# Combining Module Selection and Replication for Throughput-Driven Streaming Programs

Jason Cong, Muhuan Huang, Bin Liu, Peng Zhang and Yi Zou
Computer Science Department, University of California, Los Angeles

***Abstract*—Streaming processing is widely adopted in many data-intensive applications in various domains. FPGAs are commonly used to realize these applications since they can exploit inherent data parallelism and pipelining in the applications to achieve a better performance. In this paper we investigate the design space exploration problem (DSE) when mapping streaming applications onto FPGAs. Previous works narrowly focus on using techniques like replication or module selection to meet the throughput target. We propose to combine these two techniques together to guide the design space exploration. A formal formulation and solution to this combined problem is presented in this paper. Our objective is to optimize the total area cost subject to the throughput constraint. In particular, we are able to handle the feedback loops in the streaming programs, which, to the best of our knowledge, has never been discussed in previous work. Our methodology is evaluated with high-level synthesis tools, and we demonstrate our workflow on a set of benchmarks that vary from module kernel design such as FFT to large designs such as an MPEG-4 decoder.**

## I. Introduction

The primary goal of implementing streaming applications is to meet throughput requirements. Streaming programs are usually modeled in graphs, where each node in the graph represents an *actor*. Previous approaches to increasing throughput focus on replicating the bottleneck actors [1]–[7]. Those approaches point out that stateless actors, which have no dependence between consecutive executions, offer much data parallelism opportunities that can be exploited by replication of these actors. This strategy is targeted at a multicore architecture [1]–[6] or a multicore architecture with accelerators such as GPU [7]. For example, a greedy algorithm for mapping actors to different cores was proposed in [2]. If there are fewer actors than cores, the partitioner considers the actors in decreasing order of their computational requirements and replicates the candidate actors until all the cores are occupied. Conversely, if there are more actors than cores, the partitioner repeatedly fuses the least demanding actors until all actors fit in the multicore. In the context of the FPGA platform, a similar idea was also exploited. [8] [9] extend the idea of actor replication to the FPGA platform with consideration of resource limitations. For example, in [8] the proposed algorithm essentially performs maximal replication of all the stateless actors bounded only by the FPGA resource, then fuses the actors that do not affect the throughput. While replication does increase the throughput, such a mechanism is quite resource-consuming since occupied resources increase quickly as the number of replicas increases. High resource utilization might lead to a timing closure problem in FPGA implementation. Another disadvantage of this scheme is that it does not evaluate different possible implementation options for the actors to improve the performance; thus, only limited design space is explored.

In this paper we address the following issue: how do we design area-efficient FPGA implementations for streaming programs under the throughput constraints? We notice that most actors in streaming programs could have different implementations that trade off the processing throughput and area. Their combinations provide great opportunities for optimization since much area can be saved by selecting slower implementations for non-critical actors. This idea of module selection has been extensively explored in the past (for example in [10]–[14]). However, these studies target a fine-grained circuit level, and thus replication techniques are not employed. Moreover, most studies focus on the tradeoff between area and time, and do not consider throughput as a constraint.

In this work we are interested in building a library of implementations and selecting the best implementation from the library to replicate. Thus, our strategy could be viewed as a combination of module replication and module selection. As far as we know, we are the first to tackle the DSE problem by combining these two techniques. In contrast to previous work on design space exploration [1] [8] [9] that considers only streaming programs without feedback loops, we also take feedback loops into consideration since these are usually significant throughput bottlenecks in streaming programs [15].

The remainder of the paper is organized as follows. Section II provides background and our motivating example. Section III details the formulation and our methodology for the design space exploration problem. Section IV presents experiment results, and we conclude in Section V.

## II. Background and Motivating Example

In this work we use a synchronous data flow (SDF) [16] as our computation model. Streaming applications are represented as SDF graphs (SDFGs.) Nodes in the graph are actors, and the edges are called channels.

