Multipumping Flexible DSP Blocks for Resource Reduction on Xilinx FPGAs

Bajaj Ronak, *Student Member, IEEE,* and Suhaib A. Fahmy, *Senior Member, IEEE*

***Abstract*—For complex datapaths, resource sharing can help reduce area consumption. Traditionally, resource sharing is applied when the same resource can be scheduled for different uses in different cycles, often resulting in a longer schedule. Multipumping is a method whereby a resource is clocked at a frequency that is a multiple of the surrounding circuit, thereby offering multiple executions per global clock cycle. This allows a single resource to be shared among multiple uses in the same cycle. This concept maps well to modern field-programmable gate arrays (FPGAs), where hard macro blocks are typically capable of running at higher frequencies than most designs implemented in the logic fabric. While this technique has been demonstrated for static resources, modern digital signal processing (DSP) blocks are flexible, supporting varied operations at runtime. In this paper, we demonstrate multipumping for resource sharing of the flexible DSP48E1 macros in Xilinx FPGAs. We exploit their dynamic programmability to enable resource sharing for the full set of supported DSP block operations, and compare this to multipumping only multipliers and DSP blocks with fixed configurations. The proposed approach saves on average 48% DSP blocks at a cost of 74% more LUTs, effectively saving 30% equivalent LUT area and is feasible for the majority of designs, in which clock frequency is typically below half the maximum supported by the DSP blocks.**

***Keywords*—Digital signal processing, field programmable gate arrays, design automation, arithmetic synthesis.**

I. Introduction

Modern FPGAs include a number of embedded hard blocks, including memory blocks, DSP blocks, and embedded processors that offer performance, power, and area benefits over "soft" implementations of the same functions [1]. The DSP blocks in modern Xilinx FPGAs support a range of arithmetic functions, selected through control signals that can be dynamically set at runtime, though this is not typically exploited by vendor tools. Since hard blocks are a limited resource, it is prudent to share these resources where possible. Traditionally, operations scheduled in non-overlapping schedule times (STs) can be mapped to the same hardware resource in the binding stage by adding multiplexers at the inputs and de-multiplexers at the outputs of the block. However, this generally increases schedule length.

Multipumping is another technique that reduces hard block utilisation, without increasing schedule length and initiation interval (II). The shared block is run at a frequency that is a multiple of the surrounding circuit, hence offering multiple computational cycles per global cycle. This is possible with DSP blocks since they support much higher frequencies than the typical complete circuit, and therefore can be clocked to enable multiple operations to be scheduled in the same clock cycle. Canis *el al.* [2] demonstrated the technique by mapping two multiply operations onto a single multipumped DSP (mpDSP) block per global clock. The multiplier in the DSP block, becomes a shared resource that can be mapped to by finding multiple multiplications that can be scheduled in the same cycle. Multipumping has also been used to enable multiported memories with fewer resources [3].

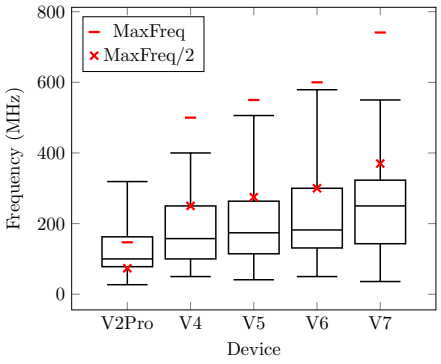

Fig. 1: Reported frequencies on Xilinx Virtex devices for over 350 papers (1100 designs) published in recent FPGA conferences.

The DSP blocks in modern Xilinx FPGAs support frequencies of over 500 MHz on a Virtex 6 [4], while complete systems will typically have a frequency of around 150–250 MHz. Multipumping relies on there being a significant difference between overall circuit frequency and the supported frequency of the hard block to be multipumped. A factor of two makes multipumping feasible.

To demonstrate the feasibility of multipumping, we analysed FPGA designs presented from 2010 onwards at four key FPGA conferences.

1) The ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA).
2) The IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM).
3) The International Conference on Field Programmable Logic and Applications (FPL).
4) The International Conference on Field Programmable Technology (FPT).

Fig. 1 shows a box plot of the reported operating frequencies for designs in all papers analysed, split across Xilinx Virtex device families. The median (indicated by the line inside the box) and third quartile (top of the box) frequencies are of particular interest here. The datasheet maximum operating frequency of

the DSP blocks for each family and half this frequency are also shown. We can see that the median design frequency has not increased at the same rate as supported DSP block frequencies in recent device generations, and that over three quarters of designs are comfortably below half the supported DSP block frequency for newer families. The outliers are typically small designs, or those manually optimised around these hard blocks for maximum performance. A similar study in [5] analysed papers at FCCM between 1995 and 2014 and similarly concluded that despite embedded blocks significantly improving, overall design performance has lagged. The results in Fig. 1 mean multipumping (or specifically dual-pumping) of DSP blocks is feasible for most designs.

DSP blocks have increased in complexity, supporting more operations. The method in [2] considers only the multiplier in the DSP block, while in previous work [6], we showed how the multiple sub-blocks could be multipumped through a brute-force (BF) schedule analysis. In this paper, we show how functional flexibility offers much improved opportunities for multipumping over fixed-function DSP blocks. This flexibility allows different configurations of supported datapaths to share the same DSP block. We present a tool that incorporates new scheduling techniques to exploit this extended sharing and generate more efficient implementations of datapaths with DSP block usage reduced by a half. This is the first work in which multipumping has been applied to dynamically configurable DSP blocks.

The main contributions of this paper are:

1) A reconfigurable multipumped DSP block architecture using the Xilinx DSP48E1 primitive.
2) Multipumping scheduling techniques that exploits the dynamic programmability of DSP blocks.
3) An approach combining the concepts of traditional resource sharing and multipumping, that reduces DSP block usage by half.
4) Integration of these techniques into an automated tool.
5) Evaluation of multipumping techniques across a suite of 18 benchmarks with varied complexity.

II. Related Work

A significant amount of research has been done on resource sharing at the register-transfer level (RTL) level as well as in high-level synthesis [7], [8], [9], [10]. A typical high-level synthesis (HLS) tool flow consists of three major steps: 1) allocation; 2) scheduling; and 3) binding. Cardoso [8] proposed an algorithm combining temporal partitioning, resource sharing, scheduling, allocation, and binding to obtain resource efficient implementations. Instead of partitioning the design first and then applying resource sharing for each partition, resource sharing is explored in each temporal partition to minimise resource requirements, resulting in a reduced number of partitions and more logic implemented in each partition. Heuristics for global resource sharing were proposed in [9], which focuses on inter-basic-block sharing. Computational modules across basic blocks are analysed to minimise connections and functional resources. Patterns for combining resources are extracted and prioritised, resulting in more effective sharing than when considered individually. This is similar to the mapping of multiple compute nodes to compound resources like DSP blocks.

The work in [10] combined module selection and resource sharing to minimise area while achieving throughput requirements. For a given throughput constraint, the proposed technique explores implementations with different frequencies and IIs to achieve a target throughput. A mapping-aware pipeline scheduling approach based on an MILP formulation of modulo scheduling is proposed in [11]. The algorithm accepts multiple constraints like clock period, II, resource constraints, and generates a schedule satisfying all the constraints while considering LUT mapping.

Generally, HLS tools use static scheduling to determine the extent of resource sharing possible. Work in [12] proposed a source-to-source transformation which improves the efficiency and II using dynamic scheduling techniques. Dynamic scheduling can exploit the extent of resource sharing on-the-fly, however, extra logic required for complex decision making during execution results in a resource overhead. A recent algorithm proposed in [13] attempts to optimise resource usage and II for different loops in a design to achieve maximum throughput. Instead of optimising different loops individually, a global resource sharing approach is proposed, enabling resource sharing across different loops. A method to reduce resource usage by determining a pattern of operations, which is then used for efficient binding was proposed in [14]. Studies in [15] and [16] analysed the impact of resource sharing on the performance of FPGA designs. They show the cases for which resource sharing is advantageous and where it can adversely impact performance.

