Task-FIFO Co-Scheduling of Streaming Applications on MPSoCs with Predictable Memory Hierarchy

Qi Tang*, Twan Basten†‡, Marc Geilen†, Sander Stuijk† and Ji-Bo Wei*
*Department of Electronic Science and Engineering
National University of Defense Technology, Changsha, Hunan, China
Email: t.qi@tue.nl, wjbhw@nudt.edu.cn
†Department of Electrical Engineering
Eindhoven University of Technology, Eindhoven, Netherlands
Email: a.a.basten@tue.nl, m.c.w.geilen@tue.nl, s.stuijk@tue.nl
‡Embedded Systems Innovation, TNO, Eindhoven, Netherlands

***Abstract*—Multi-processor systems-on-chips are widely adopted in implementing modern streaming applications to satisfy the ever increasing computing requirements. Predictable memory hierarchies, which make memory access predictable, can better satisfy the strict timing requirements of streaming applications. However, different levels of the memory hierarchy vary in latency and capacity. Hence, the system performance not only depends on the task schedule but also closely relates with the FIFO size distribution and FIFO allocation, which makes the scheduling problem much more complex. We propose an efficient Iteration-based Task-FIFO Co-Scheduling algorithm to optimize the FIFO size distribution and task/FIFO assignment. Randomly generated Synchronous Dataflow Graphs with different sizes and a set of practical applications are used to evaluate the performance of the proposed method. The experimental results demonstrate that the proposed algorithm outperforms the load balancing method and the Highest Access Frequency First algorithm.**

I. Introduction

Modern streaming applications are computation-intensive and have strict timing requirements. Since a large proportion of the systems that run these applications are powered by battery, Multi-Processor Systems-on-Chips (MPSoCs) are widely used to obtain a better trade-off between the computational capacity and power consumption. To diminish the effect of the memory wall, contemporary processors are equipped with multi-level memory architectures. For example, the memory pyramid of a typical processor consists of registers, L1 cache, L2 cache and off-chip memory [1]. In such a memory architecture, the cache plays a critical role in hiding the delay of the off-chip memory with higher latency. Nevertheless, the cache is complex in structure, resulting in larger chip size, more power consumption and lower access speed [2]. Besides, the occurrence of cache misses makes the memory access unpredictable. To overcome these problems, predictable memory hierarchies [3], which for example use scratch pad memories (SPMs) [2] rather than caches, are gathering more and more attention. Many processors, such as CompSoC [4], SB3500, CELL and Fermi, also use SPMs to improve performance or/and provide predicability. In contrast to caches, SPMs need to be explicitly controlled by the compiler or programmer, making it critical to design appropriate algorithms to make full use of it.

This work was partially supported by China Scholarship Council (CSC).

Streaming applications, such as video en/decoding, voice processing, communication protocols, software defined radio, generally operate on large or indefinite sequences of data items [5]. To schedule and analyze the performance of streaming applications on a platform, a computation model is required. The expressivity of Synchronous Data Flow (SDF) [6], [7] fits well with the features of streaming applications and hence SDF graphs (SDFGs) are widely used in literature [8], [9], [10] and also in this paper. In an SDFG, tasks communicate by the FIFO allocated to each edge. The throughput of the SDFG or application is influenced by the size of the FIFOs, i.e., FIFO size distribution [11]. On MPSoCs with predictable memory hierarchy, memories on different levels differ in capacity and latency, so the FIFO size distribution and FIFO allocation have an important impact on the system performance. Besides, the task allocation also has significant influence on task parallelism and hence the throughput, making it important to combine these aspects in the schedule. In this paper, we investigate how to exploit the MPSoC with predictable memory hierarchy that comprises SPMs and off-chip memory to optimize the schedule of streaming applications modeled by SDFGs, such that the throughput is maximized.

Though a lot of literature researches SDFG scheduling on MPSoCs [12], [13], [14] and data allocation on multiple memory banks/modules [15], [16], [17], these works have not considered SDFG scheduling with FIFO sizing and allocation. We propose the Iteration-based Task-FIFO Co-Scheduling (ITFCS) algorithm to schedule streaming applications while taking into account the memory hierarchy. The novel contribution of this paper is a method that solves SDFG scheduling on MPSoCs with a predictable memory hierarchy without assuming that FIFOs all fit in the local SPM. Besides, in our method, the instance collocation rule [18], [19] that binds each task to only one processor is obeyed. This rule provides many advantages [18], e.g., simplifying data management, reducing memory consumption and avoiding graph transformation.

The proposed method is implemented in SDF3 [20], and is evaluated by a set of practical applications and randomly generated SDFGs. The effectiveness of the proposed method is demonstrated by comparing the throughputs generated by our method and other algorithms, including load balancing [8], [21] and Highest Access Frequency First [17].

In the remainder of this paper, we use the following notations. $\mathbb{Z}$, $\mathbb{Z}^+$ and $\mathbb{Z}_0^+$ denote the set of integers, positive integers and non-negative integers respectively. We use boldface capitals to denote vectors/sets and corresponding italic lowercase letters to denote elements in them. For a vector or set, we use $|\cdot|$ to denote the number of its elements.

The remainder of the paper is organized as follows. In Section II, we discuss the related work. The models and definitions are described in Section III. In Section IV we formalize the problem to be solved. The algorithm is elaborated in Section V and the experimental results are presented in Section VI. We conclude the paper in Section VII.

II. Related Work

Scheduling of Directed Acyclic Graphs (DAGs) on multiprocessors is extensively studied in [12] and [22] that focus on scheduling without and with communication overhead respectively. These works form the basis for SDFG scheduling. However, SDFGs [6], [7], which are also called Weighted Marked Graphs in Petri Net theory [23], differ significantly from DAGs, because they can support both cyclic dependencies between tasks and multi-rate dependencies. So, SDFGs are more complex than DAGs and the DAG scheduling method can not be applied to SDFGs directly. In [12], SDFGs are scheduled by converting them into equivalent homogeneous SDFGs (HSDFGs) and further transforming the generated HSDFGs into Acyclic Precedence Graphs (APGs) [12]. Clustering is used in [13], [14] to reduce the scheduling complexity by clustering tasks in an SDFG into various groups, each of which is an indivisible scheduling element. After clustering the SDFG into a new consistent SDFG with smaller graph size, this new SDFG is transformed into an APG on which DAG scheduling algorithms are used. Because the SDFG is converted to an APG in the above methods, instances of the same task may be allocated to multiple processors in the schedule and thus the instance collocation rule [19] is violated. So, these methods can not be applied to our problem. In [8], [21], load balancing is utilized to determine the assignment of the SDFG by balancing the computation load, communication bandwidth and memory consumption. In these works, instances of the same task are bound to the same processor. However, the memory hierarchy is not taken into account. Since their task allocation strategy fits well with the problem in this paper, we adapt it to our problem as a comparison.

