PRE-SYNTHESIS AREA ESTIMATION OF STREAMING VECTOR PROCESSOR-BASED RECONFIGURABLE ACCELERATORS*

Somsubhra Mondal
EECS Department
Northwestern University

Seda Ogrenci Memik
EECS Department
Northwestern University

Nikolaos Bellas
Embedded Systems Research Labs
Motorola Inc.

Abstract- One of the major challenges in automated synthesis of reconfigurable accelerators is to create efficient designs that conform to the resource capacity of the target device. This work concerns estimation of the hardware cost before actually attempting the synthesis of a streaming accelerator on reconfigurable logic. Specifically, our proposed framework tackles the problem of pre-synthesis estimation of data queuing cost and functional unit area cost, while incorporating the potential impact of resource constraints and different operator bitwidths on the final implementation. We present a probabilistic push-and-pull approach for register queue size estimation and a bitwidth aware functional unit area estimation of a streaming data flow graph. We evaluated our techniques using an industrial toolflow. For the register queue sizes our estimations are within the range of -14.4% to 12.4% on an average, for various resource constraints on a set of multimedia applications. The estimated area of the functional units is 13.6% higher on average than that of the synthesized designs.

1. INTRODUCTION

Synthesis targeting reconfigurable logic faces the challenge of creating designs that comply with the resource and storage capacity of the target device. Synthesis tools primarily optimize latency and/or throughput, and often push the utilization of the target device to its capacity, and thereby possibly leading to infeasible designs. For faster design closure and effective design space exploration it is helpful to have an early estimate of the resource requirements of a design. In this paper, we present an area cost estimation technique during automatic synthesis of reconfigurable accelerators for multimedia applications.

With the increasing popularity of portable devices, there is a growing demand for multimedia applications. These applications are computationally intensive and often streaming in nature. Reconfigurable logic is an effective medium for creating pipelined hardware as well as for exploiting parallelism. The Reconfigurable Streaming Vector Processor (RSVP™ II[1]) [1, 2] is a pipelined vector coprocessor architecture that has been implemented on reconfigurable fabric targeting multimedia applications. Such streaming accelerators employ a set of functional units (FU) and they utilize functional pipelining heavily where it is essential to register the inputs and outputs of FUs. This is because a FU has to retain its results from previous iterations (until they have been passed on to all consumers), while it is busy computing for successive iterations. In addition, for higher throughput, modulo scheduling [3, 4] is a widely used scheduling technique used for such applications. However, higher throughput of modulo scheduling is achieved at the cost of higher number of registers [5]. Therefore, register queues at the outputs of FUs is one of the major building blocks that enable communication between FUs. Finally, the third major component in this template is the multiplexer network enabling the routing of data into and among various FUs.

In this work, we present an early estimation tool to assess the hardware complexity of realizing such a template on reconfigurable logic for a given application. Early estimation, before actually attempting the costly synthesis and physical design tasks, is crucial for the following reasons. The reconfigurable accelerator will be customized by a design space exploration tool, where several kernels extracted from a complex application need to be evaluated for their potential speedup if implemented with this accelerator. This requires a fast comparison of expected hardware cost for numerous candidate kernels. Second, for each individual kernel, further dimensions need to be explored such as different resource constraints (i.e. different allocation of functional units).

We present an early cost estimation tool that provides effective means to quickly explore the design space during automated synthesis of streaming accelerators. Our techniques enable an accurate estimation of expected cost without going through the lengthy design cycle spanning behavioral synthesis and physical synthesis. Specifically, our proposed technique tackles the problem of pre-synthesis estimation of data queuing cost, FU area cost, and multiplexing cost while incorporating the potential impact of resource constraints and bitwidth variation of different functional units on the final implementation.

Our main contributions in this paper are as follows:

- We propose a queue size estimation technique for an unscheduled streaming data flow graph, and
- We propose a bitwidth aware functional unit area and multiplexer area estimation technique for an unscheduled streaming data flow graph.

The remainder of this paper is organized as follows. We overview related work in Section 2. In Section 3 we describe our pre-synthesis estimation paradigm. Section 4 describes the proposed pre-scheduling register queue estimation technique. The details of our proposed FU area estimation technique are presented in Section 5. In Section 6 we present our experimental methodology and results. Section 7 summarizes our conclusions.

2. RELATED WORK

A compile-time FPGA area estimation approach is proposed by Kulkarni et al. [6], where the compiler user is provided with feedback of how much space is used. Hardware compilers apply extensive transformations that exploit parallelism, and their area estimation approach takes into account such compiler optimizations. Brandolese et al. [7] presented a parametric area estimation method at System-C level for FPGA-based designs. Their goal is to reduce the effort of the area estimator to adapt to the changes in the EDA design environments. An area estimation of Look-Up Table (LUT) based designs is proposed by Hamed et al. [8], where VHDL is transformed into a Boolean network, and then upper and lower bounds on the number of required LUTs is estimated. Area, time, and power estimation methodology by Bilavarn et al. [9]

* An additional page has been used with permission from the program chair and in compliance with the submission guidelines, which allow up to two additional pages in addition to six pages.

