DESIGN SPACE EXPLORATION STRATEGIES FOR FPGA IMPLEMENTATION OF SIGNAL PROCESSING SYSTEMS USING CAL DATAFLOW PROGRAM

Ab Al-Hadi Ab Rahman, Richard Thavot, Simone Casale Brunet, Endri Bezati, Marco Mattavelli

SCI-STI-MM, École Polytechnique Fédérale de Lausanne
Station 11, CH-1015, Lausanne

ABSTRACT

This paper presents some strategies for design space exploration of FPGA-based signal processing systems that are specified using the CAL dataflow language. The actor-oriented, high-level of abstraction provided by CAL allows flexible exploration and consequently results in a wide range of feasible design implementations. We have applied and extended the existing techniques for refactoring and pipelining actors and actions by means of critical path analysis, and introduced some new buffering techniques based on heuristics. The combinations of these techniques have been applied on the CAL specification of the MPEG-4 video decoder, and synthesized to HDL for evaluation in the design implementation space. Results show that using our configuration for the exploration of 48 design points, a throughput range of roughly 8x has been achieved, for slice, block RAM, frequency, and latency range of 1.3x, 2.5x, 2.5x, and 2.9x respectively.

Index Terms— Dataflow, CAL, exploration, MPEG, FPGA

1. INTRODUCTION

Design exploration is one of the most important aspects in the implementation of signal processing systems. Essentially, an implementation of a system should always meet or exceed the target performance, which for hardware implementation, is typically that of system throughput, power consumption, and/or implementation area. In some cases, multiple objectives are required, for example in mobile applications where a design should exhibit a minimum throughput using the lowest possible power. For cost minimization, it is also crucial to use as small silicon area as possible for a given throughput and/or power requirement. The different feasible implementations of a system are often defined as design points in the multidimensional space; the exploration of these points spanned by the objectives is called the design space exploration.

In the context of hardware-based design of signal processing systems, there is now a growing interest in specifying at a high-level of abstraction compared to low-level Register Transfer language (RTL), mainly for fast design-cycle, less tedious and error prone, higher degree of code-reuse, and easier to incorporate advanced algorithms. There exists a variety of high-level of abstraction languages and models that are capable of synthesizing its specification to hardware. For example, the work of Lahiri et al [1] uses pre-configured IP blocks in a dataflow environment. Although the blocks are generally in the optimized form, it puts a restriction on the designer to explore different architecture of the instantiated blocks. Synthesizing hardware from imperative languages such as C/C++ has also been a topic of intensive research, for example the GAUT tool [2] of LABSTICC. However, imperative programs are designed to run sequentially and lack the concept of time, therefore, are difficult to analyze for potential parallelism [3]. SystemC extends the C language with a subset of synthesizable constructs, but mainly used for rapid cycle-based simulations. In this work we use the CAL dataflow language [4] which was specified as part of the Ptolemy project [5], and have proven to generate efficient hardware implementation such as the works in [6] and [7].

CAL is a domain-specific language for the design of dataflow actors. The core of the language is based on data tokens consumption and production by the actors, which lend itself naturally to signal processing systems. It is based on the Kahn Process Network (KPN) [8], where actors communicate via unbounded FIFO channels, but transformed to bounded sizes for implementation. Individual actors in a network are designed to execute in parallel, therefore allowing designers to explicitly specify the desired parallelism. Each actor in the network should contain at least one action, with the constraint that at any given time, an actor selects only a single action to fire. The selection of how and which action to fire is at the core of the Model of Computation (MoC), and is explained in section 2.3. CAL is also designed to be platform independent and retargetable to a rich variety of platforms for software (using orcc [9]), hardware (using openForge [6]), and co-design environments [10].

In this work, we contribute to the application and extension of existing techniques, as well as introducing new techniques for improving the throughput of complex CAL dataflow programs for hardware implementation. All the techniques are then combined to obtain a design space exploration that explores the trade-off between design throughput and resources, operating frequency, and system latency for

the FPGA implementation of MPEG4 decoder. Three design space exploration strategies have been applied and implemented: refactoring and buffering to reduce system latency, and pipelining to obtain higher operating frequency.

2. DESIGN SPACE EXPLORATION STRATEGIES

2.1. Trace Critical Path & Refactoring

Refactoring of an actor/action essentially means splitting, replicating, or modifying its computational elements such that an increase in parallelism is obtained. In a complex dataflow network, the challenge is to find the *critical actors/actions*, such that when the refactoring is performed on them, overall system throughput is improved. The critical actors/actions reside in the *execution trace critical path* (CP), which is the longest weighted path from source to sink node of the system.

