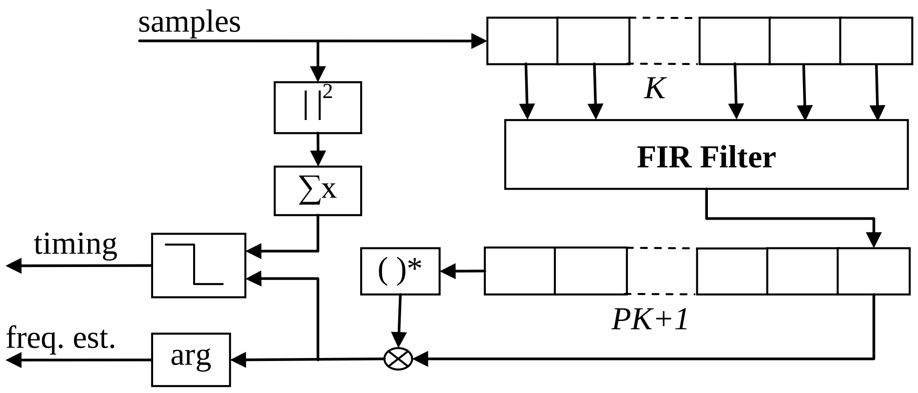


Figure 16. Architecture of preamble detection unit

5.2 Template Matching

Template matching is used to detect objects in an image of known shape, size and orientation. A straightforward and effective method to find a template is to convolve a filter mask whose coefficients constitute a sub-image that matches the desired object across the image. When this mask is located at the image object, the convolution result is an intensity peak. Therefore the presence and the location of the desired objects are determined by peak detection. In [7] the authors propose an FPGA based platform to implement an Automatic Target Recognition (ATR) device. The challenge of ATR is to analyze input images or video sequences in order to automatically locate and identify all objects within the scene of interest. The input image should be correlated with a large number of templates to find the objects of interest. FPGAs capable of dynamic partial reconfiguration are particularly well suited for this kind of application because of high-level parallelism that they deliver and because of the reconfiguration ability which makes it possible to implement successive highly specific mask filters. In our experiments on template matching, we implement 10 filter masks representing 20 x 20 pixel templates. The templates are 8-bit grayscale, meaning that each pixel is represented by 8 bits. The filter masks are applied one at a time to the input image.

As part of our experimental setup we have developed a tool which accepts a sequence of coefficient sets representing each of the reconfigurable filters. The filters are optimized according to the optimization techniques explained in this work and VHDL code for the hyper-nodes of the filter is generated. We used Xilinx ISE design suite to synthesize and place and route the RTL.

5.3 Single Filter Design

Figure 17 shows the result of applying clustering (originally proposed in [1]) to solve RSAM problem for template matching. On the X axis in the figure, $LI_0 \ldots LI_4$ represent the levels of implementation for the filter. NS abbreviates “No Static”, meaning that the whole filter structure is being optimized (no static section). Y axis shows the size of the filter (in multiple of 1000 slices). The curve marked as “Reconfigurable” is the size of the reconfigurable section of the filter while "Total" shows the total size of the filter. As can be observed, by increasing the level of implementation the total size of filter is monotonically decreasing. This reflects the fact that a larger section of filter is being optimized.