[1] RSVP is a trademark of Motorola Inc. Other product or service names are the property of their respective owners.

converts a behavioral description in C to a hierarchical Control/Data Flow Graph (HCDFG). Area estimation from MATLAB code is presented in [10]. A macro-model based area estimation is presented by Jiang et al. [11]. Another high-level FPGA area estimation technique is proposed by Enzler et al. [12] targeted for telecommunication and multimedia applications. However, all these work primarily focus on the area of functional units only, and do not take into consideration the area of the register queues at the output of the functional units – a major building block for streaming accelerator architecture. Moreover, in most of these work, bitwidth of functional units are also not considered with the exceptions [10-12].

Moreno et al. [13] proposed a register estimation method for unscheduled data flow graphs. However, their estimation assumes register reuse, which is not the case for the architecture that we are targeting.

In this paper, we propose a unified framework for estimating both the register requirements and the functional unit area at the pre-synthesis stage. Prior work in area cost estimation for reconfigurable hardware generally assumes a one-on-one mapping of tasks from the intermediate representation (Data Flow Graph, Control Data Flow graph) to functional units [6, 8, 11]. Our work distinguishes itself in the fact that our estimation techniques can take a given resource constraint into account. Thereby, our estimation is sensitive to the impact of resource binding and resource sharing onto hardware cost. In addition, we provide additional estimation techniques to account for building blocks specific to streaming architectures, namely, the data buffers attached to functional units. To the best of our knowledge, this is the first of its kind unified and general estimation framework for the popular reconfigurable accelerator family of streaming accelerators.

3. ESTIMATION PARADIGM

In this section, we present our pre-synthesis estimation paradigm for streaming accelerators. First, we present a brief overview of streaming accelerators. We illustrate various components of such architecture, and discuss their significance in our framework.

3.1. Overview of Streaming Accelerators

Streaming applications are characterized by a high degree of spatial locality and rather poor temporal locality of data streams. Moreover, data access patterns are often known in advance which allows (pre)fetching of data streams ahead of computations. These distinctive features of streaming data are the key to an effective streaming vector architecture design. Such streaming applications are often represented as streaming data flow graphs (sDFG). A sDFG is a DFG where I/O and internal communication edges are data streams, and not just simple variables.

The RSVP™ II is an example to such a stream-oriented vector processing architecture that completely decouples data access and data processing. The PICO-N system [14, 15] automatically synthesizes similar non-programmable hardware accelerators. However, RSVP™ II is targeted at configurable platforms, specifically FPGAs. Figure 1 shows the template of the RSVP™ II architecture [16]. In this paper, we have used this architecture template and the industrial automated synthesis tool developed to generate accelerators based on this template as a reference of comparison for our early estimation tool.

The two main components of the RSVP™ II architecture are: (i) the streaming interface unit, and (ii) the datapath unit. The streaming interface consists of the Address Generation Unit (AGU), the Address Line and Bus Line buffers, and a Stream Queue. The major components of the datapath unit are the functional units, the associated multiplexers at their inputs, and the register queues at their outputs. The streaming interface unit communicates with the datapath unit via FIFO queues, and it is completely decoupled from the datapath unit. Efficient memory bandwidth usage by the streaming interface is ensured by prefetching vector streams from the system memory (or peripherals). In this paper, our estimation framework focuses at the datapath unit.

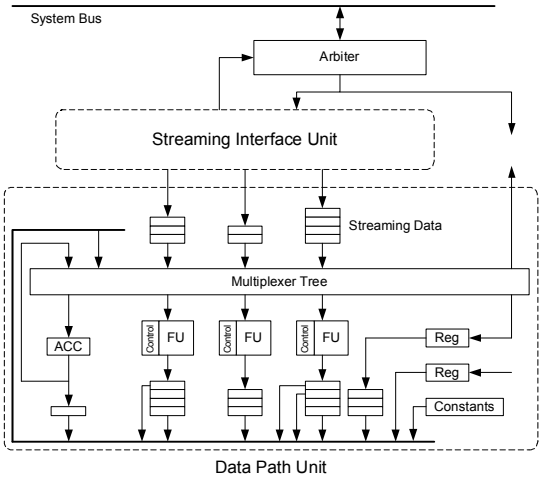

Figure 1. RSVP™ II Architecture

3.2. Methodology

Our pre-synthesis estimation framework consists of two steps: Given, an unscheduled sDFG, $G = (V, E)$, and a set of resource constraints R, our goal is to estimate, (i) the total number of registers in the output queues of all the functional units, and (ii) the area of the functional units and associated multiplexers.

4. REGISTER QUEUE SIZE ESTIMATION

Our register queue size estimation technique is a probabilistic approach at the pre-scheduling stage. The actual queue sizes at the outputs of FUs will depend on the scheduling and binding of operations. However, scheduling is a complex task, and to avoid scheduling for each and every possible solution in the design space we propose a fast queue size estimation method. Figure 3 shows our register queue estimation flowchart. In the following, we will discuss the steps in our queue estimation technique in detail.