The first tool that was developed to find the CP for CAL programs is known as the CAL Design Suite (CDS), which has been successfully used for optimizing software-based MPEG4 SP decoder [11] and hardware-based MPEG4 SP intra decoder [12]. The tool has been superseded by a new one called *TURNUS* that improves the technique for finding the critical actors/actions by performing dynamic analysis of dataflow programs directly at the dataflow level, instead of at software executable level as done in CDS. This results in faster convergence and higher details on the analysis. The following describes the methodology for trace critical path analysis using TURNUS.

The analysis is based on the *execution trace* $MDAG(V,E)$ that is generated performing a platform independent co-simulation of the dataflow design and it is defined as a multi directed acyclic graph such that every node $v_i \in V$ is a single firing of an action, and every edge $e_{i,j} \in E$ is a dependency of node v_j from node v_i (i.e. v_i must be executed first before v_j [13]). The dependences are fundamental for defining constraints on the execution order between any couple of fired action describing a scheduler-independent design behaviour because they impose an implicit execution order between the two connected actions (i.e. if it exists a dependence $e_{i,j}$ from node v_i to node v_j this implies that the firing of v_j can occur only after the firing of v_i). Moreover, from node v_i to node v_j could exist more than one dependences $\{e_{i,j}, \forall i,j : e_{i,j} \in \delta_i^+ \cap \delta_j^-\}$.

After that the generation of the $MDAG(V,E)$ is done, this latter is weighted by assigning for each node v_i a weight w_i, and for each edge $e_{i,j}$ a weight $w_{i,j}$ as well. All the weights are implementation specific: for example, in an hardware implementation a node weight w_i is defined as the latency (i.e. number of clock cycles) to execute the node v_i and an edge weight $w_{i,j}$ is the communication latency from the end of the firing of node v_i to the start of the firing of node v_j. Essentially, the trace represents a platform independent behaviour of the design and the CP can be figured out only after the weights assignment.

In order to reduce the algorithmic overhead for the analysis, an augmented graph $MDAG(\tilde{V},\tilde{E}) \supset MDAG(V,E)$ is defined where two new fictitious nodes are added: the source node v_S and the sink node v_T (both with weight $w_S = w_T = 0$). All the nodes v_i that have not incoming edges $|\delta_i^-| = 0$ are connected to v_s with a fictitious connection $e_{s,i}$ with weight $w_{s,i} = 0$; The same is done for all the nodes v_i that have not outgoing edges $|\delta_i^+| = 0$ where they are connected to v_T with a fictitious connection $e_{i,T}$ with weight $w_{i,T} = 0$. The topological order of the nodes remain the same $\{v_i < v_{i+1}, \forall i : v_i, v_{i+1} \in \tilde{V}\}$ by assigning to v_s and v_T respectively to the lowest and the highest topological index of $\tilde{V}$. After all the nodes, edges, and weights are annotated on the graph, the trace critical path can be evaluated. There exists several techniques on evaluating the trace such as in [14] and [15]. TURNUS implements the algorithm proposed in [15] because it provides reduced complexity and more detailed profiling information; Moreover, all of the required operations can be done in $O(|V| + |E|)$ by following the topological order of the $MDAG(\tilde{V},\tilde{E})$.

In this direction, for each node of the trace, four parameters are defined: 1) the *Early Start time* (ES) defines the earliest possible time that a node can start executing; 2) The *Latest Start time* (LS) defines the latest possible time that a node can start executing such that overall system latency does not change; 3) the *Early Finish time* (EF) defines the earliest possible time that a node can finish executing; 4) The *Latest Finish time* (LF) defines the latest possible time that a node can finish executing such that overall system latency does not change.

Furthermore, for each node v_i and edge $e_{i,j}$, the *Slack* S_{v_i} and $S_{e_{i,j}}$ are defined, representing the maximum delay that a node or edge could tolerate without increasing the system latency (i.e. if $S = 0$ then any increase in latency for the node or edge results in an increase in latency of the entire system). Nodes $v_i^* = \{v_i \in \tilde{V} : S_{v_i} = 0\}$ and edges $e_{i,j}^* = \{e_{i,j} \in \tilde{E} : S_{e_{i,j}} = 0\}$ are defined as critical. The sets of the critical executed action and the critical dependences are defined respectively as $C_A = \{v_i^*, \forall i : v_i^* \in V\}$ and $C_D = \{e_{i,j}^*, \forall i,j : e_{i,j}^* \in E\}$.

