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

Qi Tang, National University of Defense Technology
Twan Basten, Eindhoven University of Technology and TNO Embedded Systems Innovation
Marc Geilen, Eindhoven University of Technology
Sander Stuijk, Eindhoven University of Technology
Ji-Bo Wei, National University of Defense Technology

This paper studies scheduling real-time streaming applications on multi-processor systems-on-chips with predictable memory hierarchy. An Iteration-based Task-FIFO Co-Scheduling framework is proposed for this problem. We obtain FIFO size distributions using Pareto space searching, based on which the task-to-processor mapping is obtained with the potential FIFO allocation being taken into account; then, the FIFO-to-memory allocation is optimized to minimize the total memory access cost; finally, a self-timed throughput analysis method that considers memory and DMA contention is utilized to analysis the throughput. Our methods are validated by a set of synthesized and practical applications on different platforms.

CCS Concepts:•**Computer systems organization** → **Embedded systems;**

Additional Key Words and Phrases: SDFG, FIFO, scratch pad memory, self-timed, scheduling

ACM Reference Format:
Qi Tang, Twan Basten, Marc Geilen, Sander Stuijk and Ji-Bo Wei, 2015. Task-FIFO Co-Scheduling of Streaming Applications on MPSoCs with Predictable Memory Hierarchy. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 0 (2016), 24 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Modern streaming applications belong to real-time applications that often have strict timing requirements. To guarantee the real-time performance, it is a necessity to analyze the worst case performance. However, traditional processors use the cache that is complex in structure, resulting in larger chip size, more power consumption and lower access speed [Banakar et al. 2002]. Besides, the memory access is unpredictable since the occurrence of cache miss, making it hard to estimate the worst case performance. To overcome these problems, the predictable memory hierarchy, which for example uses scratch pad memories (SPMs) [Banakar et al. 2002] rather than caches, are gathering more and more attention. Like cache, SPM is usually implemented using SRAM that provides high access speed. The difference is that the SPM guarantees single-cycle access latency while the cache can not provide such guarantee due to cache miss. 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 them.

Streaming applications, such as video en/decoding, voice processing, communication protocols and software defined radio, generally operate on large or indefinite sequences

Author's addresses: Q. Tang and J.-B. Wei, Department of Electronic Science and Engineering, National University of Defense Technology; T. Basten, Department of Electrical Engineering, Eindhoven University of Technology and TNO Embedded Systems Innovation, Eindhoven, The Netherlands; M. Geilen and S. Stuijk, Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands.

DOI: 0000001.0000001

of data items. To schedule and analyze the worst case performance of streaming applications on a platform, a computation model is required. The expressivity of Synchronous Data Flow (SDF) [Lee and Messerschmitt 1987] fits well with the features of streaming applications and hence SDF graphs (SDFGs) are widely used in literature [Stuijk et al. 2007; Geilen 2010] and also in this paper. In an SDFG, tasks communicate by the FIFO allocated to each edge. The throughput of the SDFG is influenced by the FIFO size, i.e., FIFO size distribution [Stuijk et al. 2006a]. On Multi-Processor Systems-on-Chips (MPSoCs) with a predictable memory hierarchy, memories on different levels differ in capacity and latency, so the FIFO size distribution and FIFO allocation have a great 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 a predictable memory hierarchy that comprises SPMs and global 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 [Sriram and Bhattacharyya 2012; Pino et al. 1995] and data allocation on multiple memory banks/modules [Leupers and Kotte 2001; Zhuge et al. 2002; Zhang et al. 2010], these works have not considered SDFG scheduling with FIFO sizing and allocation. In this paper, we propose an Iteration-based Task-FIFO Co-Scheduling (ITFCS) framework to schedule streaming applications on MPSoCs 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 [Bilsen et al. 1994; Moreira and Corporaal 2014] that binds each task to only one processor is obeyed. This rule provides many advantages [Bilsen et al. 1994], e.g., simplifying data management, reducing memory consumption and avoiding graph transformation. An earlier version of this paper appeared in [Tang et al. 2015]. The current paper extends the scheduling method to construct more compact schedules while taking into account various resource conflicts; besides, the FIFO allocation algorithm is simplified.

The proposed method is implemented in SDF3 [Stuijk et al. 2006b], and is evaluated by a set of practical applications and randomly generated SDFGs. The effectiveness of the proposed method is demonstrated by comparing the throughput generated by the proposed method and other algorithms, including the load balancing algorithm [Stuijk et al. 2007; Ambrose et al. 2013], the Highest Access Frequency First algorithm [Zhang et al. 2010], the blocked schedule based throughput analysis method [Tang et al. 2015] and the MILP-based method [Morteza et al. 2011], .

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 2, we discuss the related work. The models and definitions are described in Section 3. In Section 4 we formalize the problem to be solved. The algorithm is elaborated in Section 5 and the experimental results are presented in Section 6. We conclude the paper in Section 7.

2. RELATED WORK

Scheduling of Directed Acyclic Graphs (DAGs) on multiprocessors is extensively studied in [Sriram and Bhattacharyya 2012] and [Sinnen 2007] that focus on scheduling without and with communication overhead respectively. These works have established the foundation for SDFG scheduling. However, SDFGs [Lee and Messerschmitt 1987]

differ significantly from DAGs in several aspects, e.g., SDFGs support both cyclic dependencies between tasks and multi-rate dependencies. Therefore, SDFGs are more complex than DAGs and the DAG scheduling method can not be applied to SDFGs directly. In [Sriram and Bhattacharyya 2012], SDFGs are scheduled by converting them into equivalent homogeneous SDFGs (HSDFGs) and further transforming the generated HSDFGs into Acyclic Precedence Graphs (APGs) [Sriram and Bhattacharyya 2012]. Clustering is used in [Pino et al. 1995] 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 [Moreira and Corporaal 2014] is violated. Hence, these methods can not be applied to the problem solved in this paper. In [Stuijk et al. 2007; Ambrose et al. 2013], a load balancing method is proposed to determine the task-to-processor 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.

Memory access is a key factor that affects the system performance. Recently, memory allocation, including data and code allocation, has attracted great attention. [Leupers and Kotte 2001] investigates allocating data to dual banks of a single-processor and [Zhuge et al. 2002] extends it to multiple memory modules by proposing the variable independence graph to model the data parallelism more accurately. [Zhang et al. 2010] investigates 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 the proposed ITFCS framework. [Che and Chatha 2010; Choi et al. 2012] investigate how to cope with the SPM for SDFGs. [Che and Chatha 2010] studies 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 [Choi et al. 2012] to multiprocessors and data overlay. [Morteza et al. 2011] proposes an MILP formulation for scheduling with code and data prefetch, assuming the data are moved between the global memory and SPM. However, these works consider dynamic memory overlay, while this paper combines static FIFO allocation and dynamic memory access together.

3. MODELS AND DEFINITIONS

This section introduces the platform model and the application model for specifying the MPSoC and the streaming application. Besides, some relevant concepts and definitions about the SDFG are also presented.

3.1. Platform Model

This paper studies how to schedule streaming applications on MPSoCs with a predictable memory hierarchy, of which the memory access is predictable, thus enabling worst case performance analysis. We typically consider the predictable memory hierarchy that is comprised of SPMs and global memory that are controlled by the compiler or programmer. The platform model is abstracted from CompSoC [Goossens et al. 2013] and is defined as follows.

Definition 3.1. **(Platform Model)** An MPSoC with a predictable memory hierarchy is composed with a set of processors, memories and the interconnect. We model it as

a five-tuple: $PM = (\mathbf{T}, glb, NIC, ini, d, s)$, where $\mathbf{T}$ is a finite set of tiles, glb is the global memory, and NIC is the interconnect. Each tile $t \in \mathbf{T}$ is a pair: $t = (p, spm)$, where p is the processor and spm is the SPM. We use $\mathbf{M} = \{spm_0, spm_1, ..., spm_{|\mathbf{T}|-1}, glb\}$ to represent the shared memory that is available to all processors on the platform and $\mathbf{P} = \{p_0, p_1, ..., p_{|\mathbf{T}|-1}\}$ to represent the set of processors on the platform. ini, d are functions, $ini : \mathbf{P} \times \mathbf{M} \rightarrow \mathbb{Z}_0^+$, $d : \mathbf{P} \times \mathbf{M} \rightarrow \mathbb{Z}^+$. $ini(p, m)$ 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 function, $s : \mathbf{M} \rightarrow \mathbb{Z}^+$, representing the memory capacity (in words).

