Presynthesis Area Estimation of Reconfigurable Streaming Accelerators

Seda Ogrenci Memik, *Senior Member, IEEE*, Nikolaos Bellas, *Member, IEEE*, and Somsubhra Mondal

***Abstract*—In this paper, we propose algorithms for presynthesis estimation of hardware cost of a streaming accelerator. Our proposed estimation method helps to accelerate the design-space-exploration phase by orders of magnitude by eliminating the need to perform logic and physical synthesis in each iteration. We present algorithms to perform early cost estimation of resources that are specific to a streaming accelerator, and we evaluate our techniques using an industrial tool flow and a set of streaming benchmarks. For the register-queue sizes, our estimations are in the range of 28%–9% of actual synthesis results on average, depending on the given resource constraints, while the datapath area estimations are within 14%. A typical estimation requires less than a minute, while generating the configuration bitstream of a streaming accelerator can take as much as 30 min according to our experiments. Considering several repetitions of the synthesis stage for the design space exploration, our estimation framework yields an order of magnitude speedup.**

***Index Terms*—Design automation, field-programmable gate arrays (FPGAs), high-level synthesis, reconfigurable architectures.**

I. Introduction

FIELD-PROGRAMMABLE gate-array (FPGA)-based accelerators are becoming increasingly attractive, particularly in response to the growing importance of performance-per-watt efficiency of multicore systems-on-chip (MCSoCs). Such systems can benefit from the extensive parallelism possible through the use of reconfigurable accelerator hardware. The streaming programming model is an emerging embedded domain in which an application can be viewed as a collection of independent kernel computations that communicate over explicit data channels [1]. Stream kernels exhibit large degrees of data and task level parallelism, with regular or even statically defined communication patterns. Therefore, the input and output data for each kernel can be prefetched and delivered from/to a stream register file instead of the main memory. This model decouples the datapath from the memory unit and enables extensive parallelism and heavy pipelining.

Processor architectures following this computation paradigm have been developed recently. Bellas *et al.* [2] have recently proposed an automated design flow to customize hardware accelerators which allows prototyping of streaming accelerators on FPGAs. The datapath of the streaming accelerators can be characterized by a template, consisting of a set of functional units (FUs). In order to achieve high throughput, modulo scheduling is used to exploit instruction level parallelism. In this template, register queues at the outputs of FUs is one of the major building blocks that enable communication between FUs and maintain intermediate results. The third major component in this template is the network of multiplexers enabling the routing of data among various FUs.

Automated design flows have proven effective for rapid prototyping of accelerators on reconfigurable logic. On one hand, the process of identifying the optimal configuration of this well-defined template presents a perfect opportunity for automation. On the other hand, this requires a design space search involving multiple iterations of the time-consuming logic and physical-synthesis stages.

In this paper, we propose presynthesis cost-estimation algorithms for the fundamental building blocks of the accelerator template. Our estimation techniques enable the designer to quickly assess the hardware cost of a certain template configuration without actually performing the time-consuming synthesis and physical-mapping steps.

The problem of automating the design flow for template-based streaming accelerators poses unique opportunities and challenges. The process of identifying the optimal configuration of this well-defined template presents a perfect opportunity for automation, as demonstrated by earlier work. 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. In addition, for each individual kernel, further dimensions need to be explored such as different resource constraints (RCs) (i.e., different allocation of FUs). Logic-synthesis and physical-design stages combined constitute an excessively lengthy process, and it may not be possible to synthesize all plausible solutions in the solution space in a reasonable amount of time. For example, if we target a Xilinx Virtex-II XC2VP2 FPGA for mapping streaming accelerators for two image-processing applications hpf_med_cc and lpf_gc_rgb (these applications are described in Section V), none of the applications

Manuscript received August 2, 2007; revised March 4, 2008 and July 8, 2008. Current version published October 22, 2008. This work was supported in part by a grant from the Alumnae of Northwestern University and in part by the National Science Foundation under Grant CNS-0546305. This paper was recommended by Associate Editor J. Lach.

S. O. Memik is with the Electrical Engineering and Computer Science Department, Northwestern University, Evanston, IL 60208-3118 USA (e-mail: seda@eecs.northwestern.edu).

N. Bellas was with the Embedded Imaging Systems Laboratory, Motorola Inc., Schaumburg, IL 60196-1079 USA. He is now with the Department of Computer and Communications Engineering, University of Thessaly, 38221 Volos, Greece.

S. Mondal was with the Electrical Engineering and Computer Science Department, Northwestern University, Evanston, IL 60208-3118 USA. He is now with Neokast Inc., Evanston, IL 60201 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2008.2006097

will fit in the device. This FPGA has only 1408 slices, while hpf_med_cc and lpf_gc_rgb require more than 4000 slices in the template configuration targeting highest bandwidth. In such cases, it is very useful to have an early estimate of the resources required because the synthesis of each of these applications starting from scheduling to logic synthesis and placement and routing can run in the range of a couple of hours. After synthesis fails, the designer has to reiterate the entire procedure to check whether the resource requirements are met. On the other hand, our proposed early cost estimation takes only a few minutes for these benchmarks. Thereby, if the estimated resources do not meet the requirements, the designer can simply change the user-constraints and reestimate again. When the estimate finally meets the capacity of the FPGA device, the designer can proceed with synthesis. Therefore, in our proposed framework, there is only one synthesis run as opposed to the multiple synthesis runs for the traditional approach.

Since our area estimation method precedes modulo scheduling, our methodology will abstract out any heuristics that a modulo scheduler may potentially be using, only keeping the fact that the schedule will optimize for throughput and will attempt to minimize the iteration interval (II). This postulate does not limit generality, since high throughput is the main objective in most streaming applications. The estimated area of the streaming accelerator is a random variable whose sample space is defined by the RCs set by the user and by the streaming-accelerator template.

The remainder of this paper is organized as follows. We overview related work in Section II. In Section III, we delineate the streaming-accelerator template architecture, and we present a brief overview of the CAD tool for streaming-accelerator synthesis. Section IV describes the proposed prescheduling register-queue estimation and the FU area-estimation techniques. In Section V, we present our experimental methodology and results. Section VI summarizes our conclusions.

II. Related Work

A compile-time FPGA area estimation approach is proposed by Kulkarni *et al.* [12], where the compiler user is provided with feedback on the area complexity. Hardware compilers apply extensive transformations that exploit parallelism, and their area estimation approach takes into account such compiler optimizations. Brandolese *et al.* [5] 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.* [9], where VHDL is transformed into a Boolean network, and then, upper and lower bounds on the number of required LUTs are estimated. Area, time, and power estimation methodologies by Bilavarn *et al.* [4] convert a behavioral description in C to a hierarchical control/data-flow graph (HCDFG). Area estimation from MATLAB code is presented in [15]. A macromodel-based area estimation is presented by Jiang *et al.* [11]. Another high-level FPGA area-estimation technique is proposed by Enzler *et al.* [8] targeted for telecommunication and multimedia applications. However, all these work primarily focus on the area of FUs only and do not take into consideration the area of the register queues at the output of the FUs—a major building block for streaming-accelerator architecture. Moreover, in most of these work, bitwidth of FUs are also not considered with some exceptions [8], [11], [15]. Moreno *et al.* [14] proposed a register-estimation method for unscheduled DFGs. Memory-unit estimation in addition to FU area has also been addressed for technologies other than FPGAs [7]. Finally, we have presented a queue-size-estimation algorithm for streaming accelerators in our previous work [13]. This paper only addressed the streaming queues of the accelerator architecture and did not provide any methods or results for other components. Furthermore, we did not evaluate the impact of the estimation on the design space exploration concerning the configuration of the template for a kernel.

In this paper, we propose a unified framework for estimating both the register requirements and the FU area at the presynthesis stage. Prior work in area cost estimation for reconfigurable hardware generally assumes a one-on-one mapping of tasks from the intermediate representation (DFG, control DFG) to FUs [9], [11], [12]. This paper distinguishes itself in the fact that our estimation techniques can take a given RC 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 FUs. We also evaluate the impact of our estimation technique on the runtime of the overall architectural exploration phase for the streaming accelerators.