The CP can be defined as a path from v_T to v_S where all the nodes and edges are critical. The algorithm for finding such a path is depicted in Fig. 1: the graph is walked-back starting from v_T and at each iteration i, a critical node v_j^* is reached from the critical edge $e_{i,j}^*$. The CP is completely determined when v_S is reached and its weight is defined as $CP = LF_{v_T} \leq \sum\{w_i, \forall i : v_i \in C_A\} + \sum\{w_{i,j}, \forall i,j : e_{i,j} \in C_D\}$: one such path always exists [16].

The application of TURNUS finding the execution trace CP and the list of critical actors for our case study of MPEG4 SP decoder is described in Section 3.

```
v(i) = v(T)
// while source node has not been reached
while (v(i) != v(S))
 v(j) = walkBackCriticalPath(v(i))
 v(i) = v(j)
end
procedure walkBackCriticalPath(v(i))
 for each node j with v(i) dependency
  if (Sv(j) == 0 and Se(j,i) == 0)
   return v(j)
  end
 end
end
```

Fig. 1. Algorithm for evaluating the trace critical path.

2.2. Action Critical Path & Pipelining

Pipelining could also be considered as refactoring an action, but performed specifically to increase pipeline parallelism. In pipelining, the output of a computational element is the input to the next, separated by some memory buffers. The parallelism occurs when the computational elements execute in parallel. For CAL designs, the computational elements are considered to be the actors that should all fire at the same time for a given pipeline depth. This requires that every actor in the pipeline depth is a single-action SDF [1] actor.

A semi-automated tool that performs pipeline synthesis and optimization for CAL programs is given in [17]. The basic steps are given in the flow chart of Fig. 2. Starting from a CAL program, it is first synthesized to Hardware Description Language (HDL) using openForge, and then to RTL implementation using tools such as XST or Synplify. From this, we obtain the *action critical path* that defines the action in the network with the longest combinatorial path. If the action is not part of a single-action SDF actor, then it is converted to one. The SDF actor is then sent to the automatic pipeline synthesis and optimization tool which takes in the throughput requirement and generates the pipelined CAL actors using the global minimum pipeline resources.

The pipeline synthesis and optimization tool attempts to synthesize single-action SDF actor into k-parts as equally as possible in terms of the required length of the combinatorial path using minimum pipeline registers. It first generates the *asap* and *alap* schedules for the action based on the operator-input, operator-output, and operator-precedence relations. From this, operator mobility is determined and operators are arranged in order of mobility. This is then used in the *coloring* algorithm that generates all possible (and valid) pipeline schedules based on the operator conflict and nonconflict relations. For each pipeline schedule, total register width is estimated, and the least among all schedules is taken as the optimal solution, which is finally used to generate pipelined CAL actors. In [18], the tool has been used to pipeline the MPEG4 SP intra decoder with overall throughput improvement of more than 3x using minor additional resources.

[1] Static dataflow, where actors consume a constant number of tokens on its output and input ports at every firing

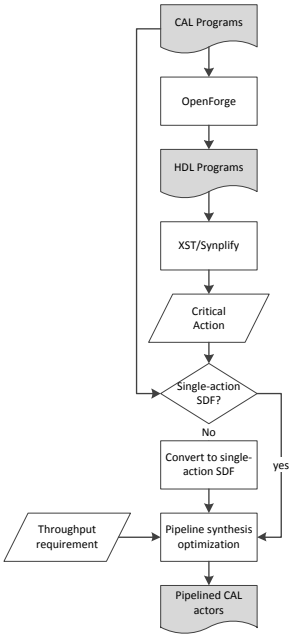

Fig. 2. Methodology for pipelining CAL dataflow programs.

2.3. FIFO Interconnections & Buffering

In CAL dataflow network, actors are interconnected using FIFO buffers. The selection of the sizes of each FIFO buffers in the network is crucial as it impacts not only the functionality (deadlock or deadlock-free), but also its performance.

Since actors can execute in parallel, a high throughput system is obtained if as much actors as possible are executed at a given time. An action in an actor fires if enabled by: 1) availability of input tokens, 2) value of input tokens, controlled by guard conditions, 3) the actor scheduler, 4) the action priority, and/or 5) the availability of free space to store output tokens. In order to ensure that actions are enabled and fired as quick as possible (hence results in higher throughput), conditions (1) and/or (5) have to be met as fast as possible. Systems with large buffer sizes between actors would always satisfy these conditions (for both actors) since input tokens are rapidly available from the buffers, and output tokens can always be generated due to large output buffers. However, setting all buffers to large values may not result in area-efficient implementation. On the other hand, buffer sizes that are too small between actors may introduce system deadlock. This is a condition when one or more actor stalls while waiting for input tokens that will never arrive, or actions that could not fire due to an always full output buffer.

