Throughput Driven Transformations of Synchronous Data Flows for Mapping to Heterogeneous MPSoCs

Anastasia Stulova, Rainer Leupers, Gerd Ascheid
Institute for Communication Technologies and Embedded Systems
RWTH Aachen University
Aachen, Germany
Email: {stulova,leupers,ascheid}@ice.rwth-aachen.de

Abstract—Due to energy efficiency requirements of modern embedded systems, chip vendors are inclined towards multicore architectures with different types of processing engines and non-uniform interconnect fabrics. At the same time multiple applications are intended to run concurrently on the devices with such heterogeneous architectures. This rapid growth in the complexity of the hardware and its use cases imposes new challenges on the software development tools. To overcome this complexity, model of computation based approaches are becoming increasingly promising. Synchronous Data Flow (SDF) is a popular specification formalism for streaming applications with inherently concurrent nature. However, the parallelism expressed in the original representation is often not sufficient to maximally exploit the potential of multicore platforms. In this paper we present a holistic methodology for improving the throughput of streaming applications while mapping them onto heterogeneous architectures. The approach uses transformations that adapt the parallelism in SDF according to available platform resources. We use a genetic algorithm to explore SDF instances with the objective of maximizing throughput on a target platform. Our model supports architecture heterogeneity and multi-application scenarios. The experiments indicate that our approach outperforms other techniques for exploiting parallelism on a single application in most of the test cases and enables concurrent applications optimization.

I. INTRODUCTION

Many applications for embedded systems are based on streaming of data. Heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) architectures have become prevalent in most of the embedded systems. Examples of such MPSoCs are the Texas Instruments OMAP [1], or the Platform 2012 initiative [2] from ST Microelectronics. These chip architectures usually contain a mixture of dedicated programmable cores, various hardware accelerators, possibly non-uniform memory hierarchy, and interconnect fabrics. Software tools must be aware of the heterogeneity to account for variable computation and communication latencies during the application execution.

In modern embedded systems applications run on the same device concurrently. This significantly complicates the optimization process as applications can affect each other during execution. At the same time, sharing of resources among applications is desirable to utilize systems efficiently.

Streaming computations are traditionally described in the format of data flows such as SDF [3]. Together with the Models of Computation (MoCs) different specification styles were developed, for instance, language based such as StreamIt [4], or CPN [5], and graphical environments namely System Studio, or SPW from Synopsys [6].

In this paper we propose a methodology for improving throughput of stream programs described in SDFs on heterogeneous platforms using graph transformations. During optimization we explore different variations of an SDF. We replicate computations in case they can be mapped to the target platform efficiently. By means of replication we propose a way to exploit data level parallelism available in SDF. Most of works on SDF only consider pipeline and task level parallelism by binding a computation to a platform resource [7]. If parallelism incurs high overhead we reduce it by sequentializing the computations. The main contributions of this work are:

- Language/framework independent transformation techniques applied directly to SDF for modifying the amount of exposed parallelism. The use of MoCs provides desirable properties of analyzing and verifying specification at design time. As a result the approach can be adopted by various tools or compilation flows.
- An integrated approach for selecting optimal transformations and mapping of SDF graphs to heterogeneous architectures which is applicable to a wide range of platforms while providing a solution in predictable time. The optimization methodology supports multi-application scenarios where multiple programs are sharing the resources available on the platform.
- An experimental evaluation of the proposed methodology for single and multiple applications on a heterogeneous MPSoC with a light-weight Operating System (OS) kernel.

The remainder of this paper is organized as follows. We first describe the SDF model and its usage as an application specification for an execution on MPSoCs in Section II. After the discussion of related work in Section III, we present two transformations of the SDF graphs that modify parallelism exposed by the model in Section IV. Section V shows how these transformations can be used to drive optimizations of programs compliant to SDF. Experimental evaluation is explained in Section VI. We conclude and give some insights into possible future work in Section VII. A glossary is provided in Section VIII.
II. BACKGROUND

**SDF Graphs.** An SDF graph (SDFG) is a tuple \( \xi = (\alpha, f, p, c, d) \), where \( \alpha \) represents a finite set of nodes (actors), \( f \) is a set of directed edges (channels), \( p \) and \( c \) are sets of integer values representing production rates and consumption rates. Each set \( d \) specifies data items (tokens) located in the channel \( f_i \) prior to an execution of a graph. An actor performs a computation characterized by a function. The results of computations are exchanged among actors via channels: unidirectional First In, First Out (FIFO) buffers. A channel is described by a pair of actors \((\alpha_{\text{prod}}, \alpha_{\text{cons}})\) representing producer and consumer of the channel respectively. Production and consumption rates describe the amount of tokens written to/read from channels at once. They are defined for each edge of the graph. One invocation of an actor with consumption of all input tokens and production of all output tokens is called firing. SDFs are characterized by the periodic behavior. A repetition vector \( q \) determines a minimal number of times each actor is to be fired to return the graph to its initial state [3]. We refer to one complete execution of the graph, in which each actor \( \alpha_i \) is fired \( q_i \) times as a periodic phase.