In general, actor replication is not an area-efficient design technique for increasing throughput because control logics, which do not directly contribute to throughput, are also duplicated. Moreover, some computing logics might be wasteful after replication. For example, consider an actor that contains

978-3-9810801-8-6/DATE12/©2012 EDAA

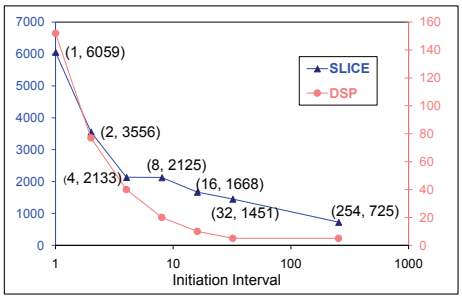


Fig. 1. Tradeoff between performance and area of one actor in benchmark "filterbank."

two add and one multiply operation, while the slowest implementation would use one adder and one multiplier to do the computation. When we duplicate such an implementation, two adders and two multipliers are generated, and one multiplier is wasted.

When mapping an actor in streaming programs onto FPGAs, by setting different design goals and deploying different optimization configurations we can generate "functionally equivalent" implementations that differ in area and performance. Therefore, we could generate a library of implementations for each actor so that design space is enlarged. For example, *initiation interval* (*II*) is a well-known parameter that trades off between area and throughput. In streaming applications, the initiation interval specifies the number of cycles between the consecutive firing of an actor and indicates to what extent the actor is pipelined. A pipelined actor offers high throughput, but usually at a large area cost because resource sharing opportunities are reduced. Fig. 1 plots the initiation interval and area cost of our FPGA-based implementations for one actor in the StreamIt benchmark "filterbank" [17]. The initiation interval of these design points are 1, 2, 4, 8, 16 and 254, respectively. We can see from the figure that decreasing the initiation interval results in more DSP logics and slices. To achieve a throughput of one firing per cycle, we could either replicate the slowest implementation (with *II* = 254) into 254 copies, or use the implementation with *II* = 1 directly. Obviously the latter option is more area efficient.

It's worthwhile to mention that pipelining does not guarantee that the throughput target can be satisfied, and replication is still needed in cases where a fully pipelined design cannot satisfy the high throughput target. There are also situations when hardware designers choose to use the predefined IP cores to avoid high development cost, and they do not have many choices for different implementations.

## III. Problem Formulation and Solutions

Consider a stream graph that has $M$ actors. When mapping actor $f_m$ in the stream graph to a physical module on FPGAs, we can have different implementations $P_m^1$, $P_m^2$,...,$P_m^{S_m}$, where each implementation $P_m^s$ can perform the functionality of $f_m$ with area cost $A(P_m^s)$, initiation interval $\delta(P_m^s)$, and execution time $t(P_m^s)$. $S_m$ denotes the number of implementations we have for actor $f_m$. Moreover, to meet the throughput target, $f_m$

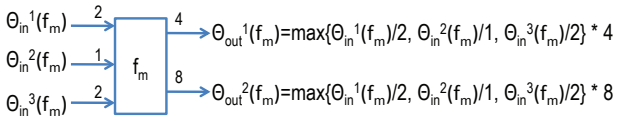


Fig. 2. Throughput target propagation.

can be implemented as several replicas of one implementation selected from the library. The DSE problem for streaming applications requires selecting implementations for each actor under throughput constraint while minimizing the total area cost. Such a throughput-driven DSE problem for streaming applications differs from traditional DSE problems in that we need to maintain a balance between the consecutive actors; *i.e.*, it is not beneficial to increase the throughput of a producer actor when its producer rate is higher than the highest consumer rate that the following actor can sustain.

### A. Throughput Calculation

For actor $f_m$ and its implementation $P_m^s$, the maximal input throughput $\theta_{in}(P_m^s)$ and output throughput $\theta_{out}(P_m^s)$ of $f_m$ are calculated as

$$\theta_{in}(P_m^s) = \frac{In(f_m)}{\delta(P_m^s)},\ \theta_{out}(P_m^s) = \frac{Out(f_m)}{\delta(P_m^s)}, \tag{1}$$

where $In(f_m)$ and $Out(f_m)$ equals the number of data tokens that $f_m$ consumes on the input data channel and produces on the output data channel during each firing. And $\delta(P_m^s)$ is the initial interval of $P_m^s$. This definition is different from [8] in which the authors substitute $\delta(P_m^s)$ with $t(P_m^s)$, because their definition does not consider pipelined implementations. If an actor has multiple input channels, the channel with the lowest data consumer rate is used in function (1).