In the following, we present two automatic buffering techniques based on heuristics that finds 1) the close-to-minimum required buffers for deadlock-free execution with

lower throughput using an extended version of [19], and 2) larger buffers for deadlock-free execution with higher throughput using a modified version of [20].

2.3.1. Close-to-minimum

This evaluation can be performed on a KPN model of computation graph for extracting minimum buffer size that guarantees deadlock-free executions at Transaction Level Modeling (TLM). This analysis is performed on an untimed simulation and it focuses on the communication channel. The assumption of such approach is a demand-driven scheduling [19] strategy, known to minimize the buffer size requirement by trying to mimic a perfect scheduler. The algorithm is shown in Fig. 3. In order to build a demand-driven scheduler, actors are ordered using a *topological sorting algorithm*. This algorithm then uses the *strongly connected components algorithm* for solving cycle path in a graph. The demand-driven scheduler tries to run actor by actor from the sink to the source until an actor is executed. Once an actor has been executed then the scheduler restarts from the sink. During the simulation analysis, the model of computation uses the communication mechanisms of the TLM FIFO. Those FIFOs are modelled as abstract channels and the FIFOs reallocate their size according the the demand driven computation. Consequently, the maximum size evaluated by the re-allocation gives the minimal size needed for a deadlock-free implementation.

```
eos := false //end of simulation flag
instances := toposort(V)
index := instances.last.index
while(!eos)
 instance := instances[index];
 //input tokens and state variables are available
 if(instance.isSchedulable)
  for each input edges of instance
   store edge.buffer.maxSize
  end
  for each edges of instance
    edge.buffer.reallocSize
  end
  instance.schedule // run the instance
  index := instances.last.index //restart graph scheduler
 elsif(instances[index] != instances.first)
  index := index - 1 // select the previous instance
 else
 eos := true // nothing is schedulable
 end
end
```

Fig. 3. Algorithm for the close-to-minimum buffering technique.

2.3.2. Deadlock-increment

The deadlock-increment technique iteratively updates all buffers that causes the system to deadlock at any given time. The algorithm is shown in Fig. 4. Initially, all buffers are set to the smallest value, i.e. 1. While the system is deadlocked, hardware simulation is ran to find the list of buffers that are full (i.e. that causes the system to deadlock). Each of these buffers are then doubled, and the while loop is repeated until the system is deadlock free.

```
for each channel i from 0 to m
 //initialize each channel capacity to 1
 buffer_size(i) = 1
end
deadlock_free = false
//while system is deadlock
while (!deadlock_free)
 run hardware simulation
 //check for deadlock
 deadlock = check_for_deadlock
 if (deadlock)
  for each full buffer j from 0 to n
   buffer_size(j) = buffer_size(j) * 2
  end
 else //deadlock free
  deadlock_free = true
  get_clock_cycles_latency
 end
end
```

Fig. 4. Algorithm for the deadlock-increment buffering technique.

Note that there are infinitely many ways that the buffers can be incremented in every iteration; for example they can be incremented by any value either by addition, multiplication, etc. There are also ways to select which buffers should be updated, either by the smallest or largest width, all or partial buffers, etc. In this work we choose to double the buffers due to the dataflow RTL architecture that takes only buffer sizes in power of 2, and update all buffers that causes the system to deadlock in order to obtain a fairly large total buffer size that would give a reasonably high throughput.

The algorithm in Fig. 4 runs a hardware simulation to check for deadlocks each time that buffer sizes are updated. This implies that the algorithm performs a dynamic analysis where the results are specific to a particular input stimulus. This is due to the assumption that there is at least one dynamic actor (DDF [2]) in the network, which is not possible to be analyzed statically. The algorithm has been implemented using TCL script for Modelsim hardware simulator, which also calls a Java program to automatically update the buffers in the HDL file, and provide logging of results to a text file.

3. CASE STUDY: MPEG4 SIMPLE PROFILE DECODER

The design space exploration strategies discussed in section 2 have been applied on the Reconfigurable Video Coding (RVC) MPEG4 Simple Profile (SP) decoder [21]. The decoder is specified in CAL and based on the new RVC standard in the ISO/MPEG, which proposes a new paradigm for specifying and designing complex signal processing systems. Essentially, the standard enables specifying new codecs by

[2]Dynamic dataflow, execution condition depends on the input data

assembling blocks from a Video Tool Library (VTL), which results in higher flexibility, reusability, and modularity.