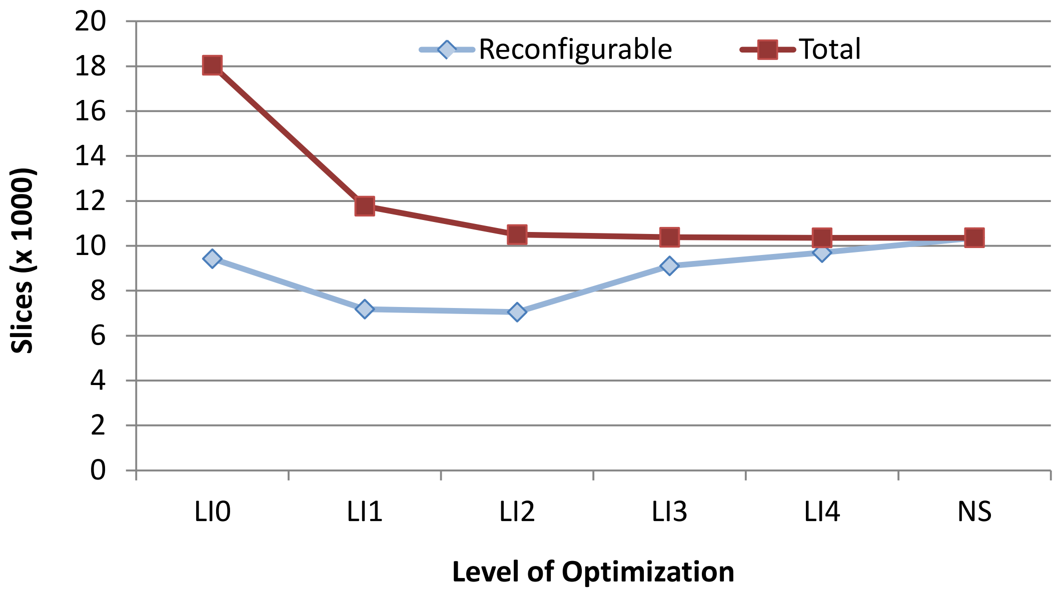


Figure 17. Template filter mask optimization for different levels of implementation

We expect the size of reconfigurable section to monotonically increase by increasing LI, however the trend shown in Figure 17 is different. As can be observed the size of reconfigurable section is a convex curve, decreasing first and then starts increasing. The reason is that increasing the level of implementation provides opportunity to merge more inputs and further decrease the number of multipliers. Figure 18 shows the reduction in filter size achieved after applying the proposed optimal solution in this paper. The optimal solution reduces the size of reconfigurable section by 17.2% and in average 5.5%. In our experiments the result of applying Theorem 5 and applying optimal MAT generation technique had the largest impact in size reduction compared to the clustering technique previously proposed.