The system throughput is measured at the first actor in the SDFG. It measures how many data tokens the system can consume in one clock cycle. We notice that for the SDFG that does not contain feedback loops, we are able to express the relationship between the system throughput target and the throughput target of each actor. Thus, we can break down the original problem into independent module selection subproblems for each actor and greatly reduce the problem size. For a SDFG with feedback loops, system throughput is related to the total execution time of the actors on the feedback path, and thus we need to consider the operation semantics of these actors. We will describe the detailed formulations of these two scenarios in the following two subsections.

### B. SDFGs Without Feedback Loops

Let us first discuss how to propagate the input throughput target to the output throughput target for a single actor. Denote the number of input and output channels for actor $f_m$ as $numIn(f_m)$ and $numOut(f_m)$. The throughput target on the input/output channel is denoted as $\theta_{in}^j(f_m)$/$\theta_{out}^k(f_m)$, where $1 \le j \le numIn(f_m)$ and $1 \le k \le numOut(f_m)$. Now, given the input throughput target $\theta_{in}^j(f_m)$, the output throughput target

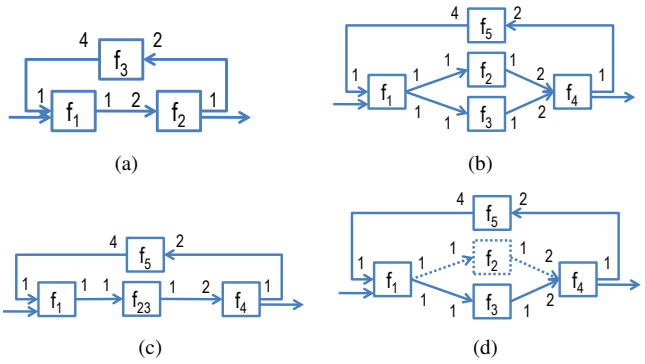


Fig. 3. (a) An example of the simplest form of feedback loop. (b) An example of a more complicated form of feedback loop. (c) and (d) are two ways to transform (b) into a simpler form. In (c), $f_2$ and $f_3$ are merged into one actor $f_{23}$. In (d), assume $f_3$ has a longer execution time than $f_2$, then we only need to consider $f_3$ in the feedback path.

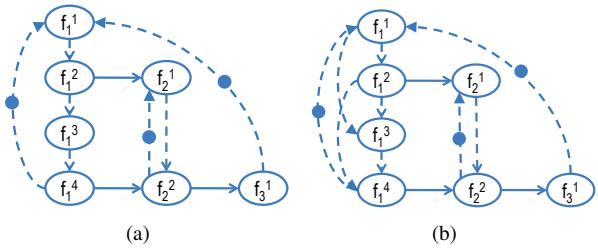


Fig. 4. Precedence graphs of feedback loop in Fig. 3(a). According to the data dependency, in each iteration actor $f_2$ could fire once after every two firings of actor $f_1$. In Fig. 4(a), each actor has only one physical implementation, thus each pair of consecutive firings is connected with a dashed edge. In Fig. 4(b), actor $f_1$ has two physical modules. According to the cyclic scheduling policy, the first and third firings are scheduled to one physical implementation, while the second and the fourth firings are scheduled to another physical implementation.

$\theta^k_{out}(f_m)$ is calculated as

$$\theta^k_{out}(f_m) = \max_j \left\{ \frac{\theta^j_{in}(f_m)}{In^j(f_m)} \right\} \cdot Out^k(f_m),$$
$$1 \leq k \leq numOut(f_m). \tag{2}$$

The use of max presumes that the selected physical implementations of $f_m$ could sustain the highest input throughput target. Fig. 2 illustrates the throughput target propagation process.

The system throughput target is enforced on the input channels of the first actor. Using formulation (2), we can propagate the throughput target to each input and output channel in the graph. We can also derive the initial interval target ($\delta^{tgt}(f_m)$) for actor $f_m$ as

$$\delta^{tgt}(f_m) = \min_j \left\{ \frac{In^j(f_m)}{\theta^j_{in}(f_m)} \right\}. \tag{3}$$

Therefore, given an SDFG and the system throughput target, we can compute the initial interval target for each actor.