The above works have not taken memory capacity and latency into account, except for [8] that tries to balance the memory consumption while binding the tasks. Recently, memory allocation, including data and code allocation, captures much attention. Reference [15] investigates allocating data to dual banks of a single-processor so that instruction parallelism is optimized. The authors proposed interference graphs to model the data parallelism and transform the problem into partitioning the interference graph into two parts. This is a max-bisection problem and is solved by ILP. This work, however, applies only to single-processor and dual-bank memory; moreover, it can not be applied to SDFGs used in this paper. The authors of [16] extended the work in [15], proposing variable independence graphs to model the data parallelism more accurately and applying it to multiple memory modules. The result still suffers the drawbacks of [15]. [17] researches how to schedule tasks and partition data onto multiprocessors with virtually shared SPMs. The authors proposed Highest Access Frequency First (HAFF) to allocate variables to SPMs for a given schedule. We adapt HAFF to our problem and use it as a comparison in this paper to justify the importance of FIFO allocation and the effectiveness of our ITFCS algorithm. While the above works use DAGs, recently, [24], [25] investigate similar problems for SDFGs. [24] researches how to use the SPM of a single-processor machine to overlay the task code of the application modeled by an SDFG such that the execution cost of the application is minimized. This work is extended by [25] to multiprocessors and data overlay. However, these two works consider dynamic memory overlay, differing from FIFO allocation considered in this paper, which is in fact a kind of static allocation.

III. Models and Definitions

In this section, we introduce the platform model and application model for specifying the MPSoC and streaming application. Besides, some relevant concepts and definitions about the SDFG are also presented.

A. Platform Model

In this paper, we investigate how to schedule streaming applications on MPSoCs with a predictable memory hierarchy, of which the memory access time is predictable. We typically consider the kind of predictable memory hierarchy that is comprised of SPMs and off-chip memory, which are all controlled by the compiler or programmer. CompSoC is a platform of this kind. The platform model is presented by Definition 1.

Definition 1. ***(Platform Model)*** *An MPSoC with a predictable memory hierarchy is modeled as a five-tuple:* $\text{PM} = (\boldsymbol{T}, OCM, NIC, ini, d, s)$*, where* $\boldsymbol{T}$ *is a finite set of tiles,* OCM *is the off-chip memory, and* NIC *is the interconnect. Each tile* $t \in \boldsymbol{T}$ *is a pair:* $t = (p, spm)$*, where* p *is the processor and* spm *is the SPM. We use* $\boldsymbol{M} = \{spm_0, spm_1, ..., spm_{|\boldsymbol{T}|-1}, OCM\}$ *to represent the shared memory that is available to all processors on the platform and* $\boldsymbol{P} = \{p_0, p_1, ..., p_{|\boldsymbol{T}|-1}\}$ *to represent the set of processors on the platform.* ini, d *are mappings,* $ini : \boldsymbol{P} \times \boldsymbol{M} \rightarrow \mathbb{Z}_0^+$*,* $d : \boldsymbol{P} \times \boldsymbol{M} \rightarrow \mathbb{Z}^+$*.* $ini(p, m)$ *represents the set up time (in clock cycles) for one memory access and* $d(p, m)$ *represents the memory access latency (in clock cycles per word) of processor* p *for memory* m*. We use the linear model* $ini + d * n$ *to represent the memory access time (in clock cycles), where* n *is the data size.* s *is a mapping,* $s : \boldsymbol{M} \rightarrow \mathbb{Z}^+$*, representing the memory capacity (in words).*

For the above platform, each processor is also equipped with an instruction and data memory and one direct memory access (DMA) controller. The instruction and data memory are used for hosting code and internal data, e.g., heap and stack, respectively. The DMA controller is used to move data between local SPM and other SPM or off-chip memory on the platform, such that the computation can occur in parallel with memory accesses. Each processor can access the memories on the same tile directly, the off-chip memory and the SPM on other tiles by the use of the DMA. We call the access of local SPM local access, the access of SPM on other

tiles remote access, and the access of off-chip memory off-chip access. Generally speaking, the latencies of local access, remote access and off-chip access differ an order of magnitude. Fig.1 illustrates one example platform consisting of two tiles. Each tile comprises an instruction memory IMEM, a data memory DMEM, a scratch pad memory SPM and a direct memory access controller DMA. For the example, for the sake of simplicity, we assume that each SPM has size 24, set up time zero, and we assume that the latencies of remote access and off-chip access are one and two clock cycles per word respectively. In this paper, we use the above platform configuration to elaborate the example.

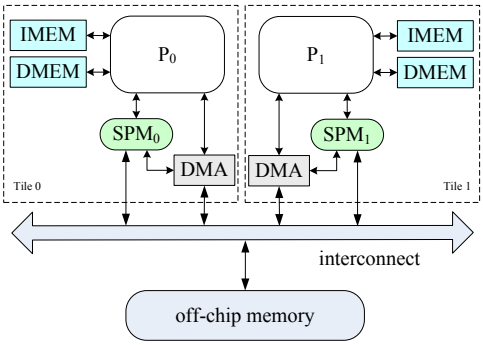

Fig. 1. The structure of the example MPSoC.

B. Application Model

DAGs are widely used in literature to model applications. Recently, data flow graphs like SDFG and Scenario Aware Dataflow Graph (SADFG) [10], [26], [27], [9] gain much attention due to their powerful combination of expressivity and analyzability. In this paper, we use SDF to model streaming applications like software defined radio and multimedia applications. SDFGs can capture application execution features, e.g., multi-rate, and also provide some useful analytical properties, e.g., deadlock, repetition vector and memory requirement analysis, making it more attractive than DAG and other data flow models. The SDFG model is presented by Definition 2.

Definition 2. ***(SDFG)*** *A synchronous data flow graph is a directed graph and is denoted by $G = (\boldsymbol{V}, \boldsymbol{E})$, where $\boldsymbol{V}$ is a finite set of nodes or vertices representing tasks or actors of an application, and $\boldsymbol{E}$ is a finite set of directed edges denoting the communications between tasks. Each node $v \in \boldsymbol{V}$ is associated with a cost $c(v)$ representing the number of clock cycles needed to complete an execution of the task. Each edge $e \in \boldsymbol{E}$ is defined as a tuple $(src, p, dst, q, iniTok, tokSiz)$, where src is the source task, p is the production rate, dst is the destination task, q is the consumption rate, $iniTok$ is the initial token number on the edge and $tokSiz$ is the token size (in words). For a given edge e, we use the notions $src(e)$, $p(e)$ etc., to denote its elements. When the source task $src(e)$ finishes its execution, it produces $p(e)$ tokens on the edge and the destination task $dst(e)$ consumes $q(e)$ tokens from the edge when it is invoked. We also refer to edge e as the output edge of task $src(e)$ and the input edge of task $dst(e)$.*

An edge whose source task and destination task are the same is called a self-edge, representing a constraint on auto-concurrency. In this paper, we do not consider FIFO requirements of self-edges. However, it is straightforward to incorporate it in our method. To simplify the elaboration, in the remaining part of this paper, when we refer to SDFG, except when stating otherwise explicitly, it means an SDFG without self-edges. Though, it should be kept in mind that auto-concurrency is not possible in our approach, due to the instance collocation rule, i.e., each task is bound to one and only one processor and all its instances should be executed on it.