Scheduling is a critical step as it determines the degree of possible resource sharing. Various heuristics have been proposed including list scheduling [17], force-directed scheduling (FDS) [18], and a recent scheduler based on system of difference constraints (SDC) [19]. We are not aware of any work that focuses on multicycle flexible hard blocks like the DSP48E1. These present unique challenges in their ability to share different computations on the same hardware, and the complex latency constraints enforced by their pipeline configuration.

The concept of multipumping has been applied previously in other areas, such as double-data-rate memories, that allow read/write data at double the system clock frequency. It has been extensively used in designing register files [20], and multiported memories [21], [22]. A whitepaper by Xilinx [23] used multipumped DSP blocks with lower input data rates than the DSP block throughput. However, this capability has not been incorporated in the Xilinx Vivado HLS tool. Multipumping has been used to reduce DSP block utilisation [2] in the open-source LegUp high-level synthesis tool [24].

Our paper differs from that in [2] primarily in that we consider the DSP blocks as fully featured blocks supporting different configurations that can be dynamically reconfigured rather than just multipliers. In our previous work [6], we showed that multipumping only multipliers can have a detrimental effect on area as other sub-block operations mapped to DSP blocks must then be implemented in logic. We also showed that it is

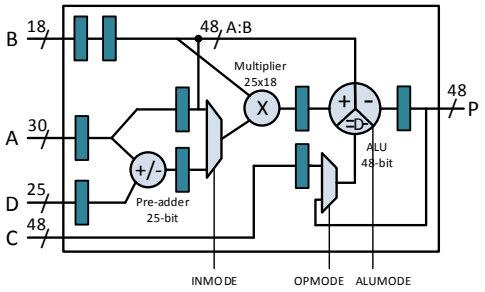

Fig. 2: DSP48E1 primitive structure.

TABLE I: Maximum frequency in MHz of DSP48E1 for different blocks used and pipeline stages (Virtex 6 XC6VLX240T-1).

	Pipeline Depth		
Sub-blocks used	**2**	**3**	**4**
Multiplier	236	473	473
Pre-adder, Multiplier	196	292	473
Multiplier, ALU	263	473	473
Pre-adder, Multiplier, ALU	196	292	473

possible to multipump the DSP block including its other sub-blocks with a BF schedule analysis. In this paper, we extend the approach to take advantage of the dynamic programmability of the DSP block, showing that this offers more opportunities to take advantage of multipumping, further reducing area.

III. Xilinx DSP48E1 Primitive

The simplified architecture of the DSP48E1 primitive in modern Xilinx FPGAs is shown in Fig. 2. Inputs A, B, C, and D are of different wordlengths: 30, 18, 48, and 25 bits, respectively. The DSP48E1 comprises multiple sub-blocks: a 25-bit pre-adder, a 25×18-bit multiplier, and a 48-bit ALU, which can be combined to perform different functions. The desired operations are set using three configuration inputs: 1) INMODE; 2) OPMODE; and 3) ALUMODE. DSP Blocks are internally pipelined with up to four stages allowing maximum throughput to be achieved even when all three sub-blocks are used. Table I shows the maximum achievable frequency for different configurations. The maximum frequency is achieved with three pipeline stages if the pre-adder is not used, or four pipeline stages if it is used.

A key feature of the DSP48E1 primitive is its dynamic programmability that allows functionality to be modified at runtime in each clock cycle by changing the configuration inputs. This greatly enhances the capabilities of the DSP48E1 primitive, as it can be reprogrammed and multiple operations can be mapped to a single DSP block. This flexibility has previously been demonstrated as enabling the design of small high speed soft processors [25] and overlay architectures [26].

We conducted a number of experiments with Xilinx RTL and HLS tools, mapping designs that could fit in a single DSP block with dynamic programmability, but found that

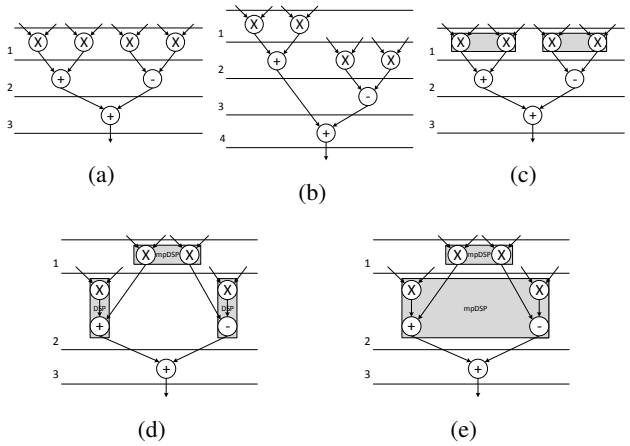

Fig. 3: **(a)** Input dataflow graph **(b)** Traditional resource sharing (Number of multipliers available = 2) **(c)** Multipumping multipliers only **(d)** Multipumping fixed configuration DSP blocks **(e)** Multipumping DSP blocks with dynamic programmability.

neither flow exploited this capability. The only sharing found was multiplexing DSP block inputs, and when different sub-blocks beyond the multiplier are required, these are always implemented in LUTs. We show that DSP block flexibility allows for improved application of multipumping, resulting in increased resource sharing beyond what the vendor tools can achieve.

IV. Resource Sharing and Multipumping

Traditional resource sharing allows the same resource to be shared in a time-multiplexed manner by multiplexing its inputs, and demultiplexing its outputs. Complex embedded blocks in FPGAs are an ideal target for resource sharing since they are scarce compared to other resources. Multipliers consume significant resources and offer poor performance when implemented using LUTs and registers, though adders and subtractors can be efficient in general logic. Resource sharing is not advisable for simple adders since the multiplexing overheads negate the benefits of sharing [15]. In traditional resource sharing, we search for independent uses of the same resource that are not scheduled at the same time, and add the resource sharing circuitry around the resource. If there are M multipliers available, for example, we can schedules the dataflow graph such that, in each ST, there are no more than M multiplication operations, but this can result in a longer schedule.

Multipumping achieves resource reduction by mapping two operations onto the same resource running at twice the clock rate, thereby giving it two execution cycles in the time of one global cycle. We illustrate this using a simple dataflow graph, which is part of larger design, shown in Fig. 3a. Assume all multiply operations are implemented using fully pipelined (four stage) DSP blocks and add/sub operations are either

merged with multipliers into DSP blocks or mapped to LUT based adders with a latency of two clock cycles. Without any resource sharing, the dataflow graph can be implemented using four DSP blocks and three LUT based adders, with a schedule length of eight clock cycles. Using traditional resource sharing, the graph can be implemented using two DSP blocks and three LUT based adders, reusing the DSP blocks, but at the expense of increased latency (12 clock cycles) and II, as shown in Fig. 3b. With multipumped DSP block multipliers, all four multiplication operations can be mapped to two DSP blocks [Fig. 3c], saving two DSPs without an increase in schedule time (compared to no resource sharing). Two operations sharing the same block are shown in shaded rectangles. Adder requirements remains same. When we use all sub-blocks in the DSP blocks, nodes implementing the same set of operations can be multipumped together. However, this could increase DSP block usage since opportunities for multipumping could be reduced if there are no matches for the full set of sub-blocks. Multipumping DSP blocks with the same configuration requires three DSP blocks and one LUT based adder, with a latency of ten clock cycles [Fig. 3d]. Implementations in Fig. 3c and Fig. 3d provide a trade-off between DSP block and generic FPGA resource usage. By exploiting the dynamic programmability of DSP blocks, resource usage can be further minimised, as different configurations of the DSP block can also be multipumped together, as shown in Fig. 3e. Multipumping with dynamic programmability requires two DSP blocks with one LUT based subtractor.

The primary condition for multipumping to be feasible as a resource reduction approach is that the embedded block should support a frequency that is double the frequency requirement of the overall design. To maximise multipumping, the dataflow graph should be scheduled such that an even number of DSP block operations can be scheduled in each ST, so that these can be shared across multipumped DSP blocks.