Then, the DSE problem can be described as follows: Given a throughput (or initial interval) constraint for an actor, which implementation should be selected and how many replicas are needed so that the aggregated throughput of all the replicas is no less than the throughput constraint? That is, for actor $f_m$ we want to get binary integers $x_1$, $x_2$,...,$x_{S_m}$ indicating which implementation is selected, and integer $u_m$ indicating number of replicas needed. Then the formulation of the problem is:

$$\begin{aligned} \text{minimize} \quad & u_m \cdot \sum_{i=1}^{S_m} A(P^i_m) x_i \\ \text{subject to} \quad & \frac{1}{u_m} \cdot \sum_{i=1}^{S_m} \delta(P^i_m) x_i \leq \delta^{tgt}(f_m) \\ & \sum_{i=1}^{S_m} x_i = 1 \end{aligned} \tag{4}$$

where $\delta^{tgt}(f_m)$ is calculated as in function (3).

To solve this problem, we could simply enumerate all the possible values of $x_i$, compute the number of replicas needed based on the throughput constraint, compute the area cost, and select the implementation that consumes the least area. The complexity of this method is $O(S_m)$, where $S_m$ is the number of implementations in the library for actor $f_m$.

### C. SDFGs With Feedback Loops

*1) Feedback Loops:* In this section we will discuss the simplest form of feedback loop — actors are sequentially connected with each other along the feedback path. Despite strong restrictions, many SDFGs that do not meet such requirements can still be transformed into this simple form as shown in Fig. 3. Fig. 3(a) shows examples of the SDFGs that we are dealing with. Fig. 3(b) is an SDFG that violates the requirement — it has two actors firing concurrently rather than sequentially. We can either merge these two actors (Fig. 3(c)) or only consider the bottleneck actor in the loop (Fig. 3(d).)

The *iteration* of a feedback loop is a set of actor firings with as many firings as in the corresponding entry in the repetition vector **q** [18]. After one iteration, the SDFG returns to the same state; *i.e.* the number of tokens on each channel remains the same. In Fig. 3(a), the repetition factor **q** is [4;2;1].

We also assume that the initial data tokens on the feedback loop are located at the input edge of the first actor in the feedback path. Those feedback tokens, or in other words, dependency distance, are intrinsic in the algorithm. In fact, such distance is 1 in many streaming applications. Therefore, we also assume that the number of feedback tokens is just enough for the first actor to fire for one iteration.

*2) Formulation:* In the feedback loop, the throughput target imposes a constraint on the execution time of all the actors in the loop. Semantics of how the actors are fired are the key to estimate the latency of the loop. To delineate such a firing sequence, we construct a precedence graph as in [19]. The weight of edge denotes by how many cycles two firings should be separated.

Fig. 4 shows the precedence graph of a feedback loop in Fig. 3(a). Each node in the graph represents an actor firing. Solid edges denote the data dependency between adjacent actors. Dashed edges reflect the physical resource conflict between consecutive firings of the same actor. Physical resource conflict depends on the number of physical modules

(implementations) selected, their initiation intervals, and how the firings are scheduled to these physical modules. We adopt the cyclic scheduling policy, which assigns a firing to the first available physical module. Two firings that are scheduled to the same physical module should be separated by at least as many cycles as the initiation interval. In the precedence graph, edges with a dot denote the dependence across the iteration.

Our problem has two parts: module selection and scheduling. In module selection, we select one implementation and decide the number of replicas for each actor, thus finalizing the area cost, module execution time, initiation interval and specific structure of the precedence graph. In module scheduling, we schedule the start time of each node in the precedence graph to see if it can satisfy the throughput target. Usually scheduling is performed after the module selection, but in our proposed formulation we perform module selection concurrently with scheduling. Moreover, we should generate a scheduling that is feasible among all iterations, although we are only scheduling the firings in one iteration. Thus, we unfold the precedence graph to include the $f_1$'s firings in the second iteration (as shown in Fig. 5). The reason is that if $f_1$ has the same firing timeline in iteration 2 as in iteration 1, then all the other actors would also have the same timeline in the first two iterations. And if the second iteration has the same firing timeline as the first iteration, we are guaranteed to get a periodical scheduling.