In the platform, each tile comprises an instruction memory (IMEM), a data memory (DMEM), a scratch pad memory (SPM) and a direct memory access controller (DMA). The IMEM and DMEM are used for hosting code and internal data. The DMA is used to move data between the local SPM remote memories, including SPMs on other tiles or the global memory [Goossens et al. 2013]. The DMA enables parallel computation and memory access. The platform is a distributed shared memory system, where each processor can access the memories on the same tile directly, and remote memories by the use of DMA. Therefore, both the SPM and the global memory in the platform are sharable memories. We call the access of local SPM local access, the access of SPM on other tiles remote access, and the access of global memory global access. Generally speaking, the latencies of local access, remote access and global access differ an order of magnitude. Fig.1 illustrates one example platform consisting of two tiles. We assume that the size of each SPM is 24 words, the set up time is zero, and the latencies of remote access and global access are one and two clock cycles per word respectively. This paper uses this platform configuration to elaborate the example.

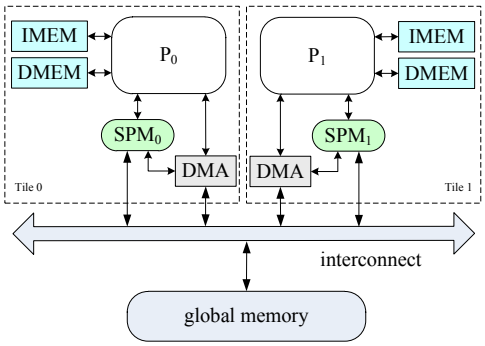

Fig. 1. The structure of the example MPSoC.

The interconnecting architecture is a critical part for the system on chip, especially for data-intensive applications. The Network-on-Chip (NoC) [Benini and Micheli 2002] is a typical interconnect infrastructure. The NoC can be designed to have various shapes, e.g., 2D mesh network [Hu and Marculescu 2004; Goossens et al. 2013]. The NoC has higher bandwidth and scalability compared to a traditional bus-based interconnect. Data transport in the NoC has two styles, i.e., contention-free or connection oriented [Stefan et al. 2014]. Connectionless packet switching NoCs typically do not offer latency and bandwidth guarantees, providing only best-effort service. While the connection-oriented, circuit-switching NoCs, e.g., the dAElite [Stefan et al. 2014], establish the route before data transportation, thus providing guaranteed bandwidth

and latency. This paper considers connection-oriented, circuit-switching NoCs such that the timing behaviour of data transport can be accurately analyzed. To avoid communication contention, any two communication transactions using routes that partly or totally overlap should mutually exclude in transportation time, meaning that only one communication transaction is allowed to transfer on conflict routes at each time. Since the NoC uses a connection-oriented, circuit-switching mechanism, communications are guaranteed to be contention-free. Since multiple connections may share the same link, accesses of the link should be arbitrated to avoid contention. We assume that the NoC uses a TDM scheme [Stefan et al. 2014] to allocate the link bandwidth to different connections, thus providing guaranteed bandwidth through assigning each connection with fixed time slices in the TDM time wheel. The transfer delay on a given route depends on the data length and the number of hops of the route. If the link is shared by multiple connections, the delay is also relative with the number of slices allocated to the connection.

3.2. Application Model

DAGs are widely used in literature to model applications. Recently, data flow graphs like SDFG and Scenario Aware Dataflow Graph (SADFG) [Geilen 2010; Damavandpeyma et al. 2013] gained great attention due to their powerful combination of expressivity and analyzability. This paper uses SDFGs to model real-time streaming applications like software defined radio and multimedia applications. SDFGs can capture the application execution feature, e.g., multi-rate execution, and also provide some useful analytical properties, e.g., deadlock, repetition vector and memory requirement analysis, making it more attractive than other computation models.

Definition 3.2. **(SDFG)** A synchronous data flow graph is a directed graph and is denoted by $G = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is a finite set of nodes or vertices representing tasks or actors of an application, and $\mathbf{E}$ is a finite set of directed edges denoting the communications between tasks. Each node $v \in \mathbf{V}$ is associated with a cost $c(v)$ representing the worst case execution time (number of clock cycles) needed to complete an execution of the task. Each edge $e \in \mathbf{E}$ 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 number of initial tokens on the edge and $tokSiz$ is the token size (in words). We use the notions $src(e)$, $p(e)$ etc., to denote elements of each edge e. We also refer to edge e as the output edge of task $src(e)$ and the input edge of task $dst(e)$. When a task starts execution, it consumes $q(e_i)$ tokens from each input edge e_i, and produces $p(e_o)$ tokens to each output edge e_o when it finishes execution.

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 [Sinnen 2007] should also be taken into account. SDFGs are multi-rate computation model, so tasks may execute in different frequencies and fire different numbers of times in the schedule. SDFG iteration and repetition vector can capture these features of the SDFG. The iteration is also related with the throughput. The throughput of an SDFG is defined as the number of iterations finishes per unit of time, representing how fast the input data stream can be processed.

Definition 3.3. **(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 3.4. **(Repetition vector)** The repetition vector $\mathbf{R}$ of an SDFG with n tasks numbered from 0 to $n-1$ is a column vector of length n, and the $k\text{-}th$ element of

$\mathbf{R}$ equals the number of instances of task k in an iteration. For a task v, we refer to the number of its instances by $\mathbf{R}(v)$.

The repetition vector can be calculated by solving the balance equation [Lee and Messerschmitt 1987]. For an SDFG, if the balance equation have non-trial solutions [Lee and Messerschmitt 1987], then $\mathbf{R}$ exists and the SDFG is said to be consistent [Lee and Messerschmitt 1987]. An incorrectly constructed SDFG may be dead-lock while executing. This paper only considers SDFGs that are consistent and deadlock-free. A consistent SDFG can always be converted to an equivalent HSDFG [Sriram and Bhattacharyya 2012] in which all rates equal to one. However, this conversion can result in exponential increase in graph size. The repetition vector provides information as to how many times each task has to be executed in an iteration. Every time the task is started, we say one instance of it is fired. $\mathbf{R}(v)$ is also called the instance number of task k in an iteration.

While executing the SDFG, its edge needs to store the output data. Since the data stored by the edge is consumed by the destination task of the edge in a fist-in-first-out style, the edge is often modeled as a FIFO. The throughput of an SDFG depends on the FIFO size of each edge. Since the token size of each edge may differ, we use FIFO size expressed in words in this paper. We also use the 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 3.5. **(FIFO size distribution)** The FIFO size distribution of an SDFG G = ($\mathbf{V}$, $\mathbf{E}$) is a function, $fs : \mathbf{E} \rightarrow \mathbb{Z}^+$, 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 $e \in \mathbf{E}$.

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.

An edge in the SDFG whose source task and destination task are the same is called a self-edge, representing a constraint on auto-concurrency. This paper does not consider FIFO requirements of self-edges. However, it is straightforward to incorporate it in the method. To simplify the elaboration, in the remaining part of this paper, the SDFG denotes that has no self-edges except when stating explicitly. It should be kept in mind that auto-concurrency is not possible due to the instance collocation rule, i.e., each task is bound to one and only one processor, so all task instances should be executed sequently.

Fig. 2 shows one example SDFG, which is used 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. The repetition vector of the SDFG 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.

4. PROBLEM FORMALIZATION

This paper investigates 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 2, find the optimal FIFO size distribution, task and FIFO allocation, such that the throughput is maximized.

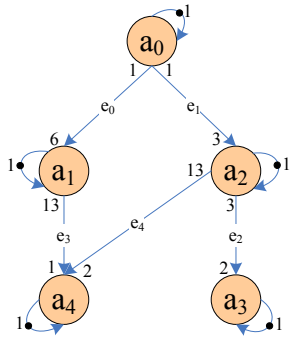

Fig. 2. The structure of the example SDFG.

This paper considers static scheduling on the platform with predictable behaviour. We assume the worst case execution time for both task execution, memory access and communication, therefore, the scheduling anomaly is avoided.

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 global 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.