V. Multipumped DSP Block Architecture

Our multipumped DSP block (mpDSP) instantiates the Xilinx DSP48E1 primitive, exploiting the full set of sub-blocks and dynamic programmability. We assume the mpDSP runs at double the speed of surrounding logic, requiring two clock domains. A block diagram of the mpDSP is shown in Fig. 4. $Clk2$ is aligned with and exactly twice $Clk1$. $Clk1\,Follower$ follows the system clock ($Clk1$), and is fed to the multiplexer select signal to choose between inputs to the DSP48E1 primitive. We do not use $Clk1$ directly to avoid possible hold-time violations [23]. Theoretically, an application at lower frequency could offer $4\times$ multipumping, however, overheads incurred by the data and control multiplexers and the increased complexity of identifying sharing possibilities in the schedule would mean diminished benefits.

The preadder and ALU can be enabled/disabled/reconfigured in each clock cycle, depending on the logic to be mapped to the mpDSP. The multiplier is always enabled. All four pipeline stages of the DSP48E1 primitive are enabled to achieve maximum frequency for $Clk2$. In configurations for which the ALU block is used, two extra

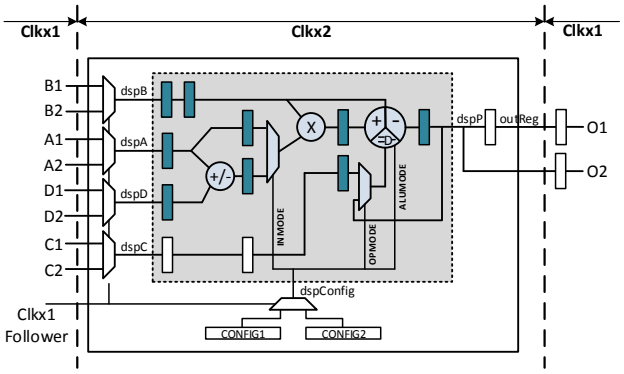

Fig. 4: Multipumped DSP block (mpDSP) architecture.

registers are added to align the C input. The configuration word for the DSP48E1 primitive is 17 bits long, consisting of 5-bit INMODE, 7-bit OPMODE, 4-bit ALUMODE, and 1-bit CARRYIN signals. CARRYIN is the carry input to the ALU sub-block, and must be set to 1 when the output of the multiplier is subtracted from input C.

The mpDSP has at most eight inputs and two outputs, when both temporal configurations utilise all three sub-blocks. Configurations of the DSP48E1 primitive are passed through parameters. If a configuration does not utilise either the preadder or ALU sub-blocks, the corresponding inputs are held at zero in the instantiation of the mpDSP, and these are then optimised away during synthesis. Fig. 5 shows the timing diagram of the mpDSP block. At each rising edge of the system clock ($Clk1$), input sets I_1 (A1, B1, C1, D1) and I_2 (A2, B2, C2, D2) arrive at the multiplexers. For the first half of the cycle, $Clk1\,Follower$ causes the multiplexer to pass the I_1 inputs and the corresponding configuration bits are applied. The I_2 inputs are selected in the second half of the cycle. The latency of the mpDSP is equivalent to 3 system clock cycles, after which the outputs corresponding to both sets of inputs arrive at $O1$ and $O2$.

The maximum frequency for a design using mpDSP blocks is calculated as $min(f_{Clk1}, f_{Clk2}/2)$. On more modern devices where the DSP block can reach up to 700 MHz, a system clock of over 300 MHz (after taking into account the delays due to multiplexers) can be achieved, which is above the maximum achievable frequency for most larger designs as shown in Fig. 1.

VI. Scheduling for Multipumping

To utilise the full potential of multipumping, the DSP dataflow graph (DDFG) (discussed in more detail in [6]) should be scheduled in such a way that in each schedule time (ST) i, an even number, $2M_i$, of DSP nodes are scheduled. $2M_i$ DSP48E1 primitive templates then can be mapped to M_i mpDSP blocks. A simple BF search was proposed in [6], but that did not scale well to large graphs and was unsuitable when considering the full flexibility of the DSP block as in this paper, as the search space becomes too large. Hence, we present two scheduling techniques that can determine a

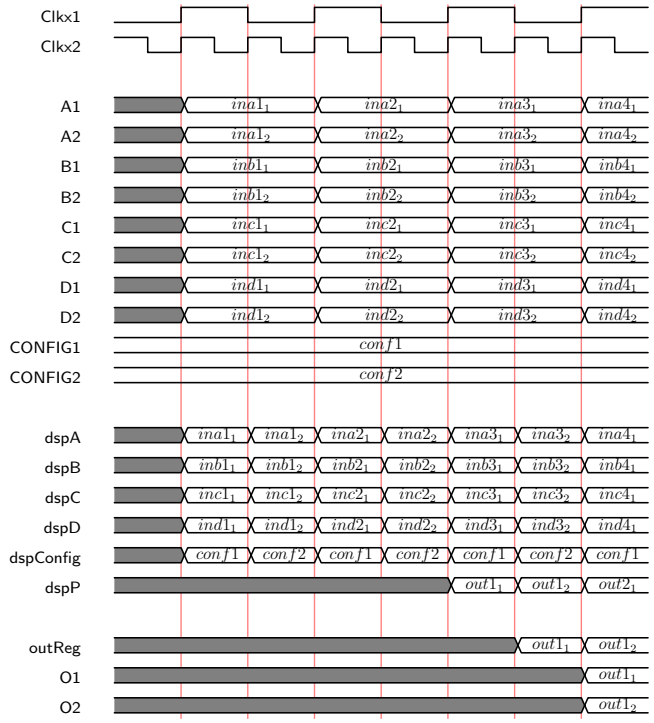

Fig. 5: Timing diagram showing mpDSP operation.

multipumping schedule in deterministic time, the first extends SDC scheduling, while the second adapts FDS. The FDS-based approach is shown to be able to prioritise multipumping of identical DSP block configurations, resulting in reduced LUT overhead compared to the SDC-based approach, as discussed in more detail in Section VI-D.

A. Brute-Force Scheduling

A BF schedule is determined as follows. First, the as soon as possible (ASAP) and as late as possible (ALAP) schedules of the DDFG are determined to compute the mobility of each node. This is the measure of scheduling flexibility for the node; the difference between the ALAP and ASAP STs. Nodes with zero mobility are those which must be scheduled in a particular ST to maintain data dependencies. Nodes with nonzero mobility can be exploited to arrive at a schedule which maximises opportunities for multipumping. All possible schedules are generated for these mobilities, ignoring dependencies. In the next step, schedules that do not satisfy dependencies are discarded. For the remaining valid schedules, we calculate the mpDSP block requirement for each schedule and track the minimum mpDSP (minmpDSP) block requirement. Schedules using more mpDSPs than minmpDSP are then discarded. Out of those remaining, the one requiring the minimum number of balancing registers is then selected as the final schedule. Although this approach results in an optimised schedule, the exhaustive search does not scale well to large dataflow graphs, and adding DSP block flexibility complicates the search further.

Algorithm 1: SDC based multipumping

```
def sdcMpSchedule(ddfg, schObjective, λ):
    Data: DSP Dataflow Graph (ddfg), schObjective, λ
    Result: Scheduled ddfg (schDDFG)
    begin
        asap(ddfg)
        alap(ddfg)
        #generate edmond matching dataflow graph and
        determine multipumping matchings
        EMDDFG = generateEMDDFG(ddfg)
        matchings = getMatchings(EMDDFG)
        lp = initialiseLP(ddfg) #initialise LP problem
        lp = addMulticycleConstraints(lp, ddfg)
        lp = addDependencyConstraints(lp, ddfg)
        lp = addMultipumpConstraints(lp, ddfg, matchings)
        lp = addObjFunc(lp, schObjective, λ) #add objective
          function to LP
        schDDFG = solveLP(lp) #solve formulated LP
        #remove infeasibility if formulated LP is infeasible
        while True:
            if (schDDFG != -1):
                break
            incMatch = identifyIncorrectMatching(lp)
            EMDDFG = updateEMDDFG(EMDDFG,
              incMatch)
            newMatchings = getMatchings(EMDDFG)
            lp = updateLP(lp, ddfg, newMatchings)
            schDDFG = solveLP(lp)
        return schDDFG
```

B. SDC-Based Scheduling

SDC scheduling is based on the SDCs, which formulates the scheduling problem mathematically as a set of linear constraints that can be solved using a linear programming (LP) solver. The scheduling algorithm is detailed in Algorithm 1.