An example of an SDFG is given in Fig. 1a. *Sink* and *Source* are actors interfacing with the external environment and they are not depicted in the graph. The repetition vector for actors \( A, B, C \) is \( q = \{1, 1, 2\} \). There are 3 delay tokens on the channel \( f_1 \). After executing each actor \( q_i \) number of times (one periodic phase) the amount of tokens remains unaltered in all buffers.

**Target Execution Environment.** In the scope of this paper we analyze applications given in SDFG format and their execution on an embedded MPSoC. Typically, a lightweight OS runs on top of such custom MPSoCs to arbitrate one or multiple applications. Each task (elementary unit of work) is represented as an actor. Communications are realized using OS primitives e.g. semaphores/locks for a shared memory system or message exchange for a distributed memory system. Tasks are scheduled dynamically by the OS to achieve higher system utilization. As discussed in [8], multiple application use cases are frequently known at design time and can be prescribed in the format of a concurrency scenario. Such scenario contains a list of applications executed concurrently.

III. RELATED WORK

There are a number of transformations implemented in the StreamIt compiler [4] aiming at modifying task and data level parallelism in SDF-like programs. For instance, on actor replication special split and join filters are inserted to distribute and merge the streams computed by replicas. Such filters introduce additional communication and synchronization overhead as data streams have to be sent to them and then also to the replicas. The inserted filters have to be taken into account during mapping and scheduling. The overhead of split/join filters has been reduced in [9] by replicating a region of connected actors. The major limitation is that all actors in that region must have the same replication number. This is not always efficient for heterogeneous architecture as such actors in a region can have different speed on different processors. The transformation on SDFG presented in Section IV avoids overheads of duplicated data transfers completely by sending data only to where they are needed for computations. In contrast to StreamIt, we are not attached to any particular language semantic or implementation style, but rather work on the SDF MoC that provides useful property of analyzability. Our replication approach is somewhat similar to unfolding [7] of Homogeneous SDF. The main difference is that we do not create a copy of each actor in the graph, but only of those for which it is beneficial to do so. This eliminates negative impact of excessive actor increase that affects design and analysis time, and also causes run time overhead for handling actor execution. With our technique we also achieve code size reduction of the resulting program. Feasibility study for actor merging in SDFs is presented in [10], [11]. Their main motivation is reducing the complexity/size of SDF. We reuse this technique for modifying exposed parallelism.

To identify which program transformation is to be applied in StreamIt, [4], [9] proposed several heuristics targeting homogeneous platforms. CCCP [12] proposes to use Integer Linear Programming (ILP) for identifying data level parallelism in StreamIt. Flextream [13] extends this work to heterogeneous platforms by using two steps: replication heuristic and optimal mapping with ILP. We do not rely on optimal ILP formulation since it has exponential runtime. Moreover, decoupling replication from mapping phase can have negative impact on the solution quality for platforms with different types of processors as our experiments indicate. Flextream targets Cell based architecture with compilations of streams only to Synergistic Processing Elements (SPEs). Due to SPE homogeneity there is no difference in the actor execution time on various processors. This fact is not considered during the replication. Another drawback is that Flextream relies on the initial mapping which is not aware of the replication. Such mapping can limit the replication. Finally, none of the aforementioned approaches targets multiple application optimization.

Evolutionary algorithms have been widely used for mapping stream programs onto heterogeneous multicore platforms in the DOL framework [14]. Multiple application use cases have been studied for mapping in [8] and an application configura-
tions selection from the list of pre-parallelized options in [15]. Our approach is orthogonal to [15], as we aim at seeking for a parallel representation of an application.

IV. SDFG TRANSFORMATIONS

Frequently, only pipeline and task level parallelism are exploited on SDFG. In some cases such parallelism does not suitably match the target architecture e.g. if there are more processing elements than actors or vice versa, or if there is a hot spot actor that bounds the performance. A little attention was dedicated to modifying amount of exposed parallelism, especially the problem of exploiting data level parallelism via actor replications. We propose two SDFG transformations: actor replication and merging. Both modify the parallelism while producing a new SDFG. To bridge the gap between modeling and actual implementation we target two goals with our transformations: (i) we preserve the SDF properties such that all usual analysis procedures are applicable on the transformed graphs e.g. schedule and buffer size computation, or throughput estimation (ii) it is possible to automatically rewrite the application from the transformed graph in case the implementation compliant to the model is given. The second aspect allows keeping correspondence between the model and the actual running code.