III. Architectural Synthesis of Streaming Accelerators

In this section, we will present an overview of the stream-processing paradigm and the streaming-accelerator architecture. We will also describe the industrial synthesis tool flow, which automatically generates template-based streaming accelerators. Our estimation framework has been integrated into this flow.

A. Stream Processing

Stream processing exploits the "arithmetic intensity" [1] of very large scale integration technology by clustering execution units together and exposing data movement and staging to the programmer. Streaming applications are characterized by a high degree of spatial locality but rather poor temporal locality of data streams. Moreover, data access patterns in most streaming applications are often known in advance, which allows prefetching 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 DFGs (sDFGs), which are DFGs where I/O and internal communication edges are data streams and not just simple variables. The operations and the dependences among them are represented in just the same manner as a DFG representation. Each node in the sDFG represents basic arithmetic and logical operations, and the

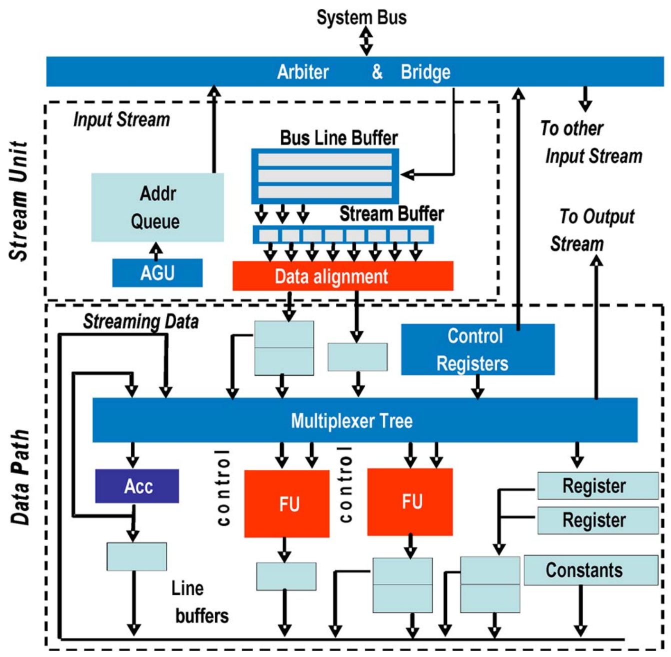


Fig. 1. Streaming accelerator template including the stream unit and the datapath.

directed edges represent the dependence of one operation on the output of a previous operation. Each node in the sDFG is denoted by a descriptor, which specifies the operation type, the precision of the output, and whether the operation is signed or unsigned.

Input data streams are represented by stream descriptors. A stream descriptor is comprised of a tuple (Type, Start_Address, Stride, Span0, Skip0, Span1, Skip1, and Size). The Stride, Span, Skip, and Type fields define the shape of the data stream in the main memory layout. Start_Address points to the location of the first data element in the memory. Each such stream passes through the Stream Unit and, after the final data alignment stage, gets presented at the input queues of FUs in the datapath.

Fig. 1 shows the template of the underlying streaming-accelerator architecture. In this paper, this architecture template and the industrial automated synthesis tool are used to generate accelerators as a reference of comparison for our early estimation tool [2].

The two main partitions of the architecture of Fig. 1 are as follows: 1) the streaming interface unit and 2) the datapath unit. The streaming interface consists of the address generation unit, the address-line and bus-line buffers, and a stream buffer. The major components of the datapath unit are the FUs, the associated multiplexers at their inputs, and the register queues at their outputs. Note that, in Fig. 1, the datapath contains two FUs for illustrative purposes. The datapath may contain an arbitrary number of FUs.

The stream unit transfers streams from a system memory, or peripheral, through a system bus and presents them in order to the accelerator. It also transfers processed output streams back to the memory. The stream buffer and the data-alignment unit store the incoming stream data and present them to the datapath in order. The number of storage elements, their size, and their interconnect depend on the stream shape and the requested bandwidth of the data. Efficient memory-bandwidth usage by the streaming interface is ensured by prefetching vector streams from the system memory (or peripherals).

Our estimation framework focuses at the datapath unit shown in Fig. 1. The area complexity of each stream unit does not change considerably across different sDFGs and can, therefore, be approximated as a constant regardless of the functionality of the sDFG.

Fig. 2 shows a sequence of representations of a quantization function which may be part of a video-compression algorithm like MPEG-4 encoding. As shown in Fig. 2(b), the input and output streams are denoted with the "vld" and "vstr" (load and store actions) operations, corresponding to nodes $v1$ and $v0$ in the sDFG. Internal operation nodes represent computation such as extracting the sign of a number, subtraction, multiplication, and arithmetic shift. In this particular example, if we assume that the architecture template will contain one ALU, one multiplier, and one shifter, the organization of these resources could be depicted as shown in Fig. 2(d). The ALU used for sign check, subtraction, and addition.

If the RC is given as one multiplier, one ALU, and one shifter, a mapping of this sDFG onto the streaming accelerator would be able to achieve an II of two cycles. The resulting implementation is shown in Fig. 2(d). Each FU is supported with an output queue of size one. The shared resources have multiplexers at their input ports.

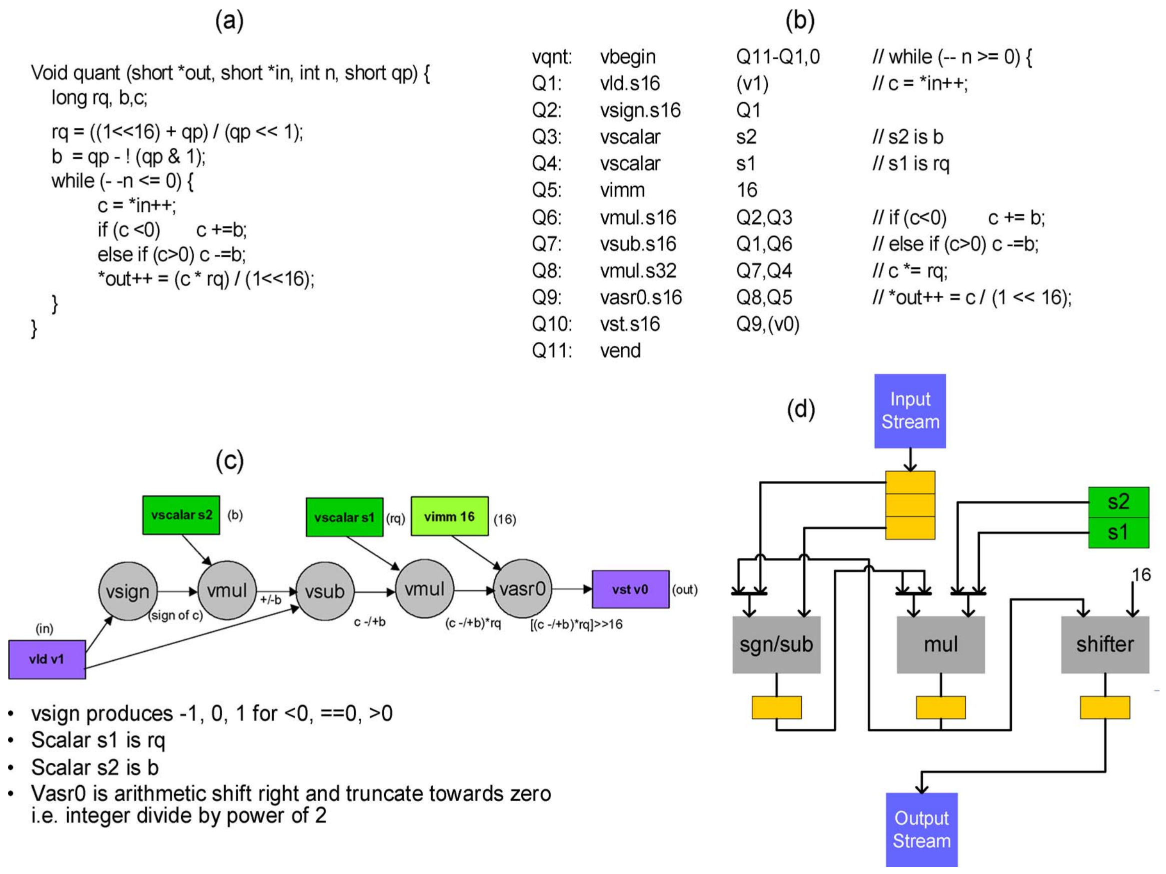
Our aim is to estimate the area complexity of the resulting accelerator shown in Fig. 2(d) (minus the I/O stream units), given the user RCs, without resorting in expensive synthesis and physical mapping in an FPGA fabric (for more information on the architecture of the streaming-accelerator architecture, the interested reader should consult prior literature [2], [6]).