5. ITERATION-BASED TASK-FIFO CO-SCHEDULING FRAMEWORK

This section introduces the Iteration-based Task-FIFO Co-Scheduling (ITFCS) framework to produce near-optimal solutions for the problem stated in Section 4. Owing to the complexity of the problem, a decomposing methodology that decomposes the problem as several sub-problems is adopted. This scheduling framework comprises four main steps and one iteration. Fig. 3 outlines the components of the scheduling framework. Given an application modeled by an SDFG, the FIFO-throughput Pareto space is found first. Then, the FIFO size distribution is selected, and the FIFO Allocation Aware Task Assignment (FAATA) algorithm is used to find the task-to-processor binding. Subsequently, we use the Global FIFO Allocation Optimization (GFAO) algorithm to optimize the FIFO-to-memory assignment. Finally, a self-timed throughput analysis method is utilized to produce the final periodic static order schedule based on the FIFO size distribution and FIFO/task mapping, and evaluate the throughput. The throughput analysis method comprises of four steps. First, it uses the Memory Access Aware SDFG (MAASDFG) Construction algorithm to model the memory access into the SDFG; second, the Mapping-Aware Earliest Task First (MAATF) scheduling algorithm is used to produce the periodic static order schedules of the tasks and memory accesses; then the task and memory schedule is modeled into the MAASDFG to enable throughput analysis; finally, the self-timed state space search is used to compute the

throughput of the system. Since the exact relation between the FIFO size distribution and the throughput on a hardware platform is unclear, an iteration strategy is utilized to assess the performance of each FIFO size distribution using the proposed scheduling method. The iteration strategy can be used in different ways in practical usage. On the one hand, the iteration can be terminated if the optimum is not a necessity and a typical timing requirement is given. On the other hand, it can be used to obtain the optimal FIFO distribution by exhaustively evaluating the quality of each distribution, which is also used in this paper to show the advantage of considering distributions in the scheduling process rather than the optimum one in terms of ideal throughput. In the following subsections, we elaborate each step of the scheduling framework.

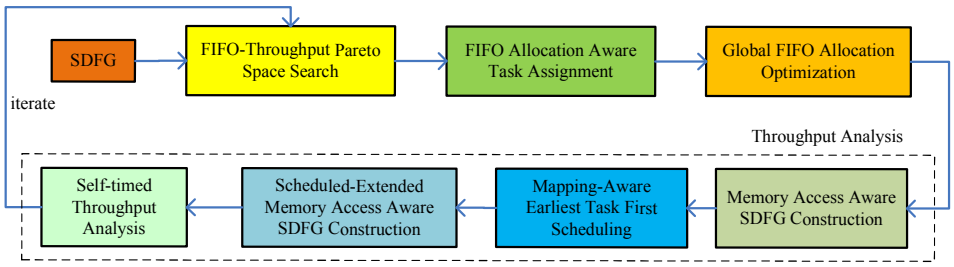

Fig. 3. The Iteration-based Task-FIFO Co-Scheduling framework.

5.1. FIFO-Throughput Pareto Space Search

FIFO sizing is a critical step in implementing streaming applications on embedded systems. For one thing, the FIFO size distribution of an SDFG has significant impact on the timing behavior of the application, e.g., the throughput [Stuijk et al. 2006a; Yang et al. 2011], the achievable throughput of the application is constrained by the size of each FIFO. For another thing, memories, especially high-speed memories, are expensive and power consuming, making it important to configure the platform with appropriate memory sizes and make full use of them while deploying the application onto the platform.

There are two usage patterns with respect to how FIFOs use the memory. One pattern shares the memory space among all FIFOs [Yang et al. 2011], and the other allocates a specific memory space to each FIFO exclusively [Stuijk et al. 2006a]. We use non-shared FIFO allocation [Stuijk et al. 2006a] in which the FIFO assigned to each edge exclusively occupies a memory block, which simplifies the FIFO management and complies with the concept of modular programming.

[Stuijk et al. 2006a] shows that the application throughput is positively related with the FIFO size. However, the memory capacity 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 [Stuijk et al. 2006a], if this makes the FIFO too large to reside in the SPM, then it should be allocated to the global memory, which would worsen the performance. Our experiments also demonstrate the above notion. So, rather than using the technique introduced in [Stuijk et al. 2006a] to obtain the FIFO size distribution leading to maximum throughput without resource constraint, we use it to find all possible Pareto points (a 2-tuple comprising the FIFO size distribution and the corresponding throughput) and perform iterative analysis on them. Since each task is mapped to only one processor, auto-concurrency of any task is impossible while executing the application on

the platform. To take it into account, each task in the SDFG is extended with a self-edge with one initial token to avoid auto-concurrency, as in Fig. 2, before searching the FIFO-throughput Pareto space so as to obtain more accurate FIFO size distributions. Since the technique introduced in [Stuijk et al. 2006a] has pruned the Pareto space, the number of iterations need to be carried out is reduced, thus improves the efficiency. The complexity of the Pareto space search is highly application-related. However, available works [Stuijk et al. 2006a; Yang et al. 2011] have made great contributions to speed up this process. The iteration strategy proposed in this paper sacrifices the computation time. The time complexity of the iteration strategy is roughly the iteration number times that when no iteration is carried out.

Table I shows two FIFO size distributions of the SDFG in Fig. 2. As shown in the table, the total FIFO size (memory requirement) and the throughput of the first distribution are 7.32% and 20% larger than the second distribution respectively, showing that the throughput increases with the total FIFO size. However, as shown later, this is not the case when considering platforms with resource constraints.

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	6	4	4	13	17	44	0.0222
2	6	3	4	13	15	41	0.0185

5.2. FIFO Allocation Aware Task Assignment

Having obtained the FIFO size distribution using the technique introduced in the former section, this subsection proposes the FIFO Allocation Aware Task Assignment (FAATA) algorithm, as shown in Algorithm 1. FAATA aims at generating the task-to-processor assignment and determining to which processor each task should be assigned. Rather than only exploiting computation parallelism as the load balancing method does, the potential FIFO allocation is also taken into account such that a good balance can be made between computation and memory access. The idea behind the algorithm is that a task should be allocated to the processor that makes the load be balanced while leading to more local memory access. The load on a processor is denoted by the normalized load, and the local memory access is denoted by the normalized estimated potential local communication cost. The estimated potential local communication cost is computed according to the recursive Equation 3 by a dynamic programming technique. To combine these two aspects, the localization coefficient is introduced.

Algorithm 1 first constructs a Resource-Aware SDFG (RASDFG) [Yang et al. 2011] based on the application model and the FIFO size distribution generated by the FIFO-throughput Pareto space search method. 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 $(dst(e), q(e), src(e), p(e), \boldsymbol{fs}(e) - iniTok(e), tokSiz(e))$ 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 [Sriram and Bhattacharyya 2012; Stuijk et al. 2007], we use the estimated maximum cycle mean (MCM), as in [Stuijk et al. 2007], to compute the task priority. The estimated MCM of task v is computed by Equation 1,

ALGORITHM 1: FIFO Allocation Aware Task Assignment Algorithm

Input: application model $G(\mathbf{V}, \mathbf{E})$, platform model $PM = (\mathbf{T}, glb, NIC, ini, d, s)$ 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 1 ;
sort the tasks in non-increasing order of priority and obtain task list $\mathbf{Q}$;
repeat
  $a \leftarrow$ pop-up the first element in $\mathbf{Q}$;
  for $i = 0$ *to* $|\boldsymbol{P}| - 1$ **do**
    tentatively assign task a to processor p_i;
    compute the computation load $l(p_i)$ of p_i according to Equation 2;
    compute the estimated potential local communication cost $eplcc(p_i)$ according to Equation 3;
    compute the metric value $metric(p_i)$ from $l(p_i)$ and $eplcc(p_i)$ according to Equation 4;
  end
  $proc(a) \leftarrow \arg\min_{p_i \in \mathbf{P}} \{metric(p_i)\}$;
until $\boldsymbol{Q} = \varnothing$;

$$emcm(v) = \max_{C_v \in \mathbf{C}_v} \frac{\sum_{v' \in \mathbf{V}_c} \mathbf{R}(v') * c(v')}{\sum_{e' \in \mathbf{E}_c} iniTok(e')/q(e')} \tag{1}$$

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.

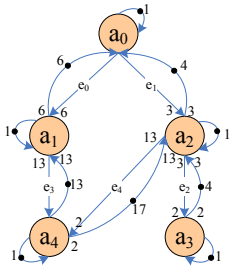

Fig. 4. The RASDFG of the example SDFG.