A. Actor Replication

The replication transformation generates SDFGs with replicated actors from the original specification, preserving functional equivalence. Functional equivalence here means that for a given sequence of input tokens both the original and the transformed graph produce the same sequence of tokens.

To construct an SDFG with replicas, a vector $\gamma$ of integer values with replica numbers has to be specified. For the nodes $A, B, C$ in Fig. 1a and $\gamma = \{2, 2, 1\}$, the resulting graph with replicas is shown in Fig. 1b. This graph exhibits more parallelism, as there are more actors that can execute concurrently. According to the operational semantic of SDF, all computations are now evenly distributed among replicas of $A$ and $B$. The actor $C$ is not replicated and it reads tokens from both the replicas. The production and consumption rates for $A^i$ and $B^i$ are unchanged. The actor $C^0$ upon its firing, instead, reads 2 and writes 4 tokens. The increase in the consumption rate of the input channels of $C^0$ is needed to keep the original functionality of the graph when generating the actual running code and it is explained below. As more tokens are obtained by $C^0$ at each invocation, more computations must be performed at once and, thus, more output tokens are generated. If the computations of $A, B$ and $C$ take 4, 4 and 1 time unit respectively, then $C^0$ requires 4 time units for execution. The repetition rate $q_i$ for all actors in Fig. 1b is 1. If all actors are executed in parallel, we observe for the graph with replicas twice the amount of tokens appear at Sink during the time interval of 4 units. Further replication of $A$ and $B$ does not bring any benefit as the execution will be bounded by the computation time of $C$. The computational load has changed in the resultant graph.

Algorithm 1 describes a formal procedure for derivation of the graph with replicas. Initially we construct a new set of actors by adding replicas (lines 2-4). In lines 5-20 we recreate the edges to connect the actors that have just been added. Lines 6-7 calculate common terms. Operators prod and cons return an index of producer and consumer actors for a given channel of the graph. The number of edges that have to be inserted to substitute each channel in the original graph are calculated in line 8. For each channel of the new graph we assign a consumer and a producer preserving strict alternation that aims at work balancing among actors (shown in line 11). The delay tokens are taken into account here by shifting the consumer of each new edge by the number of delays (computed in line 9). In the new production rate calculation (line 12), we consider the least common multiple (lcm) among the replication numbers of the producer and its adjacent actors to ensure the smallest possible integer value is obtained. With this calculation we also prevent introducing inconsistency in SDFG. Inconsistency [3] is caused by the consumption/production rate assignments that requires unbounded memory during SDF execution. An operator adjacent returns a set of replication numbers of the nodes connected to a given node by an edge. Our replication technique exploits data level parallelism among various periodic phases of SDFG. Therefore, we take into account repetition rates $q$ of $\xi$ (in the term $pt$) while computing the new production rate. The consumption rates are computed similarly (line 13). In lines 14-17 we distribute delays equally among the new edges preserving their indexing order. The delay distribution and actor/channel indexing are done in such a way that functionality of the original graph can be preserved during the code generation. More details are
provided at the end of this subsection.

Practically, an unlimited number of replicas can be created for an actor in a cycle-free path of a graph. If a graph has cycles the transformation explained above for certain replication vectors may result in the graph with deadlocks due to insufficient delays. For SDF there exists a procedure that can perform liveness check [3] very efficiently on most of the practical cases.

Implementation Rules. Algorithm 1 creates a graph topology. Different analysis and verification procedures can directly be applied to it. To derive a valid implementation of an application from this graph, we present a set of rules below. The functionality of each actor is commonly implemented as a set of actions including accesses to channels and performing actual computation (see example Listings 1a,1b). We use read and write primitives for removing or adding tokens to a buffer. Computation is described by a function e.g. \( \text{funcC} \) for the actor \( C \). The first rule we impose is that accesses to channels have to be made in ascending order of their indexing specified in Algorithm 1. The second rule is that an actor has to complete accessing the number of tokens specified by the new production and consumption rates from the current channel before moving to the next channel.

Both rules are needed to enforce the original order in which tokens are produced by the graph and to simplify the code generation procedure. In Listing 1c, \( C^0 \) first reads 2 tokens from channels \( f^0_2, f^0_3 \) and only after 2 tokens from channels \( f^1_2, f^1_3 \). We now explain the increase of the consumption rate of the actor \( C^0 \) and reasoning for the two rules we have just specified. In Fig. 1a the first 2 reads of the actor \( C \) remove tokens produced by the first firing of \( A \) and \( B \). The repetition rate of \( C \), as defined in \( q \), is twice that of \( A \) and \( B \). If we assume that in the transformed example \( C^0 \), still, reads 1 token per input channel, its first firing would consume tokens from two different firings of both \( A \) and \( B \) of the original graph. Considering that all tokens are written to the FIFO buffers just before the firing ends, the tokens read by \( \text{Sink} \) would arrive in different order.