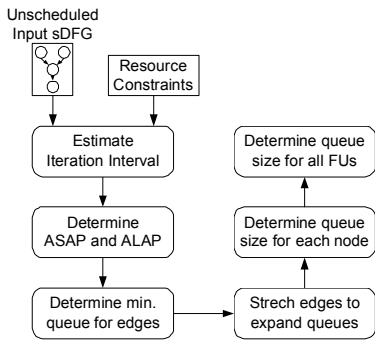

Figure 3. Queue size estimation flowchart.

4.1. Iteration Interval Estimation

The first step of our estimation scheme is to determine the iteration interval of a sDFG based on the resource constraints. The lower bound of the iteration interval is estimated based on the technique by Hwang et al. [17].

Let N_i be the number of operations of type i in the sDFG, which can be implemented using a functional unit of type i, and let M_i be the number of such functional units. Then the lower bound of the iteration interval, given t types of functional units, $ItIr$, is calculated as,

$$ItIr = \max_{1 \le i \le t}(\lceil N_i / M_i \rceil) \quad (1a)$$

However, in presence of cycles in the sDFG, iteration interval can be calculated as follows. Let an instance of operation op_i at iteration It_A be denoted by op_i @ It_A, and lat_i be its latency. Also, let deg_e be the associated *degree* of each edge $e(i, j)$ which is the number of iterations after which the result produced by op_i will be consumed by op_j. If there is a dependency cycle op_1 @ 1 → op_2 @ 1 → ... → op_n @ 1 → op_1 @ 2, the lower bound on $ItIr$ due to this cycle is $\Sigma_{1 \le i \le n}(lat_i)$. In a given sDFG, if there are cycles c_1, c_2, ..., c_k, then $ItIr$ will be given by,

$$ItIr = \max_{1 \le i \le k}(\lceil L_i / D_i \rceil) \quad (1b)$$

where, $L_i = \Sigma_{op_m \in c_i}(lat_m)$ and $D_i = \Sigma_{e_m \in c_i}(deg_m)$.

4.2. Estimation of Initial Lower Bounds for Queues

The next step is to determine the ASAP and ALAP schedules of the given sDFG. We have used the ASAP latency of the sDFG as the upper bound latency for the ALAP schedule. Note that computing the earliest and latest start times of operations (i.e. ASAP and ALAP schedules) is a significantly simpler task than actual resource constrained scheduling, which will take place during synthesis and will employ a much more complex optimizing heuristic to solve the intractable resource constrained scheduling. Let $ASAP(v)$ and $ALAP(v)$ be the ASAP and ALAP times of node $v \in V$. Once we have both the ASAP and ALAP schedules, we designate a lower bound on queue sizes to each edge of the sDFG,

$$Q^{edge}Min(i, j) = ASAP(j) - ALAP(i) - lat(i),$$

$$i, j \in V, e(i, j) \in E \quad (2a)$$

It may so happen that $ASAP(j)$ is actually less than $ALAP(i) + lat(i)$ which yields a negative queue size for edge $e(i, j)$. However, queue sizes cannot be negative, and there must be a queue at the output of every FU. Therefore, for such cases we have,

$$Q^{edge}Min(i, j) = 1, i, j \in V, e(i, j) \in E \quad (2b)$$

The lower bounds on the queue sizes assigned to each edge in this step are not very tight. Remainder of our efforts in queue size estimation is to further tighten these lower bounds using various novel steps as described in the following subsections.

4.3. Refinement of Queue Sizes of Edges

We aim to refine the lower bounds on queue sizes under the given resource constraints. Our main tool is based on the likelihood estimation that the source node may actually be producing data before its ALAP time, and likewise, the sink node may actually be consuming data after its ASAP time. The likelihood of the source and sink nodes of an edge being moved up and down respectively during the actual scheduling depends primarily on resource constraints of the design and criticality of the nodes. In addition, it will be affected by the heuristics that a particular scheduler applies to optimize the throughput by reducing the register or interconnect pressure. We propose a probabilistic ***push-and-pull*** approach, which estimates the amount by which a sink node is expected to be *pushed down* and the source node to be *pushed up* for an edge during scheduling. This will denote an increase in the initial lower bound of queue size assigned to each edge. For each edge $e(i, j) \in E$, we define Δi as the number of cycles by which node i is expected to be pulled up, and similarly, Δj as the number of cycles by which node j is estimated to be pushed down.

4.3.1. *Pull Force and Δi Computation*

Figure 4 shows the probabilistic *push-and-pull* queue expansion of an edge, where each node n is marked with a set of values, [$ASAP(n)$, $ALAP(n)$, $slack(n)$], and $slack(n)$ is given by $ALAP(n) - ASAP(n)$.