The top level network is shown in Fig. 5. The encoded stream of bytes are first serialized to bitstreams to the parser, which then provides data, control, and motion vectors to the texture and motion decoder actors. The texture decoding and motion compensation are divided into three separate parts: one for luminance (Y) and two for chrominance (U and V) for parallel processing potential. At the end of processing (texture and/or motion), the merger combines the decoded bitstream from each three parts to form a complete YUV 4:2:0 video output. The decoder had been designed using 60 actors that contain both static and dynamic types.

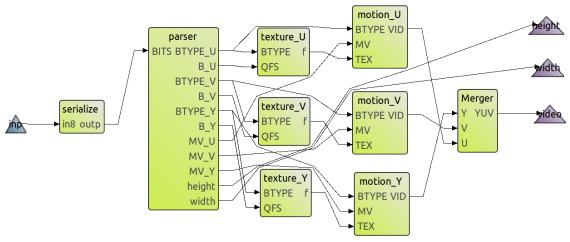

Fig. 5. CAL dataflow network of the RVC MPEG4 SP.

Among the three design exploration strategies, pipelining and buffering are not design specific, i.e. the techniques can be used and generalized for any designs. As for the refactoring technique, TURNUS only provides the list of critical actions/actors that are in the trace critical path of the design. The technique on how to refactor the critical actions/actors for system improvement is design specific and based on the discretion of the designer. The following presents the results from TURNUS for the analysis of the RVC MPEG4 SP and techniques on refactoring the critical actors.

The trace critical path is found to cross the y-branch texture inverse scan (*texture_IS_y*), y-branch texture inverse ac prediction (*texture_IAP_y*), and y-branch motion addressing (*motion_addr_y*). These three actors are therefore, the critical actors. Increasing the level of parallelism of these actors should result in a reduction in design latency.

The refactoring techniques for *texture_IS_y*, *texture_IAP_y*, and also *texture_IDP_y* (inverse dc prediction) have been reported in [12], and the basic mechanism is as follows: the y-branch texture decoder processes the y-macroblock video frames based on four blocks, i.e. blocks 0, 1, 2, and 3. In the original implementation, the blocks are processed in sequence; block 0, followed by block 1, then block 2, and finally block 3. In the improved parallel implementation, block 0 and block 3 are processed in parallel, followed by block 1 and block 2 in parallel. In theory, this would result in latency reduction of 2x, i.e. from 4 blocks to 2 blocks processing in sequence. However, since the video decoder input is serial, the gain that is achieved in practice have found to be somewhat less, but nonetheless, a significant gain. Note that all four blocks could not be processed altogether in parallel since there are dependencies among the blocks as specified in the MPEG standard [3]. Note also that it is also possible to refactor the *texture_IDP_y* using this technique, which we will prove that it will not result in any throughput gain since it is found to be outside the trace critical path.

The refactoring technique for *motion_addr_y* is as follows: In the original implementation of the motion compensation, one decoded pixel is dedicated to one memory location. This is inefficient in terms of latency since each memory access (i.e. reading or writing each pixel) requires two clock cycles. The addressing part is improved by first packing four pixels into four bytes, and then storing the packed pixels as 32-bit word in memory. This reduces memory access by 4x, which translates to a significant reduction in system latency. Note that it is also possible to apply this technique to the uv-branch of motion compensation (*motion_addr_u* and *motion_addr_v*), but again will be proven to result in no gain since they are not in the trace critical path.

4. EXPERIMENTAL RESULTS

This section presents experimental results of the design space exploration strategies (section 2) applied on the MPEG4 SP decoder (section 3). The CAL specification of the RVC decoder has been taken and synthesized to HDL using *openForge*. The actor for storing inter frames for motion compensation has been replaced by a memory controller that interfaces to a Cypress Semiconductor CY7C1354C SRAM. The synthesized decoder has been analyzed for clock-cycle latency using Modelsim for *Foreman QCIF* video frames (resolution 176x144), and verified for Xilinx Virtex-5 FPGA implementation using the XST synthesis tool.

The representation of the results are as follows. Each design point is assigned a prefix of either r for refactoring, or p for pipelining. Furthermore, the points are also assigned a two-digit subscript. For refactoring design points r_{xy} for $x \in \{0,1\}$ and $y \in \{0,1,...,9\}$, x represents the buffering technique used for that design point where $x = 0$ for *close-to-minimum* and $x = 1$ for *deadlock-increment* buffering techniques; and y represents which actors to be refactored as shown in table 1. For example, the design point r_{01} represents a *close-to-minimum* buffering technique with a refactored *motion_addr_y*. In total, 20 refactoring design points have been explored. Note that refactoring is performed for all combinations of the critical actors: *motion_addr_y*, *texture_IS_y*, and *texture_IAP_y*. The refactoring of the actors *motion_addr_u*, *motion_addr_v*, and *texture_IDP_y* are not analyzed for combinations since they are found to be outside the list of critical actors (This is also proven in the following graphs).