Listing 1 Excerpts of the actors pseudo code

(a) actor \( B \) from Fig. 1a

\[
\begin{align*}
a &= \text{read}(f_1); \\
b &= \text{funcB}(a); \\
&\text{write}(f_3, b); \\
&\text{for } (i=0;i<4;i++) \\
&\quad \text{if } (i<2) \\
&\quad \quad a = \text{read}(f_2_0); \\
&\quad \quad b = \text{read}(f_3_0); \\
&\quad \quad \text{else} \\
&\quad \quad a = \text{read}(f_2_1); \\
&\quad \quad b = \text{read}(f_3_1); \\
&\quad \quad c = \text{funcC}(a,b); \\
&\quad \text{write}(f_4_0, c); \\
&\quad \text{write}(f_4, c); \\
&\end{align*}
\]

(b) actor \( C \) from Fig. 1a

\[
\begin{align*}
a &= \text{read}(f_2); \\
b &= \text{read}(f_3); \\
c &= \text{funcC}(a,b); \\
&\text{write}(f_4, c); \\
&\end{align*}
\]

(c) actor \( C^0 \) from Fig. 1b

\[
\begin{align*}
&\text{for } (i=0;i<4;i++) \\
&\quad \text{if } (i<2) \\
&\quad \quad a = \text{read}(f_2_0); \\
&\quad \quad b = \text{read}(f_3_0); \\
&\quad \quad \text{else} \\
&\quad \quad a = \text{read}(f_2_1); \\
&\quad \quad b = \text{read}(f_3_1); \\
&\quad \quad c = \text{funcC}(a,b); \\
&\quad \text{write}(f_4_0, c); \\
&\end{align*}
\]

(d) actor \( BC \) from Fig. 2

\[
\begin{align*}
a &= \text{read}(f_1); \\
b &= \text{funcB}(a); \\
c &= \text{funcC}(a,b[i]); \\
&\text{write}(f_4, c); \\
&\end{align*}
\]

Delays represent shifts of token processing in the graph. In Algorithm 1 the delay tokens are distributed to the channels such that shifts are adjusted according to the original computation order. In the example in Fig. 1a at each firing \( A \) removes token \( t_i \) from the input to the graph while \( B \) removes token \( t_{i-3} \). Thus, \( B \) is delayed by three tokens due to three initial delays. \( C \) instead gets tokens \( t_i \) on \( f_2 \) and \( t_{i-3} \) on \( f_3 \). Let us assume the delays in Fig. 1a are given by the set \( d^0_1 = \{d_{10}, d_{11}, d_{12}\} \). Then the distribution of tokens produced by the replication procedure is as follows \( d^3_1 = \{d_{11}\} \) and \( d^3_2 = \{d_{10}, d_{12}\} \). If we adhere to the implementation rules defined above, such delay distribution will lead to the equivalent order of computations as in the original graph.

B. Merging of Actors

Merging is an aggregation of nodes into one actor that performs the computations of all its constituents. We apply it to eliminate channels of SDFGs and, thus, to reduce communication related cost and actor execution handling overhead (described in Section V-A). In [10], [11] it is shown that actor merging can result in inconsistent and deadlocked graphs. To avoid the first problem we take the repetition rate into account during merging. The example in Fig. 2 illustrates merging of actors \( B \) and \( C \). The consumption rate on the channel \( f_2 \) is increased w.r.t to the original graph in Fig. 1a as \( C \) has twice the repetition rate of \( B \). The implementation of the composite actor \( BC \) is shown in Listing 1d. The result of the merging procedure has to be, however, checked for deadlocks [3].

V. OPTIMIZATION APPROACH

We employ a genetic algorithm (GA) to seek for transformations to be applied to the original SDFG for the following reasons. It is general enough to represent our problem. The solution search time can be configured based on the termination criterion. We can easily take into account several aspects such as possibility of both replication and merging, multiple application execution, and limiting factors: runtime,
synchronization and communication overheads. Our objective is throughput maximization - \( \max T_{\xi} \). The mapping of an application graph to a platform is taken simultaneously into account. We now describe a platform and an application model used in our problem formulation.

Heterogeneous MPSoCs are composed of different processor types that can communicate with each other via various interconnect resources such as buses or point-to-point links. There might be multiple memory components as depicted in Fig. 3. Let \( \pi^\psi_i \) denote the \( i \)-th processing element (PE) of the type \( \psi \). \( N_\pi \) is the total number of PEs in the platform. As communications among processors usually involve interconnect and storage, we use an abstract logical communication primitive. A value \( \kappa \) reflects communication cost of such a primitive in terms of latency for transmitting an elementary data unit (a byte, or a word). Each actor \( \alpha_i \) is characterized by a computation cost per processor type \( \psi \) defining an execution time \( \epsilon_{i\psi} \) of one firing. Each buffer \( f_i \) is characterized by the size of transferred token \( \delta_i \).