Fig. 2 shows one example SDFG with five tasks. We use this SDFG to elaborate the problem solved in this paper. We assume that each task in Fig. 2 has an execution time of three and each edge has a token size of one.

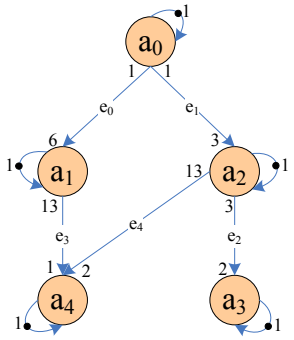

Fig. 2. The structure of the example SDFG.

Generally speaking, scheduling is the process of mapping tasks onto the platform, ordering the executions of tasks bound to the same processor and determining task start/finish times. If communication latency and contention are considered, edge scheduling [28], [22] or memory accesses should also be incorporated into the scheduling. However, SDFGs differ significantly from DAGs. For DAGs, the schedule objects are tasks and edges which appear only once in the schedule, while for SDFGs, multi-rate causes tasks to execute with different frequencies and thus appear different numbers of times in the schedule. We use the notions of SDFG iteration and repetition vector to capture these features of SDFGs.

Definition 3. ***(SDFG iteration)*** *An SDFG iteration is defined as the process of executing each task the minimum positive number of times so that the token count on each edge returns to the initial value.*

Definition 4. ***(Repetition vector)*** *The repetition vector $\boldsymbol{R}$ of an SDFG with n tasks numbered from 0 to $n-1$ is a column vector of length n, and the k-th element of $\boldsymbol{R}$ equals the number of instances of task k in an iteration. For a task v, we refer to the number of its instances by $\boldsymbol{R}(v)$.*

The repetition vector can be calculated by solving the balance equations [6], [7]. For a DAG, each entry of **R** equals one. For an SDFG or HSDFG, **R** exists only when the balance equations of the graph have non-zero solutions [6], [7]. This so-called SDFG consistency can be easily verified [7]. In this paper, we only consider consistent SDFGs that do not deadlock and have non-zero repetition vector. A consistent SDFG can always be transformed into an equivalent HSDFG [12]. However, this conversion might result in exponential increase in graph size. The repetition vector of the SDFG in

Fig. 2 is $[6, 1, 2, 3, 13]^T$, meaning that in one iteration tasks a_0, a_1, a_2, a_3 and a_4 have to execute $6, 1, 2, 3$ and 13 times respectively.

The throughput of the SDFG depends on the maximum token number or the FIFO size of each edge. Since the token size of each edge may differ, we use FIFO size in this paper. We also use FIFO size distribution to denote the size of each FIFO. Different distributions vary in throughput, which can be captured by the FIFO-throughput Pareto space, each point of which records the distribution and its throughput.

Definition 5. ***(FIFO size distribution)*** *The FIFO size distribution of an SDFG $G = (\boldsymbol{V}, \boldsymbol{E})$ is a mapping, $fs : \boldsymbol{E} \rightarrow \mathbb{Z}^+$, representing the size (in words) of the FIFO allocated to each edge. We also use $fifo(e)$ and $fs(e)$ to represent the FIFO and FIFO size of edge $e \in \boldsymbol{E}$.*

It should be noted that an application iteration also implies data access with respect to FIFOs of the input/output edges of each task. Since we use FIFOs for inter-task communication, the number of FIFO writes/reads is the same as its source/destination task occurrence in an iteration for each FIFO.

IV. Problem Formalization

In this paper, we investigate how to schedule an SDFG on an MPSoC while exploiting the predictable memory hierarchy. The problem is stated as follows:

Given a streaming application modeled by an SDFG and the platform configuration modeled according to Definition 3, find the optimal FIFO size distribution, task and FIFO allocation, such that the throughput is maximized.

In the above problem, the tasks in an SDFG interact by the use of FIFOs that are allocated to each edge. The FIFO size is not given a priori and should be determined by the algorithm. Since different memories differ in capacity and latency, the throughput not only depends on the FIFO size distribution, but also closely relates with the FIFO allocation. Besides, the task assignment has an important impact on the task parallelism, making proper allocation a necessity. Therefore, all the above aspects should be considered such that we can obtain an optimal throughput bound.

In this paper we make the following assumptions. We assume that the capacity of the off-chip memory is large enough such that there is enough memory to accommodate all the FIFOs. Besides, the IMEM and DMEM on each tile are assumed to be large enough to accommodate the code and stack/heap of all the tasks bound to the processor on this tile. Moreover, for each tile, we assume that there is enough reserved space in the local SPM for one invocation of each task assigned to it. The reserved memory is not taken into account in the platform model and our algorithm.

V. Iteration-based Task-FIFO Co-Scheduling

In this section we introduce the Iteration-based Task-FIFO Co-Scheduling (ITFCS) algorithm to find near-optimal solutions for the problem stated in Section IV. This algorithm comprises four steps and one iteration. Fig. 3 outlines the framework of our algorithm. Given an application modeled by an SDFG, the FIFO-throughput Pareto space is computed first. Then, one FIFO size distribution is selected, and the FIFO Allocation Aware Task Assigning (FAATA) algorithm is used to find the task binding. Subsequently, we use the Global FIFO Allocation Optimizing (GFAO) algorithm to optimize the FIFO assignment. Finally, the Earliest Task First Scheduling (ETFS) algorithm is used to find the task and memory access ordering and timing. In the following subsections, each step is elaborated.

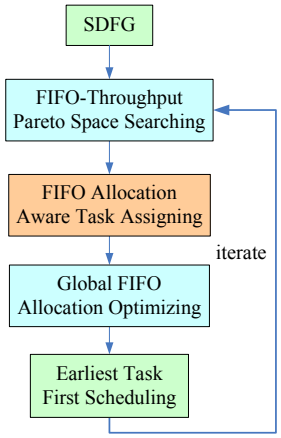

Fig. 3. The framework of the Iteration-based Task-FIFO Co-Scheduling Algorithm.

A. FIFO-Throughput Pareto Space Searching

References [11], [29] show that the FIFO size distribution of an SDFG has significant impact on the throughput. We use non-shared FIFO allocation [11] in which the FIFO assigned to each edge exclusively occupies a memory block. This kind of FIFO usage pattern simplifies the FIFO management and complies with the concept of modular programming. In our algorithm, each task in the SDFG is extended with a self-edge with one initial token to avoid auto-concurrency when doing FIFO-throughput Pareto space searching. This is because each task is allocated to one and only one processor in our method, which makes auto-concurrency impossible. So, by adding self-edges, the FIFO size distributions found are more accurate than without self-edges. Reference [11] shows that there is a positive correlation between throughput and total FIFO size. However, the memory capacity constraints and access latency are not taken into account, making this result inapplicable to our problem. For example, assuming that increasing the size of one FIFO can improve the throughput under the model in [11], if this makes the FIFO too large to reside in the SPM, then it should be allocated to the off-chip memory, which would worsen the performance. Our experiments also demonstrate the above. So, rather than using the technique introduced in [11] to obtain the FIFO size distribution leading to maximum throughput without resource constraint, we use it to find all possible Pareto points and perform iterative analysis on them.