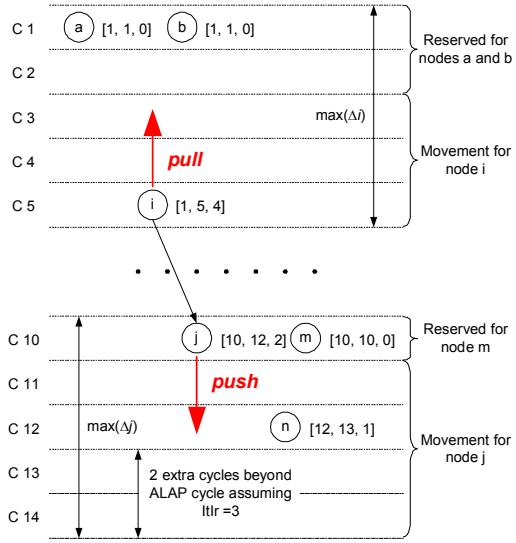

Figure 4. Probabilistic queue expansion by *push-and-pull*

Now let us consider node i in Figure 4. Node i can be pulled up by the scheduler because it may actually be scheduled earlier, therefore produce data before its ALAP time. Let $P(i)_k$ be the probability that node i is scheduled in cycle k. So, assuming that node i can be pulled up only up to $ASAP(i)$, we have,

$$\sum_{k=ASAP(i)}^{ALAP(i)} P(i)_k = 1 \text{, and } k < ASAP(i) \vee k > ALAP(i) \Rightarrow P(i)_k = 0$$

If the scheduler primarily tries to optimize latency, it will try to pull up node i as high as possible from its ALAP cycle. But due to the resource constraints, it can pull up node i only by a certain extent, depending upon the number of more critical nodes in the interval [$ASAP(i)$, $ALAP(i)$] and the number of similar operations in each cycle of the ALAP schedule in that interval. On the other hand, the scheduler will try to reduce the register burden by trying to schedule node i as close as possible to its ALAP cycle. These two counter-acting forces ultimately determine the total pull force exerted on node i.

Let $MoreCrit^{ALAP}(i)$ be the number of more critical nodes in the interval [$ASAP(i)$, $ALAP(i)$] of the ALAP schedule, and let $N(i)$ be the number of FUs available to implement the operation performed by node i, then, we define $MaxUp(i)$ as the cycle up to which node i can be pulled up from its ALAP cycle, and is given by,

$$MaxUp(i) = ASAP(i) + \lfloor MoreCrit^{ALAP}(i) / N(i) \rfloor \quad (4)$$

It can so happen that that $MaxUp(i)$ is greater than $ALAP(i)$ because of larger number of more critical nodes than node i and/or fewer available resources. In such cases we have, $MaxUp(i) = ALAP(i)$.

In Figure 4, assuming we have only one functional unit that can implement operations a, b, and i, we have $MaxUp(i)$ equal to 3, because nodes a and b are more critical than node i. In other words, node i cannot be pulled up any further above cycle 3. Therefore, both $P(i)_1$ and $P(i)_2$ equal 0, because our technique assumes that cycles 1 and 2 are reserved for nodes a and b.

Next, we will calculate $P(i)_k$ for each k within the interval $[MaxUp(i), ALAP(i)]$. The probability that node i is scheduled in cycle k depends on the contention for resources within cycle k. Also, since the scheduler will try to alleviate the register burden of each node, it will try to schedule it as close as possible to its ALAP cycle.

Let $N_k(i)$ be the number of operations of the same type as node i in cycle k of the ALAP schedule, then the pull force on node i by cycle k will be given by,

$$Pull(i)_k = (k - MaxUp(i) + 1) / N_k(i) \tag{5}$$

Based on the pull forces, we compute $P(i)_k$ as,

$$P(i)_k = \frac{Pull(i)_k}{\sum_{m=MaxUp(i)}^{ALAP(i)} Pull(i)_m} \tag{6}$$

We then compute the expected value of Δi, which is the number of cycles that node i is likely to be pulled up, based on $P(i)_k$ as,

$$E[\Delta i] = \sum_{m=0}^{ALAP(i)-MaxUp(i)} m \cdot P(i)_{ALAP(i)-m} \tag{7}$$

4.3.2. *Push Force and Δj Computation*

Now let us consider node j in Figure 4. Node j can be pushed down by the scheduler because it may actually be scheduled later therefore consume data after its ASAP time. Let $P(j)_k$ be the probability that node i is scheduled in cycle k. We assume that node j can be pulled down up to cycle $ALAP(j) + ItIr - 1$, we have,

$$\sum_{k=ASAP(j)}^{ALAP(j)+ItIr-1} P(j)_k = 1 \text{, and}$$

$$k < ASAP(j) \vee k > ALAP(j)+ItIr-1 \Rightarrow P(j)_k = 0$$

Let $MoreCrit^{ASAP}(j)$ be the number of more critical nodes in the interval $[ASAP(j), ALAP(j) + ItIr - 1]$ of the ASAP schedule, and let $N(j)$ be the number of FUs available to implement the operation performed by node j. We define $MinDown(j)$ as the cycle up to which node j is estimated to be pushed down from its ASAP cycle,

$$MinDown(j) = ASAP(j) + \lfloor MoreCrit^{ASAP}(j) / N(j) \rfloor \tag{8a}$$

It can so happen that $MinDown(j)$ is greater than $ALAP(j) + ItIr - 1$ because of larger number of more critical nodes than node j and/or fewer resources available. In such cases we have,

$$MinDown(j) = ALAP(j) + ItIr - 1 \tag{8b}$$

By similar arguments, for the push force, we calculate $P(j)_k$ for each k within the interval $[MinDown(j), ALAP(j) + ItIr - 1]$ in a similar manner.