For pipelining design points p_{ki} for $k \in \{0,1,2,3\}$ and

[3] http://mpeg.chiariglione.org

$i \in \{1, 2, ..., 8\}$, $k = 0$ refers to pipelining the point r_{00}, $k = 1$ for the point r_{09}, $k = 2$ for the point r_{10} and $k = 3$ for the point r_{19}. The subscript i represents pipeline iterations as shown in table 2. For pipelining starting from point p_{01}, 6 iterations are possible before the combinatorial path is not anymore dominated by the action, but the routing delay and scheduler registers. Similarly, 7, 7, and 8 iterations are possible for pipelining from the points p_{11}, p_{21}, and p_{31} respectively. In total, 28 design points have been analyzed for pipelining space exploration.

Table 1. Refactoring design points r_{x0} to r_{x9} with refactored action(s). $x = 0$ for *close-to-minimum* and $x = 1$ for *deadlock-increment* buffering techniques.

Design point	Refactored actors(s)
r_{x0}	-
r_{x1}	motion_addr_y
r_{x2}	motion_addr_y motion_addr_u motion_addr_v
r_{x3}	texture_IS_y
r_{x4}	texture_IAP_y
r_{x5}	texture_IDP_y
r_{x6}	texture_IS_y texture_IAP_y
r_{x7}	motion_addr_y texture_IS_y
r_{x8}	motion_addr_y texture_IAP_y
r_{x9}	motion_addr_y texture_IS_y texture_IAP_y

The results have been analyzed for throughput based on four parameters: FPGA slice, block RAM, maximum operating frequency, and clock-cycles latency. Fig. 6 shows the graph of FPGA slice versus throughput for all design points. The refactoring points (r_{xy}) for both buffering techniques and all refactoring strategies show similar pattern, where r_{x5} (texture_IDP_y refactoring) and r_{x2} (motion_addr_y, motion_addr_u, and motion_addr_v refactoring) are inferior points with higher slice requirement for the same throughput compared to some other points, and r_{x9} are points with the highest throughput and slice. Pipelining of all four points (r_{x0}) and (r_{x9}) for both $x = 1$ and $x = 0$ show roughly linear increase in both slice and throughput. The best throughput also shows the highest slice requirement (point p_{38}) with slice of roughly 31000 and throughput of 1700 QCIF frames/s. Compared this to the original point r_{00}, it results in about 8x higher throughput with 30% more slice.

The graph in Fig. 7 shows the required FPGA block RAM (BRAM) versus throughput for all design points. The

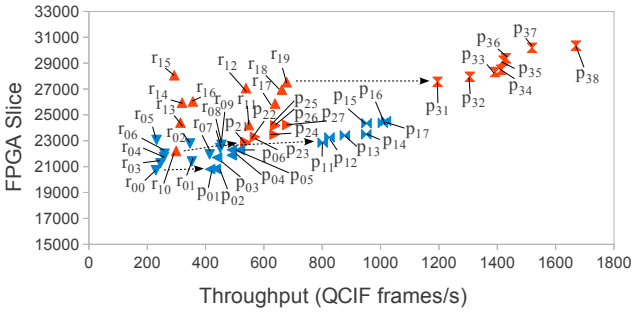

Fig. 6. FPGA slice versus throughput for all design points.

distribution of BRAM for the refactoring techniques (r_{xy}) are almost similar for both buffering techniques $x = 0$ and $x = 1$. Design points r_{x4} and r_{x6} require similar BRAM, but between points r_{x8} and r_{x9}, r_{08} require less BRAM compared to r_{09}, while r_{18} and r_{19} require the same number of BRAM. Furtermore, we can see that using the *close-to-minimum* buffering technique ($x = 0$), results in about 2x less BRAM compared to the *deadlock-increment* technique. As for pipelining, it can be observed that BRAM does not increase with every iteration. This implies that pipelining only takes the slice as additional resource. For the overall design space exploration with throughput range of 8x, the range for BRAM is about 2.5x between the original point r_{00} and highest-throughput point p_{38}.

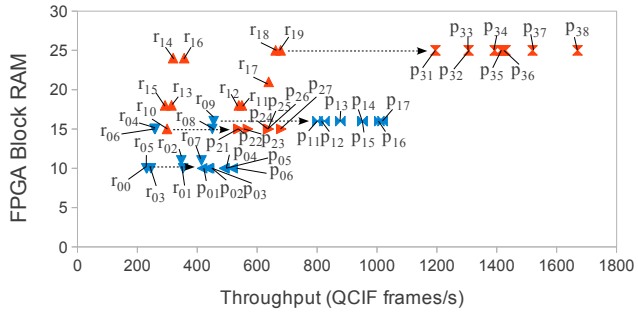

Fig. 7. FPGA block RAM versus throughput for all design points.