Table I shows two FIFO size distributions of the SDFG in Fig. 2. From the table, we can see that the total FIFO size and the throughput of the first distribution are 4.35% and 7.69% larger than the second one respectively. It shows that the throughput increases with the total FIFO size in ideal

$$F(i,j) = \begin{cases} 0, & if\ i = 0\ or\ j = 0 \\ F(i-1,j), & if\ fs(e_i) > j,\ i > 0,\ j > 0 \\ max\{F(i-1,j), F(i-1,j-fs(e_i)) + p(src(e_i)) * \mathbf{R}(src(e_i)) * tokSiz(e_i)\}, & else \end{cases} \quad (1)$$

TABLE I. FIFO SIZE DISTRIBUTIONS OF THE EXAMPLE SDFG

Distribution index	Edges					Total size	Throughput
	e_0	e_1	e_2	e_3	e_4		
1	8	6	4	14	16	48	0.025641
2	7	5	4	13	17	46	0.0238095

scenarios. However, as shown later, this is not the case when considering the concrete platform in which memories vary in capacity and latency.

B. FIFO Allocation Aware Task Assigning

In this subsection, we introduce the FIFO Allocation Aware Task Assigning algorithm that is shown in Algorithm 1. The goal of this algorithm is to find the task assignment to tiles, determining to which processor each task should be assigned. Rather than only considering computational parallelism, which can only guarantee the performance without inter-task communication, the potential FIFO allocation is also taken into account such that a good balance can be made between computation and memory access. To combine these two aspects and make the algorithm FIFO allocation aware, we propose a coefficient called localization coefficient to balance the computation load and local communication. The estimated potential local communication cost is computed according to the recursive Equation 1 by a dynamic programming technique. By utilizing it, a good compromise is reached. Our experiments also demonstrate the effectiveness of it.

Algorithm 1 FIFO Allocation Aware Task Assigning Algorithm

Input: application model $G(\mathbf{V}, \mathbf{E})$, platform model $PM = (\mathbf{T}, OCM, NIC, ini, d, s)$ and FIFO size distribution $fs : \mathbf{E} \rightarrow \mathbb{Z}^+$.
Output: task to processor assignment $proc : \mathbf{V} \rightarrow \mathbf{P}$.

```
construct Resource-Aware SDFG.
compute task priority according to Equation 2.
sort the tasks in non-increasing order of priority, obtain task list Q.
while Q ≠ ∅ do
    pop-up the first element in Q, denote it as a.
    for i = 0 to |P| − 1 do
        tentatively assign task a to processor p_i.
        compute the computation load l(p_i) of p_i according to Equation 3.
        compute the estimated potential local communication cost eplcc(p_i) according to Equation 1.
        compute the metric value metric(p_i) from l(p_i) and eplcc(p_i) according to Equation 4.
    end for
    proc(a) ← arg min_{p_i ∈ P} {metric(p_i)}.
end while
```

In Algorithm 1, we first construct a Resource-Aware SDFG (RASDFG) [29] based on the application model and the FIFO size distribution generated by FIFO-throughput Pareto space searching. The RASDFG is created by adding FIFO size constraining edges and self-edges to the original SDFG. For each edge $e \in \mathbf{E}$ of the original SDFG, we add a FIFO size constraining edge $(dst(e), q(e), src(e), p(e), fs(e) - iniTok(e), tokSiz(e))$ to $\mathbf{E}$, and add a self-edge $(v, 1, v, 1, 1, 1)$ for each task $v \in \mathbf{V}$. The generated RASDFG corresponding to Fig. 2 is shown in Fig. 4 with the FIFO size distribution one in Table I. The generated RASDFG is strongly connected. Since the throughput of an SDFG is limited by its critical cycle of the corresponding HSDFG [12], [8], we use the estimated maximum cycle mean (MCM), which is also used in [8], to compute the task priority. The estimated MCM of task v is computed by Equation 2,

$$emcm(v) = \max_{C_v \in \mathbf{C}_v} \frac{\sum_{v' \in \mathbf{V}_c} \mathbf{R}(v') * c(v')}{\sum_{e' \in \mathbf{E}_c} iniTok(e')/q(e')} \quad (2)$$

where $\mathbf{C}_v$ represents the set of cycles that include task v, and $\mathbf{V}_c, \mathbf{E}_c$ are the sets of tasks and edges of cycle C_v respectively.

For example, the priorities of tasks a_0, a_1, a_2, a_3 and a_4 of the RASDFG in Fig. 4 are $18, 42, 45, 15$ and 45 respectively. Task assigning is performed according to non-increasing order of task priority. Each time we pop-up the first task in the ordered task list and find the best processor where it should be assigned. We introduce a composed metric to evaluate the attractiveness of each processor. The metric is composed by normalized computation load and normalized estimated potential local communication cost. The computation load is computed by Equation 3,

$$l(p_i) = \sum_{v \in \mathbf{V}_i} \mathbf{R}(v) * c(v) \quad (3)$$

where $\mathbf{V}_i$ represents the set of tasks that are assigned to processor p_i tentatively.

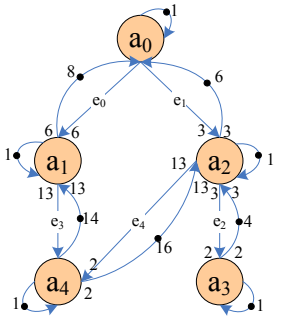

Fig. 4. The RASDFG of the example SDFG.