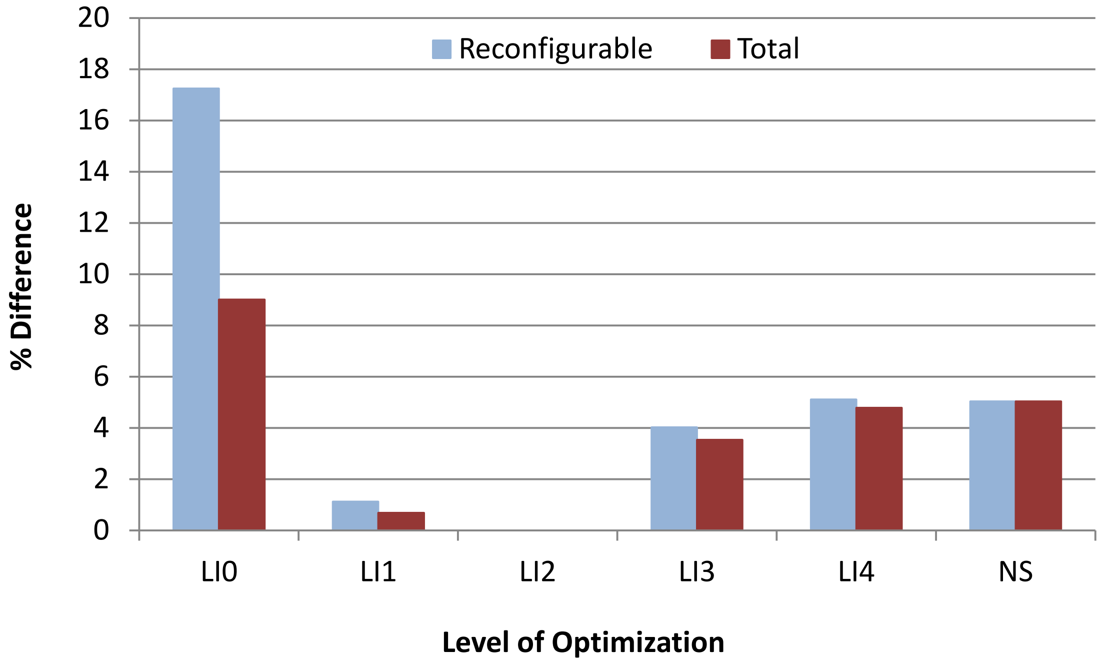


Figure 18. Optimal template filter mask design

5.4 Joint Filter Design

Figure 19 shows the combined effect of tuning LI and RF for a sequence of 10 filter masks designed using JFD for template matching. X axis in Figure 19 shows the average area of filter masks while Y axis shows the average size that is reconfigured across filter masks. Each point in Figure 19 represents one multi-mode filter designed to support 10 modes of operation (corresponding to the number of image templates). The scope of Figure 19 is twofold. First, it aims at exploring the impact of Reconfiguration Factor (RF) on the area and the reconfiguration overhead. This is illustrated with the points marked as *"RF Exp"* (RF Exploration) varied from 0 to 1, for a given Level of Implementation (LI_3). Secondly, it aims at exploring the impact of Level of Implementation (LI) on the area and the reconfiguration overhead. This is illustrated with the points marked as *"LI Exp"* (LI Exploration) varied from LI_0 to LI_5, for a given Reconfiguration Factor (RF = 0.6).

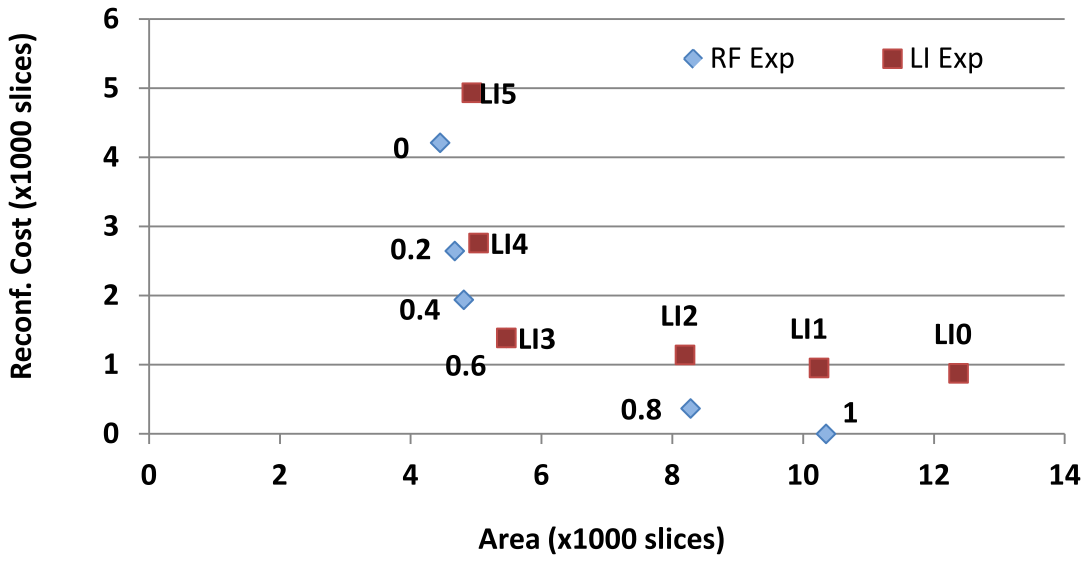


Figure 19. Trade-off of Area and Reconfiguration cost while exploring RF and LI for template matching application

In *"RF Exp"* experiment as is clear from Figure 19, by increasing the value of RF from 0 to 1, the average area of the filters are increasing while the reconfiguration cost is decreasing until it reaches 0 for RF = 1. This is elaborated further, while showing the results of the next set of experiments. As for *"LI Exp"* experiment, increasing the LI from 0 to 5 (LI_5 results in a design without static section) results in a considerable decrease of the size of circuit. This is because the circuit becomes more optimized based on the coefficient set of the filter. At the same time, the average reconfiguration cost is increasing since the filters become coefficient specific and less similar.

Given a sequence of filters that have to be reconfigured one after another one we design each filter using the Joint Optimization approach we proposed in Section 4. As mentioned in Section 4.2 we define a profit function to find the value for OCCS for a given *n*-tuple. Figure 20 shows the effect of increasing the RF (Reconfiguration Factor) for each level of implementation.