An individual solution (chromosome) encodes a graph and its mapping in the format of a binary vector of a length \( N_\alpha \times N_\pi, N_\alpha = |\alpha| \). Each bit (gene) \( g_i \) represents whether a replica of the \( (i \mod N_\alpha) \)-th actor is mapped to the \( \lfloor i/N_\pi \rfloor \)-th processor. The encoding for the chromosome is depicted in Fig. 4. The highlighted area represents the encoding for one application (i.e., one SDFG). From the chromosome we extract the replication vector \( \gamma \): \( \forall i = 0 \ldots N_\alpha - 1, \gamma_i = \sum_{k=0}^{N_\alpha-1} q_{k\times N_\alpha + i} \) and apply the replication transformation to the original graph \( \xi \). We also extract the mapping \( \mu \) of actors to PEs. As a final step we will merge any pair of connected actors \( \alpha_{pr}, \alpha_{ca} \) mapped to the same PE.

For the graph in Fig. 1a and for a three processor platform, the chromosome:

\[ \chi = 110 110 001 \]

is decoded as follows. The actors \( A \) and \( B \) are replicated twice and both mapped to the processors 1 and 2. The actor \( C \) is not replicated and is mapped to the processor 3. The SDFG encoded by the chromosome is shown in Fig. 1b. Each pair of nodes \( A \) and \( B \) is merged at the last step as they are to be executed on the same processor.

The GA iteratively proposes a population of candidate solutions generated by evolutionary operators such as crossover and mutation [16]. There are a number of constraints that each solution has to satisfy. The condition \( \gamma_i \geq 1 \) is used to ensure that at least one copy of each actor \( \alpha_i \) is to be created. As discussed in Section IV, the validity of the transformed SDFG has to be checked. More constraints may arise from the target platform. As every MPSoC contains limited memory, the communication buffers are constrained by the memory size.

A. Throughput Computation

For each transformed SDFG \( \xi' \) and its mapping \( \mu \) throughput estimation has to be done. General techniques for the throughput computation of SDFG have been studied in [7], [17]. They are, however, known to be computationally expensive. For SDFGs that contain cycles only as self-edges the computation can be done more efficiently based on the Initiation Interval, as proposed in [12]. In practice very few real life SDFGs contain parallelism in a path with a cycle. We, therefore, assume for the rest of the paper that the application contains only self-cycles (states). Taking into account this restriction, we can reuse simple and efficient heuristic in [18] to derive the required buffer sizes for a given SDFG.

Actors can run on different processors concurrently. We target systems with dynamic scheduling [7]. As all buffers are bounded, there will be a steady state of execution in which the throughput is bounded by the processor with the highest load and can be approximated based on the time required to execute one full periodic phase. The time spent by the processor \( \pi^\psi_j \) to perform the assigned computations is given by:

\[ \tau^\psi_{i\psi} = \sum_{\alpha_i \in \mu^\psi_j} (\tau^E_{i\psi} + \tau^K_{i\psi} + \tau^\Delta_{i\psi}) \]  

where \( \tau^E_{i\psi} \) accounts for the total computation time of an actor \( \alpha_i \) performed during the period on the processor \( \pi^\psi_j \) and \( \tau^K_{i\psi} \) accounts for the cost of communication and synchronization. The term \( \tau^\Delta_{i\psi} \) represents the overhead of handling an execution of an actor (a function call or a thread context switch). Its value is measured once per processor type and is assumed to be constant during the optimization. In Section IV, it has been shown that during the graph transformation the granularity of actor computations may change. We present a formula for an actor execution cost in the transformed graph \( \xi' \) below. The following computation applies to the actor \( \alpha_i' \) that is a replica of some actor \( \alpha_i \) of the original SDFG with the repetition count \( q_i \) and the execution time per firing \( \epsilon_{i\psi} \) on the processor \( \pi^\psi_j \):

\[ \tau^E_{i\psi} = \frac{q_i \times \epsilon_{i\psi}}{\gamma_i} \]  

In the denominator we account for the speedup each replica gets from cooperative work among replicas of the same actor. In fact, each replica is fired less number of times than the
original actor. For an actor $\alpha_n$ that is composed of a pair of merged actors $\alpha_{pr}$ and $\alpha_{co}$ we first compute individual costs $\tau^{E}_{(pr)}$ and $\tau^{E}_{(co)}$ and then simply add these two costs together.