The total area cost, execution time, and initiation interval of the selected implementation for actor $f_m$ is denoted as $A_m$, $t_m$ and $\delta_m$ respectively. We introduce binary integers $x_m^i$ to denote if implementation $P_m^i$ is selected and use $u_m$ to denote the number of replicas. Then we have

$$A_m = u_m \cdot \sum_{i=1}^{S_m} A(P_m^i) x_m^i, \tag{5}$$

$$\delta_m = \sum_{i=1}^{S_m} \delta(P_m^i) x_m^i, \tag{6}$$

$$t_m = \sum_{i=1}^{S_m} t(P_m^i) x_m^i, \tag{7}$$

$$\sum_{i=1}^{S_m} x_m^i = 1. \tag{8}$$

In the unfolded precedence graph, the set of nodes in the graph is $V$, and the set of edges is $E$. For $e(u,v) \in E$, denote the weight as $w(u,v)$. If $u$ and $v$ are the firings of different actors $f_m$ and $f_{m+1}$, then

$$w(u,v) = t_m. \tag{9}$$

If $u$ and $v$ are different firings $f_m^{k,j}$ and $f_m^{k,j'}$ of the same actor $f_m$, then $w(u,v)$ can be expressed as:

$$\forall j, \forall j', j < j', w(f_m^{k,j}, f_m^{k,j'}) = \begin{cases} \delta_m, if\ j' - j = u_m \\ 0, \quad otherwise \end{cases}. \tag{10}$$

Since $u_m$ is an unknown variable, every two firings of the same actor can be connected with an edge. And since the number of firings for $f_m$ in one iteration is $\mathbf{q}[m]$, the number of such constraints is $O(\mathbf{q}[m]^2)$. In realistic cases, we usually know the maximum possible number of replicas ($u_m^{max}$). Thus the number of constraints is $O(\mathbf{q}[m] \cdot u_m^{max})$.

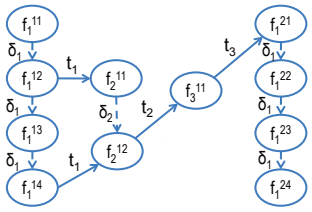


Fig. 5. Unfolded precedence graph of feedback loop in Fig. 3(a).

Denote the start time of the $j$th firing of actor $f_m$ in iteration $k$ as $T(f_m^{k,j})$. We can formulate the problem as follows,

$$\begin{aligned} &\text{minimize} \sum_{m=1}^{M} A_m, \\ &\text{subject to} \\ &\forall j \in \{1,2,...,\mathbf{q}[m]-1\}, \\ &T(f_m^{1,(j+1)}) - T(f_m^{1,j}) = T(f_m^{2,(j+1)}) - T(f_m^{2,j}), \quad (11) \\ &\forall j \in \{1,2,...,\mathbf{q}[1]\}, \\ &T(f_1^{2,j}) - T(f_1^{1,j}) \le \frac{In(f_1) \cdot \mathbf{q}[1]}{\theta_{in}^{tgt}(f_1)}, \quad (12) \\ &\forall e(u,v) \in E,\ T(v) - T(u) \ge w(u,v). \quad (13) \end{aligned}$$

where $\theta_{in}^{tgt}(f_1)$ is the throughput target of the feedback loop, which is measured at actor $f_1$. Constraints (11) and (12) guarantee that the schedule is periodical and should meet the system throughput. Constraint (13) ensures the dependency that firing $v$ follows $u$ by at least $w(u,v)$ cycles. $w(u,v)$ is calculated as shown in functions (9) and (10).

Functions (5) to (13) present our formulation of the DSE problem for an SDFG with feedback loop.

*3) Solution:* The above set of formulas are difficult to solve because function (5) and (10) contain nonlinear expressions. Therefore, we propose the following techniques to transform the formulation into linear expression so that we can use an ILP solver to solve the problem.

Let us look at function (10) first. We could enumerate the possible values of $u_m$ as 1, 2,...,$u_m^{max}$ and introduce binary integer variables $y_m^i$ to denote which one is the actual value of $u_m$. Thus $w(f_m^{k,j}, f_m^{k,j'})$ and $u_m$ is:

$$w(f_m^{k,j}, f_m^{k,j'}) = \delta_m \cdot y_m^{j'-j}, \tag{14}$$

$$u_m = \sum_{i=1}^{u_m^{max}} i \cdot y_m^i, \tag{15}$$

$$\sum_{i=1}^{u_m^{max}} y_m^i = 1. \tag{16}$$

Combining function (6) and (14), we have:

$$w(f_m^{k,j}, f_m^{k,j'}) = (\sum_{i=1}^{S_m} \delta(P_m^i) x_m^i) \cdot y_m^{j'-j}. \tag{17}$$