First, the ASAP and ALAP schedules of the DDFG are determined to compute the mobility of each node. Before formulating the LP problem to determine the schedule, we must identify pairs of DSP blocks that can be multipumped together. Each DSP block can possibly be paired with any DSP blocks that can be scheduled in same ST. We pair DSP blocks to maximise the number of pairs. In graph theory, a *matching* of a graph is a set of edges such that no edge shares a vertex with any other, i.e., each vertex is connected to a maximum of one edge in the matching. *Maximum matching* is a matching comprising the maximum number of edges, i.e., covering the maximum number of vertices. Identification of valid pairs of DSP blocks for multipumping can be formulated as finding the maximum matching for a graph where each vertex is a DSP block and those sharing a ST are connected via edges. DSP blocks at the endpoints of each edge in the matching can then be multipumped and mapped to an mpDSP. We use the Edmond Matching (EM) algorithm [27] to determine the maximum number of DSP block pairs.

The DDFG includes DSP blocks and add/sub blocks as vertices with edges representing data dependencies. A separate

EMDDFG, including only DSP blocks is derived from this. In the EMDDFG, an edge connecting vertices v_i and v_j indicates that the DSP blocks can be multipumped together. An edge is added between v_i and v_j if:

1) v_i and v_j do not depend on each other, i.e., there is no path connecting the output of v_i to v_j and vice versa;
2) The ST of v_i and v_j overlap, to allow multipumping. The ST of a node in the ASAP schedule is the earliest a node can be scheduled and the ALAP ST is the latest ST for a node. Nodes are considered as overlapping if the ASAP ST of v_i is less than the ALAP ST of v_j and the ASAP ST of v_j is less than the ALAP ST of v_i.

After determining the set of DSP block pairs which can be multipumped, the algorithm proceeds to SDC scheduling. Generating the schedule using SDC is done in four steps.

1) Initialise LP Problem: The LP problem is initialised and scheduling variables for each node in the DDFG are added. The number of scheduling variables associated with each node is equal to the latency of the node, one for each pipeline stage.

2) Model Scheduling Constraints: Next, the scheduling constraints are modeled. Constraints added to the LP problem are as follows.

a) *Multicycle constraints:* Constraints are added such that difference between the scheduling variables of a node is always 1.
b) *Dependency constraints:* To ensure the correct flow of operations, constraints are added for each dependent pair of nodes such that the start time of the destination node is always greater than end time of the source node.
c) *Multipumping constraints:* DSP block pairs identified for multipumping using the EM algorithm should be scheduled in the same ST. For each DSP blocks pair to be multipumped, a constraint is added such that the start times for both the DSP blocks is the same.

3) Formulate Objective Function: Thirdly, the objective function is formulated. Both ASAP and ALAP scheduling objectives are supported, selectable by the user. As ALAP scheduling maximises the objective function, constraints for maximum ST (λ) for each output node are also added, such that the end time of each output node is less than or equal to λ.

4) Solve LP: The LP with the objective function defined in the previous step is solved, subject to the defined constraints, using an LP solver. We used the open-source "lpsolve" [28].

Detailed mathematical formulations of the LP and its constraints for SDC are discussed in [19].

In the EMDDFG, independent overlapping nodes are connected by edges, but, information on data dependencies between nodes is not captured. Due to this, some matchings generated by the EM algorithm can result in infeasible LP formulations, for which no solution exists that satisfies all the constraints. In order to resolve this, we iteratively perform the following four steps until the formulated LP results in a solution.

1) Identify incorrect matching, i.e., a multipumping constraint due to which the LP is infeasible.
2) Remove the edge corresponding to the identified matching from the EMDDFG.
3) Rerun the EM algorithm with the updated EMDDFG, resulting in a new set of matchings.
4) Update the multipumping constraints according to the new matchings and solve the LP.

An infeasible LP implies that for one of the multipumping pairs, the start times of both the DSP blocks cannot be equal. In order to identify this incorrect matching (step 1), we modify the multipumping constraints one-by-one and attempt to solve the LP. Multipumping constraints are of the form $sv_{start}(v_i) - sv_{start}(v_j) = 0$, where $sv_{start}(v)$ is the start time variable of vertex v, and v_i and v_j are DSP block vertices to be multipumped. To relax the multipumping constraints, instead of forcing the start times to be equal, the 0 on the right-hand side of the constraint equation is replaced by an unbounded variable α. The relaxed constraint for which the LP results in a feasible solution after the above replacement is the incorrect matching, and the corresponding edge must be removed from the EMDDFG.

The time complexity of EM algorithm is $O(n(n+m)m)$, where n is number of DSP block nodes in the graph and m is number of edges in the EMDDFG. The LP model used for SDC scheduling can be solved in $O(p^2(q + p\log(p))\log(p))$, where p is number of scheduling variables and q is number of constraints. Considering the iterative method of removing infeasibility and pair-wise computation of mpDSP blocks, total worst case time complexity of SDC-based scheduling algorithm is $O(n^2(n(n+m)m + p^2(q + p\log(p))\log(p)))$.

C. FDS-Based Scheduling

FDS [18] is a heuristic method for generating a schedule using a deterministically greedy approach without backtracking. Although FDS follows a greedy approach, all possible STs of the node being scheduled are explored with consideration for the effect on other nodes before a ST is assigned, resulting in a satisfactory schedule.

We use FDS with a modification to generate multipumping optimised schedules. Instead of directly selecting the ST of minimum force for a node, we explore the possibilities of multipumping the node with previously scheduled nodes. If a match is found, that ST is selected for the node, otherwise the minimum force ST is selected. The FDS-based scheduling algorithm is detailed in Algorithm 2.

First, the ASAP and ALAP schedules of the DDFG are generated to compute the mobility of each node. We then create a priority list of nodes in the DDFG, which orders the traversal for scheduling. We sort nodes according to their ASAP ST, and ALAP STs are used as a tie-breaker. We initialise the DDFG schedule by assigning nodes with zero mobility an ST equal to their ASAP schedule time, and -1 for other nodes that are yet to be scheduled.

We traverse the nodes in the DDFG according to the priority determined above and schedule one unscheduled node in each iteration through three stages. In the first stage, we create a distribution graph (DG) of the operation of the current node. Each node can be of two types. It can either be a DSP node

Algorithm 2: Modified FDS for multipumping

```
def fdsMpSchedule(ddfg):
    Data: DSP Dataflow Graph (ddfg)
    Result: Scheduled ddfg
    begin
        asap(ddfg)
        alap(ddfg)
        #for each dsp node n in ddfg
        for n in ddfg:
            n[mobility] = n[t_alap] − n[t_asap]
        #assign schedule time to nodes with zero mobility; -1
          for other nodes
        initialiseSchedule(ddfg)
        #initialise an empty list of nodes which are paired for
          multipumping
        matchedNodes = [ ]
        for n in ddfg:
            if n[schTime] != -1:
                continue
            else:
                currDG = getDistributionGraph(n[type], n)
                nF = [0]*(n[mobility] + 1)
                #for each schedule time of node n
                for i in (n[t_asap], n[t_alap] + 1):
                    nF[i] += calcSelfF(currDG,i,ddfg)
                    nF[i] += calcPredF(currDG,i,ddfg)
                    nF[i] += calcSuccF(currDG,i,ddfg)
                minFIndex = nF.index(min(nF))
                if n[type] is addsub:
                    n[schTime] = n[t_asap] + minFIndex
                else:
                    [matchedNode,fIndex] = findMpNode(nF,
                      ddfg)
                    if matchedNode:
                        n[schTime] = n[t_asap] + fIndex
                        matchedNodes.append(n)
                        matchedNodes.append(matchedNode)
                    else:
                        n[schTime] = n[t_asap] + minFIndex
        return ddfg
```

implementing a set of operations using a DSP48E1 primitive or it can be an add/sub node, to be implemented using a LUT based adder/subtractor. The DG is a set of sums of probabilities of scheduling an operation in a particular ST. For each operation Op, $DG(i) = \sum_{n=1}^{N} Prob(n,i)$, where N is the total number of nodes in the DDFG, and $Prob(n,i)$ is $1/(n[mobility]+1)$ if $n[t_{asap}] \leq i \leq n[t_{alap}]$, 0 otherwise.