The frequency versus throughput plot is shown in Fig. 8. As expected, refactoring techniques do not increase the operating frequency, where it is almost constant at around 40MHz. Pipelining the original point r_{00} results in frequency range of between 76MHZ to 94MHz, point r_{09} with range between 75MHz to 96MHz, point r_{10} with range between 75MHz to 95MHz, and point r_{19} with range between 75MHz to 105MHz. This implies that pipelining increases system throughput by increasing the frequency at every iteration. For low power applications where the frequency should be kept

Table 2. Pipelining design points p_{0i}, p_{1i}, p_{2i}, and p_{3i} with pipelined action at every iteration i.

Iterations(i)	Pipelined actions			
	p_{0i}	p_{1i}	p_{2i}	p_{3i}
1	texture_IDP_readintra	texture_IDP_readintra	texture_IDP_readintra	texture_IDP_readintra
2	texture_IDP_readintra	motion_addr_readaddr	texture_IDP_readintra	motion_addr_readaddr
3	motion_addr_readaddr	texture_IDP_readintra	motion_addr_readaddr	texture_IDP_readintra
4	texture_IDP_readintra	texture_IDP_readintra	texture_IDP_readintra	texture_IDP_readintra
5	texture_dequant_ac	texture_dequant_ac	texture_dequant_ac	texture_dequant_ac
6	texture_dequant_ac	texture_dequant_ac	texture_dequant_ac	texture_dequant_ac
7	-	texture_IDP_readintra	texture_IDP_getdcinter	texture_IDCT_rowcalc
8	-	-	-	texture_IDP_readintra

as small as possible, pipelining strategies may not be feasible. On the other hand, buffering and refactoring strategies provide throughput improvement at a constant frequency.

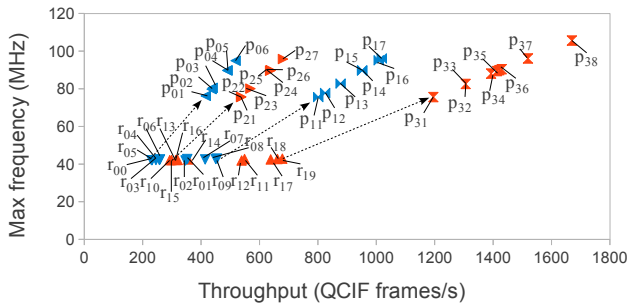

Fig. 8. maximum frequency versus throughput for all design points.

Fig. 9 shows clock-cycles-per-frame (c.c./f) latency versus throughput for all design points. In contrast to the frequency plot where refactoring techniques do not increase the operating frequency, latency can be reduced for most refactoring techniques. Using the *close-to-minimum* buffering technique, the original point r_{00} requires 183000 c.c./f, with the best point r_{09} with only 94000 c.c./f. On the other hand using the *deadlock-increment* buffering technique, the point r_{10} requires 140000 c.c./f while the best point r_{19} requires 63000 c.c./f. Comparison between r_{00} and r_{19} shows a latency reduction of roughly 3x. This is particularly important for systems that require as small operating frequency as possible to minimize power; for a given throughput requirement, systems with low latency requires a lower operating frequency compared to that of high latency system. Another interesting observation is that the latency of $r_{x5} = r_{x0}$ and $r_{x2} = r_{x1}$, which implies that the refactoring techniques of r_{x5} and r_{x2} do not result in any performance increase. As for pipelining, latency does not change for all points, as expected.

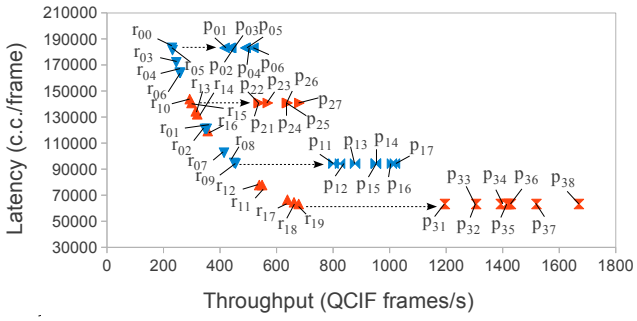

Fig. 9. latency versus throughput for all design points.