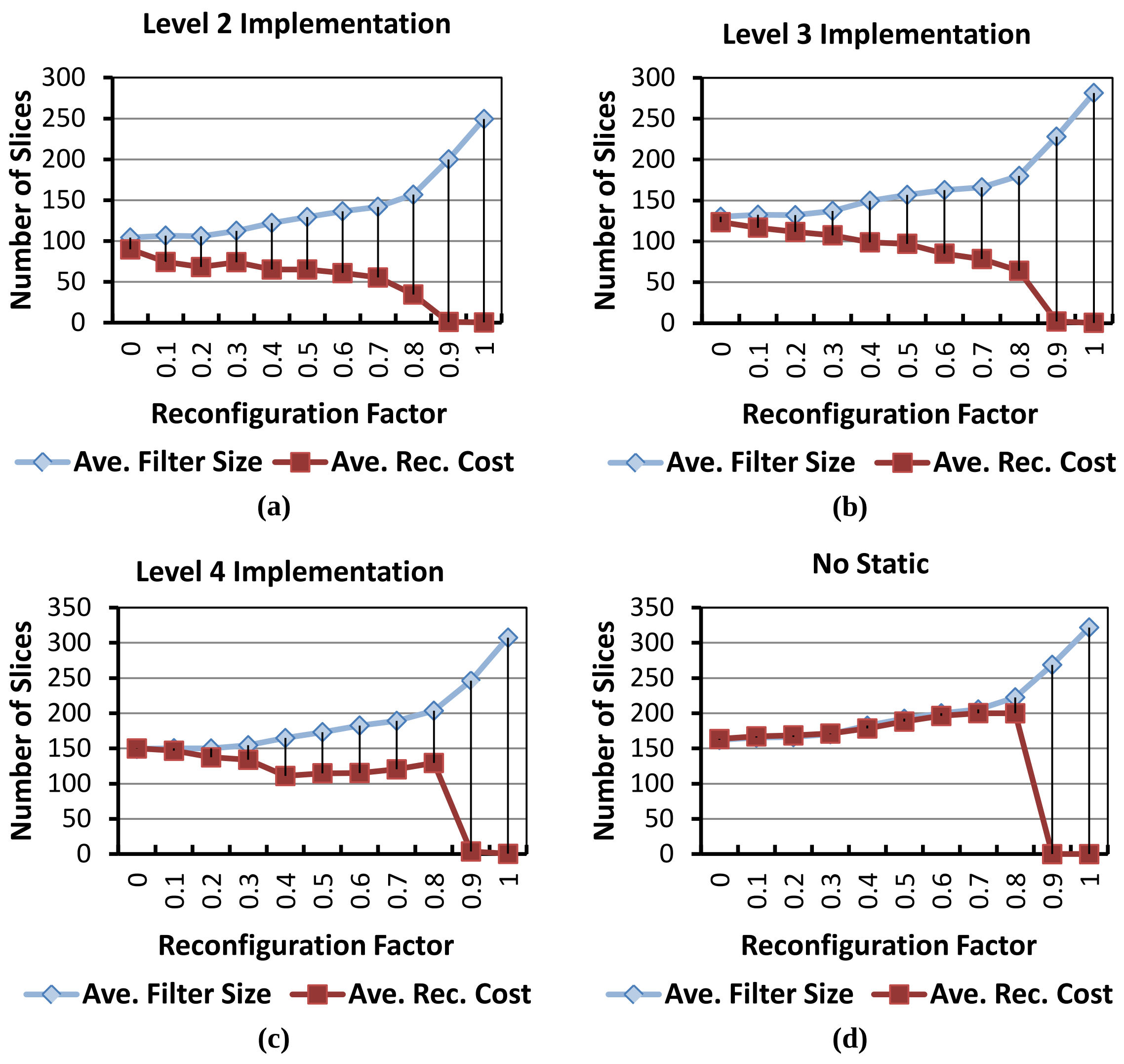


Figure 20. Average design size increases by increasing the Reconfiguration Factor while average reconfiguration cost decreases

In the figures the X axis represents the RF while the Y axis represents the average size of the reconfigurable sections of the designs in number of slices. The RF is varied between 0.0 to 1.0 in the scale of 0.1 at every step. As can be observed in the figure, for each level of implementation, increasing the RF results in designs that are larger in size.

However the benefit is in average reconfiguration cost. As we are increasing the RF the individual designs are becoming more similar in structure which requires no reconfiguration for some/most of the clusters.