$$Push(j)_k = (ALAP(j) + ItIr - k) / N_k(j) \tag{9}$$

$$P(j)_k = \frac{Push(j)_k}{\sum_{m=MinDown(j)}^{ALAP(j)+ItIr-1} Push(j)_m} \tag{10}$$

Finally, we compute the expected value of Δj, which is the number of cycles that node j is likely to be pushed down as,

$$E[\Delta j] = \sum_{m=MinDown(j)-ASAP(j)}^{ALAP(j)+ItIr-MinDown(j)} m \cdot P(j)_{ASAP(j)+m} \tag{11}$$

Now, the new expanded queue size for each edge $e(i, j)$ will be, $Q^{edge}Expand(i, j) = Q^{edge}Min(i, j) + E[\Delta i] + E[\Delta j]$

From the queue sizes of the edges we calculate the queue size of nodes. The expanded queue size of each node will simply be the maximum queue size of all outgoing edges of that node. Therefore, based on that we have,

$$Q^{node}Expand(i) = \max\{Q^{edge}Expand(i, j) : e(i, j) \in E\} \tag{12}$$

Now, that we have $Q^{node}Expand(i)$, because of the iteration interval ($ItIr$) this will be reduced by a factor of $ItIr$. If $ItIr$ equals 1, i.e. a new iteration starts every cycle, the queue sizes are at their maximum. If $ItIr$ equals 2, then a new iteration starts every other cycle, and in that case the required queue size of all nodes are halved and so on. Therefore, the final queue size of a node i is given by,

$$Q^{node}(i) = Q^{node}Expand(i) / ItIr \tag{13}$$

4.4. Final Refinement of Queue Sizes of FUs by Resource Constraint Correction Factor

Because of resource constraints, more than one node of the sDFG may be mapped onto a single FU. Let t be the number of types of FUs, and there are M_1, M_2, ..., M_t FUs of each type. If the sDFG has N_1, N_2, ..., N_t nodes that are mapped onto these FUs, the average number of nodes mapped per FU of type i will be $N_i/M_i \ \forall \ 1 \le i \le t$. When multiple nodes are mapped onto a single FU, the queue at its output is also shared by the nodes.

With resource sharing, the queue size of an FU will be determined by the Q^{node} values of the nodes mapped onto that FU. Therefore, queue size of an FU has to be at least the maximum of Q^{node} values, and at most the sum of the Q^{node} values. However, there are two counteracting factors that determine the actual queue size of an FU. First, with fewer resources, more nodes are mapped onto a particular FU, and this tends to reduce the queue size at the output of each FU. Second, as resources become scarce, there will be more contention for resources, and hence the result produced by a node has to be in the queue for a longer duration before it can be consumed. This denotes an increase in the queue size of an FU. Based on these observations, we experimented with various resource constraints to determine a *Resource Constraint Correction Factor* (*RCCF*) for type t of FU, given by,

$$RCCF_t = \frac{1}{\ln(\lfloor N_t / M_t \rfloor + e)} \tag{14}$$

The natural logarithmic base e is introduced in Equation (14) to make sure that *RCCF* is at most 1, which is the case when there are more resources available than operations.

Since our estimation is at a pre-scheduling stage, it is not known how the nodes are distributed among the FUs. Therefore, we estimate Q^{Total}, which is the total queue size of all FUs. The total queue size of all FUs of a particular type is determined by the sum of all nodes of that type

multiplied by the *RCCF* of that type. Finally, the total queue size for all FUs is simply the sum of queue sizes of all such types of FUs. Therefore Q^{Total} is given by,

$$Q^{Total} = \sum_{i=1}^{t}(RCCF_i \cdot \sum_{j=1}^{N_i} Q^{node}(j)) \quad (15)$$

5. BITWIDTH SENSITIVE FUNCTIONAL UNIT AREA ESTIMATION

The pre-synthesis area estimation technique presented in this paper performs a fast area estimation of a streaming kernel. Similar to the queue size estimation technique, the input to the FU area estimation is also a sDFG and a set of resource constraints. In addition, the input to the tool will also include knowledge of resource usage of operations mapped onto FUs. Figure 5 shows our FU area estimation methodology.

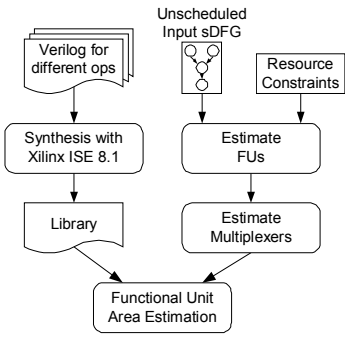

Figure 5. Functional Unit area estimation.

5.1. Library Creation

The first step of our functional unit area estimation is to create a library that contains the area cost information of different operations. The entries in the library file are obtained by synthesizing designs that only use that particular resource for which we are trying to estimate the area. For example, to estimate the area of a FU that performs a `vadd` operation (basic add operation in RSVP™ II Instruction Set Architecture), we synthesize a hardware description of the adder block (and associated registers) using Xilinx ISE 8.1. This is done for various bitwidths of the `vadd` operation.