Task assignment is performed according to non-increasing order of task priority. Each time, pop-up the first task in the ordered task list and find the best processor where it should be assigned. A composed metric is introduced to evaluate the attractiveness of each processor by combining the computation and memory access. The metric is composed by the normalized computation load and the normalized estimated potential local communication cost. The computation load is computed by Equation 2,

$$l(p_i) = \sum_{v \in \mathbf{V}_i} \mathbf{R}(v) * c(v) \tag{2}$$

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

The estimated potential local communication cost $eplcc(p_i)$ of processor p_i is calculated according to the recursive Equation 3 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 3 to store the intermediate results while computing 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 3, 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 3, 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 $F(i-1, j-\boldsymbol{fs}(e_i))$ 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 of all access operations on this FIFO. Since the access operations triggered 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.

$$F(i,j) = \begin{cases} 0, & if\ i = 0\ or\ j = 0 \\ F(i-1,j), & if\ \boldsymbol{fs}(e_i) > j,\ i > 0,\ j > 0 \\ max\{F(i-1,j), F(i-1, j-\boldsymbol{fs}(e_i)) + p(src(e_i)) * \mathbf{R}(src(e_i)) * tokSiz(e_i)\}, & else \end{cases} \tag{3}$$

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. The value of the localization coefficient depends on the application and platform. If the memory access has more importance on the performance, e.g., the application is data-intensive or the SPM size is limited, then more weight should be put on the FIFO allocation. In this paper, the value of the coefficient is obtained by experiment. For Equation 4, we assume $\frac{eplcc(p_i)}{\max\limits_{p_j \in \mathbf{P}} eplcc(p_j)} = 1$ when $\max\limits_{p_j \in \mathbf{P}} eplcc(p_j) = 0$ (which also implies $eplcc(p_j) = 0$ for all $p_j \in \mathbf{P}$). The first part of Equation 4 denotes the load on the processor, and the second part denotes the local communication on this processor. Using the coefficient c the computation and communication is combined.

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

The FAATA algorithm is composed of two loops, one enumerates all the tasks and the other tentatively maps the task to each processor, therefore, there are $|\mathbf{P}||\mathbf{V}|$ iterations. The inner loop is dominated by the computation of $eplcc(p_i)$, which is fulfilled by the dynamic programming technique, so the worst case complexity is $O(S_{fifo}S_{spm})$, where

S_{fifo} is the total size of all the FIFOs and S_{spm} is the maximum size of the SPMs. Therefore, the total complexity is $O(|\mathbf{P}||\mathbf{V}|S_{fifo}S_{spm})$.

For the SDFG in Fig. 2, the throughput values obtained by load balancing task allocation and by our method are shown in Table II. While distribution 1 has a higher throughput than distribution 2 in the ideal analysis, as shown in Table I, the latter one performs better on our example platform, with the throughput increasing by 4.71% and 6.45% 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 34.85% under the former distribution and 37.10% under the latter.

Table II. Throughput comparison of LB and ITFCS

Method	Distribution number		
	1	2	
Load balancing	0.01124	0.01176	**4.71%**
ITFCS	0.01515	0.01613	**6.45%**
	34.85%	**37.10%**	Thr impr

The task allocation with the second distribution when using load balancing and our method are shown in Table III with bold font. 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 Assignment algorithm with the localization coefficient being 0.15 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, 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_1 to balance the computation load. However, by using the method proposed in this paper, a_4 is still allocated to p_0 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 the metric of Equation 4 from 1.00 to 0.85 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, increases the throughput from 0.01176 to 0.01613, with the improvement of 37.10%.

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**	**0.85**	0.85	0.85	0.85
	p_1	1.00	0.89	**0.20**	**0.41**	**0.59**

5.3. Global FIFO Allocation Optimization

Having obtained the FIFO size distribution and task assignment, the FIFO allocation needs to be determined. Even though using the dynamic programming embedded in the FAATA, the allocation of part of the FIFOs can be determined, there are FIFOs that are not local and need to be allocated. So, the Global FIFO Allocation Optimization algorithm is proposed to allocate all FIFOs globally. The Global FIFO Allocation Optimization algorithm is based on the integer linear programming, and it aims at minimizing the total memory access cost. In the ILP formulation, not only the local

memory access, but also the remote and global memory accesses are taken into account explicitly. Besides, the memory access cost used in this subsection is different from Algorithm 1 by considering memory access latency. The computation of FIFO access cost is presented by Definition 5.1.

Definition 5.1. **(FIFO Access Cost)** The FIFO access cost $\mathit{fac}(i,j)$ is the summation of the access costs of the source task and destination task of edge e_i to FIFO $\mathit{fifo}(e_i)$ when it is allocated to memory m_j. The value depends on where $\mathit{fifo}(e_i)$, the source task and destination task of edge e_i, i.e., $src(e_i)$ and $dst(e_i)$, are allocated. If $\mathit{fifo}(e_i)$, $src(e_i)$ and $dst(e_i)$ are allocated to the same tile, then $\mathit{fac}(i,j)=0$. If $\mathit{fifo}(e_i)$ is allocated to the global 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 5. If $\mathit{fifo}(e_i)$ 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 6. If $\mathit{fifo}(e_i)$ 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 7. In these equations ini is the number of clock cycles needed to initialize the memory access.

$$\begin{aligned}\mathit{fac}(i,j) = {} & \mathbf{R}(src(e_i)) * (ini(p_s,m_j) + p(e_i) * tokSiz(e_i) * d(p_s,m_j)) \\ & + \mathbf{R}(dst(e_i)) * (ini(p_d,m_j) + q(e_i) * tokSiz(e_i) * d(p_d,m_j))\end{aligned} \tag{5}$$

$$\mathit{fac}(i,j) = \mathbf{R}(dst(e_i)) * (ini(p_d,m_j) + q(e_i) * tokSiz(e_i) * d(p_d,m_j)) \tag{6}$$

$$\mathit{fac}(i,j) = \mathbf{R}(src(e_i)) * (ini(p_s,m_j) + p(e_i) * tokSiz(e_i) * d(p_s,m_j)) \tag{7}$$

Since the the FIFO allocation problem is a combinatorial optimization problem, it is formulated as an integer linear programming model that can be solved by available solver.

In the ILP model, the binary allocating variable $m_{i,j}$ is used to denote where FIFO i is allocated to. If $m_{i,j}$ equals one, then FIFO i is allocated to memory j; otherwise, it is allocated to another FIFO.

Since each FIFO should be allocated to only one memory, so the following constraint should be satisfied:

$$\sum_{j\in[1,|\mathbf{M}|]} m_{i,j} = 1,\ \forall\, i \in [1,|\mathbf{E}|] \tag{8}$$

The memory capacity constraints should be obeyed, so that the memory can accommodates all FIFOs allocated to it:

$$\sum_{i\in[1,|\mathbf{V}|]} \sum_{j\in[1,|\mathbf{M}|]} m_{i,j} * fs(i) \le s(m_j),\ \forall\, i \in [1,|\mathbf{E}|] \tag{9}$$

Where $fs(i)$ is the size of FIFO i and $s(m_j)$ is the capacity of memory m_j.

Since the aim is to minimize the total memory access cost, the variable C is introduced to represent the sum of the memory access cost of each FIFO, so the following constraint should be satisfied:

$$\sum_{i\in[1,|\mathbf{V}|]} \sum_{j\in[1,|\mathbf{M}|]} m_{i,j} * fac(i,j) \le C \tag{10}$$

So the objective can be represented by the following equation:

$$min:\ C \tag{11}$$

To improve the efficiency of the ILP solver, we use a greedy allocation algorithm to produce an initial solution. The ILP solver can use the initial solution to prune the solution space more efficiently, thus improve the performance. The greedy allocation

algorithm comprises two steps, i.e., FIFO priority assignment and FIFO allocation. In the algorithm, the average cost-size ratio of the FIFO is used as the FIFO priority, which is defined as follows,

$$pri(i) = \frac{\sum_{j \in [1,\mathbf{M}]} fac(i,j)}{fs(i) * |\mathbf{M}|}, \ \forall \ i \in [1, |\mathbf{E}|] \tag{12}$$

Where $fs(i)$ and $pri(i)$ are the FIFO size and the priority of FIFO i, the numerator is the sum of memory access cost on each memory, and the denominator is the multiplication of the FIFO size and memory number. The FIFO size in the denominator is used to compute the cost-size ratio, and the memory number in the denominator is used to average the cost-size ratio.

Having obtained the FIFO priority, the next step is to allocate the FIFO to the memory. The FIFO is allocated in non-increasing order of the priority. For the current selected FIFO that has the highest priority in the set of non-allocated FIFOs, it is allocated to the memory that has sufficient remaining space to accommodate it and makes the cost the minimum.

The time complexity of GFAO is application-depended since ILP solver is used to solve the problem. However, the time complexity is not very high, partly owing to the structural feature of the FIFO access cost. As shown in the experiment, the GFAO can be efficiently applied to the problem, with the extra time burden being negligible.

5.4. Throughput Analysis

Since the objective is to optimize the application throughput, it is a necessity to construct the schedule and analyze the throughput of the application. This section elaborates the methodology for constructing the schedule and analyzing the throughput given the FIFO distribution and the task/FIFO allocation strategy produced by algorithms introduced in former sections.