B. *Tool Flow for Area Estimation*

The presynthesis area estimation procedure is an integral part of an automatic generation tool, Proteus [2], which produces synthesizable accelerators from the streaming representation (Fig. 3). The synthesis tool selects designs from a well-engineered framework, instead of generating the given hardware from a generic representation of a high-level language. The main points of the tool flow are the following:

1) a common template based on a simple data flow architecture that processes streaming data (Fig. 1);
2) an iteration engine that instantiates designs based on system parameters that meet system and user constraints to initiate the next iteration of space search;
3) a scheduler that performs sDFG scheduling and hardware;
4) an RTL constructor engine that produces optimized Verilog code for the datapath and the stream interface modules;
5) an evaluation phase that synthesizes the designs in an FPGA and produces quality metrics such as area and clock speed.

A set of DFG type of constructs are identified in the application algorithms, and these portions of the code are targeted



(a)

```
Void quant (short *out, short *in, int n, short qp) {
   long rq, b,c;
   rq = ((1<<16) + qp) / (qp << 1);
   b  = qp - ! (qp & 1);
   while (- -n <= 0) {
          c = *in++;
          if (c <0)       c +=b;
          else if (c>0) c -=b;
          *out++ = (c * rq) / (1<<16);
   }
}
```

(b)

```
vqnt:  vbegin     Q11-Q1,0   // while (-- n >= 0) {
Q1:    vld.s16    (v1)       // c = *in++;
Q2:    vsign.s16  Q1
Q3:    vscalar    s2         // s2 is b
Q4:    vscalar    s1         // s1 is rq
Q5:    vimm       16
Q6:    vmul.s16   Q2,Q3      // if (c<0)       c += b;
Q7:    vsub.s16   Q1,Q6      // else if (c>0) c -=b;
Q8:    vmul.s32   Q7,Q4      // c *= rq;
Q9:    vasr0.s16  Q8,Q5      // *out++ = c / (1 << 16);
Q10:   vst.s16    Q9,(v0)
Q11:   vend
```

- vsign produces -1, 0, 1 for <0, ==0, >0
- Scalar s1 is rq
- Scalar s2 is b
- Vasr0 is arithmetic shift right and truncate towards zero i.e. integer divide by power of 2

Fig. 2. C *quant* function is transformed to an sDFG and finally to a streaming accelerator. Note that only the code within the while loop is mapped into hardware. (a) C code. (b) sDFG text code. (c) sDFG graphical representation. (d) Resulting streaming accelerator. RCs: 1 ALU, 1 MUL, 1 shifter. II II $= 2$.

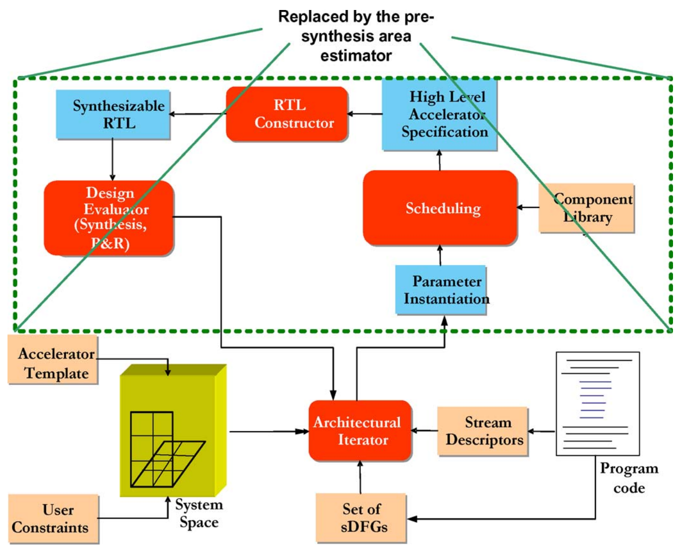


Fig. 3. Presynthesis area estimation as part of the architectural generation tool flow.

to run on the reconfigurable accelerator, while the rest of the code executes on a scalar processor. The DFGs are passed through the Proteus modulo scheduler along with resource- and system-bandwidth constraints. The output of the scheduler is a detailed hardware description of the datapath and the control path, which is then run through the hardware generator tool, which creates Verilog code. The design is then synthesized and mapped onto the reconfigurable fabric.

The area-estimation function is used to replace the upper part of Fig. 3, thus driving a faster architectural exploration closure. As shown in Figs. 1 and 2, the datapath consists of an interconnect of FUs, queue registers, and multiplexers. We will examine the contribution of each one of these elements to the area complexity of a streaming accelerator in the next section. Given an unscheduled sDFG, $G = (V, E)$, where V is the set of nodes and E is the set of data dependences between nodes, and a set of RCs R, our goal is to estimate the total number of registers in the output queues of all the FUs (Section IV-A) and the area of the FUs and associated multiplexers (Section IV-B).

IV. Presynthesis Area Estimation

In the following, we describe our proposed presynthesis cost-estimation algorithms for different building blocks of the streaming accelerator.

A. *Queue Registers Area Complexity*

Since the actual queue sizes at the outputs of FUs depend on the scheduling and binding of sDFG operations, the presynthesis area complexity is considered a random variable. Fig. 4 shows our register-queue-estimation flowchart. In the rest of

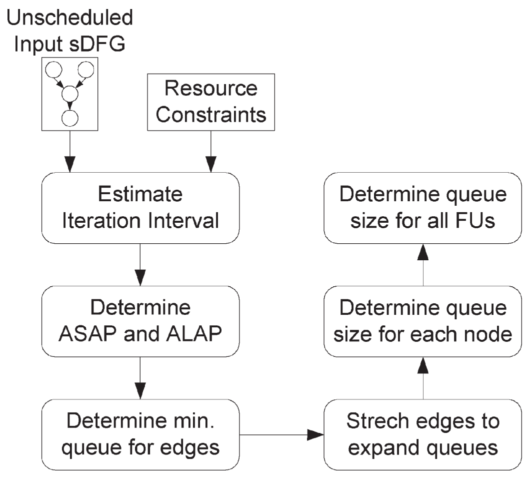

Unscheduled Input sDFG
Resource Constraints
Estimate Iteration Interval
Determine ASAP and ALAP
Determine min. queue for edges
Strech edges to expand queues
Determine queue size for each node
Determine queue size for all FUs

Fig. 4. Flow of the register-queue-size-estimation tool.

Section IV-A, we will discuss the steps in our queue-estimation technique in detail.

1) II Estimation: The first step of our estimation scheme is to determine the II of an sDFG based on the RCs defined by the user and the structure of the sDFG. One iteration of the sDFG corresponds to one iteration of the corresponding loop kernel in the behavioral specification. For the *quant* function shown in Fig. 2, this would correspond to one iteration of the while loop. The lower bound of the II is estimated based on the technique presented by Hwang *et al.* [9]. As aforementioned, we assume that any scheduler will be able to achieve this throughput, even if the particular distribution of operations within the interval may differ for each scheduler.