Due to the product of unknown variables $x_m^i$ and $y_m^{j'-j}$, the function is not linear. Thus, we propose the following theorem:

*Theorem 1:* Function $F()$ is a function of integer variable $x$ and binary variable $y_1$,$y_2$,...,$y_n$. $\sum_{i=1}^{n} y_i = 1,\ n > 1$. Given the value of binary variables, if $F()$ is a linear function of $x$, then $F(x, y_1, y_2, \ldots, y_n) \geq 0$ is equivalent to the following set of functions which are all in the linear form:

$$\begin{cases} F(x,1,0,\ldots,0) \geq (1-y_1)\cdot \inf_x F(x,1,0,\ldots,0) \\ F(x,0,1,\ldots,0) \geq (1-y_2)\cdot \inf_x F(x,0,1,\ldots,0) \\ \qquad\qquad \ldots \\ F(x,0,0,\ldots,1) \geq (1-y_n)\cdot \inf_x F(x,0,0,\ldots,1) \end{cases} \tag{18}$$

where inf means the infimum of the function. We can also derive a similar set of functions for the case where $f(x, y_1, y_2, \ldots, y_n) \leq 0$ (which is omitted here due to page limitation). This theorem can be easily proved by enumerating all the possible values of binary variables.

In function (17), the variables $x_m^i$ and $y_m^{j'-j}$ correspond to the binary variables $y_i$ and $x$ in the above theorem. We could also apply this theorem to function (5); thus all of the formulation of our problem is in a linear form, and can be solved with an ILP solver. The total number of variables is $O(\sum_{m=1}^{M} S_m + \sum_{m=1}^{M} u_m^{max})$, and the number of constraints is $O(\sum_{m=1}^{M} u_m^{max}\mathbf{q}[m])$.

## IV. Experiment Results

The experiment is carried out in two parts. For the streaming programs without feedback loops, we evaluate our strategies using StreamIt benchmarks [17]. For those containing feedback loops we examine our methodology using the MPEG-4 decoder. Implementation of these benchmarks on an FPGA is carried out by the high-level synthesis tool *Autopilot* [20] from AutoESL Xilinx (version: AutoESL_2011.1). It takes C code as input and generates synthesizable RTL code which can be fed into a *Xilinx ISE* design suite. Thus, most benchmarks are rewritten into C code. The FPGA device we use is the *Xilinx Virtex6* XC6VLX240T board. The target FPGA clock cycle is set at 100 MHz. Initiation intervals are specified in the C program by Autopilot pragmas. Estimated execution time and resource usage (*i.e.*, DSP, Block RAM, FF and LUT) are provided by the Autopilot synthesis report. The area metric is empirically estimated as the maximum utilization percentage of any of these four kinds of resources.

### A. Results on StreamIt Benchmarks

StreamIt benchmarks provide us with high-level structure and behavior for realistic streaming applications. The actors in StreamIt benchmarks are restricted to have only one input/output channel, and data distribution and merging are implemented as split-join nodes. Since we allow multiple input/output channels in SDF, we modify the benchmarks to embed the split-join nodes into the actors. For example, if an actor $f_m$ is followed by a split node, we embed the split node into $f_m$. Within $f_m$, data tokens are distributed to multiple output channels. We compare our methodology with two baselines: 1) the strategy in previous work [8] [9] that simply replicates the non-pipelined actors to reach high throughput; 2) the strategy that all the actors are fully pipelined. In the first case, throughput is usually achieved at a large area cost, while in the second case, non-bottleneck actors are over-performed and thus occupy more area than they really need. The comparison results are shown in Table I. For each benchmark we show the area of baseline 1, baseline 2 and our method that combines module selection and replication (CMSR for abbreviation). We also show the area reduction.

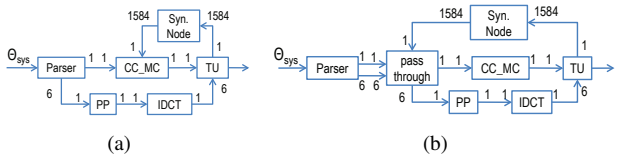


Fig. 6. SDFGs of MPEG-4 decoder.