Given a task and FIFO allocation strategy, it is still hard to find the optimal schedule in the view of throughput. It should be noted that the throughput differs with the execution style. One execution style is the so-called blocked schedule [Sriram and Bhattacharyya 2012], in which a block is composed of one or several iterations. The iteration number in a block is called the blocking factor. In the blocked schedule, different blocks cannot overlap. In a block, different iterations may interleave rather than overlap, hence improving the performance. Using the above execution style, it's important to find the optimal blocking factor to obtain a good performance. However, as the blocking factor increases, the overhead for controlling the system increases. In [Tang et al. 2015], the throughput is calculated using the blocked schedule with the blocking factor being one, which prohibits pipelining executions, thus underestimating the throughput. As an alternative, we assume that the system executes according to a Periodic Static-Order Schedule (PSOS) [Damavandpeyma et al. 2013] that forces the tasks to fire one by one in a given order and makes the system execute according to the self-timed schedule. In this kind of execution, different iterations may overlap, proceeding in a pipelining style. The throughput of the self-timed system can be analysed utilising the state space based method [Ghamarian et al. 2006] or via max-plus algebra [Geilen 2010]. Since the self-timed schedule would finally enter the periodic state after the transient state, the system could run in a time-driven style to guarantee the performance.

The state space based throughput analysis method of SDFGs without resource constraints is proposed in [Ghamarian et al. 2006]. In this method, the token distribution and task execution state specify the state. Since the SDFG is forced to be strongly connected and the range of each element of the state is finite, the state space is finite. Af-

ter a finite number of transitions, some states will be revisited, hence forming a cycle. Having obtained the cycle in the state space, the throughput can be calculated directly. Another method for computing the throughput is to model the self-timed execution of the SDFG in max-plus algebra [Geilen 2010]. The eigenvalue of the max-plus matrix equals the reciprocal of the throughput. The above methods do not take the mapping and schedule into account, making the results inaccurate. It's possible to model the mapping and PSOS in the SDFG [Damavandpeyma et al. 2013] or the IPCG (Inter-Processor Communication Graph) [Bambha et al. 2002]. After embedding the schedule into the application model, the self-timed execution of the new model complies with the schedule exactly while still obeying the execution specification of the application. Therefore, by applying the methods introduced in [Ghamarian et al. 2006; Geilen 2010] on the schedule-aware application model [Damavandpeyma et al. 2013; Bambha et al. 2002], the application throughput with resource constraints can be analyzed. However, the PSOS and the method for modeling the PSOS into the application model in [Damavandpeyma et al. 2013; Bambha et al. 2002] have only considered processing resource contention on each processor. While [Bambha et al. 2002] has modeled inter-processor communication into the model, other kinds of resource contention have not been considered. Owing to the features of the platform, other resource contentions including buffer size constraints, memory access contention and DMA contention should also be taken into account. To capture these aspects, we first introduce the memory access aware SDFG that models the remote memory access operations; then the algorithm for constructing the contention-aware PSOS is introduced; finally, we extend the technique introduced in [Damavandpeyma et al. 2013], and model the contention-aware PSOS into the memory access aware SDFG, thus enabling throughput analysis.

Since we assume that only remote memory access would consume resources, including DMA and the memory port, the local memory access is ignored for simplicity. Though link contention is also possible in the NoC, we assume that the transportation on the NoC is contention-free, e.g., by using the TDMA arbitrary policy to assign each connection a fixed bandwidth, thus avoiding the communication contention in the NoC. While in [Poplavko et al. 2003] the authors assumed that it takes extra time to copy the data from/to the input/output buffer, we assume the processor can access all the local memory directly without incurring any delay, as the CompSoC platform does. We model each remote memory access (writing and reading) as a task, reconstruct the precedences of the new SDFG, thus producing a memory access aware SDFG. While constructing the memory access aware SDFG, we start from each edge, excepting self-edges, of the original SDFG. For each edge, based on the locations of the source task, destination task and the FIFO, the edge is reconstructed by adding extra memory access tasks and additional precedence edges. Algorithm 2 illustrates the procedures for constructing the memory access aware SDFG. First, the computation tasks are added to the model, as the first for-loop shows; then for each edge, as illustrated by the second for-loop, extra memory access tasks and precedence edges are added; finally, self-edges are added to avoid self-concurrency in the last for-loop.

Fig. 5 depicts four cases for an edge with respect to the relative locations of the shared FIFO and the tasks accessing this FIFO. In the figure, task b depends on task a, with the production rate and consumption rate being 2 and 3 respectively, and the initial token number being 2. We use fs to represent the FIFO capacity assigned to this edge, and we use ifs and ofs to denote the local input and output buffer size. The local input/output buffer is used when the read or produced data of a task has to be transferred from/to the remote FIFO. The input buffer size should be larger than the consumption rate of the edge and the output buffer size should be larger than the production rate. While a larger output buffer can make the task execute freely without waiting for buffer space to write output data, a smaller output buffer may serialize the

ALGORITHM 2: Memory Access Aware SDFG Construction Algorithm

Input: application model, platform model, task and FIFO allocation.
Output: memory access aware SDFG.
construct empty SDFG $G_{maa}(\mathbf{V}_{maa}, \mathbf{E}_{maa})$;
for $v \in \boldsymbol{V}$ **do**
  add task v to $\mathbf{V}_{maa}$;
end
for $e \in \mathrm{E}, src(e) \neq dst(e)$ **do**
  if $islocal(proc(src(e)), mem(fifo(e)))$ && $islocal(proc(dst(e)), mem(fifo(e)))$ **then**
    add edge $(src(e), p(e), dst(e), q(e), iniTok(e))$ to $\mathbf{E}_{maa}$;
    add edge $(dst(e), q(e), src(e), p(e), fs(e))$ to $\mathbf{E}_{maa}$;
  else if $!islocal(proc(src(e)), mem(fifo(e)))$ && $!islocal(proc(dst(e)), mem(fifo(e)))$ **then**
    add memory accessing tasks $w(e)$ and $r(e)$ to $\mathbf{V}_{maa}$;
    add edge $(src(e), p(e), w(e), p(e), 0)$ to $\mathbf{E}_{maa}$;
    add edge $(w(e), p(e), src(e), p(e), ofs(e))$ to $\mathbf{E}_{maa}$;
    add edge $(w(e), p(e), r(e), q(e), iniTok(e))$ to $\mathbf{E}_{maa}$;
    add edge $(r(e), q(e), w(e), p(e), fs(e))$ to $\mathbf{E}_{maa}$;
    add edge $(r(e), q(e), dst(e), q(e), 0)$ to $\mathbf{E}_{maa}$;
    add edge $(dst(e), q(e), dst(e), q(e), ifs(e))$ to $\mathbf{E}_{maa}$;
  else if $!islocal(proc(src(e)), mem(fifo(e)))$ && $islocal(proc(dst(e)), mem(fifo(e)))$ **then**
    add memory accessing task $w(e)$ to $\mathbf{V}_{maa}$;
    add edge $(src(e), p(e), w(e), p(e), 0)$ to $\mathbf{E}_{maa}$;
    add edge $(w(e), p(e), src(e), p(e), ofs(e))$ to $\mathbf{E}_{maa}$;
    add edge $(w(e), p(e), dst(e), q(e), iniTok(e))$ to $\mathbf{E}_{maa}$;
    add edge $(dst(e), q(e), w(e), p(e), fs(e))$ to $\mathbf{E}_{maa}$;
  else
    add memory accessing task $r(e)$ to $\mathbf{V}_{maa}$;
    add edge $(src(e), p(e), r(e), q(e), iniTok(e))$ to $\mathbf{E}_{maa}$;
    add edge $(r(e), q(e), src(e), p(e), fs(e))$ to $\mathbf{E}_{maa}$;
    add edge $(r(e), q(e), dst(e), q(e), 0)$ to $\mathbf{E}_{maa}$;
    add edge $(dst(e), q(e), dst(e), q(e), ifs(e))$ to $\mathbf{E}_{maa}$;
  end
end
for $v \in \boldsymbol{V}_{maa}$ **do**
  add a self-edge with one initial token to $\mathbf{E}_{maa}$;
end