Let N_i be the number of operations in the sDFG that can be implemented using a FU of type i, l_i be the latency to start a new operation for each resource type i,[1] and M_i be the number of available FUs of type i. The lower bound of the II for an acyclic graph $\mathrm{II}_{\mathrm{ac}}$, given t types of FUs, is calculated as

$$\mathrm{II}_{\mathrm{ac}} = \max_{1\le i\le t}\left(\left\lceil \frac{l_i N_i}{M_i}\right\rceil\right). \tag{1}$$

In the presence of cycles in the sDFG, the II also depends on the maximum latency among all cycles in the graph. Let an instance of operation $\mathrm{op_i}$ at iteration $\mathrm{It_A}$ be denoted by $\mathrm{op_i}$ at $\mathrm{It_A}$ and l_i be its latency. In addition, let d_i be the associated weight label of each edge $e(\mathrm{op_i}, \mathrm{op_j})$, which is the number of iterations after which the result produced by $\mathrm{op_i}$ will be consumed by $\mathrm{op_j}$. In a given sDFG, if there are k such cycles $c_1, c_2, \ldots, c_k$, then $\mathrm{II}_{\mathrm{cyc}}$ will be given by

$$\mathrm{II}_{\mathrm{cyc}} = \max_{1\le i\le k}\left(\left\lceil \frac{L_i}{D_i}\right\rceil\right) \tag{2}$$

where

$$L_i = \sum_{\mathrm{op}_m\in c_i} l_m \quad D_i = \sum_{\mathrm{edge}_m\in c_i} d_m.$$

[1]Also known as initiation interval of a functional unit.

Essentially, L_i is the summation of operation latencies around a cycle c_i, and D_i is the summation of edge weights around the cycle c_i. Finally, the value of the II will be given by

$$\mathrm{II} = \max(\mathrm{II}_{\mathrm{ac}}, \mathrm{II}_{\mathrm{cyc}}). \tag{3}$$

Although this is the minimum attainable II $\mathrm{II}_{\mathrm{min}}$, we will be using this value with the understanding that any scheduler that optimizes throughput will be able to obtain such a schedule.

An example illustrating the computation of the II lower bound is shown in Fig. 5. There are two loop-carried dependences in this example: one having an iteration distance of two and the other having an iteration distance of one. Two cycles, c_1 and c_2, are formed due to these loop dependences. Assuming the latency of the nonpipelined multiplier to be two cycles and that of an adder to be one cycle, L_1 would be equal to $(1+1+2) = 4$. L_2 would be equal to $(2+1) = 3$. Similarly

$$D_1 = (0+0+2) = 2 \quad D_2 = (0+1) = 1.$$

As a result

$$\mathrm{II}_{\mathrm{cyc}} = \max\left(\left\lceil\frac{4}{2}\right\rceil, \left\lceil\frac{3}{1}\right\rceil\right) = 3.$$

The computation of $\mathrm{II}_{\mathrm{ac}}$would be as follows:

$$\mathrm{II}_{\mathrm{ac}} = \max\left(\left\lceil\frac{1\times 3}{2}\right\rceil, \left\lceil\frac{2\times 2}{2}\right\rceil\right) = \max(2,2) = 2.$$

As a result, the lower bound on the II would be determined as $\max\{\mathrm{II}_{\mathrm{ac}}, \mathrm{II}_{\mathrm{cyc}}\} = \max\{2,3\} = 3$. A possible schedule obeying the RCs and achieving this II is shown in Fig. 5(c).

2) Estimation of Initial Lower Bounds for Queues: The next step is to determine a lower bound for the number of the queues based only on the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules of the given sDFG (V, E). 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

$$Q_{\min}^{\mathrm{edge}}(e_{ij}) = \mathrm{ASAP}_j - \mathrm{ALAP}_i - l_i \tag{4}$$

where $i, j \in V$, and $e_{ij} \in E$. It may so happen that ASAP_j is actually less than $\mathrm{ALAP}_i + l_i$, which yields a negative queue size for edge e_{ij}. However, queue sizes cannot be negative, and there must be a queue at the output of every FU. Therefore, in general, we have

$$Q_{\min}^{\mathrm{edge}}(e_{ij}) = \max\left\{(\mathrm{ASAP}_j - \mathrm{ALAP}_i - l_i), 1\right\}. \tag{5}$$

The lower bounds on the queue sizes assigned to each edge in this step are not very tight. The remainder of our

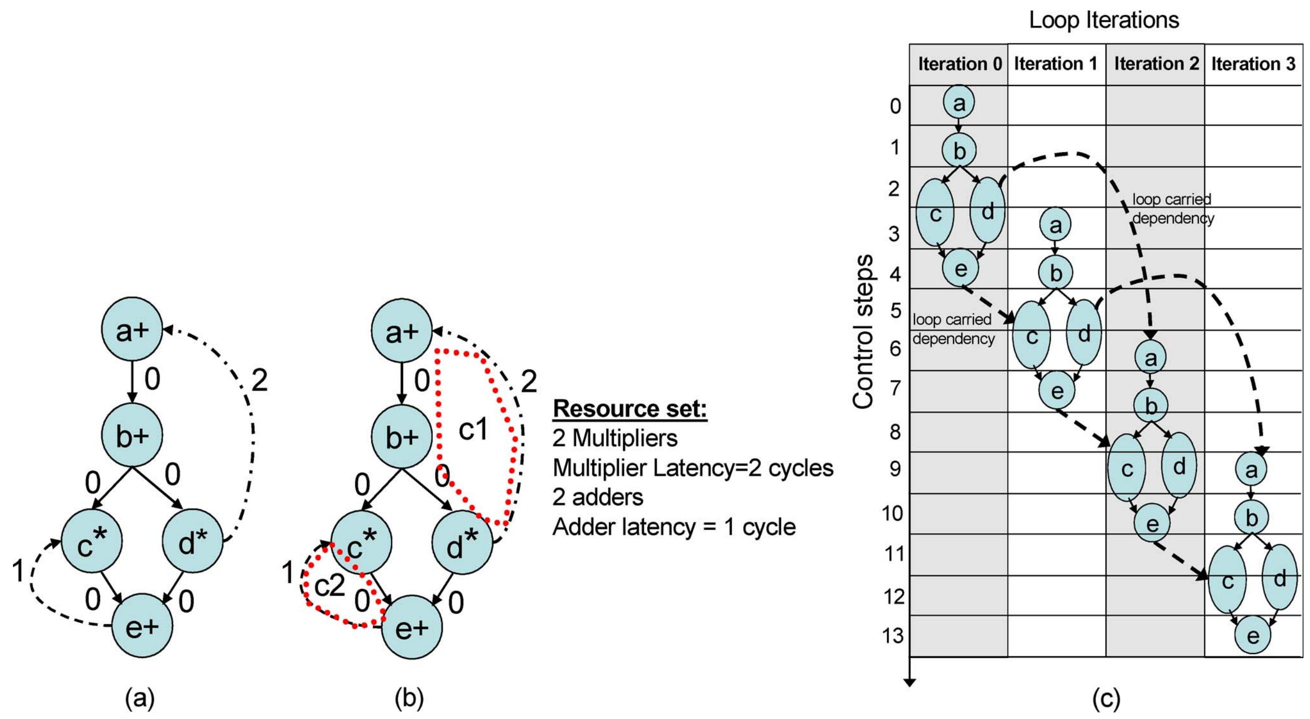


Fig. 5. (a) Cyclic sDFG, where operations a and d have a data dependence with an iteration distance of two iterations and operations c and e have a dependence for each iteration of the loop. The dotted loop edges are annotated with weights corresponding to these interiteration distances. (b) Edges (a, b), (b, d), and (d, a) form a cycle. The edges (c, e) and (e, c) form a second cycle in the graph. These cycles are denoted as c1 and c2. (c) For the RC of two multipliers and two adders, the cyclic graph can be scheduled with an II of three control steps.

efforts in queue-size estimation will be devoted to further tighten these lower bounds as described in the following sections.