5. CONCLUSION & FUTURE WORK

In this paper, we have presented several strategies to explore the design space of signal processing systems that are designed using the CAL dataflow language. The strategies have been applied on the CAL specification of the RVC MPEG4 SP decoder, and synthesized to HDL for the exploration in the design space. The first strategy examines the trace critical path to find the list of critical actors where refactoring would result in feasible implementations. In this context, we have improved the technique on finding the trace critical path using a new tool called TURNUS, and introduced a technique to refactor a critical actor in the MPEG motion compensation. The second strategy examines the action critical path and applies the semi-automatic pipelining methodology for pipeline parallelism. The third and final strategy explores design tradeoff in the exploration space by introducing two heuristic techniques on assigning buffer sizes for deadlock-free execution. All these strategies are combined in such a way to achieve feasible design points for evaluation in the design space.

The design points in the exploration space for the case study given in this paper were obtained using heuristics; other refactoring, pipelining, and buffering techniques exist that would possibly give superior implementation points. This will be further explored in the immediate future.

6. REFERENCES

[1] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, pp. 768–783, 2001.

[2] E. Martin, O. Sentieys, H. Dubois, and J. L. Philippe, "Gaut: An architectural synthesis tool for dedicated signal processors," in *European Design Automation Conference - Proceedings*, 1993, pp. 14–19.

[3] G. De Micheli, "Hardware synthesis from c/c++ models," in *Design, Automation and Test in Europe Conference and Exhibition 1999*, 1999, pp. 382–383.

[4] J. Eker and J. Janneck, *CAL Language Report: Specification of the CAL Actor Language*, University of California-Berkeley, December 2003.

[5] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127 – 144, jan 2003.

[6] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs," *Journal of Signal Processing Systems*, vol. 63, pp. 241–249, 2011.

[7] A.A.-H. Ab Rahman, R. Thavot, M. Mattavelli, and P. Faure, "Hardware and software synthesis of image filters from cal dataflow specification," in *Ph.D. Research in Microelectronics and Electronics (PRIME), 2010 Conference on*, july 2010, pp. 1 –4.

[8] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing*, pp. 471–475, 1974.

[9] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan, "Software code generation for the rvc-cal language," *J. Signal Process. Syst.*, vol. 63, no. 2, pp. 203–213, May 2011.

[10] G. Roquier, R. Thavot, and M. Mattavelli, "Methodology for the hardware/software co-design of dataflow programs," in *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, oct. 2011, pp. 174 –179.

[11] C. Lucarz, G. Roquier, and M. Mattavelli, "High level design space exploration of rvc codec specifications for multi-core heterogeneous platforms," in *Proceedings of the 2010 Conference on Design and Architectures for Signal and Image processing (DASIP)*, October 2010.

[12] H. Amer, A.A.-H. Ab Rahman, I. Amer, C. Lucarz, and M. Mattavelli, "Methodology and technique to improve throughput of fpga-based cal dataflow programs: Case study of the rvc mpeg-4 sp intra decoder," in *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, oct. 2011, pp. 186 –191.

[13] J.W. Janneck, I.D. Miller, and D.B. Parlour, "Profiling dataflow programs," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, 2008, pp. 1065–1068.

[14] Cui-Qing Yang and Barton P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *ICDCS*, 1988, pp. 366–373.

[15] Cedell Alexander, Donna Reese, and James C. Harden, "Near-critical path analysis of program activity graphs," in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, Washington, DC, USA, 1994, MASCOTS '94, pp. 308–317, IEEE Computer Society.

[16] Reinhard Diestel, *Graph Theory*, vol. 173 of *Graduate Texts in Mathematics*, Springer-Verlag, Heidelberg, third edition, 2005.

[17] A.A.-H Ab Rahman, A. Prihozhy, and M. Mattavelli, "Pipeline synthesis and optimization of fpga-based video processing applications with cal," *EURASIP Journal on Image and Video Processing*, vol. 2011, pp. 1–28, 2011.

[18] A.A.-H. Ab Rahman, H. Amer, A. Prihozhy, C. Lucarz, and M. Mattavelli, "Optimization methodologies for complex fpga-based signal processing systems with cal," in *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, nov. 2011, pp. 1 –8.

[19] Nan Guan, Zonghua Gu, Wang Yi, and Ge Yu, "Improving scalability of model-checking for minimizing buffer requirements of synchronous dataflow graphs," in *Proceedings of Regular paper accepted by the 14th Asia and South Pacific Design Automation Conference, Jan. 19-22 2009. Yokohama, Japan*. 2009, IEEE computer society.

[20] T. M. Parks, *Bounded Scheduling of Process Networks*, PhD Thesis-University of California-Berkeley, December 1995.

[21] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 159–164+167, 2010.