The estimated potential local communication cost $eplcc(p_i)$ of processor p_i is calculated according to the recursive Equation 1 by a dynamic programming technique. The value of $eplcc(p_i)$ represents the maximum achievable intra-processor

communication and indicates the priority of mapping the task to processor p_i. For processor p_i, we use $\mathbf{V}_i$ to denote the set of tasks that are assigned to it tentatively and use $\mathbf{E}_i$ to denote the set of edges between these tasks. The edges are indexed from 1 to $|\mathbf{E}_i|$. We use the cost matrix F in Equation 1 to store the intermediate results of the maximum local communication cost. Each element $F(i,j)$ of the cost matrix F represents the maximum cost when trying to assign the first i FIFOs to the local SPM with size j. The element of F is initialized to be zero when the FIFO number or the SPM size is zero. After recursively computing the remaining elements of $F(i,j), i \in [1,|\mathbf{E}_i|], j \in [1,s_i]$, by Equation 1, we obtain the estimated potential local communication cost $eplcc(p_i) = F(|\mathbf{E}_i|, s_i)$, where s_i is the size of the SPM residing on the same tile where processor p_i belongs to. As shown in Equation 1, if the FIFO size of edge e_i is larger than the current SPM size j, then this FIFO can not reside in the SPM and thus the cost is the same as $F(i-1,j)$; on the other hand, the cost is the maximum of two costs. The first is $F(i-1,j)$ which means the FIFO of edge e_i is not assigned to the SPM. The second is the sum of $F(i-1, j-fs(e_i))$ and the access cost of the FIFO, meaning that the FIFO of edge e_i is assigned to the SPM. The access cost of a FIFO is the total access operations on this FIFO. Since the access operations by the source and destination tasks are the same and the memory reading and writing are assumed to have the same access latency, only the operations triggered by the source task are counted.

After obtaining the computation load and estimated potential local communication cost, a metric value for each processor p_i is computed by Equation 4, in which $c \in [0,1]$ is the localization coefficient used to weight the computation and local communication.

$$\begin{aligned} metric(p_i) =& (1-c) * \frac{l(p_i)}{\max\limits_{p_j \in \mathbf{P}} l(p_j)} \\ &+ c * (1 - \frac{eplcc(p_i)}{\max\limits_{p_j \in \mathbf{P}} eplcc(p_j)}), c \in [0,1] \end{aligned} \tag{4}$$

The complexity of the FAATA algorithm is $O(|\mathbf{P}||\mathbf{V}|S_{fifo}S_{spm})$, where $|\mathbf{P}|$ is the number of processors, $|\mathbf{V}|$ is the number of tasks, S_{fifo} is the total size of all the FIFOs and S_{spm} is the maximum size of the SPMs.

For the SDFG in Fig. 2, the throughput obtained by load balancing and our method are shown in Table II. While distribution one has a higher throughput than distribution two in the ideal analysis, as shown in Table I, the latter one performs better on our example platform, with the throughput increasing by 19.15% and 16.00% when using load balancing and our algorithm respectively. It shows that a good FIFO distribution in the ideal analysis does not necessarily perform well on a practical platform, proving the necessity to search the FIFO-throughput Pareto space. The awareness of FIFO allocation, which is captured by our method, also improves the performance, with the throughput increasing by 28.73% under the former distribution and 25.33% the latter.

The task allocation with the second distribution by using load balancing and our method are shown in Table III with

TABLE II. THROUGHPUT COMPARISON OF LB AND ITFCS.

Method	Distribution number		
	1	2	
Load balancing	1/112	1/94	**19.15%**
ITFCS	1/87	1/75	**16.00%**
	28.73%	**25.33%**	Thr impr

TABLE III. TASK ASSIGNMENT OF LB AND ITFCS.

Method	Proc	Ordered tasks				
		a_2	a_4	a_1	a_0	a_3
Load balancing	p_0	**1.00**	1.00	**0.21**	**0.47**	**0.75**
	p_1	1.00	**0.87**	1.00	1.00	1.00
ITFCS	p_0	**1.00**	**0.80**	0.80	0.80	0.80
	p_1	1.00	0.89	**0.25**	**0.43**	**0.61**

bold font, and the corresponding schedules are shown in Fig. 5. When the load balancing method is used, a_2, a_1, a_0 and a_3 are allocated to p_0 and a_4 is allocated to p_1. However, if the FIFO Allocation Aware Task Assigning algorithm is used, the task allocation is changed, with a_2 and a_4 being allocated to p_0 and a_1, a_0 and a_3 being allocated to p_1. Because the ordered task list is a_2, a_4, a_1, a_0, a_3, so a_2 is assigned first, processor p_0 is selected arbitrarily since both the processors are the same. Then, a_4 is to be allocated. When using load balancing, a_4 is allocated to p_2 to balance the computation load. However, by using the method proposed in this paper, a_4 is still allocated to p_1 since there is an edge, i.e., e_4, between a_2 and a_4, as shown in Fig. 2. The existence of e_4, as shown in Table III, changes the attractiveness of p_0 defined by metric of Equation 4 from 1.00 to 0.80 while p_1 increases from 0.87 to 0.89, so, a_4 is allocated to p_0. This new allocation, as shown in Table II and Fig. 5, decreases the schedule length from 94 to 75, with the throughput improving by 25.33%.

In Fig. 5, both the schedules are illustrated by Gantt charts. The executions of different tasks are represented by rectangles of different colors and background patterns, and each remote memory access is represented by the rectangle of the same color and background pattern with the task that triggers this memory access. In Fig. 5(a), *fifo*(e_1) and *fifo*(e_4) are assigned to spm_1, and the others are assigned to spm_0. For this allocation, the overhead of transferring the output data of a_2 is quite large, which increases the schedule length. In Fig. 5(b), *fifo*(e_2) and *fifo*(e_4) are assigned to spm_0, and the others are assigned to spm_1. For this allocation, task a_2 and task a_4 are allocated to the same processor and *fifo*(e_4) is assigned to the local SPM, so the overhead of transferring the output data of a_2 and the schedule length is smaller.

C. Global FIFO Allocation Optimizing

Given the FIFO size distribution and task assignment, we try to optimize the FIFO allocation by the Global FIFO Allocation Optimizing algorithm. As demonstrated by our experiments, the system throughput depends strongly on the FIFO allocation. We design a dynamic programming based Global FIFO Allocation Optimizing algorithm shown in Algorithm 2 to find the optimal FIFO allocation by minimizing the total memory access cost. While Algorithm 1 considers only potential local memory access, the algorithm introduced in this subsection takes into account not only local memory access, but also remote and off-chip memory accesses. Besides, the memory access cost used in this subsection is different from

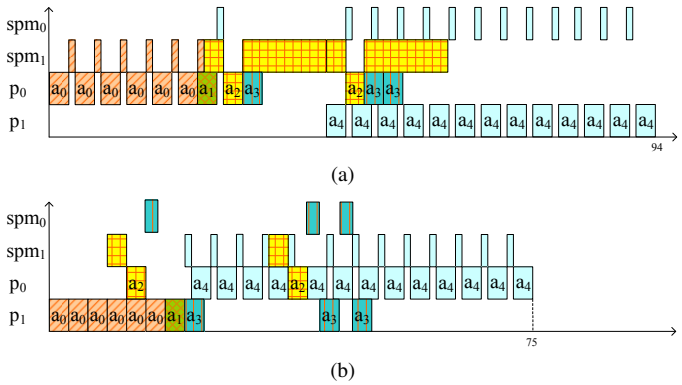

Fig. 5. Schedules of the example SDFG with distribution 2 by load balancing (a) and ITFCS (b).