3) Refinement of Queue Sizes of Edges: 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 RCs 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 pulled up for an edge during scheduling due to RCs.

This will denote an increase in the initial lower bound of queue size assigned to each edge. For each edge $e_{ij} \in E$, 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.

Pull margin and Δi computation: Fig. 6 shows the probabilistic push-and-pull queue expansion of an edge, where each node v is marked with a set of values $\{\text{ASAP}_v, \text{ALAP}_v, mobility_v\}$, and $mobility_v$ is given by $\{\text{ALAP}_v - \text{ASAP}_v\}$.

Now, let us consider node i shown in Fig. 6. Node i can be pulled up by the scheduler because it may actually be scheduled earlier, therefore, producing data before its ALAP time. Let $P_i(k)$ be the probability that node i is scheduled in cycle k.

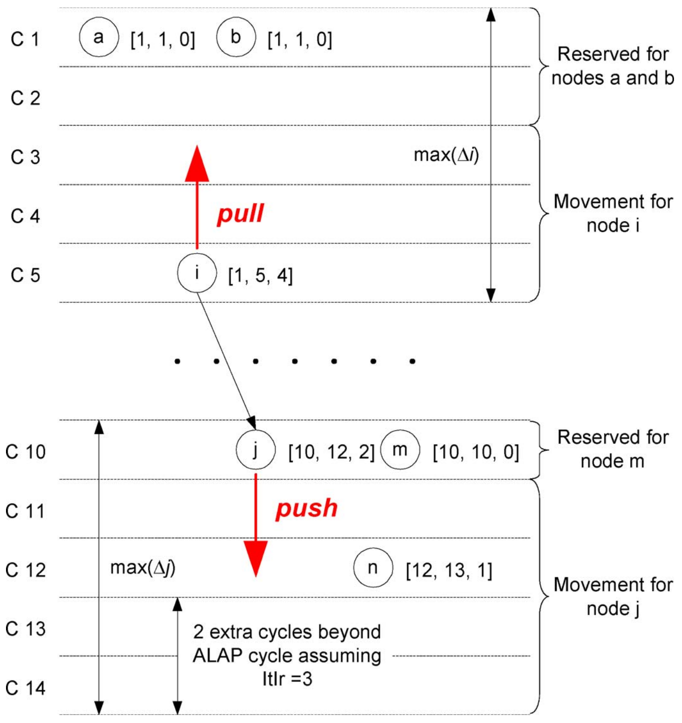


Fig. 6. Probabilistic queue expansion by push-and-pull.

Therefore, assuming that node i can be maximally pulled up only up to ASAP_i, we have

$$\sum_{k=\text{ASAP}_i}^{\text{ALAP}_i} P_i(k) = 1$$
$$P_i(k) = 0 \qquad \forall k : k < \text{ASAP}_i \vee k > \text{ALAP}_i.$$

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 RCs, it can pull up node i only by a certain extent, depending upon the number of more critical nodes in the interval 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 counteracting forces ultimately determine the total pull margin observed on node i.

Node i is more critical than a node j if $mobility_i < mobility_j$. Let R_i be the number of nodes that are more critical then node i in the interval $[\text{ASAP}_i, \text{ALAP}_i]$ of the ALAP schedule, and let M_i be the number of FUs available to implement the operation performed by node i; then, we define C_i as the cycle up to which node i can be pulled up from its ALAP cycle as follows:

$$C_i = \min\left\{\left(\text{ASAP}_i + \left\lfloor \frac{R_i}{M_i} \right\rfloor\right), \text{ALAP}_i\right\} \tag{6}$$

since node i cannot be scheduled beyond ALAP_i.

In Fig. 6, we have C_i equal to three, because nodes a and b are more critical than node i, assuming we have only one FU that can implement operations a, b, and i. Since node i cannot be pulled up any further above than cycle three, both $P_i(1)$ and $P_i(2)$ are equal to zero, because our technique assumes that cycles one and two are reserved for nodes a and b.

Note that a node with large mobility may be artificially constrained closer to its ALAP cycle even if it can normally be scheduled closer to its ASAP cycle, if there are enough open slots. We chose to simplify the determination of the available scheduling cycles by making this assumption, rather than having to consider the relative positions of all overlapping nodes.

Next, we will calculate $P_i(k)$ for each k within the interval $[C_i, \text{ALAP}_i]$. The probability that node i would be scheduled in cycle k depends on the contention for resources within cycle k.

Let $N_i(k)$ be the number of operations of the same type as node i in cycle k of the ALAP schedule, then the pull margin on node i by cycle k is defined as

$$\text{Pull}_i(k) = \frac{(k - C_i + 1)}{N_i(k) + 1}. \tag{7}$$

The pull toward earlier cycles tends to increase if the node i were to be scheduled closer to its ALAP value, since the number of available cycles in this case is larger. This means that there is a larger space above operation i, which would be sufficient to satisfy all resource contention caused by other competing (more critical) nodes. As a result, operation i may actually have a chance to be placed earlier in time in the actual schedule. The likelihood of an operation actually being scheduled earlier (i.e., being moved up in time) by the scheduler is small if we are considering the possibility of moving an operation from just one cycle below its ASAP time. On the other hand, the possibility would be higher, if we consider a pessimistically late (closer to the ASAP time) time for the same operation. This is captured by (7).

Using the concept of the $\text{Pull}_i(k)$ for each potential schedule step, we compute $P_i(k)$ as a summation of progressive likelihoods of observing displacement between the start times of what is predicted about an operation and what the actual scheduler may decide. This is shown in

$$P_i(k) = \frac{\text{Pull}_i(k)}{\sum\limits_{m=C_i}^{\text{ALAP}_i} \text{Pull}_i(m)}. \tag{8}$$

$P_i(k)$ essentially represents the likelihood of an operation to be placed in a particular control step by the actual resource-constrained scheduler. By using C_i, we further limit the range of such possible steps, which now falls between the interval defined by C_i and ALAP_i.

Finally, the expected value of Δi is computed as the expected number of cycles that node i will be pulled up, based on $P_i(k)$ as

$$E[\Delta i] = \sum_{m=1}^{\text{ALAP}_{i-C_i}} m P_i(\text{ALAP}_i - m). \tag{9}$$

The overall meaning of Δi is to provide an estimate for the impact of RCs and scheduling on the final timing of the operation start times. Δ_i represents symmetrically the second component of this consideration.

Push margin and Δj computation: Now, let us consider node j shown in Fig. 6. Node j can be pushed down by the scheduler due to RCs and, hence, consume data after its ASAP time. Let $P_j(k)$ be the probability that node j is scheduled in cycle k. We assume that node j can be pulled down by up to cycle $\text{ALAP}_i + \text{II} - 1$, and therefore

$$\sum_{k=\text{ASAP}_j}^{\text{ALAP}_j+\text{II}-'} P_j(k) = 1$$

$$P_j(k) = 0 \qquad \forall k : k < \text{ASAP}_j \vee k > \text{ALAP}_j + \text{II} - 1.$$

Let R_j be the number of nodes that are more critical then node j in the $[\text{ASAP}_j, \text{ALAP}_j + \text{II} - 1]$ interval of the ASAP schedule, and let M_j be the number of FUs available to implement the operation performed by node j. We define C_j as the cycle up to which node j is estimated to be delayed from its ASAP cycle

$$C_j = \min\left\{\left(\text{ASAP}_j + \left\lfloor \frac{R_j}{M_j} \right\rfloor\right), \text{ALAP}_j + \text{II} - 1\right\} \tag{10}$$

since node j cannot be scheduled after $\text{ALAP}_j + \text{II} - 1$.