Channels incur communication and synchronization overhead. Communication latency estimation differs significantly for various types of interconnects. For example, on shared memory it can be derived on the basis of the number of elementary data items that have to be loaded $N_{ld}$ or stored $N_{stb}$ when an actor is fired. These numbers are obtained from the information on the token size $\delta$ and production/consumption rates per channel. We assume without any loss of generality that there is only one communication primitive in the platform with latency $\kappa$. If an input channel $f_i$ of $\alpha_i$ on $\xi$ is represented by a set of input channels $f_i^m, m = 0, \ldots, lcm(\gamma_{prod}(f_i, \xi), \gamma_{cons}(f_i, \xi))$ in the graph $\xi'$, the overall communication cost for each channel on the processor $\pi_j$ is:

$$\tau_{\pi_j}^{K} = q_i \times \left( S_{rd}^{\psi} + N_{ld}^{n} \times \kappa \right)$$

(3)

where $S_{rd}^{\psi}$ is the synchronization overhead caused by each single read access to a buffer (acquiring locks, calling send or receive primitives). We assume this synchronization overhead is constant per processor type $\psi$ and can be measured before the optimization starts. For the output channel the computation is done similarly. The overall communication cost $\tau_{\pi_i}^{K}$ is a sum of costs caused by all input and output channels connected to the actor $\alpha'_i$.

Considering infinite repetitive actor firings in SDF, the throughput of cycle-free SDF with possible states is bounded by the processor with the highest load. We, therefore, use the following computation for our optimization objective:

$$T_{\xi'} = \frac{1}{\max_{\forall \pi_j \in \pi} \{ \tau_{\pi_j}^{K} \}}$$

(4)

The replication can increase throughput as individual execution time of actors becomes smaller (see Eq. 2), but it can also have negative impact if the maximum workload $\tau$ becomes too large. The merging reduces the overhead related to channels (see Eq. 3) as no synchronization is required and communication can be done directly on the processor via registers or local memory. It also decreases the execution overhead by reducing the total number of $\tau^A$ terms in Eq. 1.

\begin{center}
\begin{tikzpicture}
\node [coordinate] (a) at (0,0){RF};
\node [coordinate] (b) at (1.5,0){FFT};
\node [coordinate] (c) at (3,0){DEMAP};
\node [coordinate] (d) at (4.5,0){DENTRL};
\node [coordinate] (e) at (6,0){CHDEC};
\node [coordinate] (f) at (7.5,0){SNK};
\node [coordinate] (g) at (0.75,-1.5){CHEST};

\draw (a) to (b);
\draw (b) to (c);
\draw (c) to (d);
\draw (d) to (e);
\draw (e) to (f);
\draw (d) to (g);
\end{tikzpicture}
\end{center}

Fig. 5: SDFG of MIMO OFDM

\begin{center}
\begin{tikzpicture}
\node [coordinate] (a) at (0,0){VLD};
\node [coordinate] (b) at (1.5,0){IQ};
\node [coordinate] (e) at (4,0){DCT};
\node [coordinate] (f) at (5.5,0){DISP};
\node [coordinate] (g) at (3.25,-1.5){I};

\draw (a) to (b);
\draw (b) to (g);
\draw (g) to (f);
\draw (c) to (e);
\end{tikzpicture}
\end{center}

Fig. 6: SDFG of MJPEG

\subsection{B. Multi-Application (MA) Scenario}

The optimization concept above can be generalized for a MA scenario as described in this section. Let us suppose a set of concurrent applications is given $\{A_1, \ldots, A_n\}$. Each application $A_i$ is presented in the format of an SDFG $\xi_{A_i}$. We apply the chromosome encoding described earlier on the set of actors $\alpha = \alpha_{\xi_{A_1}} \cup \alpha_{\xi_{A_2}} \cdots \cup \alpha_{\xi_{A_n}}$ to perform concurrent exploration of instances of each application graph $\xi_{A_i}$. This encoding represents a concatenation of multiple chromosomes of each single application (Fig. 4). For multiple application, we target finding the most fair way to utilize the platform treating equally all the applications. We achieve this with the objective function represented as a product:

$$\prod_{i=1}^{n} T_{\xi_{A_i}}$$

(5)

The absolute workload per periodic phase can significantly differ among $\xi_{A_i}$. In [19] it is shown that the objective function of this form is suitable to handle such situations. Each $T_{\xi_{A_i}}$ is computed according to Eq. 4. The only difference is that in the denominator we do not find the maximum workload among all the processors but only among those to which at least one actor of application $A_i$ is mapped.