Algorithm 2 Global FIFO Allocation Optimizing Algorithm

Input: application model $G(\mathbf{V},\mathbf{E})$, platform model $PM = (\mathbf{T}, OCM, ini, d, s)$, FIFO size distribution $fs : \mathbf{E} \to \mathbb{Z}^+$.
Output: FIFO allocation.

```
define vector L, L_i = s(m_i), ∀i ∈ [0, |M| − 1].
let Δ = ∅, add root node (|E| − 1, L) to Δ.
constructDPTree(|E| − 1, L, Δ).
trace Δ and find the FIFO allocation.

function constructDPTree(i, L, Δ) :
for j = 0 : |M| − 1 do
  if fs(e_i) ≤ L_j then
    let L' = L.
    let L'_j = L'_j − fs(e_i).
    if node (i − 1, L') ∉ Δ then
      add node (i − 1, L') to Δ.
      if i > 1 then
        constructDPTree(i − 1, L', Δ).
      else
        compute the cost of (0, L') by Equation 5.
      end if
    end if
    add directed edge ((i, L), (i − 1, L')).
    compute the edge weight fac(i, j) by Definition 6.
    store the FIFO mapping indicated by the edge:
    fifo(e_i) → m_j.
  end if
end for
compute the cost of (i, L) by Equation 6.
```

Algorithm 1 by considering memory access latency. The FIFO access cost is defined by Definition 6.

$$cost(0,\mathbf{L}) = \min_{j\in[0,|\mathbf{M}|-1],\mathbf{L}_j\geq fs(e_0)} \{fac(0,j), +\infty\} \quad (5)$$

Definition 6. ***(FIFO Access Cost)*** *The FIFO access cost $fac(i,j)$ is the summation of the access costs of the source task and destination task of edge e_i to FIFO $fifo(e_i)$ when it is allocated to memory m_j. The value depends on where $fifo(e_i)$, the source task and destination task of edge e_i, i.e., $src(e_i)$ and $dst(e_i)$, are allocated. If $fifo(e_i)$, $src(e_i)$ and $dst(e_i)$ are allocated to the same tile, then $fac(i,j) = 0$. If $fifo(e_i)$ is allocated to the off-chip memory or a tile different from both tiles where $src(e_i)$ and $dst(e_i)$ are assigned, then the cost is computed by Equation 7. If $fifo(e_i)$ and $src(e_i)$ are assigned to the same tile, while $dst(e_i)$ is assigned to another tile, then the cost is computed by Equation 8. If $fifo(e_i)$ and $dst(e_i)$ are assigned to the same tile, while $src(e_i)$ is assigned to another tile, then the cost is computed by Equation 9. In these equations ini is the number of clock cycles needed to initialize the memory access. For the off-chip memory, if DRAM is used, then the refresh time should be added to ini so that hard real-time requirements can be guaranteed.*

We use backward dynamic programming to solve the FIFO allocation problem. In Algorithm 2, a dynamic programming tree or the FIFO allocation tree Δ is constructed first by the use of recursive function $constructDPTree$. Each path starting from the root of Δ and ending at its leaf represents one possible FIFO allocation strategy, which is dictated by the edges of the path. The cost of each node is computed by the use of Definition 6 while constructing the tree. At last, the optimal FIFO allocation is determined by tracing Δ from root to leaf node. The cost of each node $(i,\mathbf{L})$ in Δ represents the minimal memory access cost when allocating the first i FIFOs to the memory hierarchy with the memory sizes represented by the vector $\mathbf{L}$, the index of which is the index of the memory. The cost of each node is recursively calculated based on its child nodes and the edges between them, as shown in Equation 6, being aware that $(i-1,\mathbf{L}-\mathbf{L}')$ is the child node of $(i,\mathbf{L})$ and $fac(i,j)$ is the weight of the edge between them. After recursively computing the cost of each node in Δ, the minimal cost of FIFO allocation is obtained, with the value equaling the cost of the root node, and the best FIFO allocation strategy can be found by tracing from the root to the leaf nodes of Δ. The tracing process is as follows. Set the root node as the current node. Then iterate the following process until the allocation of all FIFOs are determined. Find the current node's child node, whose cost sums with the weight of the edge between it and the current node equals the cost of the current node, then the FIFO allocation is determined by this edge. Store the above result, set this child node as the current node and iterate the above process. If the current node is a leaf node, e.g., $(0,\mathbf{L})$, then $fifo(e_0)$ is allocated to memory m_j, for which $fs(e_0) \leq \mathbf{L}_j$ and $cost(0,\mathbf{L})$ equals $fac(0,j)$.

Fig. 6 shows one example of GFAO. Suppose we have two memories, i.e., m_0, m_1, and three FIFOs, i.e., $fifo(e_0), fifo(e_1), fifo(e_2)$. The sizes of the memories and FIFOs and the corresponding FIFO access costs are shown in the left side of the graph. The right side is the FIFO allocation tree constructed by the GFAO algorithm. Each node and edge of the FIFO allocation tree has a label. The pair of the label denotes the node or FIFO allocation (the first element is the FIFO index and the second is the index of the memory to which the FIFO is allocated), and the number below the pair is the cost of the node or the weight of the edge. For this example, the root node $(2,[6,8])$ is constructed first, and then the tree is built recursively. For the root node, the child node $(1,[1,8])$ is added since $fs(e_2)$ is smaller than 6. For node $(1,[1,8])$, it only has one child node, since $fs(e_1)$ is larger than 1. Then the leaf node $(0,[1,6])$ is added. Its cost is computed by Equation 5, i.e., 2. The edge between node $(1,[1,8])$ and node $(0,[1,6])$ is added. The temporary allocation of $fifo(e_1)$ is recorded and

$$cost(i,\mathbf{L}) = \min_{j\in[0,|\mathbf{M}|-1],\mathbf{L}_j\geq fs(e_i),\mathbf{L}'=\mathbf{0},\mathbf{L}'_j=\mathbf{L}_j} \{cost(i-1,\mathbf{L}-\mathbf{L}') + fac(i,j), +\infty\} \tag{6}$$

$$\begin{aligned} fac(i,j) = {} & \mathbf{R}(src(e_i)) * (ini(p_s,m_j) + p(e_i) * tokSiz(e_i) * d(p_s,m_j)) \\ & + \mathbf{R}(dst(e_i)) * (ini(p_d,m_j) + q(e_i) * tokSiz(e_i) * d(p_d,m_j)) \end{aligned} \tag{7}$$

$$fac(i,j) = \mathbf{R}(dst(e_i)) * (ini(p_d,m_j) + q(e_i) * tokSiz(e_i) * d(p_d,m_j)) \tag{8}$$

$$fac(i,j) = \mathbf{R}(src(e_i)) * (ini(p_s,m_j) + p(e_i) * tokSiz(e_i) * d(p_s,m_j)) \tag{9}$$

the weight of this edge is computed by Definition 6, i.e., $(1,1)$ and 3. At this time, all the children of node $(1,[1,8])$ have been constructed. So the cost of this node is computed, i.e., 5. After that, an edge between $(2,[6,8])$ and $(1,[1,8])$ is added and labeled. Proceed with the above process and we can obtain the FIFO allocation tree. Trace the tree and we can find the path with the edges in red dash line that results in the cost of the root node. So the final FIFO allocation is as follows: $fifo(e_0), fifo(e_1) \rightarrow m_0$, $fifo(e_2) \rightarrow m_1$.