For RF = 0.9 and RF = 1.0, the average reconfiguration size overhead is negligible prompting that all the individual filters are the same. In the case of generic design, we also pay negligible reconfiguration overhead to change the filter structure. However compared to 662 slices to implement a generic filter, we can implement the filter using JFD with RF = 1 which requires 320 slices. This is more than 50% improvement for the same quality of design.

By increasing the level of implementation we can observe from Figure 20.[a, b, c and d] that the gap between the average reconfiguration cost and the average size of the filters has tendency to get close. This is clear from the rate of reconfiguration cost reduction in each figure. While in Figure 20.a (LI_2) the reconfiguration cost is almost monotonically decreasing by increasing the reconfiguration cost, the rate is obviously not monotonic in Figure 20.c (LI_4). In Figure 20.d (No Static) interestingly the average reconfiguration cost follows the exact same pattern of the size of the individual designs until RF = 0.9 where the average reconfiguration cost drops abruptly. The reason for this behavior is as follows: Increasing the level of reconfiguration results in generating less number of clusters, thus there is less chance of finding similar clusters across the designs. So increasing the level of implementation forces the individual designs to be less similar to each other. On the other hand increasing the RF has the opposite effect meaning that increasing the RF tends to make the individual designs in the sequence more similar to each other. This trade-off is clearly seen in Figure 20.d. NS is when we need to generate just one cluster which is indeed fully optimizing the individual filters. As can be observed up to RF = 0.9 the average reconfiguration cost is the same as the average size of the filters, suggesting that we have to fully reconfigure individual filters to implement the next filter in the sequence. However when RF = 0.9 and RF = 1.0 the reconfiguration cost drops.

6 Conclusion

In this work we introduced a framework to optimize multi-mode FIR-like structures. We proposed splitting the filter structure to reconfigurable and static sections. In this framework we proposed a polynomial time optimization algorithm to optimize the reconfigurable section of the filter. We showed the trade-off between reconfiguration overhead and the area of filter for different LIs. We furthermore showed that by applying the optimal solution we can improve the results by 17.2% and in average 5.5% compared to the clustering technique previously proposed in [1]. For a given sequence of filters, we introduced techniques to jointly optimize the filter structures. We also showed that, by tuning RF we can get the same quality of design as generic design (in terms of reconfiguration overhead) with only 48% of resources.

7 Bibliography

[1] GHOLAMIPOUR A. H., ESLAMI H., ELTAWIL A., KURDAHI F. J., “Size-Reconfiguration Delay Tradeoffs for a Class of DSP Blocks in Multi-mode Communication Systems”, FCCM 2009

[2] http://www.itl.nist.gov/div897/sqg/dads/HTML/perfectBinaryTree.html

[3] KNUTH D. E., “*Art of Computer Programming*”, Volume 4, Fascicle 3, Addison-Wesley, 2005.

[4] J. KLEINBERG, E. TARDOS. “*Algorithm Design*”, Addison-Wesley, 2005.

[5] LIM D., PEATTIE M., “*Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*” Xilinx Application Note XAPP290, V1.0, Xilinx.Inc (May 2002)

[6] LYSAGHT P., BLODGET B., MASON J., YOUNG J., BRIDGFORD B., “Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfigurable of Xilinx FPGAs”, FPL 2006

[7] J. Villasenor, B. Schoner, K.N. Chia, and C. Zapata, H.J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "*Configurable Computing Solutions for Automatic Target Recognition*", FCCM 1996

[8] F. TUFVESSON, O. EDFORS AND M. FAULKNER, “Time and Frequency Synchronization for OFDM Using PN-Sequence Preambles,” VTC, 1999.