### B. Results on MPEG-4 Decoder

MPEG-4 is a collection of methods and standards to perform audio and video coding. The reference C code for the MPEG-4 decoder is provided by Xilinx. The C code has already been developed with synthesis in mind, and most unsynthesizable constructs (such as dynamic allocation) are avoided. The design description and block diagram are presented in [21]. The modules are connected by either FIFOs or shared memory.

The MPEG-4 decoder can be represented in an SDFG as shown in Fig. 6(a). There are five actors in the graph: *Parser*, *CC_MC* (copy control and motion compensations), *TU* (texture update), *PP* (pre-processor) and *IDCT*. The data format is set as 4CIF and the data token granularity is set at the macroblock (16x16 pixels) level. So, a 4CIF-size image frame (704x576) corresponds to 44x36 macroblocks. The firing of *CC_MC* requires the output data from *TU* in the previous frame; thus there is a synchronization node between *CC_MC* and *TU*. While maintaining the correctness of the algorithm, we can transform the SDFG in Fig. 6(a) into Fig. 6(b) — a pass-through actor is added. There are two feedback paths in the new SDFG, and we select the path that takes the longer execution time (estimated from the Autopilot synthesis report) as the feedback loop.

We use function block pipelining inside some actors to generate hardware modules with different *II*. For example, *IDCT* is further broken into two modules, *IDCT_row* and *IDCT_col*, and they can be pipelined. The initial interval of *IDCT* is the larger value between the execution time of *IDCT_row* and *IDCT_col*. The implementation library we generated for *Parser*, *PP*, *IDCT*, *CC_MC* and *TU* is shown in Table II. In Table III we show that the minimum area cost for a given throughput target that varies from 30fps to 60fps. For each actor, we list the selected implementation, number of replicas, and total area usage for the actor.

As we can see from the table, bottleneck actors are *Parser*, *PP* and *IDCT*, which require a larger number of replicas and implementation with smaller *II*. It is worthwhile to point out that Autopilot uses the worst case latency as the estimated ex-

TABLE I
DESIGN SPACE EXPLORATION FOR THREE STREAMIT BENCHMARKS.

| | Filterbank | | | FFT | | | Autocor | | |
|---|---|---|---|---|---|---|---|---|---|
| Throughput (# of data per cycle) | 0.5 | 0.25 | 0.125 | 0.5 | 0.2 | 0.1 | 0.5 | 0.25 | 0.125 |
| Baseline1 Area | 56.6 | 28.6 | 14.6 | 38.8 | 19.4 | 18.9 | 83.5 | 42.6 | 24.5 |
| Baseline2 Area | 62.8 | 55.2 | 51.4 | 21.5 | 20.6 | 20.2 | 61.2 | 60.3 | 59.9 |
| CMSR Area | 39.9 | 20.9 | 11.5 | 20.7 | 19.4 | 18.9 | 61.2 | 36.75 | 24.51 |
| Reduct1 | -29.5% | -26.8% | -21.1% | -46.6% | 0 | 0 | -26.6% | -13.8% | 0 |
| Reduct2 | -36.5% | -62.1% | -77.6% | -3.7% | -5.8% | -6.4% | 0 | -39.1% | -59.1% |

TABLE II
IMPLEMENTATION LIBRARY FOR MPEG-4 DECODER KERNELS

| | Parser | | PP | | | IDCT | | | CC_MC | | | TU | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | v1 | v2 | v1 | v2 | v3 | v1 | v2 | v3 | v1 | v2 | v3 | v1 | v2 |
| II | 3278 | 2798 | 590 | 339 | 262 | 1250 | 378 | 250 | 4495 | 907 | 733 | 604 | 250 |
| FF | 1071 | 1218 | 1084 | 1014 | 1098 | 185 | 619 | 1055 | 602 | 629 | 666 | 300 | 371 |
| LUT | 2614 | 2947 | 2294 | 2311 | 2333 | 463 | 739 | 1699 | 1213 | 1260 | 1610 | 414 | 456 |

TABLE III
DESIGN SPACE EXPLORATION FOR MPEG-4 DECODER.