By similar arguments, for the push margin, we calculate $P_j(k)$ for each k within the interval $[C_j, \text{ALAP}_j + \text{II} - 1]$ as

$$\text{Push}_j(k) = \frac{(\text{ALAP}_j + \text{II} - k)}{N_j(k) + 1} \tag{11}$$

$$P_j(k) = \frac{\text{Push}_j(k)}{\sum\limits_{m=C_j}^{\text{ALAP}_j+\text{II}-1} \text{Push}_j(m)}. \tag{12}$$

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

$$E[\Delta j] = \sum_{m=C_j-\mathrm{ASAP}_j}^{\mathrm{ALAP}_j+\mathrm{II}-1} mP_j(\mathrm{ASAP}_j + m).$$

The new expanded queue size for each edge e_{ij} will then be

$$Q^{\mathrm{edge}}(e_{ij}) = Q^{\mathrm{edge}}_{\min}(e_{ij}) + E[\Delta i] + E[\Delta j]. \tag{13}$$

From the queue sizes of the edges, we determine the queue size of nodes. The expanded queue size of each node will simply be the maximum queue size among all outgoing edges of that node, i.e.,

$$Q^{\mathrm{node}}_{\max}(n_i) = \max\left\{Q(e_{ij}) : \forall e_{ij} \in E\right\}.$$

The $Q^{\mathrm{node}}_{\max}(n_i)$ will be reduced by a factor of II, because of the effect of the II. If II equals one, i.e., a new iteration starts every cycle, the queue sizes are at their maximum. If II equals two, 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

$$Q^{\mathrm{node}}(n_i) = \left\lceil \frac{Q^{\mathrm{node}}_{\max}(n_i)}{\mathrm{II}} \right\rceil. \tag{14}$$

4) Final Refinement by RCCF: Because of RCs, more than one node of the sDFG may be mapped onto a single FU. Again, let N_i be the number of operations in the sDFG, that can be implemented using a FU of type i, and let M_i be the number of available FUs of type i. Then, the average number of nodes mapped in a FU of type i will be N_i/M_i. 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 a FU. First, with fewer available FUs, there will be less queues, because a queue can only be attached to the output of a FU. Second, as resources become scarce, there will be more contention for resources, and hence, the result produced by a node has to reside in the queue for a longer duration before it can be consumed. This denotes an increase in the queue size of a FU. Based on these observations, we experimented with various RCs to determine a resource-constraint correction factor (RCCF) for type t of FU, given by

$$\mathrm{RCCF}_t = \frac{1}{\ln\left(\left\lfloor \frac{N_t}{M_t} \right\rfloor + e\right)}. \tag{15}$$

The natural logarithmic base e is introduced in the equation earlier to make sure that RCCF is at most one, which is the case when there are more resources available than operations.

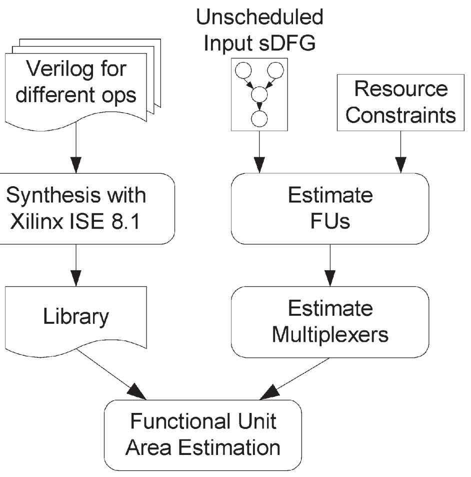

Unscheduled Input sDFG
Verilog for different ops
Resource Constraints
Synthesis with Xilinx ISE 8.1
Estimate FUs
Library
Estimate Multiplexers
Functional Unit Area Estimation

Fig. 7. FU area-estimation flowchart.

Since our estimation is at a prescheduling 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

$$Q^{\mathrm{total}} = \sum_{i=1}^{t} \left(\mathrm{RCCF}_i \sum_{j=1}^{N_i} Q^{\mathrm{node}}(j) \right). \tag{16}$$

B. FU Area Complexity

Similar to the queue-size-estimation technique, the input to the FU area estimation is also an sDFG and a set of RCs. Fig. 7 shows our FU area-estimation methodology. The library database contains information on the area complexity of a (relatively small) number of FUs with selected parameters. The library is used to interpolate the area complexity of FUs of arbitrary parameter settings based on the area of neighboring data points.

1) Library Creation: The first step of our FU 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 an add operation, we synthesize a hardware description of the adder block (and associated internal registers, if any) using Xilinx ISE 8.1 for various bitwidths.

Although each operation is synthesized for different bitwidths 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 interpolates the resource usage. For example, if the library has entries for 16- and 24-b add operation, but an estimate is needed for 20 b, the tool interpolates the number of required FPGA slices from the 16- and 24-b add 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 (nops). 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 least-square curve-fitting function based on the input size, bitwidth, and the number of nops. This method is described in more detail in the Appendix.

2) Estimation of FU Area Complexity: In the first phase of our FU area estimation, we estimate the II of the given sDFG based on the given RCs, as described in Section IV-A1. Then, we estimate the number of FUs of each type that is required. Let N_i and M_i have the same definitions as in the previous sections. Then, the estimated number of FUs m_i of type i is given by

$$m_i = \left\lceil \frac{N_i}{\text{II}} \right\rceil \qquad \forall i,\ 1 \le i \le t,\ m_i \le M_i. \tag{17}$$

Once we estimate the number of required FUs, 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, bitwidth allocation is performed first to achieve minimal operator bitwidths for the required computation accuracy. Such an optimization pass would already be applied to the input sDFGs before starting our estimation phase.

We categorize operations into a small number of subsets based on their bitwidths. For instance, we partition operations into four groups in terms of their bitwidths (1–8, 9–16, 17–24, and 25–32 b). Next, we estimate what sizes of FUs are likely to be used by the scheduler to map these operations from these categories. The estimated number of FUs 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 an sDFG onto reconfigurable logic, the scheduler takes advantage of the fact that on reconfigurable logic bitwidths of individual FUs 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 FU.

In our streaming 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 prescheduling 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 an sDFG that can be mapped onto that FU.

For example, Table I shows the four operations that are implemented by the ALU. The number of slices depends on the number of inputs of the operation and their bitwidths (16-b in this case). As seen from Table I, operation *vneg* requires only 8 slices, whereas the *vsub* operation requires 24 slices.

TABLE I
RESOURCE REQUIREMENTS FOR DIFFERENT OPERATIONS

Operation	# of inputs	FPGA slices
vabs	1	16
vadd	2	9
vneg	1	8
vsub	2	24

In cases in which a FU implements more than one operation, we choose the one with the largest number of slices for our estimation. For example, if a FU implements *vadd* and *vabs* operations, we estimate the area of that FU as 16 slices.

3) Estimation of Multiplexer Area Complexity: Next, we estimate the size of the multiplexers at the inputs of the FUs, as well as the multiplexers used for the control of each FU to select between the various operations performed in different cycles. In total, we need $n+1$ multiplexers for an n-input FU. As stated in Section IV-B, the area of multiplexers is based on three parameters: 1) number of inputs, 2) bitwidth of inputs, and 3) number of nops.

We start our analysis with the input multiplexers. The number of inputs of a multiplexer will be equal to the II because, in each cycle of the II, the multiplexer has to choose the appropriate input from a set of available signals. Therefore, the number of inputs of each multiplexer mux is $\text{IP}_{\text{mux}} = \text{II}$, and the number of selector bits for the multiplexer is $\text{SB}_{\text{mux}} = \lceil \log_2 \text{II} \rceil$.

The bitwidth of the multiplexers at the inputs of a FU FU_t of type t will be determined from the bitwidth specification of the operations as specified in the sDFG.