Although each operation is synthesized for different bitwidths (8, 16, 24, and 32) of the inputs, synthesizing each operation for all possible bitwidths will require substantial amount of time and resources. Therefore, whenever the estimation tool does not find a corresponding entry in the library, it inter/extrapolates the resource usage. For example if the library has entries for 16-bit and 24-bit `vadd` operation, but an estimate is needed for 20-bit, the tool interpolates the number of required FPGA slices from the 16-bit and 24-bit `vadd` entries.

The estimation of multiplexer area is more complicated because the number of FPGA slices depends not only on the bitwidth and the number of inputs, but also on the number of *don't care* inputs (*no-ops*) and their relative positions. Therefore synthesizing multiplexers to create library entries with all these variations is time consuming and inefficient. Hence, to estimate multiplexer area, we resorted to a standard curve-fitting technique. The number of FPGA slices for multiplexers is generated by a second degree least square curve fitting function based on the input size, bitwidth, and the number of *don't care* inputs.

5.2. Estimation of Functional Units

In the first phase of our functional unit area estimation, we estimate the iteration interval *ItIr* of the given sDFG based on the given resource constraints, as described in Section 4.1. Once we estimate the iteration interval of the given sDFG we need to estimate the number of FUs of each type that are required. Let N_i be the number of operations of type i in the sDFG, which can be implemented using a functional unit of type i. Also, let there be t types of functional units, and M_i be the number of available functional units of type i as specified by the resource constraints. Then the estimated number of functional units of type i, m_i, is given by,

$$m_i = \lceil N_i/ItIr \rceil \quad \forall i : 1 \le i \le t \Rightarrow m_i \le M_i \quad (16)$$

Once we estimate the number of required functional units, the sDFG is scanned to gather information about what operations are performed and their bitwidths. Note that the input sDFG is annotated with required bitwidths of individual operations. In many synthesis flows bidwith allocation is performed first to achieve minimal operator bitwidths for a required computation accuracy. Such an optimization pass would already be applied to the input sDFGs before starting our estimation phase. We then categorize operations into a small number of subsets based on their bitwidths. For instance, we categorize operations into four groups in terms of their bitwidths (1-8 bits, 9-16 bits, 17-24 bits, and 25-32 bits). Next, we estimate what sizes of functional units are likely to be used by the scheduler to map these operations from these categories. The estimated number of functional units, m_i, is distributed proportionately among each such bitwidth group. The rationale behind this is to emulate the bitwidth-aware optimization that will be performed by the actual scheduler. When mapping a sDFG onto reconfigurable logic, the scheduler takes advantage of the fact that on reconfigurable logic bitwidths of individual functional units can be customized in order to create an area efficient design. To achieve this, the scheduler will try to group operations of similar bitwidths together and assign them to the same functional unit.

In the RSVP™ II architecture, a single type of FU can implement several types of operations. Each of these operations will have different resource requirements when mapped to a FU, depending on their complexity. However, at the pre-scheduling stage it is not known which particular FU will perform which set of compatible operations. Therefore, in order to make an area estimation of a particular type of FU, we use the maximum area usage among all operations of a sDFG that can be mapped onto that FU. Let FU_t be a functional unit of type t that implements operations op_{t1}, op_{t2}, ..., op_{tn} of a particular bitwidth group, and their respective area usage be $Area^{op(t1)}$, $Area^{op(t2)}$, ..., $Area^{op(tn)}$. Therefore, the estimated area of such a functional unit, $Area^{FU(t)}$, in terms of FPGA slices, is given by,

$$Area^{FU(t)} = \max_{1 \le i \le n}(Area^{op(ti)}) \quad (17)$$

Table I. Resource requirements for different operations

Operation	*# inputs*	*# slices*
`vabs`	1	16
`vadd`	2	9
`vneg`	1	8
`vsub`	2	24

Table I shows the four operations that are implemented by the *ALU* FU. The number of slices depends on the number of inputs of the operation and their bitwidth (16-bit in this case). As seen from Table I, operation `vneg` requires only 8 slices, whereas the `vsub` operation requires 24 slices. In such cases, if a FU implements more than one such operation, for our estimation we choose the one with the largest number of slices. For example, if a FU

implements `vadd` and `vabs` operations, we estimate the area of that FU as 16 slices.

5.3. Estimation of Multiplexers

Once we have estimated the number of different functional units of each type, we need to estimate the number and size of the multiplexers at the inputs of those functional units. In addition, we will also need a multiplexer for each functional unit to choose between the different operations it performs in different cycles. Therefore, for an n-input functional unit, we need n+1 multiplexers. The bitwidth of the multiplexers at the inputs of functional units will be determined from the bitwidth specification of the operations as specified in the sDFG.

The number of inputs of a multiplexer will be equal to the iteration interval because in each cycle of the iteration interval the multiplexer has to choose the appropriate input from a set of available signals. So the number of inputs of the multiplexers, $Mux^{\#IP} = ItIr$, such that the number of selector bits for the multiplexer $Mux^{\#SelBits} = \lceil \log_2(ItIr) \rceil$.