The second stage is to calculate the *force* for the node, for each possible ST. This is a measure of the cost of scheduling the node in a particular ST, and is the product of the value of the DG of the ST and the change in the operation's probability if it is scheduled in that ST. Force for an operation assigned ST i is calculated as, $Force(i) = DG(i) \times \Delta Prob(Op,i)$, where $\Delta Prob(Op,i)$ is the change in probability. Three types of force are associated with each node. First is the *self force*, which is the sum of *forces* for each possible ST, calculated using the change in probabilities of the current node being scheduled. While calculating the *self force* for ST i, the probabilities of the node changes to 1 for ST i, 0 otherwise; these are the probability changes used. Assigning the node in a ST can affect the mobility of its predecessor and successor nodes. Similar to *self force*, *predecessor force* and *successor force* are calculated for nodes whose mobility is affected due to the current node ST. The total force for the current ST is the sum of all three forces.

In the third stage, traditional FDS selects the ST with minimum force. For the multipumping optimised FDS, we modify this stage. Starting from the ST with minimum force, we check if there are nodes scheduled in the same ST that are not yet paired with any other node for multipumping. If a match is found, we assign the ST of the matched node. We continue to check STs with ascending force to find a match. Both the current node and the matched node are flagged as *matched* and are not considered for matching in subsequent iterations. If no match is found, the node is assigned to the ST with minimum force. The worst case time complexity of the FDS-based scheduling algorithm is $O(n^2)$, where n is number of nodes in the graph, due to the pair-wise force computations [18].

D. Comparing SDC-Based and FDS-Based Scheduling

We ensure functional correctness of the generated schedules for both techniques. For SDC-based scheduling, dependency constraints ensure that while determining opportunities for multipumping, the order of operations remains the same. For FDS-based scheduling, the priority list generated prior to scheduling for graph traversal ensures functional correctness. Furthermore, when the ST for a node is fixed, the probabilities of dependent nodes change accordingly. Hence scheduling never affects the order of nodes or dependencies in the graph.

The SDC- and FDS-based scheduling approaches can generate schedules in deterministic time while also exploiting the dynamic programmability of DSP blocks to maximise multipumping. One advantage of the FDS-based approach over the SDC-based approach is that it can prioritise multipumping of identical DSP block configurations, whereas the EM algorithm used for matching in the SDC approach treats all DSP block configurations identically.

The overheads of using mpDSP blocks include the multiplexers required to select two different sets of inputs, the control to switch DSP block configuration, three 48-bit registers for outputs, and, when the ALU sub-block is used, two extra 48-bit registers to balance input C. Multipumping with the same configuration results in savings in terms of LUTs as there is no need for the configuration control circuitry, and where a sub-block is unused, a multiplexer is saved. Consider a scenario where four DSP blocks can be scheduled in one ST, mapping to two mpDSPs. Among the four DSP blocks, two utilise only the multiply sub-block (*mul*) and the other two utilise all three sub-blocks (*add-mul-add*). In SDC, the identified pairs could each be *mul* and *add-mul-add*, requiring an extra input and control register for both the mpDSPs.

However, in FDS, the *mul* operations will be paired and the *add-mul-add* paired. Thus, only one mpDSP block will require extra registers compared to both the blocks for SDC, and the configuration circuitry is optimised away, saving LUTs.

E. Further Resource Sharing

Ideally, for a dataflow graph with n DSP blocks, multipumping should result in a DSP block reduction from n to $\lceil \frac{n}{2} \rceil$. However, this is not always feasible due to data dependencies and the structure of the input dataflow graph that may result in STs with odd numbers of DSP blocks. In such cases, the $(2n+1)$ DSP blocks are mapped to n mpDSPs and a single DSP48E1 primitive. In large graphs with multiple STs with odd numbers of DSP blocks, this can limit the benefits of multipumping.

We overcome this by further sharing these lone DSP blocks using an mpDSP. An additional pass searches for non-mpDSP blocks in different STs and pairs them into an mpDSP. Thus, without affecting the rest of the datapath (including pipeline balancing registers) and II, and without requiring any extra control logic, DSP blocks scheduled in different STs can also be multipumped and mapped onto mpDSPs.

VII. Integration into Tool Flow

We adapt the tool flow in [29] to generate synthesisable RTL for multipumped implementations. The tool has been modified to accept a computational kernel description in C rather than the previous proprietary input. LLVM passes are used to translate the input C file into a set of DOT files, which are parsed to generate a dataflow graph that is partitioned into subgraphs representing DSP block configurations. The partitioned graph, called the DDFG, comprises nodes that are either DSP48E1 primitive configurations or adders/subtractors to be implemented in FPGA logic. These adder nodes are those that cannot be merged with multipliers in the original graph to map to DSP block configurations. The tool also generates required balancing registers to align inputs to all nodes in the DDFG. The scheduling approaches described in Section VI are added to the flow, and the RTL generation stage is rewritten to use mpDSP blocks as described in Section V.

The DDFG is transformed into an multipumped DDFG (mpDDFG), with each node representing one of these three type of blocks: an mpDSP block, a DSP48E1 primitive, or a LUT based add/sub block. To achieve a DSP block reduction of a half, the number of DSP block nodes in each ST should be a multiple of two, though this is not always possible due to dependency constraints. For the BF approach, DSP nodes in the same ST with the same configuration are mapped to mpDSPs. For the SDC and FDS based techniques (Sections VI-B and VI-C), if the number of nodes scheduled in the DDFG is even $(2M)$, we utilise M mpDSP blocks. Ports corresponding to the nodes in the DDFG (up to $4 \times 2M$) are mapped to the corresponding $8 \times M$ ports of M mpDSPs. If an odd number of DSP blocks $(2M+1)$ is scheduled in a ST, $2M$ DSP blocks are mapped to M mpDSPs in the mpDDFG, and the remaining DDFG node is mapped directly to a DSP block with the correct configuration running at the system clock frequency. Although the single DDFG node can also be mapped to a mpDSP with input set I_2 left unconnected, mapping the node directly to a DSP48E1 primitive saves on the extra circuitry required in a mpDSP. Finally, lone DSP blocks scheduled in different schedule times are combined into mpDSPs as described in Section VI-E.

From the mpDDFG, Verilog RTL instantiations of the mpDSP blocks, DSP48E1 primitives, and adder blocks are generated along with pipeline balancing registers. For mpDSP blocks, the two configurations are passed as parameters (which can be the same) for each positive and negative edge of the system clock, as discussed in Section V. If the two multipumped configurations are identical, the configuration is hard-wired.

TABLE II: Graph nodes I/O and operations.

Graph	Inputs	Outputs	Adders/Subs	Muls
Chebyshev	1	1	2	3
Mibench2	3	1	8	6
FIR2	17	1	15	8
SG Filter	2	1	6	6
Horner Bezier	12	4	6	8
Poly1	2	1	5	4
Poly2	2	1	3	5
Poly3	6	1	4	6
Poly4	5	1	3	3
Poly5	3	1	14	11
Poly6	3	1	19	23
Poly7	3	1	18	17
Poly8	3	1	16	15
Quad Spline	7	1	4	13
ARF	26	2	12	16
EWF	21	5	26	8
Motion Vector	25	4	12	12
Smooth Triangle	29	14	20	17

VIII. Experiments and Analysis

To explore the effectiveness of the proposed methods for multipumping, we implemented a number of multiply-add flow graphs. These include the Mibench2 filter, quadratic spline, and Savitzky-Golay filter from [30]; the ARF, EWF, Horner Bézier, motion vector, and smooth triangle extracted from MediaBench [31]; and eight polynomials of varied complexity from the Polynomial Test Suite [32]. Table II shows the number of inputs, outputs, and number of each type of operation for each of the benchmarks. All the implementations target the Virtex 6 XC6VLX240T-1 FPGA found on the ML605 development board, using Xilinx ISE 14.6 and Xilinx Vivado HLS 2013.4 tools on an Intel Xeon E5-2695 running at 2.4 GHz with 16 GB RAM.