execution and data transportation. A similar case also applies to the input buffer. For a task, we say a FIFO is local for it if the FIFO is allocated to the memory that is hosted on the same tile as the processor where the task is mapped onto. Fig. 5(a) shows how to reconstruct the edge when the FIFO is local to both the source task and destination task. Since the FIFO is allocated to the local memory, both the tasks a and b can access the memory directly free of delay; therefore, only a reverse edge directing from b to a needs to be added to constrain the FIFO capacity. The initial token number of the reverse edge is the same as the FIFO capacity, and the rates are just the opposite of the original edge. Fig. 5(b) illustrates the case when the FIFO is remote to both the tasks. In this case, the output of task a should be moved from the local output buffer to the FIFO located in the remote memory by the use of DMA, and task b has to move the data from the remote FIFO to the local input buffer to execute. We assume the local buffer is pre-assigned, and the size of it is known a priori. We add tasks w and r to represent the above remote FIFO accesses. We assume the system uses a coarse-grained memory access pattern, i.e., for an edge, the reading and writing are in terms of consumption rate and production rate respectively. Hence, w and r would be fired the

same number of times as a and b in long-term execution. Since the local output buffer of a is limited in size, the task can not fire freely when w has not moved the data to the remote FIFO. To capture this constraint, in Fig. 5(b), an extra edge is added between w and a, with the initial token number being the size of the local output buffer. The rates of the edge between w and a are the same as the production rate of the original edge, i.e., 2, to make w and a have the same firing frequency. The relation between a and b in the original edge is moved to w and r; therefore, the firings of w and r are constrained by the FIFO capacity (fs), and the FIFO access pattern remains the same. Fig. 5(c) shows when the FIFO is local to task b but remote to task a. In this case, task b can access the FIFO directly; however, the output data of task a needs to be transferred to the FIFO from the local output buffer by the use of DMA. As shown in the figure, the local output buffer is limited in size, as captured by the reverse edge from w to a. To constrain the FIFO access pattern, the edge between w and b is the same as the original edge. Finally, Fig. 5(d) depicts the case when the FIFO is remote to the reading side and local to the writing side.

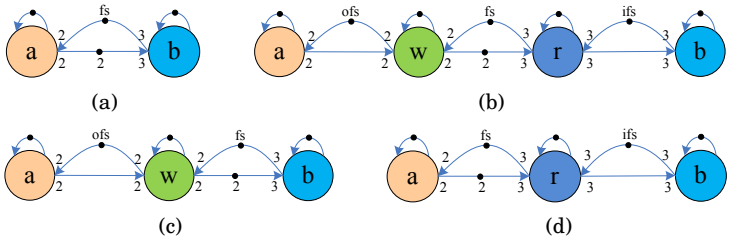

Fig. 5. Reconstructed SDFG models for the edge with different task and FIFO allocation strategies, (a) local FIFO, (b) remote FIFO for both sides, (c) remote FIFO for the writing side, (d) remote FIFO for the reading side.

Using the method introduced above, the memory access aware SDFG can be constructed. However, the self-timed execution of the memory access aware SDFG does not take into account resource contention. To solve this problem, we construct the contention-aware PSOS and model it into the memory access aware SDFG.

To construct the contention-aware PSOS, three kinds of resource have to be arbitrated. The first one is the arbitration when tasks compete for the processor, which is extensively investigated in [Damavandpeyma et al. 2013]. To take into account memory contention and DMA contention, when self-timed executing the memory access aware SDFG, not only should the processor contention be detected, but also the memory access contention and DMA contention. Memory access contention is also called port contention. The memory can support parallel memory access if it has more than one port, otherwise, the operations should be serialized. We assume each memory provides only one port for remote access, so, only one remote transaction can be fired on a typical memory at the same time. The SPM is configured with two ports, one for local access and one for remote access, so it supports two parallel accesses. However, any remote access on it would occupy its external port exclusively; hence, all remote operations on it should also be serialized. For each processor, reading or writing a remote memory is fulfilled by the DMA device on the same tile. Therefore, all remote memory accesses initiated by each processor should be serialized to avoid DMA contention.

We use the Mapping-Aware Earliest Task First scheduling algorithm (MAETF) (Algorithm 3) to construct the contention-aware PSOS. MAETF has taken into account the FIFO size constraints, memory access latency, port contention when accessing the memory, and yields both task ordering on each processor and memory access ordering

on each memory and DMA. MAETF is a list-based algorithm. As shown in Algorithm 3, first, the SDFG is transformed to an APG using the technique introduced in [Sriram and Bhattacharyya 2012]; second, the tasks in the APG are ordered in non-increasing order of bottom level, which is the length of the longest path starting from the task; finally, the tasks are scheduled one by one while taking into account their mapping information. To avoid resource contention, while scheduling the task in the APG, resource contention is arbitrated. For the task to be scheduled, on the resource it depends on, all the time intervals that have already been occupied by other tasks scheduled earlier are found, and the new task is scheduled to an earliest unoccupied time interval using an insertion strategy [Sinnen 2007]. By the above way, the ordering of computation tasks and memory accesses on each resource (the contention aware PSOS) is obtained.

ALGORITHM 3: Mapping-Aware Earliest Task First Scheduling Algorithm

Input: memory access aware SDFG.
Output: contention-aware PSOS.
transform the memory access aware SDFG to APG;
order the tasks in non-increasing order of bottom level, obtaining task list **A**;
repeat
  $v \leftarrow$ pop-up the first element in **A**;
  if v *is a computation task* **then**
    let $proc(v)$ be the processor where task v is mapped onto;
    schedule v on processor $proc(v)$;
  else
    let $dma(v)$ be the DMA that task v uses;
    let $mem(v)$ be the memory that task v accesses;
    schedule v on $dma(v)$ and $mem(v)$;
  end
until $\boldsymbol{A} = \varnothing$;

The contention aware PSOS can be modeled into the memory access aware SDFG using the technique introduced in [Damavandpeyma et al. 2013]. During the modeling process, three kinds of resource contentions captured by the contention aware PSOS, i.e., the processor contention, the DMA contention and memory port contention, should be considered. The tasks in the memory access aware SDFG are categorized as three kinds based on the resource they use, i.e., computation task, DMA task and memory task. The computation task uses a processor to process data, the DMA task uses DMA to access data in remote memory, and the memory task occupies one port of the remote memory to access data. Each task in the memory access aware SDFG is associated with the resources it uses. We use Res to denote the set of resources in the platform, including processors, DMAs and memories. So, the decision state in [Damavandpeyma et al. 2013] is a state when multiple tasks use the same resource in Res. After obtaining the decision state, each opponent actor set that use the same resource is detected. According to the contention aware PSOS, for each opponent actor set, the actor of choice is picked out, then extra edges are added to the model between the actor of choice and other tasks in the decision state to make the self-timed execution comply with the PSOS.

6. EXPERIMENTS AND RESULTS

In this section, the proposed methods are evaluated experimentally by comparing with the non-iterative edition of the scheduling framework, the load balancing method [S-

tuijk et al. 2007], the HAFF algorithm [Zhang et al. 2010], the blocked schedule based throughput analysis method [Tang et al. 2015] and the MILP-based schedule algorithm [Morteza et al. 2011]. The remaining section first introduces the platform configuration and the benchmark used in the experiments. Then, the algorithm performance is evaluated in terms of throughput.

6.1. Platform Configuration and Benchmark

We use several MPSoCs with different configurations in the experiment to test the algorithm performance. The MPSoCs are configured with different processor numbers, ranging from two to six. We assume the processors and memories are interconnected by a network-on-chip (NOC) [Goossens et al. 2013; Stefan et al. 2014] and the NOC links are reserved for each connection, e.g., by using the TDMA policy.

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 [Stefan et al. 2014] 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 [Banakar et al. 2002], so, the memory access delay of a remote SPM is $4 + 2n$. For the global memory, we assume the worst-case memory access delay is $60 + 9n$. In each experiment, we reserve a part of the SPM for the application to be scheduled. On each SPM, the reserved capacity is of the same size, and the total reserved capacity equals the total FIFO requirement.

A set of practical applications and randomly generated SDFGs with different sizes are used as benchmarks for performance evaluation. The practical applications include the H.263 decoder [Stuijk et al. 2007], H.263 encoder [Oh and Ha 2004], samplerate conversion [Bhattacharyya et al. 1999], bipartite [Bhattacharyya et al. 1999], MPEG-4 SP decoder [Geilen 2010], MP3 decoder granule/block [Geilen 2010], modem [Bhattacharyya et al. 1999] and channel equalizer [Ritz et al. 1995]. The open source tool SDF3 [Stuijk et al. 2006b] is used to generate random SDFGs of 10, 20 and 30 tasks. The repetition vector is also randomly generated with a constraint 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 50 to 150 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 (CCR) which is defined as the ratio between the number of memory access operations and the total task execution time. In the experiments, different CCRs are used to configure the application to evaluate the effect of CCR on performance.