As stated in Section 5.1 the area of multiplexers in our estimation is based on three parameters: (i) Number of inputs, (ii) Bitwidth of inputs, and (iii) Number of no-ops. Now, since we do not know which operations are mapped to which functional units, we assume uniform distribution of operations among FUs. Therefore for N_t operations of type t, and m_t FUs of type t, we assume that each FU implements N_t/m_t operations. So, the number of no-ops among its inputs $Mux^{\#NoOps} = Mux^{\#IP} - \lfloor N_t/m_t \rfloor$.

Finally, if $Mux^{OpndBW(t)}$ is the bitwidth of a multiplexer, the area of a multiplexer at the inputs of a functional unit of type t is estimated as a function f of these three parameters, as,

$$Area^{OpndMUX(t)} = f(Mux^{\#IP(t)}, Mux^{OpndBW(t)}, Mux^{\#NoOps(t)}) \quad (18)$$

and the bitwidth of the multiplexer that chooses between the operations the FU performs is given by,

$$Mux^{OpsBW(t)} = \lceil \log_2(N_t/m_t) \rceil \quad (19)$$

Therefore, the area of the multiplexers for choosing the operations performed by the functional units is given by,

$$Area^{OpsMUX(t)} = f(Mux^{\#IP(t)}, Mux^{OpsBW(t)}, 0) \quad (20)$$

This function f is a second order curve fitting equation based on a set of multiplexer area determined by synthesis with Xilinx ISE 8.1.

Finally, the total area of the functional units and the multiplexers will be given by $Area^{DataPath}$ as,

$$\sum_{i=1}^{t} m_i (Area^{FU(i)} + n \times Area^{OpndMUX(i)} + Area^{OpsMUX(i)}) \quad (21)$$

Table II. Queue size estimation results

Application	*# nodes*	*RC Set 1*			*RC Set 2*			*RC Set 3*			*RC Set 4*		
		Synthesized	*Estimated*	*% Error*	*Synthesized*	*Estimated*	*% Error*	*Synthesized*	*Estimated*	*% Error*	*Synthesized*	*Estimated*	*% Error*
dctCol	85	103	152	47.6	103	152	47.6	36	44	22.2	36	44	22.2
dctRow	95	111	168	51.4	71	168	136.6	41	49	19.5	41	49	19.5
hpf_med_cc	157	404	347	-14.1	263	194	-26.2	76	91	19.7	77	85	10.4
lpf_gc_rgb	221	697	508	-27.1	394	293	-25.6	103	117	13.6	108	107	-0.9
lpr	67	150	110	-26.7	150	110	-26.7	58	67	15.5	39	42	7.7
open	30	71	47	-33.8	71	47	-33.8	44	34	-22.7	30	29	-3.3
quant	10	13	13	0.0	13	13	0.0	13	13	0.0	13	13	0.0
RsvpLPR	67	150	110	-26.7	150	110	-26.7	58	67	15.5	39	42	7.7
Average	**92.7**	**212.4**	**181.9**	**-14.4**	**151.9**	**135.9**	**-10.5**	**53.6**	**60.3**	**12.4**	**47.8**	**51.4**	**7.3**

5.4. Complexity

Finally, we would like to emphasize that the complexity of our estimation paradigm is largely insignificant in comparison to the complexity of the optimizing scheduler and the remainder of the behavioral and physical synthesis steps. Therefore, performing the abovementioned steps is justified in terms of computational cost as opposed to performing the actual synthesis steps. *More specifically, we observe that the combined run time of our estimation for all of the applications takes less than a few seconds. On the other hand, scheduling and synthesizing each application takes several minutes using an industrial automated front-end synthesis tool and Xilinx ISE 8.1 physical synthesis backend tools.*

6. EXPERIMENTAL RESULTS

The effectiveness of the proposed estimation paradigm is evaluated for a set of industrial applications from the multimedia domain. This benchmark suite includes various video and image compression and filtering algorithms (e.g. dctCol, hpf_med_cc, lpf_gc_rgb) as well as applications such as license plate recognition (RsvpLPR). We compare our estimates for both the steps in our framework with corresponding results generated by the toolflow proposed by Bellas et al. [16]. In the following, we present our experimental results for each of the estimation method.

6.1. Queue Size Estimation Results

We have evaluated our queue size estimation technique on a set of industrial multimedia applications. For each application we have chosen four different sets of resource constraints. We compare our results by the required queue sizes as determined by the modulo scheduler employed by an industrial automated synthesis tool [16]. Table II shows our results. In Table II, RC (Resource Constraint) Set 1 corresponds to unlimited resources, and the resources become scarce progressively across RC Set 2 through Set 4.

From Table II we observe that as the available resources decrease (From RC Set 1 to RC Set 4), the queue sizes also decrease, as discussed in Section 4.4. This is because when there are plenty of resources (e.g. RC Set 1), each node (operation) of the sDFG is assigned its own functional unit, and there is no sharing of the queue at the output of the functional unit. When resources are scarce (e.g. RC Set 4), the queue sizes are reduced substantially because of resource sharing.