A. Resource Usage and Frequency

Multipumping results in a tradeoff between DSP block and LUT usage. As DSP blocks and LUTs cannot be compared directly, and to understand overall resource usage, we compare the area in terms of equivalent LUTs, $LUT_{eqv} = nLUT + nDSP \times (196)$, where 196 is the ratio of the number of LUTs (150720) to the number of DSP blocks (768) available on

TABLE III: Resource usage and maximum frequency for mpDSP block. (PA: Pre-adder sub-block; Freq in MHz)

Sub-blocks	DSPs	LUTs	Eq LUTs	Reg	Freq
Mul	1	45	241	51	235
PA-Mul	1	70	266	51	230
Mul-ALU	1	69	265	147	227
PA-Mul-ALU	1	102	298	147	229

the target device used. This gives a proxy for overall area consumption.

Table III shows the resource usage and maximum frequency for a single mpDSP block which does not utilise dynamic reconfiguration, for different combinations of sub-blocks used. The number of LUTs required increases as we use more sub-blocks since more inputs need to be multiplexed. When DSP blocks with different configurations are mapped onto a mpDSP, a 17-bit multiplexer is required for switching between configurations (as shown in Fig. 4), consuming a maximum of up to 17 extra LUTs. This multiplexer is optimised away if the configurations of both DSP operations are the same. Registers for holding intermediate outputs and the outputs of both the operations are the same for all four combinations. However, configurations for which the ALU sub-block is used, require two extra 48-bit registers to balance the C input of the DSP block primitive.

As discussed above, a DSP48E1 primitive is equivalent to 196 LUTs in logic. Even after considering the extra 17 LUTs required to select configurations, the number of extra LUTs required by the mpDSP is always far fewer than the LUT_{eqv} of a DSP block (up to $102+17$), and thus multipumping represents an overall area saving. However, there remains a register utilisation overhead, due to balancing the internal stages and intermediate output storage. The maximum frequency achieved by all the configurations remains largely the same.

We compare four different scenarios to understand the effect of multipumping on resource utilisation and frequency. The first (*Original*) does not use multipumping but maps efficiently to fully pipelined DSP blocks, as discussed in [29]. The second (*MulOnlyMP*) multipumps only multipliers (similar to [2]). This gives us a baseline against which to compare our approach. Since only the multipliers are multipumped, all adders and subtracters are forced into the logic fabric. The third (*MP*) multipumps DSP blocks including all subblocks (similar to [6]), but with the improved scheduling approaches described in this paper. A pair of DSP blocks scheduled in the same ST, with the same configuration, i.e., implementing the same combination of operations, is implemented using a single mpDSP block. The fourth (*RTRMP*), exploits runtime programmability to allow DSP block nodes implementing different operations, including different sub-block usage, to be implemented using a single mpDSP block.

1) Baseline Multiplier Multipumping: BF scheduling is unable to generate schedules for five benchmarks (*FIR2*, *Poly6*, *Poly7*, *Poly8*, and *Smooth Triangle*). The high mobility of multiple nodes in these benchmarks results in a very large number of possible schedules, resulting in full memory utilisation on our test machine. For the remaining 13 benchmarks, compared to *Original*, *MulOnlyMP* reduces DSP block usage by 33–50%, averaging 41%, at a cost of increased LUT and register usage of 2.7× and 1.7× respectively. The significant increase in LUTs and registers is due to DSP blocks being used for multiplication only and all add/sub blocks being implemented in LUTs. Despite this significant increase, this still results in an average reduction in LUT_{eqv} of 13%, and achieves close to half the maximum frequency of *Original* (average 242 MHz).

TABLE IV: Comparing geometric mean of resource usage and frequency (normalised against *Original* implementation) across the 13 feasible benchmarks for brute-force, SDC-based, and FDS-based scheduling techniques. Freq in MHz.

	Scheduling	DSPs	LUTs	LUT_{eqv}	Regs	Freq
MulOnlyMP	BF	0.59	2.7	0.87	1.71	0.52
	SDC	0.59	2.61	0.87	1.67	0.53
	FDS	0.59	2.55	0.86	1.66	0.52
MP	BF	0.67	1.1	0.78	0.98	0.51
	SDC	0.79	0.94	0.85	0.86	0.52
	FDS	0.72	1.02	0.81	0.95	0.51

2) Fixed Function Multipumping: Considering *MulOnlyMP* as a baseline, *MP* utilises 15% more DSP blocks due to the limited possibilities for multipumping, as the DSP block configurations must agree. However, since full DSP blocks are multipumped, add/sub blocks are included, significantly reducing resource consumption. *MP* utilises 60% fewer LUTs and 43% fewer registers compared to *MulOnlyMP*, with average frequency improved by 1%. Compared to *Original*, *MP* results in a 33% reduction in DSP block usage with 9% more LUTs and 2% fewer registers, effectively saving 22% LUT_{eqv} area. This represents DSP block savings comparable to *MulOnlyMP* with significantly less impact on LUTs and registers.

Table IV shows the geometric mean of resource usage and maximum frequency for the 13 feasible benchmarks, for these three scenarios. For *MP*, we see that BF scheduling offers the best savings, but the SDC and FDS approaches can be applied to more complex benchmarks.

3) SDC-Based Flexible Multipumping: Table V shows resource utilisation and maximum frequency for all four scenarios, using SDC-based scheduling, normalised against *MulOnlyMP* implementations. Ideally, for a benchmark using n DSP blocks, multipumping can result in savings of up to $\lceil \frac{n}{2} \rceil$ DSP blocks. However, this is not always achievable due to node dependencies in the dataflow graph, and this is evident in Table V. Out of the 18 benchmarks, half of the benchmarks (*Mibench2*, *Horner Bezier*, *Poly2*, *Poly6*, *Quad Spline*, *ARF*, *EWF*, *Motion Vector*, and *Smooth Triangle*) do offer maximum DSP block reduction, while the other benchmarks save between 33–42% DSP blocks for the fully flexible *RTRMP* approach.

Table V also shows the geometric mean of normalised resource utilisation across benchmarks, for all four scenarios we have implemented. *MulOnlyMP* results in a 44% reduction

TABLE V: Resource usage and maximum frequency across all implementations, using SDC-based scheduling (normalized against *MulOnlyMP*). Freq in MHz.

Benchmarks	**Original**				**MulOnlyMP**				**MP**				**RTRMP**			
	DSPs	LUTs	Regs	Freq	DSPs	LUTs	Regs	Freq	DSPs	LUTs	Regs	Freq	DSPs	LUTs	Regs	Freq
Chebyshev (1)	1	0.78	1.06	1	1	1	1	1	1	0.22	0.27	0.62	1	0.22	0.27	0.6
Mibench2 (2)	2	0.33	0.46	2.08	1	1	1	1	1	0.53	0.68	1	1	0.53	0.68	1
FIR2 (3)	2	0.59	0.54	2.05	1	1	1	1	2	0.43	0.51	1.27	2	0.43	0.51	1.27
SG Filter (4)	1.5	0.21	0.45	2	1	1	1	1	1.25	0.17	0.3	1	1	0.27	0.44	0.98
Horner Bezier (5)	2	0.38	0.58	2.03	1	1	1	1	1.75	0.35	0.52	1.02	1	0.64	0.87	0.98
Poly1 (6)	2	0.42	0.67	2.03	1	1	1	1	1.5	0.28	0.59	1.01	1.5	0.28	0.59	1.01
Poly2 (7)	1.67	0.34	0.49	2	1	1	1	1	1.33	0.26	0.43	1	1	0.37	0.55	1
Poly3 (8)	1.5	0.38	0.6	2	1	1	1	1	1.25	0.42	0.49	1	1	0.56	0.73	1
Poly4 (9)	1.5	0.36	0.62	2	1	1	1	1	1.5	0.3	0.45	1.24	1.5	0.3	0.45	1.24
Poly5 (10)	1.71	0.25	0.45	2.07	1	1	1	1	1.71	0.17	0.27	1.28	1	0.35	0.68	0.99
Poly6 (11)	1.92	0.23	0.38	2.09	1	1	1	1	1.83	0.19	0.32	1.09	1	0.56	0.73	1.05
Poly7 (12)	1.89	0.26	0.42	2.16	1	1	1	1	1.89	0.16	0.33	1.31	1.11	0.47	0.67	1.04
Poly8 (13)	1.88	0.22	0.39	2.07	1	1	1	1	1.75	0.16	0.3	1.04	1.25	0.23	0.52	1
Quad Spline (14)	1.86	0.33	0.58	2.06	1	1	1	1	1.29	0.52	0.62	1.02	1	0.78	0.8	1.01
ARF (15)	2	0.49	0.65	2.08	1	1	1	1	1.75	0.49	0.7	1.04	1	0.84	1.07	1
EWF (16)	2	0.64	0.73	2.05	1	1	1	1	1.5	0.72	0.7	1.04	1	0.81	0.83	1.02
Motion Vector (17)	2	0.38	0.6	2.03	1	1	1	1	1	0.76	0.91	0.94	1	0.76	0.91	0.94
Smooth Triangle (18)	1.89	0.36	0.6	2.52	1	1	1	1	1.78	0.45	0.57	1.26	1	0.66	0.83	0.89
Geo Mean	1.77	0.36	0.55	1.99	1	1	1	1	1.47	0.32	0.47	1.05	1.11	0.46	0.64	0.99
Impv (%)					1	1	1	1	-47	68	53	5.2	-11	54	36	-0.9
LUT_{eqv} **Impv (%)**						1				1				14		