Since we do not know which operations are mapped to which FUs, we assume uniform distribution of operations among FUs. Therefore, for N_t operations of type t in the sDFG and m_t FUs of type t, we assume that each FU_t implements N_t/M_t operations. The number of nops among the inputs of multiplexer mux is

$$\text{noops}_{\text{mux}} = \max\left\{ \left(\text{IP}_{\text{mux}} - \left\lfloor \frac{N_t}{M_t} \right\rfloor \right), 0 \right\}. \tag{18}$$

The area of a multiplexer at the inputs of a FU of type t is estimated as a function f of these three parameters. Likewise, the area of the multiplexers to control the operation of the FUs is defined by a function, with the parameters IP_{mux}, $\text{BW}_{\text{cntMux}}$, and $\text{noops}_{\text{mux}}$. The bitwidth of the control multiplexer $\text{BW}_{\text{cntMux}}$ is given by

$$\text{BW}_{\text{cntMux}} = \left\lceil \log_2 \frac{N_t}{m_t} \right\rceil. \tag{19}$$

The analytical form of the function f is a curve-fitting polynomial with three independent variables based on a set of data points. The data points (or samples) are the area of various multiplexers synthesized with Xilinx ISE 8.1. The Appendix provides additional information on the details of the regression analysis to obtain a closed form of the function f.

Finally, the total area of the FUs and the multiplexers will be given by

$$\sum_{i=1}^{t} m_i \left(A_{\mathrm{FU}_i} + nA_{\mathrm{inMux}_i} + A_{\mathrm{cntMux}_i}\right). \tag{20}$$

V. Experimental Results

The effectiveness of the proposed estimation paradigm is evaluated for a set of streaming applications from the multimedia domain. This benchmark suite includes various video- and image-compression algorithms, such as the column and row DCT filters (*dctCol*, *dctRow*) and quantization functions (*quant*). The image-processing filters *hpf_med_cc* and *lpf_gc_rgb* implement pipelines of simpler filters such as high-pass filter, median filter and color correction and low-pass filter, gamma correction, and RGB-to-YUV conversion, respectively. Finally, the benchmarks include an imaging filter (*open filter*) and a series of smaller imaging filters used in a license-plate-recognition application (*lpr*) [3]. Table II shows the number of nodes for each benchmark. We compare our estimates for both the steps in our framework with the corresponding results generated by synthesis using the Proteus architectural synthesis tool to create synthesizable Verilog that is given as input to the Xilinx ISE 8.1 tool flow. In the following, we present our experimental results for each estimation method.

A. 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 three different sets of RCs. Table II shows our results. In Table II, RC set 1 corresponds to unlimited resources (and, thus, $\mathrm{II} = 1$). The resources become scarce progressively as we move to the right of Table II, so that the RC set 3 provides only one instance for a resource from each type (one ALU, one multiplier, etc.), whereas the RC set 2 is equivalent to the resources available in the reconfigurable streaming vector processor (RSVP) architecture [6]. From Table II, we observe that as the amount of available resources decreases (from RC set 1 to RC set 3); the queue sizes also decrease. As we discussed in Section IV, there are two counteracting factors determining the relationship between queue sizes and amount of FUs available. On one hand, since each resource will have one queue attached at its output, more FUs directly indicate more queues; hence, the total size of the queues will increase. On the other hand, with fewer FUs, the scheduler will resort to more resource sharing. While the number of FUs might be less, the average length of each queue might be higher as a result. Here, in this set of experimental results, we observe that the former aspect has a larger influence. With less stringent resource sets (i.e., more resources available), the total queue size increases. This is because, when there are plenty of resources (e.g., RC set 1), each node (operation) of the sDFG is assigned its own FU, and there is no sharing of the queue at the output of the FU. If the architecture is not taking advantage of sharing of the queues, the resulting hardware overhead becomes larger.

Moreover, our methodology is less accurate when there are plenty of resources, and $\mathrm{II} = 1$. For instance, Table II shows that, in that case, the average estimation error is 28.67%, whereas it is only 9.14% when there is a single resource for each FU type. When there are infinite resources, the algorithm assumes that a node is less constrained and can be scheduled in the whole [ASAP, ALAP] space as shown in (6). Therefore, there is more uncertainty on where an operation will actually be scheduled, which accounts for the larger error. As the resources become progressively more scarce, the accuracy of the method improves.

B. FU Area-Estimation Results

For the FU area estimation, we have evaluated our results by comparing the area obtained by our estimation to that generated by the industrial hardware generator tool [2]. The Verilog code generated is synthesized using Xilinx 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 RCs as in RC set 3 in Section V-A, which is the default configuration of the RSVP architecture [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 49 152 slices and 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 14.2%. Previous work by Kulkarni *et al.* [12] reports results that are about 10% better than ours. However, their estimation is based on a one-to-one mapping of nodes of the DFG to resources. Our estimation is trying to capture a more general case by taking RCs and sharing into account.

One of the important factors affecting accuracy for FU area estimation is the sharing of resources for different operation types. Our scheme is based on assuming the worst case, i.e., if a resource can be shared across multiple operation types, it will attain the largest area required for executing all operation types. In some benchmarks, such as dctRow, different types of operations that could possibly share a FU end up being scheduled such that they are assigned to distinctly unique resources. In those cases, the FUs tend to perform a single task uniformly; whereby, their area cost does not approach the worst-case estimation.

C. Impact on the Runtime of the Design Space Exploration

The total amount of time required by the complete design flow depends on various factors, such as the particular benchmark under consideration, other modules that exist in the FPGA (processors, peripherals, buses, etc.), and the resource capacity (number of CLBs) of the target FPGA device. If an FPGA is almost full and the benchmark is large, it may take

TABLE II
QUEUE-SIZE-ESTIMATION RESULTS

Application	*# Nodes*	*RC Set 1*			*RC Set 2*			*RC Set 3*		
		Synthesized	*Estimated*	*% Error*	*Synthesized*	*Estimated*	*% Error*	*Synthesized*	*Estimated*	*% Error*
dctCol	85	103	152	47.6	36	44	22.2	36	44	22.2
dctRow	95	111	168	51.4	41	49	19.5	41	49	19.5
quant	10	13	13	0.0	13	13	0.0	13	13	0.0
hpf_med_cc	157	404	347	-14.1	76	91	19.7	77	85	10.4
lpf_gc_rgb	221	697	508	-27.1	103	117	13.6	108	107	-0.9
lpr	67	150	110	-26.7	58	67	15.5	39	42	7.7
open	30	71	47	-33.8	44	34	-22.7	30	29	-3.3
Abs Average Error				**28.67**			**16.2**			**9.1**

TABLE III
FU AREA ESTIMATION

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

hours to synthesize, place, and route the whole SoC. The same benchmark will take minutes if the FPGA is large and/or there are no other modules in the FPGA.

In our experiments, we chose a target FPGA for which the utilization would not be pushed to extreme limits. Even in that case, the largest benchmark took close to 30 min to get past the Proteus tool (i.e., schedule and generate Verilog), and then, do synthesis, Place&Route through the Xilinx ISE tool.

If we have to explore 100 of these configurations, we would need $100^{*}30 = 3000$ min $= 50$ h for just one kernel (and there may be five to six kernels per application). One estimation run requires less than 1 min for the benchmark kernels. Hence, evaluating 100 configuration would take 100 min, followed by one synthesis run, totaling 130 min, which is more than an order of magnitude less than repeating the entire synthesis flow for each configuration.

VI. CONCLUSION

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 FUs; and 2) a library-based approach to estimate the FU area incorporating bitwidth awareness. We evaluated our estimation results based on synthesized designs using an industrial-template-based tool flow for a set of multimedia applications. For the register-queue sizes, our estimations are within 10.4% on an average. For the register-queue sizes, our estimations are in the range of 28%–9%. The amount of time required to perform the estimation is negligible in comparison to one full synthesis run. Considering a case where a large number of configurations need to be evaluated, utilizing our proposed estimation framework can help provide speedups by more than an order of magnitude.

APPENDIX

We seek to find an analytical polynomial form of the continuous function $A = f(x, y, x)$ given the values of f in a discrete set of independent data points (x_i, y_j, x_k). The unisolvence theorem states that, given a set of $(N_1 + 1) \times (N_2 + 1) \times (N_3 + 1)$ such data points, polynomial interpolation defines a linear bijection.