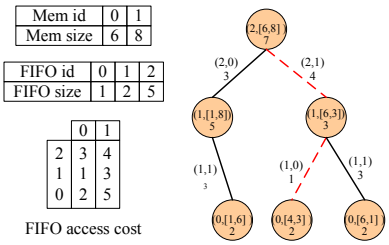

Fig. 6. The example of GFAO.

Algorithm 3 Earliest Task First Scheduling Algorithm

Input: task and FIFO allocation.
Output: task and memory access schedule.

```
let Rem = R, denoting the remaining number of invocations of each task in the SDFG.
while Rem != 0 do
    ready task list A = ∅.
    for each task v in V do
        if Rem(v) > 0 && input FIFOs of v are ready then
            tentatively schedule memory accesses of v.
            tentatively schedule v.
            store the earliest start time of v.
            push back v to A.
        end if
    end for
    select the task v' in A with the minimum start time.
    schedule task v' and its memory accesses.
    Rem(v') ← Rem(v') − 1.
end while
```

D. Earliest Task First Scheduling

In this subsection we introduce the algorithm used to construct the final schedule, including task and memory access ordering and timing. The method used is Earliest Task First by taking into account the FIFO size constraints, memory access latency and contention. As shown in Algorithm 3, each time we tentatively schedule one task instance, if available, and find its earliest start time. The instance with the minimal start time is selected and scheduled. The memory accesses required by the task instance are scheduled at the same time. The above process is stopped when a full iteration is finished. The insertion strategy, which has the potential to improve the performance, is used in our method. By the above way, a blocked schedule [12] is obtained, and we use the reciprocal of the schedule length of it as the throughput.

VI. Experiments and Results

In this section, we evaluate the proposed algorithm experimentally by comparing our method with the load balancing method [8] and the HAFF algorithm [17]. The load balancing method used for comparison binds tasks of the SDFG to processors while balancing the computation load. The tasks are bound in non-increasing order of the task priority defined as the estimated maximum cycle mean of any execution cycle containing that task. The HAFF algorithm assigns FIFOs in non-increasing order of FIFO access frequency. In the remaining section, we first introduce the platform configuration and the benchmark used in the experiments. Then, the algorithm performance is evaluated in terms of throughput.

A. Platform Configuration and Benchmark

We use two sets of MPSoCs in our experiments. One is with two processors and the other is with four. We assume the processors and memories are interconnected by network-on-chip (NOC) [4], [30] and the NOC links are entirely reserved for our application. The memory access delay is primarily composed of the NOC delay and the memory response delay. Data traversing the NOC link can be pipelined, so the delay equals $4+n$, assuming that the link and router delay are both one cycle [30] and all routes are of two hops, where n is the data size. The memory response delay depends on the memory type. For the SPM, the response delay is generally one cycle [2], so, the memory access delay of a remote SPM is $4+2n$. For the off-chip memory, considering DDR3 with a transfer rate of 128M words/s, the memory response delay is 8 cycles. Assuming the refresh time of the off-chip memory is 56 cycles and there is no memory access contention, the worst-case memory access delay is $60+9n$. So, in our experiments, we configure the set up time and latency of the remote SPM

as 4 and 2, and that of the off-chip memory as 60 and 9.

We use a set of practical applications and randomly generated SDFGs with different sizes as benchmarks for performance evaluation. The practical applications include the H.263 decoder [8], H.263 encoder [31], samplerate conversion [32], bipartite [32], and the MPEG-4 SP decoder [10]. The open source tool SDF3 [20] is used to generate random SDFGs of 5, 10 and 15 tasks. The repetition vector is also randomly generated with a constraints on the sum of the repetition vector entries. In our experiments, the constraint is set to be five times the number of tasks. Hence, there are about 25 to 75 task instances in one application iteration. The graph properties such as in-degree/out-degree and edge production/consumption rate are all randomly generated given the corresponding average value, minimum/maximum value and variation. The in-degree and out-degree are both given the average value and variation of 2, the minimum value of 0 and the maximum value of 4. The production and consumption rates are given the average and variation of 5 and 7, the minimum value of 1 and the maximum value of 9. For each graph size, 100 random graphs are generated. The SDFG parameters, including task execution time and edge token size, are randomly generated. The task execution time is uniformly distributed between 400 and 1000, and the token size is randomly generated under the constraint of communication and computation ratio which is defined as the ratio between the number of memory access operations and the total task execution time.

B. Results of Random Applications

Tables IV and V show the experimental results of randomly generated applications on the MPSoC with two and four processors respectively. The first column represents the task number of the randomly generated SDFGs. The column "Iteration" represents the throughput improvement by iterating the FIFO-throughput Pareto space over using the FIFO distribution with the maximum ideal throughput. The column "FAATA vs LB" represents the throughput improvement by the use of FAATA over the load balancing method. The column "GFAO vs HAFF" represents the throughput improvement by the use of GFAO over HAFF.

From Table IV, we can see that the iteration strategy can improve the throughput substantially for SDFGs of different sizes, with the average improvement ranging from 13% to 28% for benchmarks with different task number. It demonstrates that the FIFO size distribution affects the system performance and it is of great importance to set the FIFO size of each edge correctly to achieve a higher throughput. The column "FAATA vs LB" shows that the algorithm FAATA outperforms the load balancing method. The throughput improvement of FAATA over LB is about one third for all our benchmarks, proving that it is important to consider the potential FIFO allocation when assigning the tasks. The column "GFAO vs HAFF" shows that GFAO can produce better FIFO allocation than HAFF. The system throughput, by the use of GFAO, improves 18% to 30% for different application sets.

From Table V, we can see that the iteration strategy is also effective for platforms with more processors, with the improvement amounting to about 12% for different sets of SDFGs. However, the value is smaller than that in Table IV,

TABLE IV. THROUGHPUT IMPROVEMENT OF RANDOM BENCHMARKS ON THE MPSOC WITH TWO PROCESSORS.

Task Num	Iteration	FAATA vs LB	GFAO vs HAFF
5	28%	36%	18%
10	19%	41%	30%
15	13%	41%	21%

because the number of memories is larger in this experiment and hence the influence of FIFO size distribution is reduced. The column "FAATA vs LB" shows that the algorithm FAATA outperforms the load balancing method. The throughput improvement of FAATA over LB amounts to about one third for different benchmarks. The effectiveness of GFAO, however, is far smaller, with the throughput improvement of GFAO over HAFF reaching only 2% for the benchmark with 5 tasks. Because the processor number is comparable to the task number, hence the tasks are assigned to processors uniformly, in such a case, the HAFF and GFAO would produce similar allocations. For SDFGs with 10 and 15 tasks, the throughput improvement of GFAO reaches 8% and 13% respectively.