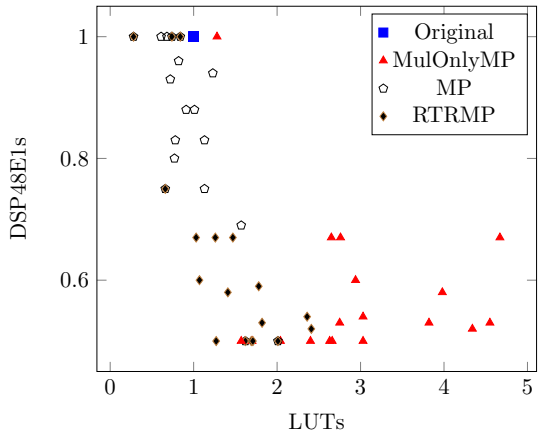

Fig. 6: DSP48E1-LUT usage trade-off for SDC-based scheduling.

in DSP utilisation compared to *Original*, however this is at the cost of a 2.8× and 1.8× increase in LUTs and Regs respectively. Note, however, that these values are for a computation kernel in a larger system, which can utilise many LUTs for the surrounding logic. Thus, the percentage increase in LUT usage for the full system may not be significant, as demonstrated in [2]. Despite the significant increase in LUTs, LUT_{eqv} is reduced by 13%. As expected, the frequency achieved using *MulOnlyMP* is in most cases close to half of *Original*.

Fig. 6 shows the tradeoff between relative DSP block and LUT usage for all variations of multipumping, for SDC-based scheduling. We normalise DSP48E1 and LUT count for each benchmark against the non-multipumped implementations. *MulOnlyMP* implementations have significantly increased LUT usage, compared to *MP* and *RTRMP*. This is due to the mapping of add/sub operations in the FPGA fabric since only the multipliers are multipumped. The LUT overheads for *MP* and *RTRMP* are significantly reduced, as full DSP block functionality is multipumped. For *MP*, DSP block usage is higher than *RTRMP* due to the limited opportunities for multipumping DSP blocks with identical configurations. We can also see that *RTRMP* tends to save more DSP blocks with a comparable LUT count to *MP*.

Compared to *MulOnlyMP*, *MP* utilises 47% more DSP blocks, however, as the sub-blocks of the DSPs are also utilised, it uses 68% fewer LUTs and 53% fewer Regs. *RTRMP* exploits both the sub-blocks and dynamic programmability of DSP blocks, thus multipumping the same number of DSP blocks as *MulOnlyMP* in most cases, with a significant reduction in LUTs and Regs of 54% and 36% respectively. Compared to *Original*, *RTRMP* results in a 37% reduction in DSP block usage, and a 27% and 17% increase in LUT and Register usage respectively, effectively saving 25% LUT_{eqv}.

4) FDS-Based Flexible Multipumping: Table VI shows the resource usage and maximum frequency across all benchmarks using FDS-based scheduling, normalised against *MulOnlyMP* implementations. Table VII compares the geometric mean of normalised results against the SDC-based approach. We see some slight improvements resulting from the better matching of DSP block configurations. As shown in Table VII, both SDC-based and FDS-based scheduling are not able to achieve DSP block reduction by half for *RTRMP* due to odd numbers of DSPs being scheduled in some STs. The additional resource sharing in Section VI-E overcomes this and is able to achieve a

TABLE VI: Resource usage and maximum frequency across all implementations, using FDS-based scheduling (normalized against *MulOnlyMP*). Freq in MHz.

B'mark	**MP**				**RTRMP**			
	DSPs	LUTs	Regs	Freq	DSPs	LUTs	Regs	Freq
1	1	0.19	0.27	0.62	1	0.19	0.27	0.62
2	1	0.58	0.71	0.99	1	0.58	0.71	0.99
3	2	0.38	0.51	1.27	2	0.38	0.51	1.27
4	1.25	0.24	0.35	1.03	1	0.29	0.47	1.03
5	1	0.62	0.85	0.98	1	0.62	0.85	0.98
6	1.5	0.36	0.59	1.03	1.5	0.36	0.59	1.03
7	1.33	0.26	0.43	1	1	0.41	0.55	1
8	1.25	0.42	0.52	1.04	1	0.54	0.74	1.03
9	1.5	0.3	0.46	1.24	1.5	0.3	0.46	1.24
10	1.25	0.2	0.35	1.02	0.88	0.46	0.69	1
11	1.36	0.23	0.4	0.96	0.93	0.38	0.71	1.04
12	1.4	0.28	0.45	1.15	1.1	0.4	0.7	1.07
13	1.3	0.2	0.37	1.05	1.1	0.17	0.46	1.04
14	1.43	0.54	0.67	1.03	1.14	0.72	0.86	0.99
15	1	0.71	1	0.99	1	0.71	1	0.99
16	1.5	0.7	0.7	1.08	1	0.77	0.82	0.82
17	1	0.51	0.91	1.09	1	0.51	0.91	1.09
18	1.67	0.41	0.53	1.21	1.11	0.62	0.75	1.15
Geo Mean	1.29	0.36	0.53	1.03	1.1	0.43	0.64	1.01
Impv (%)	-29	64	47	3.3	-10	57	36	1
LUT_{eqv} **Impv (%)**		7				15		

TABLE VII: Comparing geometric mean of resource usage and frequency (normalised against *MulOnlyMP* implementation) across all benchmarks, using SDC-based, and FDS-based scheduling techniques. Freq in MHz.

	Scheduling	DSPs	LUTs	LUT_{eqv}	Regs	Freq
MP	SDC	1.48	0.33	1	0.47	1
	FDS	1.3	0.36	0.92	0.52	1
RTRMP	SDC	1.11	0.47	0.86	0.64	0.98
	FDS	1.1	0.44	0.84	0.65	1

48% DSP block reduction for both SDC-based and FDS-based scheduling techniques. These savings are at a cost of 86% LUTs and 51% registers for SDC-based scheduling, effectively saving 29% LUT_{eqv} area. For FDS-based scheduling, as multipumping of DSP blocks with the same configurations is prioritised, LUTs and registers are marginally fewer (74% LUTs and 46% registers), with LUT_{eqv} area savings of 30%.

As discussed earlier, multipumping is feasible only if the throughput requirement of the full system is half of the maximum throughput supported by the embedded DSP blocks. Here, we are focused on the area efficient implementation of a computationally intensive inner loop of a larger system. As DSP48E1 primitives on the Xilinx Virtex 6 can run at a maximum frequency of 473 MHz (Table I), implementations with multipumping can achieve a maximum system clock frequency of up to 236 MHz (half the maximum DSP48E1 frequency), which is achieved by most of the multipumped designs. On more modern Virtex 7 devices where the DSP block can reach 700 MHz, this translates to a 300 MHz system clock which is above the maximum achievable frequency for most larger designs, as shown in Fig. 1.