\section{VI. Experimental Evaluation}

\textbf{Experimental Setup.} Experimental evaluation of our methodology has been done with two benchmarks, MIMO OFDM from digital baseband processing, and MJPEG from the multimedia domain. MIMO OFDM is a state-of-the-art wireless receiver for a 4x4-antenna system [5] that has been developed in the context of Software Defined Radio, shown in Fig. 5. A Motion JPEG (MJPEG) is a video decoder based on the image processing (Fig. 6). Its reference implementation was obtained from [20].

Our target MPSoC is a virtual prototype platform designed with the Synopsys Platform Architect [21]. It consists of two types of programmable cores: a 32-bit RISC processor and a 4-slot VLIW machine with custom instructions as described in [5]. Both processors are modeled by cycle accurate Instruction Set Simulators. Each processor has a 256 KB local data memory and a bus interface that connects them to a 32 MB shared memory using an AMBA AXI bus. In our experiment we use 3 platform configurations $Conf1$, $Conf2$ and $Conf3$ with 2, 3 and 4 cores of each type. The computation of actors is implemented as C functions, while each actor node from the SDFG is represented by an OS thread. We use a light-weight OS with hardware accelerated scheduler [22] and multi-threading support to arbitrate actors on the platform. The buffers are implemented in the shared memory using OS semaphores for synchronizations.

\textbf{Single Application.} In Fig. 7a - 7b we present a comparison of the average execution time that is required to complete one periodic phase of the original SDFG (inverse throughput) for three following cases. We first plot the solution of GA formulation without transformations (GA in Fig. 7a - 7b) i.e. optimization is achieved via optimal mapping. This
solution is compared against cases with SDF transformations for our GA formulation (GAxforms) and for two stages approach (Flexstream) with replication heuristic and optimal ILP mapping [13]. Applying transformations significantly reduces computation time. For MIMO OFDM we obtain 2.4x, 3.7x, 5.6x speedup with the proposed technique for Conf1, Conf2, Conf3 respectively. For Flexstream the speed up is 2x, 2.7x and 2.9x for the same platforms. For MJPEG the reduction is lower than for MIMO OFDM 2.2x, 2.3x, 2.7x with our GAxforms and 1.9x, 2.5x, 2.5x with Flexstream. The stateful actor VLD that performs demanding decompression operations does not benefit from replication. It becomes a bottleneck when replication is applied to IDCT and limits optimization.

Our technique outperforms Flexstream in most of the test cases, with up to 2x difference for MIMO OFDM. The reason is that the Flexstream replication heuristic always replicated only one actor CHDEC. During the first few iterations it got mapped to the VLIW core on which it has much worse runtime than on the RISC core. The heuristic proceeds with load balancing by always replicating this actor as it has the highest cost. In the solution obtained by GAxforms also FFT has been replicated. The advantage of replicating is mainly due to its relatively high cost, and its ability to exploit the instruction level parallelism of VLIW cores. For MJPEG both approaches found very similar solutions. The maximum difference in the execution time is up to 27% and mainly due to the fact that GAxforms takes merging into account. The gain from merging can be larger for OS with high multi-threading and synchronization costs. There is also a small percentage contributed to the difference between solutions found with both transformation approaches that is due to the replication of various actors. In our approach not only the most demanding actors were replicated. It improved further the load balancing as workload is better subdivided among the cores.

**Accuracy of Throughput Estimation.** Fig. 7a - 7b also show the estimate of the execution time obtained by the analytical model from Section V-A for all three solutions: GA, GAxforms and Flexstream. While we observed deviation from the measured result by 7-30%, in most cases the trend is predicted correctly. The error is mainly due to the simplicity of communication estimation. Dynamic effects such as bus congestion are not captured by the presented formulas. Therefore, the error is higher for larger number of cores (Conf2 or Conf3) and for MJPEG application due to its high communication to computation ratio.

**MA Scenario.** In the second experiment we evaluated MA case with both applications running concurrently. We compared our approach to the Flexstream technique. The replication has been done by the same heuristic on both SDFG simultaneously while mapping has been done by the GA using the objective function from Eq. 5. Our approach balances the load for both applications while Flexstream optimized MIMO OFDM at the cost of MJPEG performance degradation. This is due to the workload difference that can not be handled by the replication heuristic. The MJPEG periodic phase takes 5x less time than MIMO OFDM. In all measurements both applications were activated at the same time. We study only the effects of simultaneous MA execution and not the dynamic switching between the MA scenarios.