[9] WIRTHLIN, M. J. 2004. “Constant Coefficient Multiplication Using Look-Up Tables”, J. VLSI Signal Process. 2004

[10] DENYER P. B., RENSHAW D., “*VLSI Signal Processing, a bit-serial Approach*” Addison-Wesley Longman Publishing, 1985

[11] WHITE S. A., “Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review”, IEEE ASSP 1989

[12] LEE H., SOBELMAN G. E, "*FPGA-based FIR Filters Using Digit-Serial Arithmetic*", IEEE International ASIC Conference and Exhibit 1997

[13] "FIR Compiler V5.0", Xilinx Data Sheet DS534, Xilinx. Inc (June 2009)

[14] ANDRAKA R. J., "*FIR Filter Fits in an FPGA using a Bit Serial Approach*", 3rd Annual PLD conference and Exhibit 1993

[15] HARTLEY R. I., PARHI K. K., "*Digit-Serial Computation*", Kluwer Academic, Boston, MA, 1995

[16] PETERSEN R. J., "An Assessment of the Suitability of Reconfigurable Systems for Digital Signal Processing", Master's Thesis, Brigham Young University, 1995

[17] HOSANGADI A., FALLAH F., KASTNER R., "Algebraic Methods for Optimizing Constant Multiplications in Linear Systems". VLSI Signal Processing, 2007

[18] CHATTERJEE A., ROY R. K., AND D_ABREU M. A., "Greedy Hardware Optimization for Linear Digital Circuits using Number Splitting and Refactorization," TVLSI 1993

[19] OH Y. J., LEE H., LEE C. H., "A reconfigurable FIR filter design using dynamic partial reconfiguration.", ISCAS 2006

[20] GHOLAMIPOUR A. H., BOZORGZADEH E., BAO L., "Seamless Sequence of Software Defined Radio Designs through Hardware Reconfigurability of FPGAs", ICCD 2008

[21] BRUNEEL K., STROOBANDT D., "Automatic Generation of Run-Time Parameterizable Configurations", FPL 2008

[22] PAPADIMITRIOU K., ANYFANTIS A., DOLLAS A., "*An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems*", IEEE Transaction on Instrumentation and Measurement 2009

[23] K. Y. KHOO, A. KWENTUS, AND A. N. WILLSON, "*A Programmable FIR Digital Filter Using CSD Coefficients*", IEEE Journal of Solid-State Circuits, Jun. 1996.

[24] L. SINGHAL, E. BOZORGZADEH, "Physically-aware Exploitation of Component Reuse in a Partially Reconfigurable Architecture", IPDPS 2006

[25] J. GAUSE, P. Y. K. CHEUNG, W. LUK, “Reconfigurable Shape-Adaptive Template Matching Architectures”, FCCM 2002

[26] N. SHIRAZI, W. LUK, P. Y. K. CHEUNG, “Automating Production of Run-Time Reconfigurable Designs”, FCCM 1997

[27] GHOLAMIPOUR A. H, GORCIN A., CELEBI H., TOREYIN B. U., SAGHIR M. A. R, KURDAHI F. J., ELTAWIL A., “Reconfigurable Filter Implementation of a Matched-filter Based Spectrum Sensor for Cognitive Radio Systems”, to appear in proceedings of ISCAS 2011

[28] GHOLAMIPOUR A. H, KURDAHI F. J., ELTAWIL A., SAGHIR M. A. R., “Exploiting Architectural Similarities and Mode Sequencing in Joint Cost Optimization of Multi-mode FIR Filters”, in proceedings of FPL 2010

[29] GHOLAMIPOUR A. H., KURDAHI F. J., ELTAWIL A., SAGHIR M. A. R. “Placement-aware Partial Reconfiguration for a Class of FIR-Like Structures”, in proceedings of IEEE ICT 2010

[30] GHOLAMIPOUR A. H., DILLENCOURT M., KURDAHI F., ELTAWIL A., “Heterogeneous Mapping to Minimize Resource Usage Under Maximal Spatial Reuse Constraints for FIR-like Structures”, CECS technical report # TR11-01

[31] PROAKIS J. G., *Digital Communications*. New York, U.S.A.: McGraw-Hill International Editions, New York, 2001.

[32] YUCEK T., ARSLAN H., “A survey of spectrum sensing algorithms for cognitive radio applications,” *IEEE Communications Surveys & Tutorials*, vol. 11, no. 1, pp. 116-130, 2009

[33] 3GPP2 C.S0002-E V1.0, “Physical Layer Standard for cdma2000 Spread Spectrum Systems,” October 2010.

J. PUCHINGER, G. RAIDL, AND U. PFERSCHY, "The Multidimensional Knapsack Problem: Structure and Algorithms," Informs Journal on Computing, DOI: 10.1287/ijoc.1090.0344, 2009.