B. Power Consumption

For multipumping implementations, DSP blocks run at double the frequency of the surrounding circuit, which can affect power consumption. We have evaluated this using Xilinx Power Analyzer on post-place-and-route designs for *Original* and SDC-based and FDS-based implementations combining multipumping and resource sharing (Section VI-E), over 10000 test inputs. We find that both SDC-based and FDS-based implementations show a modest increase in power of between 0.3% to 2.5% (average 1.1%) across the different benchmarks. This demonstrates that the higher frequency of operation of the mpDSP blocks is offset by the reduced numbers of DSP blocks used in the designs.

C. Tool Runtime

As shown in Table II, the size of the dataflow graphs for the benchmarks varies from 5 nodes to 42. The time taken to generate synthesisable RTL from the input C files, including graph partitioning and scheduling, for all techniques discussed in this paper is in a range of 10–20 ms.

The BF method is slower due to its complexity, up to 390 ms for *EWF*, and fails to run for the larger benchmarks. The SDC and FDS based scheduling approaches complete in just 3.7–31 ms across all benchmarks, which is reasonable for a small step of the design flow, considering that this includes all the intermediate steps of RTL generation.

IX. Conclusions

We have demonstrated the concept of multipumping applied to the flexible DSP blocks in modern Xilinx FPGAs. Since these blocks can run at significantly higher frequencies than most large designs, we can clock them at double the system clock, allowing them to be shared by two operations in a single system clock cycle. We showed how a BF method could be applied to fixed DSP block sharing but that it could not scale to flexible DSP blocks.

We proposed two scheduling techniques, one based on the SDC framework and another based on FDS, that could both generate multipumped schedules for flexible DSP blocks in deterministic time. With improved scheduling techniques and using dynamic programmability, we showed that multipumping can result in a reduction of DSP block usage by 37% and 35% and LUT_{eqv} area by 25% and 26% for SDC and FDS based scheduling respectively.

Finally, we presented an approach for improving savings further by sharing across schedule times through multipumping, resulting in DSP block reduction by 48%, effectively saving 30% LUT_{eqv} area compared to non-multipumped implementations. All these methods were integrated into a tool flow that generates multipumped implementations from a high level description with very short runtime.

References

[1] P. Maidee, N. Hakim, and K. Bazargan, "FPGA family composition and effects of specialized blocks," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2008, pp. 101–106.

[2] A. Canis, J. H. Anderson, and S. D. Brown, "Multi-pumping for resource reduction in FPGA high-level synthesis," in *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, March 2013, pp. 194–197.

[3] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, Feb 2010, pp. 41–50.

[4] X. Inc., *UG479: 7 Series DSP48E1 Slice User Guide*, 2013.

[5] L. Shannon, V. Cojocaru, C. N. Dao, and P. Leong, "Technology scaling in FPGAs: Trends in applications and architectures," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2015, pp. 1–8.

[6] B. Ronak and S. A. Fahmy, "Minimising DSP block usage through multi-pumping," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, Dec 2015, pp. 184–187.

[7] S. Raje and R. A. Bergamaschi, "Generalized resource sharing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1997, pp. 326–332.

[8] J. Cardoso, "Novel algorithm combining temporal partitioning and sharing of functional units," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, March 2001, pp. 31–40.

[9] S. Memik, G. Memik, R. Jafari, and E. Kursun, "Global resource sharing for synthesis of control data flow graphs on FPGAs," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, June 2003, pp. 604–609.

[10] W. Sun, M. Wirthlin, and S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," *Proceedings of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 26, no. 2, pp. 254–265, Feb 2007.

[11] R. Zhao, M. Tan, S. Dai, and Z. Zhang, "Area-efficient pipelining for FPGA-targeted high-level synthesis," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2015, pp. 157:1–157:6.

[12] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–10.

[13] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong, "Resource-aware throughput optimization for high-level synthesis," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2015, pp. 200–209.

[14] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2008, pp. 107–116.

[15] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012, pp. 111–114.

[16] Y. Hara-Azumi, T. Matsuba, H. Tomiyama, S. Honda, and H. Takada, "Impact of resource sharing and register retiming on area and performance of FPGA-based designs," *Proceedings of Information and Media Technologies*, vol. 9, no. 1, pp. 26–34, 2014.

[17] S. Davidson, D. Landskov, B. Shriver, and P. Mallett, "Some experiments in local microcode compaction for horizontal machines," *Proceedings of IEEE Transactions on Computers*, vol. C-30, pp. 460–477, July 1981.

[18] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 1987, pp. 195–202.

[19] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 433–438.

[20] H. Yantir, S. Bayar, and A. Yurdakul, "Efficient implementations of multi-pumped multi-port register files in FPGAs," in *Proceedings of Euromicro Conference on Digital System Design (DSD)*, Sept 2013, pp. 185–192.

[21] F. Anjam, S. Wong, and F. Nadeem, "A multiported register file with register renaming for configurable softcore VLIW processors," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, Dec 2010, pp. 403–408.

[22] C. E. Laforest, M. G. Liu, E. R. Rapati, and J. G. Steffan, "Multi-ported memories for FPGAs via XOR," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012, pp. 209–218.

[23] R. P. Tidwell, *XAPP706: Alpha Blending Two Data Streams Using a DSP48 DDR Technique*, Xilinx Inc, 2005.

[24] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based Processor/Accelerator systems," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 33–36.

[25] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, "The iDEA DSP block-based soft processor for FPGAs," *Proceedings of ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 19:1–19:23, Sep. 2014.

[26] A. K. Jain, D. Maskell, and S. A. Fahmy, "Throughput oriented FPGA overlays using DSP blocks," in *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, March 2016.

[27] J. Edmonds, "Paths, trees, and flowers," in *Classic Papers in Combinatorics*, ser. Modern Birkhuser Classics. Birkhuser Boston, 1987, pp. 361–379.

[28] "[Online] LP Solve 5.5," http://lpsolve.sourceforge.net/5.5/.

[29] B. Ronak and S. A. Fahmy, "Mapping for maximum performance on FPGA DSP blocks," *Proceedings of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, pp. 573–585, April 2016.

[30] S. Gopalakrishnan, P. Kalla, M. Meredith, and F. Enescu, "Finding linear building-blocks for RTL synthesis of polynomial datapaths with fixed-size bit-vectors," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2007, pp. 143–148.

[31] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of International Symposium on Microarchitecture*, Dec 1997, pp. 330–335.

[32] "[Online] Polynomial Test Suite," http://www-sop.inria.fr/saga/POL/.

Ronak Bajaj received the B.Tech. degree in electronics and communication engineering from International Institute of Information Technology-Hyderabad (IIIT-H), India, in 2010 and the Ph.D. degree from the School of Computer Engineering, Nanyang Technological University, Singapore, in 2016.

From 2010 to 2011, he worked as an intern at Xilinx Research Labs, India. Since 2016, he has been working as post doctoral research fellow at Nanyang Techological University, Singapore.

Suhaib A. Fahmy (M'01, SM'13) received the M.Eng. degree in information systems engineering and the Ph.D. degree in electrical and electronic engineering from Imperial College London, UK, in 2003 and 2007, respectively.

From 2007 to 2009, he was a Research Fellow at Trinity College Dublin, and a Visiting Research Engineer with Xilinx Research Labs, Dublin. From 2009 to 2015, he was an Assistant Professor at Nanyang Technological University, Singapore. Since 2015, he has been an Associate Professor with the School of Engineering at the University of Warwick, UK. His research interests include reconfigurable computing and FPGAs, accelerators in a broad range of applications, and networked embedded systems.

Dr. Fahmy was a recipient of the Best Paper Award at the IEEE Conference on Field Programmable Technology in 2012, the IBM Faculty Award in 2013, and is also a senior member of the ACM.