$L_n : K_{N_1+1} \times K_{N_2+1} \times K_{N_3+1} \rightarrow \Pi_{N_1 \times N_2 \times N_3}$, where $\prod_{N_1 \times N_2 \times N_3}$ is the vector space of polynomials of three independent variables with degree (N_1, N_2, N_3) or less, for each one of the three independent variables.

Suppose that the polynomial p is in the form

$$\begin{aligned} p = &\, a_{N_1N_2N_3}x^{N_1}y^{N_2}z^{N_3} + a_{N_1N_2N_{3-1}}x^{N_1}y^{N_2}z^{N_{3-1}} + \cdots \\ &+ a_{N_1N_20}x^{N_1}y^{N_2} + a_{N_1N_{2-1}N_3}x^{N_1}y^{N_{2-1}}z^{N_3} + \cdots \\ &+ a_{N_1N_{2-1}0}x^{N_1}y^{N_{2-1}} + \cdots + a_{002}z^2 + a_{001}z + a_{000}. \end{aligned}$$

If we substitute the $(N_1 + 1) \times (N_2 + 1) \times (N_3 + 1)$ known data values in the polynomial equation, the resulting function value will satisfy $f(x_i, y_j, x_k) = p(x_i, y_j, x_k)$, since the polynomial is, by definition, equal to the function f in these data points.

Based on this property, we construct and solve a system of $(N_1 + 1) \times (N_2 + 1) \times (N_3 + 1)$ linear equations with as many unknowns, the polynomial parameters being

$$a_{i,j,k}: \qquad 0 \le i \le N_1,\ 0 \le j \le N_2,\ 0 \le k \le N_3.$$

For our specific case, the independent variables are the number of inputs to the multiplexer, the bitwidth of the inputs (assuming that all inputs have the same bitwidth), and the number of nop operations. To keep the size of problem, and thus the size of the linear system to be solved, small, we only synthesize multiplexers whose parameters have the following restrictions: The number of inputs to the multiplexer (which equals the II) as well as the number of nops are in multiples of four and do not exceed 16 $\{1, 4, 8, 12, 16\}$. The bitwidth can only be in the set of $\{8, 16, 32\}$.

A system of 75 equations with 75 unknowns is constructed based on the results of the synthesis tool and solved to obtain the analytical form of the function f.

REFERENCES

[1] S. Amarasinghe and B. Thies, “Architectures, languages, and compilers for the streaming domain,” *Tutorial, Int. Conf. Parallel Architectures Compilation Tech.*, 2003.

[2] N. Bellas, S. Chai, M. Dwyer, and D. Linzmeier, “Template-based generation of streaming accelerators from a high level presentation,” in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2006, pp. 345–346.

[3] N. Bellas, S. Chai, M. Dwyer, and D. Linzmeier, “FPGA implementation of a license plate recognition SoC using automatically generated streaming accelerators,” in *Proc. Reconfigurable Architectures Workshop*, 2006, pp. 2–8.

[4] G. Bilavarn, J.-L. Gogniat, and L. Philippe, “Low complexity design space exploration from early specifications,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 10, pp. 1950–1968, Oct. 2006.

[5] C. Brandolese, W. Fornaciari, and F. Salice, “An area estimation methodology for FPGA based designs at system-C level,” in *Proc. Des. Autom. Conf.*, 2004, pp. 129–132.

[6] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, “The reconfigurable streaming vector processor (RSVP),” in *Proc. Int. Conf. Microarchitecture*, 2003, pp. 141–150.

[7] J. P. Diguet, D. Chillet, and O. Sentieys, “A framework for high level estimations of signal processing VLSI implementations,” *J. VLSI Signal Process. Syst.*, vol. 25, no. 3, pp. 261–284, Jul. 2000.

[8] R. Enzler, T. Jeger, D. Cottet, and G. Troster, “High-level area and performance estimation of hardware building blocks on FPGAs,” in *Proc. Int. Conf. Field-Programmable Logic*, 2000, pp. 525–534.

[9] B. A. Hamed, A. Salem, and G. M. Aly, “Area estimation of LUT based designs,” in *Proc. Int. Conf. Elect., Electron., Comput. Eng.*, 2004, pp. 39–42.

[10] C. T. Hwang, Y. S. Hsu, and Y. L. Lin, “PLS: A scheduler for pipeline synthesis,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 9, pp. 1279–1286, Sep. 1993.

[11] T. Jiang, X. Tang, and P. Banerjee, “Macro models for high level power and area estimation in FPGAs,” in *Proc. ACM Great Lakes Symp. VLSI*, 2004, pp. 162–165.

[12] D. Kulkarni, W. Najjar, R. Rinker, and F. Kurdahi, “Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems,” in *Proc. Int. Symp. Field Programmable Custom Comput. Mach.*, 2002, p. 239.

[13] S. Mondal, S. O. Memik, and N. Bellas, “Pre-synthesis area estimation of reconfigurable streaming accelerators,” in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2006, pp. 1–4.

[14] R. Moreno, R. Hermida, and M. Fernandez, “Register estimation in unscheduled dataflow graphs,” *ACM Trans. Design Autom. Electron. Syst.*, vol. 1, no. 3, pp. 396–403, Jul. 1996.

[15] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, “Accurate area and delay estimators for FPGAs,” in *Proc. Des. Test Eur.*, 2002, p. 862.

Seda Ogrenci Memik (M’98–SM’05) received the B.S. degree in electrical and electronic engineering from Bogazici University, Istanbul, Turkey, and the Ph.D. degree in computer science from the University of California at Los Angeles, Los Angeles.

She is currently an Assistant Professor with the Electrical Engineering and Computer Science Department, Northwestern University, Evanston, IL. Her research interests include embedded and reconfigurable computing, thermal-aware design automation, and thermal management for high-performance microprocessor systems.

Dr. Memik has served as technical program committee member, organizing committee member, and subcommittee chair of several conferences, including the International Conference on Computer-Aided Design, Design, Automation and Test in Europe, International Conference on Field Programmable Logic and Its Applications, and Great Lakes Symposium on Very Large Scale Integration. She is currently serving on the Editorial Board of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS. She was the recipient of the National Science Foundation Early Career Development (CAREER) Award in 2006.

Nikolaos Bellas (M’93) received the Diploma degree in computer engineering and informatics from the University of Patras, Patras, Greece, in 1992 and the M.Sc. and Ph.D. degrees from the Electrical and Computer Engineering Department, University of Illinois at Urbana–Champaign, Urbana, in 1995 and 1998, respectively.

From 1999 to 2007, he was a Principal Member of the technical staff with the Embedded Imaging Systems Laboratory, Motorola Inc., Schaumburg, IL, where he worked on chip design for multimedia processors and CAD tools for architectural synthesis. He is currently an Associate Professor with the Computer and Communication Engineering Department, University of Thessaly, Volos, Greece, where he is currently working in the area of reconfigurable computing, embedded systems, and computer architecture. He has lead research projects on developing CAD tools to automate the design of complex SoCs and has seen his work being used in a number of industrial products. He has published numerous papers and patents on a variety of research subjects.

Dr. Bellas is a member of the Technical Chamber of Greece.

Somsubhra Mondal received the B.E. degree (with honors) in electrical engineering from Jadavpur University, Calcutta, India, in 1997, the M.S. degree in electrical engineering from Michigan Technological University, Houghton, in 2002, and the Ph.D. degree in computer engineering from the Northwestern University, Evanston, IL, in 2007.

He was an Engineering Intern at the Embedded Imaging Systems Laboratory, Motorola Inc., during the summers of 2005 and 2006. His research primarily focused on architectural optimizations and CAD tools for power-aware FPGA design and synthesis of reconfigurable streaming accelerators. He has over four years of experience in the software industry. He is currently a Software Engineer with Neokast Inc. LLC, Evanston, IL, an Internet video broadcasting startup.