TABLE V. THROUGHPUT IMPROVEMENT OF RANDOM BENCHMARKS ON THE MPSOC WITH FOUR PROCESSORS.

Task Num	Iteration	FAATA vs LB	GFAO vs HAFF
5	12%	33%	2%
10	13%	35%	8%
15	10%	37%	13%

C. Results of Practical Applications

Table VI shows the experimental results of practical applications on the MPSoC with two processors. The iteration strategy has significant performance improvement except the H.263 encoder. This is because different FIFO allocations of the H.263 encoder have minimal difference, making the iteration strategy ineffective. FAATA is effective for all real applications except the H.263 encoder and MPEG-4 SP decoder. One common feature of these two applications is that several tasks and the related FIFOs dominate the total execution time and memory requirement. In such a case, the load balancing method seems to produce a quite good task assignment. From the last column of Table VI it is shown that for real applications the FIFO allocation has more impact on the system performance. For the applications we use, the throughput improvement of CFAO over HAFF ranges from 45% to 93%, a value being far larger than that of randomly generated applications. This implies that the performance is application structure dependent. It needs further research how to incorporate the application structure into the algorithm.

TABLE VI. THROUGHPUT IMPROVEMENT OF REAL APPLICATIONS ON THE MPSOC WITH TWO PROCESSORS.

Task Num	Iteration	FAATA vs LB	GFAO vs HAFF
H.263 decoder	31%	13%	45%
H.263 encoder	0.02%	0.23%	86%
samplerate conversion	38%	7%	93%
bipartite	51%	21%	56%
MPEG-4 SP decoder	21%	0.5%	88%

VII. CONCLUSIONS

In this paper, we investigate the problem of scheduling streaming applications on multi-processor systems-on-chips with predictable memory hierarchy by taking into account

memory access latency and memory capacity constraints. We propose an efficient Iteration-based Task-FIFO Co-Scheduling (ITFCS) algorithm, consisting of FIFO-Throughput Pareto Space Searching, FIFO Allocation Aware Task Assigning (FAATA), Global FIFO Allocation Optimizing (GFAO) and Earliest Task First Scheduling (ETFS) algorithms, to solve the problem defined in this paper. Extensive experiments are carried out on both random SDFGs and practical applications. Experimental results show that our method outperforms the load balancing method and Highest Access Frequency First (HAFF) algorithm. Useful directions for future work include improving the scalability such that the method can be efficiently applied to large-scale problems, modeling the task/memory schedule in the SDFG to enable accurate throughput analysis, and using the schedule result as feedback for searching the FIFO distribution.

ACKNOWLEDGMENT

The authors would like to thank Yonghui Li for the fruitful discussions about the platform model, and the reviewers for their constructive remarks.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*. ACM, 2002, pp. 73–78.

[3] B. Cilku and P. Puschner, "Towards a time-predictable hierarchical memory architecture-prefetching options to be explored," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium on*. IEEE, 2010, pp. 219–225.

[4] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad *et al.*, "Virtual execution platforms for mixed-time-criticality systems: The compsoc architecture and design flow," *ACM SIGBED Review*, vol. 10, no. 3, pp. 23–34, 2013.

[5] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction*. Springer, 2002, pp. 179–196.

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

[7] E. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. 100, no. 1, pp. 24–35, 1987.

[8] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 777–782.

[9] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Throughput-constrained dvfs for scenario-aware dataflow graphs," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 175–184.

[10] M. Geilen, "Synchronous dataflow scenarios," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 2, p. 16, 2010.

[11] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 899–904.

[12] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2012.

[13] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for dsp applications," in *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, vol. 1. IEEE, 1995, pp. 122–126.

[14] J. L. Pino and E. A. Lee, "Hierarchical static scheduling of dataflow graphs onto multiple processors," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 4. IEEE, 1995, pp. 2643–2646.

[15] R. Leupers and D. Kotte, "Variable partitioning for dual memory bank dsps," in *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, vol. 2. IEEE, 2001, pp. 1121–1124.

[16] Q. Zhuge, B. Xiao, and E. H.-M. Sha, "Variable partitioning and scheduling of multiple memory architectures for dsp," in *Parallel and Distributed Processing Symposium, International*, vol. 2. IEEE Computer Society, 2002, pp. 0130–0130.

[17] L. Zhang, M. Qiu, W.-C. Tseng, and E. H.-M. Sha, "Variable partitioning and scheduling for mpsoc with virtually shared scratch pad memory," *Journal of Signal Processing Systems*, vol. 58, no. 2, pp. 247–265, 2010.

[18] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Static scheduling of multi-rate and cyclo-static dsp-applications," in *VLSI Signal Processing, VII, 1994.,[Workshop on]*. IEEE, 1994, pp. 137–146.

[19] O. Moreira and H. Corporaal, *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor*. Springer, 2014.

[20] S. Stuijk, M. Geilen, and T. Basten, "Sdf3: Sdf for free." in *ACSD*, vol. 6, 2006, pp. 276–278.

[21] J. A. Ambrose, I. Nawinne, and S. Parameswaran, "Latency-constrained binding of data flow graphs to energy conscious gals-based mpsocs," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 1212–1215.

[22] O. Sinnen, *Task scheduling for parallel systems*. John Wiley & Sons, 2007, vol. 60.

[23] J. L. Peterson, *Petri net theory and the modeling of systems*. Prentice-hall Englewood Cliffs (NJ), 1981, vol. 132.

[24] W. Che and K. S. Chatha, "Scheduling of synchronous data flow models on scratchpad memory based embedded processors," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2010, pp. 205–212.

[25] J. Choi, H. Oh, S. Kim, and S. Ha, "Executing synchronous dataflow graphs on a spm-based multicore architecture," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 664–671.

[26] M. Damavandpeyma, S. Stuijk, M. Geilen, T. Basten, and H. Corporaal, "Parametric throughput analysis of scenario-aware dataflow graphs," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE, 2012, pp. 219–226.

[27] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Schedule-extended synchronous dataflow graphs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 10, pp. 1495–1508, 2013.

[28] O. Sinnen and L. Sousa, "List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures," *Parallel Computing*, vol. 30, no. 1, pp. 81–101, 2004.

[29] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "Iteration-based trade-off analysis of resource-aware sdf," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 2011, pp. 567–574.

[30] R. A. Stefan, A. Molnos, and K. Goossens, "daelite: A tdm noc supporting qos, multicast, and fast connection set-up," *Computers, IEEE Transactions on*, vol. 63, no. 3, pp. 583–594, 2014.

[31] H. Oh and S. Ha, "Fractional rate dataflow model for efficient code synthesis," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 37, no. 1, pp. 41–51, 2004.

[32] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 21, no. 2, pp. 151–166, 1999.