6.2. Results of Random Applications

This subsection evaluates the performance of the algorithms proposed in this paper with respect to synthesized applications. Several denotations are used to represent the comparison between different algorithms. “Comp1” represents the throughput improvement by searching the FIFO-throughput Pareto space over using the FIFO distribution with the maximum ideal throughput, i.e., the non-iteration strategy. In this

comparison, the same task/FIFO allocation strategy and scheduling method are used for each FIFO distribution. “Comp2” represents the throughput improvement by the use of FAATA over the load balancing method, with the GFAO and pipelined schedule method being used to allocate the FIFO and make the final schedule. “Comp3” represents the throughput improvement by the use of GFAO over HAFF, and the pipelined schedule method is used to construct the final schedule. “Comp4” compares the throughput analysis method proposed in this paper with that in [Tang et al. 2015]. “Comp5” compares the scheduling framework proposed in this paper without performing iteration with that in [Morteza et al. 2011]. We also record the run time of each sub-algorithm independently in each iteration, hence, the total time taken to finish one iteration is the sum of the run time of each sub-algorithm.

To have a global view of the performance of the proposed scheduling framework, we first summarize the average throughput improvement of each algorithm in the scheduling framework over all applications and platforms with different configurations. The proposed scheduling framework outperforms other scheduling strategies in different aspects. The iteration strategy outperforms the non-iteration strategy by 38.36% in throughput on average. Even though it is achieved at the expense of runtime, for off-line optimization when runtime is not a serious concern, it is a practical method to optimize the system. Besides, using FAATA can improve throughput by 8.08% on average. The FIFO allocation also influence the performance, using which the performance can be improved by 22.47% on average. The proposed schedule strategy is an effective method to produce compact schedules. Compared with the blocked schedule based throughput analysis method proposed in [Tang et al. 2015], an average performance improvement of 6.54% can be achieved using our method. This is because using our method, the system executes in a self-timed style and different iterations can overlap, thus improving the throughput. Since the MILP model [Morteza et al. 2011] is too large for applications with more than ten tasks, we only carried experiments for 10-task application for the MILP. The proposed scheduling framework performs better than the MILP-based method, with the performance being improved by 23.71%.

Table IV illustrates the average throughput improvement between different algorithms with respect to application size. The performance improvement achieves 34.15% for the 10-task application set and increases to 40.80% for the 30-task application set. It’s because larger applications also have a larger FIFO distribution space, providing more potential to improve the performance. The performance of FAATA decreases with the application size, with the performance improvement decreasing from 11.15% for 10-task applications to 6.44% for 30-task applications. GFAO performs better than HAFF for different application sets, with the throughput being improved by over 19.41%. The proposed schedule strategy outperforms the blocked schedule. An average performance improvement of over 4.97% can be achieved for different application sets.

Table IV. Average throughput improvement with respect to application size

Task Num	Comp1	Comp2	Comp3	Comp4
10	34.15%	11.15%	24.09%	8.73%
20	40.12%	6.65%	23.90%	5.93%
30	40.80%	6.44%	19.41%	4.97%

Table V depicts the average throughput improvement between different algorithms with respect to processor number in the platform. On the 2-processor platform, the iteration strategy achieves an improvement of 38.59% over the non-iteration strategy,

and the improvement decreases to 37.54% for the 6-processor platform. The performance improvement of FAATA decreases from 9.00% to 7.55%. The FIFO allocation algorithm has a decreasing trend, with the value decreasing from 34.28% to 21.07%. For the schedule algorithm, the throughput improvement shows an increasing trend with the platform size, with the value increasing from 4.70% to 7.42%. The proposed scheduling framework outperforms the MILP for platforms with different processors, with the performance improvement being over 13.53%.

Table V. Average throughput improvement with respect to platform size

Proc Num	Comp1	Comp2	Comp3	Comp4	Comp5
2	38.59%	9.00%	34.28%	4.20%	34.14%
4	38.95%	7.70%	25.05%	6.01%	13.53%
6	37.54%	7.55%	21.07%	7.42%	23.45%

Table VI compares different algorithms with respect to the CCR of the application. For the application with larger CCR value, the overhead of accessing the memory grows, making it more important to configure the FIFO size and FIFO allocation. As demonstrated by the results in Table VI, the performance improvements of the iteration strategy and FAATA increase with the CCR value. The iteration strategy has a performance improvement of 27.52% when the CCR is 1; the improvement grows to 44.24% as the CCR increase to 5, showing that for large CCR, it's more important to search the FIFO distribution space so as to obtain a good performance. Similarly, the performance of FAATA increases with the CCR value, with the improvement increasing from 6.54% to 9.04%. GFAO outperforms HAFF for different CCRs by over 10%. The performance of the self-timed schedule decreases with the CCR value, with the performance improvement decreasing from 6.90% to 5.23%. Similarly, the proposed scheduling framework outperforms the MILP for applications with different CCRs, with the performance improvement being over 17.09%.

Table VI. Average throughput improvement with respect to CCR

CCR	Comp1	Comp2	Comp3	Comp4	Comp5
1	27.52%	6.54%	22.05%	6.90%	17.09%
3	43.31%	8.66%	21.59%	6.50%	24.17%
5	44.24%	9.04%	10.76%	5.23%	22.34%

In the experiment, the average runtime of the load balancing method is 1.09 seconds, and that of the FAATA is 7.16 seconds. Since FAATA needs to evaluate the effect of local FIFO allocation, so FAATA consumes more time. The runtime of GFAO is about three times of HAFF, the average runtime of HAFF is 4.37 microseconds and that of GFAO is 12.27 microseconds. Throughput analysis consumes the biggest part of the total computation time, the pipelined method consumes 19.01 seconds averagely, and the blocked-schedule based method consumes 16.62 seconds averagely. As the task number of the application grows, the runtime time increases at the same time. As the task number grows from 10 to 30, the runtime of FAATA grows from 0.80 seconds to 14.56 seconds, and the runtime of the pipelined method grows from 2.37 seconds to 44.14 seconds. However, the runtime of GFAO grows little, with the value increasing from 10.63 microseconds to 15.60 microseconds. The MILP-based method is time-consuming, with the solving time being more than ten minutes, and as the graph size grows, the runtime grows rapidly, e.g., even 12 hours is not enough to produce a valid solution for the 20-task applications.

6.3. Results of Practical Applications

To evaluate the practical usage of the scheduling framework proposed in this paper, we test it on a set of real applications and compare it with the available methods. The parameters of the used real applications are shown in Table VII. The task number ranges from 4 to 16; however, the instance number in an application iteration reaches up to more than 1000. We categorize these applications as two sets according to the task number in the application, one is the small application set consisting of the applications with less than ten tasks, and the other is the large application set with more tasks. We test the small application set on a 2-processor platform and the larger application set on both a 2-processor platform and a 4-processor platform.

Table VII. Parameters of the practical streaming applications

	App Name	Task Num	Edge Num	Instance Num
small apps	H.263 decoder	4	3	1190
	H.263 encoder	5	5	201
	samplerate conversion	6	5	612
	bipartite	4	4	73
	MPEG-4 SP decoder	5	8	63
large apps	MP3 decoder granule	14	18	27
	MP3 decoder block	14	18	911
	modem	16	19	48
	channel equalizer	12	20	14

Table VIII depicts the experimental results of practical applications on the MPSoC with two processors. From the table, it's shown that the proposed scheduling framework works for practical applications. The iteration strategy can improve the throughput, with the improvement ranging from 4.42% to 16.22%. FAATA is effective for all real applications, and for some applications it's more important to consider the FIFO allocation while mapping the tasks, e.g., for mp3 decoder granule, samplerate and bipartite, and the throughput improvement by using FAATA achieves more than 35%. From the fourth column of Table VIII, 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 GFAO over HAFF ranges from 17% to 71%, values being larger than those of randomly generated applications. For real applications, the self-timed schedule can still improve the performance compared with the blocked schedule, though the improvement is not very significant. Cyclic precedences in the application model constrain the overlap between different iterations, so the throughput cannot improve a lot.

Table VIII. Performance comparison with respect to real applications on the MPSoC with two processors

App Name	Comp1	Comp2	Comp3	Comp4
h263 decoder	16.22%	23.60%	37.60%	0.00%
h263 encoder	6.48%	4.28%	70.97%	0.00%
samplerate	13.72%	39.91%	48.06%	0.58%
bipartite	6.63%	35.11%	43.37%	3.15%
MPEG-4 SP decoder	4.42%	9.42%	46.48%	1.02%
mp3 decoder granule	10.06%	77.37%	17.09%	6.57%
mp3 decoder block	10.40%	13.23%	34.30%	0.05%
modem	6.09%	6.50%	21.06%	2.14%
channel equalizer	7.88%	8.33%	38.11%	0.57%

Table IX depicts the experimental results for large practical applications on the MPSoC with four processors. From the table, it's shown that on large platforms, the pro-

posed scheduling framework still works. Specifically, the iteration strategy can improve the throughput by 7.26% to 15.48%. FAATA performs better on larger platforms for modem and channel equalizer, with the throughput improvement reaching 11.62% and 7.26% respectively. For the applications we use, the throughput improvement of GFAO over HAFF ranges from 5.54% to 45.76%. The self-timed schedule can improve the performance compared with the blocked schedule, with the improvement reaching as high as 10.46% for the channel equalizer.