**GAxforms Runtime.** The termination criterion for the GA in all experiment was an accumulative difference of no more than 1% in the result of the objective function for 100 consecutive generations. With such criterion all results for our experiments were found by the GA within an interval of 2-8 seconds on an AMD Phenom host processor running at 3.2 GHz. The implementation was done in Matlab. We have also assessed the runtime of the optimization on several synthetic benchmarks. The average runtime is shown in Table Ia for the same termination criterion: 0.01 tolerance on 100 generations. For these experiments we generated several graphs (varying execution time and topology) with number of actors: 10 (row 1), 50 (row 2) and 100 (row 3). And we picked platform configurations with 10, 50 and 100 PEs (columns 1, 2 and 3 respectively). As we can see, experiments on the graphs with

<table>
<thead>
<tr>
<th>Actors</th>
<th>Processors</th>
<th>GAxforms</th>
<th>Flexstream</th>
<th>GAxforms est</th>
<th>Flexstream est</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>9.0</td>
<td>21.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>7</td>
<td>26.0</td>
<td>231.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>8</td>
<td>152.0</td>
<td>938.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Termination: 0.01 tol + 100 generations

<table>
<thead>
<tr>
<th>Actors</th>
<th>Processors</th>
<th>GAxforms</th>
<th>Flexstream</th>
<th>GAxforms est</th>
<th>Flexstream est</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.03</td>
<td>0.04</td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0.03</td>
<td>0.05</td>
<td>0.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.05</td>
<td>0.09</td>
<td>0.26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Termination: 100 generations

TABLE I: GA runtime in seconds for various problem sizes
100 actors and a platform with 100 processors showed that up to 15 minutes are needed in average for an exploration with such criterion. We consider this runtime to be, still, acceptable as the problem size is quite large. The GA runtime is more sensitive to the number of PEs (than to the number of actors) as it determines the population size. The population size affects the amount of computations performed per generation.

We have also performed a study for the case the GA was terminated after 100 generations without any tolerance (Table Ib). The runtime is then significantly lower, especially for the large problem size. We have, however, noticed the reduction in the solution quality found with this criterion as shown in Table II. The runtime of the optimization did not depend much on the number of applications and buffers as they only contribute to a small part of fitness function computation, and constraints evaluation.

### TABLE II: Performance reduction in % between the solutions found with and without the tolerance

<table>
<thead>
<tr>
<th>Actors</th>
<th>10</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>11</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>100</td>
<td>32</td>
<td>39</td>
<td>40</td>
</tr>
</tbody>
</table>

### VII. CONCLUSIONS

We presented a methodology for increasing the throughput of streaming applications compliant to SDF when mapped to heterogeneous MPSoCs. The SDF transformations to adapt parallelism and the optimization procedure have been kept framework independent and can be integrated into various software compilation flows. Experiments indicate that the approach is suitable for single applications and scenarios in which applications are executed concurrently. Our throughput computation model is currently restricted to SDFG with self-cycles only. An evaluation of our optimization technique on SDFGs with general cycles would be potentially interesting future work. One negative impact that our transformations have is an increase in the consumption/production rates of the transformed SDFGs. This can result in large buffer sizes for an application implementation. We plan to address this issue in our future work.

### REFERENCES


### VIII. GLOSSARY

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\xi$</td>
<td>SDFG</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>a set of actors (nodes of graph $\xi$)</td>
</tr>
<tr>
<td>$\alpha_i$</td>
<td>$i$-th actor (i-th node of graph $\xi$)</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>a set of buffers (edges of graph $\xi$)</td>
</tr>
<tr>
<td>$\gamma_i$</td>
<td>$i$-th buffer (i-th edge of graph $\xi$)</td>
</tr>
<tr>
<td>$\pi$</td>
<td>a set of PEs</td>
</tr>
<tr>
<td>$\pi^\psi$</td>
<td>$i$-th PE of the type $\psi$</td>
</tr>
<tr>
<td>$\epsilon_{\gamma,i}$</td>
<td>one firing execution time of $i$-th actor on processor type $\psi$</td>
</tr>
<tr>
<td>$\delta$</td>
<td>tokens sizes for the buffer set $\gamma_i$</td>
</tr>
<tr>
<td>$\delta_{\gamma,i}$</td>
<td>a token size for the buffer $\gamma_i$</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>cost of an elementary data unit transfer for a communication primitive</td>
</tr>
<tr>
<td>$\psi$</td>
<td>a processor type</td>
</tr>
<tr>
<td>$\psi_i$</td>
<td>$i$-th gene in the chromosome</td>
</tr>
<tr>
<td>$\mu$</td>
<td>an actor-PE assignment map</td>
</tr>
<tr>
<td>$\mu_{\psi,i}$</td>
<td>a set of actors assigned to $i$-th PE</td>
</tr>
<tr>
<td>$S_{\gamma,i}^\psi$</td>
<td>a synchronization cost of each read from a buffer on a processor type $\psi$</td>
</tr>
<tr>
<td>$T_{\xi_i}$</td>
<td>a throughput of $i$-th application</td>
</tr>
<tr>
<td>$A_{\xi_i}$</td>
<td>a set of actors assigned to $i$-th gene in the chromosome</td>
</tr>
</tbody>
</table>