| Throughput | Parser | | | PP | | | IDCT | | | CC_MC | | | TU | | | Total Area |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (fps) | impl | rep | area % | impl | rep | area % | impl | rep | area % | impl | rep | area % | impl | rep | area % | % |
| 60 | v2 | 3 | 5.87 | v2 | 2 | 3.07 | v2 | 3 | 1.47 | v2 | 1 | 0.84 | v1 | 1 | 0.27 | 11.52 |
| 50 | v1 | 3 | 5.20 | v2 | 2 | 3.07 | v2 | 2 | 0.98 | v2 | 1 | 0.84 | v1 | 1 | 0.27 | 10.36 |
| 40 | v2 | 2 | 3.91 | v3 | 1 | 1.55 | v2 | 2 | 0.98 | v2 | 1 | 0.84 | v1 | 1 | 0.27 | 7.55 |
| 30 | v1 | 2 | 3.47 | v2 | 1 | 1.53 | v2 | 2 | 0.98 | v2 | 1 | 0.84 | v1 | 1 | 0.27 | 7.09 |

ecution time, thus may underestimate the actor's performance in reality. *Parser* is such a case. The ILP solver we use is GLPK [22], and results are generated on the order of minutes.

## V. CONCLUSION

In this paper we studied the design space exploration problem of mapping streaming applications onto FPGAs. Our method differs from existing methods because it combines both module selection and replicationto find a better design point in design space. The method is verified with both small designs like FFT and large designs like the MPEG-4 decoder, and results can be computed in minutes using an ILP solver. In the future, we would like to investigate more on the communication cost in this design space exploration problem.

## ACKNOWLEDGMENT

This research was partially supported by the NSF Expedition in Computing Award CCF-0926127 (Center for Domain-Specific Computing), Altera and Mentor Graphics.

## REFERENCES

[1] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGOPS*, vol. 40, pp. 151–162, 2006.

[2] M. I. Gordon *et al.*, "A stream compiler for communication-exposed architectures," *SIGARCH*, vol. 30, pp. 291–303, 2002.

[3] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," *SIGPLAN*, vol. 43, pp. 114–124, 2008.

[4] A. Hormati *et al.*, "MacroSS: macro-SIMDization of streaming applications," *SIGARCH*, vol. 38, pp. 285–296, 2010.

[5] A. Hormati *et al.*, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," in *PACT*, 2009, pp. 214–223.

[6] S. Liao *et al.*, "Data and computation transformations for Brook streaming applications on multiprocessors," in *CGO*, 2006, pp. 196–207.

[7] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Synergistic execution of stream programs on multicores with accelerators," *SIGPLAN*, vol. 44, pp. 99–108, 2009.

[8] A. Hagiescu *et al.*, "A computing origami: Folding streams in FPGAs," in *DAC*, 2009, pp. 282 –287.

[9] F. Plavec, Z. Vranesic, and S. Brown, "Enhancements to FPGA design methodology using streaming," in *FPL 2009*, 2009, pp. 294 –301.

[10] M. Ishikawa and G. De Micheli, "A module selection algorithm for high-level synthesis," in *ISCAS*, 1991, pp. 1777 –1780 vol.3.

[11] I. Ahmad, M. Dhodhi, and C. Chen, "Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis," *CDT*, vol. 142, no. 1, pp. 65 –71, 1995.

[12] K. Ito, L. Lucke, and K. Parhi, "ILP-based cost-optimal DSP synthesis with module selection and data format conversion," *VLSI*, vol. 6, no. 4, pp. 582 –594, 1998.

[13] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," *TCAD*, vol. 26, no. 2, pp. 254 –265, 2007.

[14] D. Chen, J. Cong, and J. Xu, "Optimal module and voltage assignment for low-power," in *ASP-DAC*, 2005, pp. 850 – 855.

[15] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *PACT*, 2010, pp. 365–376.

[16] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, 1987.

[17] "StreamIt benchmarks," http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.

[18] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. C-36, no. 1, pp. 24 –35, 1987.

[19] Govindarajan *et al.*, "Rate-optimal schedule for multi-rate DSP computations," *J. VLSI Signal Process. Syst.*, vol. 9, pp. 211–232, 1995.

[20] J. Cong *et al.*, "High-level synthesis for fpgas: From prototyping to deployment," *TCAD*, vol. 30, no. 4, pp. 473 –491, 2011.

[21] P. Schumacher *et al.*, "A scalable, multi-stream mpeg-4 video decoder for conferencing and surveillance applications," in *ICIP 2005*, vol. 2, 2005, pp. II – 886–9.

[22] "GNU Linear Programming Kit," http://www.gnu.org/software/glpk.