Table IX. Performance comparison with respect to large real applications on the MPSoC with four processors

App Name	Comp1	Comp2	Comp3	Comp4
mp3 decoder granule	15.48%	75.40%	22.02%	3.53%
mp3 decoder block	7.97%	9.23%	45.76%	0.11%
modem	15.42%	11.62%	10.02%	5.54%
channel equalizer	7.26%	12.16%	10.71%	10.46%

The MILP-based method is also applied to real applications in the experiment. However, since most of real applications are too large, making it quite hard to solve using the MILP solver in limited time. For example, three hours are not sufficient to produce a valid solution for MPEG-4 SP decoder that has 63 task instances. So, only the results of applications with little task instances, i.e., MP3 decoder granule, modem and channel equalizer, are recorded. According to the experiment results, the proposed method outperforms the MILP-based method, the throughput are improved by over 40% for all these three applications, showing that appropriate usage of the SPM is has more effects on the performance for real applications.

The runtime of the proposed algorithm is not large for real applications. In the experiment, The average execution time of LB and FAATA are 0.096 and 1.010 respectively. The runtime of GFAO is about two times of HAFF, with the average execution time being 1.66 and 3.55 microseconds respectively. Throughput analysis consumes the biggest part of the total computation time, especially for applications many task instance, e.g, H.263 decoder samplerate, conversion MP3 and decoder block. For these three applications, the pipelined method and the blocked-schedule based method consume 15.9 and 14.9 minutes respectively. For other applications, the average execution time are 4.43 and 3.49 seconds, which are quite small. Compared with MILP-based method, the method proposed in this paper is less complex. While the MIlP-based method needs several hours to find the solution, the proposed method can be finished in several seconds.

7. CONCLUSIONS

This paper investigates the problem of scheduling streaming applications on multiprocessor systems-on-chips with a predictable memory hierarchy by taking into account memory access latency and memory capacity constraints. The resources of practical embedded systems are limited. Therefore, it's critical to utilize them appropriately to meet the timing requirements of the application. An efficient Iteration-based Task-FIFO Co-Scheduling (ITFCS) framework is proposed in this paper to solve the scheduling problem. The scheduling framework consists of FIFO-Throughput Pareto Space Search, FIFO Allocation Aware Task Assignment (FAATA) algorithm, Global FIFO Allocation Optimization (GFAO) algorithm and a self-timed throughput analysis method. Extensive experiments are carried out on both random SDFGs and practical applications. Experimental results show that our method outperforms other methods. Roughly, the iteration strategy increases the complexity a lot, since it needs to analyze the FIFO distribution iteratively; while other parts of the scheduling framework are

quite efficient, with the complexity being increased only slightly. Considering the fact that the method is used for off-line optimization, such incensement is acceptable. In the future, we will study how to use meta-heuristics to solve the problem.

REFERENCES

Jude Angelo Ambrose, Isuru Nawinne, and Sri Parameswaran. 2013. Latency-constrained binding of data flow graphs to energy conscious GALS-based MPSoCs. In *International Symposium on Circuits and Systems*. IEEE, 1212–1215.

Neal Bambha, Vida Kianzad, Mukul Khandelia, and Shuvra S Bhattacharyya. 2002. Intermediate representations for design automation of multiprocessor DSP systems. *Design Automation for Embedded Systems* 7, 4 (2002), 307–323.

Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*. ACM, 73–78.

L. Benini and G. De Micheli. 2002. Networks on chips: a new SoC paradigm. *Journal in Computer* (2002).

Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. 1999. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology* 21, 2 (1999), 151–166.

Greet Bilsen, Marc Engels, Rudy Lauwereins, and JA Peperstraete. 1994. Static scheduling of multi-rate and cyclo-static DSP-applications. In *Proceedings of the International Workshop on VLSI Signal Processing*. IEEE, 137–146.

Weijia Che and Karam S Chatha. 2010. Scheduling of synchronous data flow models on scratchpad memory based embedded processors. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 205–212.

Junchul Choi, Hyunok Oh, Sungchan Kim, and Soonhoi Ha. 2012. Executing synchronous dataflow graphs on a spm-based multicore architecture. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 664–671.

Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. 2013. Schedule-extended synchronous dataflow graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 10 (2013), 1495–1508.

Marc Geilen. 2010. Synchronous dataflow scenarios. *ACM Transactions on Embedded Computing Systems* 10, 2 (2010), 16.

Amir Hossein Ghamarian, MCW Geilen, Sander Stuijk, Twan Basten, AJM Moonen, Marco JG Bekooij, Bart D Theelen, and MohammadReza Mousavi. 2006. Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design*. IEEE, 25–36.

Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, and others. 2013. Virtual execution platforms for mixed-time-criticality systems: The compsoc architecture and design flow. *ACM SIGBED Review* 10, 3 (2013), 23–34.

Jingcao Hu and Radu Marculescu. 2004. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, Vol. 1. IEEE, 234–239.

Edward A Lee and David G Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.* 100, 1 (1987), 24–35.

Rainer Leupers and Daniel Kotte. 2001. Variable partitioning for dual memory bank DSPs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 2. IEEE, 1121–1124.

Orlando Moreira and Henk Corporaal. 2014. *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor*. Springer.

Damavandpeyma Morteza, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. 2011. Hybrid code-data prefetch-aware multiprocessor task graph scheduling. In *14th Euromicro Conference on Digital System Design*. IEEE, 583–590.

Hyunok Oh and Soonhoi Ha. 2004. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI signal processing systems for signal, image and video technology* 37, 1 (2004), 41–51.

Jose Luis Pino, Shuvra S Bhattacharyya, and Edward A Lee. 1995. A hierarchical multiprocessor scheduling system for DSP applications. In *Conference Record of the Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, Vol. 1. IEEE, 122–126.

Peter Poplavko, Twan Basten, Marco Bekooij, Jef van Meerbergen, and Bart Mesman. 2003. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems* (2003).

Sebastian Ritz, Markus Willems, and Heinrich Meyr. 1995. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *International Conference on Acoustics, Speech, and Signal Processing*, Vol. 4. IEEE, 2651–2654.

Oliver Sinnen. 2007. *Task scheduling for parallel systems*. Vol. 60. John Wiley & Sons.

Sundararajan Sriram and Shuvra S Bhattacharyya. 2012. *Embedded multiprocessors: Scheduling and synchronization*. CRC press.

Radu Andrei Stefan, Anca Molnos, and Kees Goossens. 2014. dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. *IEEE Trans. Comput.* 63, 3 (2014), 583–594.

Sander Stuijk, Twan Basten, MCW Geilen, and Henk Corporaal. 2007. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*. ACM, 777–782.

Sander Stuijk, Marc Geilen, and Twan Basten. 2006a. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd annual Design Automation Conference*. ACM, 899–904.

Sander Stuijk, Marc Geilen, and Twan Basten. 2006b. SDF3: SDF For Free. In *Sixth International Conference on Application of Concurrency to System Design*, Vol. 6. 276–278.

Qi Tang, Twan Basten, Marc Geilen, Sander Stuijk, and Ji-Bo Wei. 2015. Task-FIFO Co-Scheduling of Streaming Applications on MPSoCs with Predictable Memory Hierarchy. In *Fifteenth International Conference on Application of Concurrency to System Design*. IEEE, 90–99.

Yang Yang, Marc Geilen, Twan Basten, Sander Stuijk, and Henk Corporaal. 2011. Iteration-Based Trade-Off Analysis of Resource-Aware SDF. In *14th Euromicro Conference on Digital System Design*. IEEE, 567–574.

Lei Zhang, Meikang Qiu, Wei-Che Tseng, and Edwin H-M Sha. 2010. Variable partitioning and scheduling for MPSoC with virtually shared scratch pad memory. *Journal of Signal Processing Systems* 58, 2 (2010), 247–265.

Qingfeng Zhuge, Bin Xiao, and Edwin H-M Sha. 2002. Variable partitioning and scheduling of multiple memory architectures for DSP. In *International Parallel and Distributed Processing Symposium*, Vol. 2. IEEE Computer Society, 0130–0130.