Resource sharing can impact the queue sizes. However, this is decided by the scheduler and after our estimation. Therefore, since we are only interested in the overall register queue size of an application mapped onto an FPGA, and we have no knowledge of how resources are shared, our estimates are based on the average number of operations mapped per functional unit.

6.2. Functional Unit Area Estimation Results

For the functional unit area estimation, we have evaluated our results by comparing the area obtained by our estimation to that generated by the industrial hardware generator tool as described in [16]. The Verilog code generated is synthesized using Xilix ISE 8.1, and mapped onto Virtex 4 XC4VLX100 FPGA. Table III shows our estimation results. The results presented in Table III correspond to the same set of resource constraints as in RC Set 3 in Section 6.1, which is the default configuration of the RSVP™ II architecture.

Table III. Functional Unit Area Estimation

Application	*# Slices*		*% Error*
	Synthesized	*Estimated*	
dctCol	2175	2245	3.2
dctRow	1937	2375	22.6
hpf_med_cc	Over-mapped	3192	NA
lpf_gc_rgb	4324	5140	18.9
lpr	1520	1805	18.8
open	1132	1095	-3.3
quant	492	422	-14.2
RsvpLPR	1520	1805	18.8
Average	**1871.4**	**2126.7**	**13.6**

As seen from Table III, for the application hpf_med_cc the estimation error cannot be calculated because the design could not be synthesized on any Xilinx Virtex 4 device. The Virtex 4 XC4VLX100 has 960 user I/Os, which is the maximum I/O capability provided in this family of devices. However, this particular application requires more than 960 I/Os, and the synthesis process fails at the mapping stage as the device was overmapped. The I/O usage is one aspect that cannot be captured in our estimation framework in its current form, since we focus on area cost estimation. For the remainder of the applications, our estimation is accurate and with an average error of 13.6%. Previous work by Kulkarni et al. [6] reports results that are about 10% better than ours. However, their estimation is based on a 1-to-1 mapping of nodes of the DFG to resources. Our estimation is trying to capture a more general case by taking resource constraints and sharing into account.

7. CONCLUSIONS

In this paper we have presented a framework to provide early estimates of the implementation cost of reconfigurable streaming accelerators. Our estimation methodology has two steps: (1) A probabilistic push-and-pull approach to determine the register queue size at the outputs of functional units, (2) A library based approach to estimate the functional unit area incorporating bitwidth awareness. We evaluated our estimation results based on synthesized designs using an industrial template-based toolflow for a set of multimedia applications. For the register queue sizes our estimations are within the range of -14.4% to 12.4% on an average. The estimated area of the functional units is 13.6% higher on an average than that of the synthesized designs.

8. REFERENCES

[1] S. Chiricescu, et al., "The Reconfigurable Streaming Vector Processor (RSVP™)," International Symposium on Microarchitecture, 2003.

[2] S. Chiricescu, et al., "RSVP II: A Next Generation Automotive Vector Processor," 2005.

[3] B. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," International Symposium on Microarchitecture, 1994.

[4] K. Fan, et al., "Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System," International Symposium on Microarchitecture, 2005.

[5] A. Eichenberger, et al., "Minimum Register Requirements for a Modulo Schedule," International Symposium on Microarchitecture, 1994.

[6] D. Kulkarni, et al., "Fast Area Estimation to Support Compiler Optimizations in FPGA-based Reconfigurable Systems," IEEE Symposium on Field-Programmable Custom Computing Machines, 2002.

[7] C. Brandolese, et al., "An Area Estimation Methodology for FPGA Based Designs at SystemC-Level," Design Automation Conference, 2004.

[8] B. Hamed, et al., "Area Estimation of LUT bsed designs," International Conference on Computer Engineering & Systems 2004.

[9] S. Bilavarn, et al., "Area Time Power Estimation for FPGA Based Designs at a Behavioral Level," International Conference on Electronics, Circuits and Systems, 2000.

[10] A. Nayak, et al., "Accurate Area and Delay Estimators for FPGAs," Design, Automation and Test in Europe, 2002.

[11] T. Jiang, et al., "Macro-models for High Level Area and Power Estimation on FPGAs," Great Lakes Symposium on VLSI, 2004.

[12] R. Enzler, et al., "High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs," Int. Conf. on Field Programmable Logic and Applications, 2000.

[13] R. Moreno, et al., "Register estimation in unscheduled dataflow graphs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, pp. 396 - 403, 1996.

[14] S. Aditya, et al., "Automatic architecture synthesis of VLIW and EPIC processors," Int. Symp. on System Synthesis, 1999.

[15] R. Schreiber, et al., "High-Level Synthesis of Nonprogrammable Hardware Acclerators," *Journal of VLSI Signal Processing*, 2002.

[16] N. Bellas, et al., "FPGA Implementation of a License Plate Recognition Soc Using Automatically Generated Streaming Accelerators," Reconfigurable Architecture Workshop, 2006.

[17] C. Hwang, et al., "PLS: A scheduler for pipeline synthesis," *IEEE Transactions on CAD/ICAS*, vol. 12, pp. 1279-1286, 1993.