Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs*

S. Stuijk, T. Basten, M.C.W. Geilen and H. Corporaal
Eindhoven University of Technology, Department of Electrical Engineering
{s.stuijk, a.a.basten, m.c.w.geilen, h.corporaal}@tue.nl

Abstract. Embedded multimedia systems often run multiple time-constrained applications simultaneously. These systems use multiprocessor systems-on-chip of which it must be guaranteed that enough resources are available for each application to meet its throughput constraints. This requires a task binding and scheduling mechanism that provides timing guarantees for each application independent of other applications while taking into account the available processor space, memory and communication bandwidth.

Synchronous Dataflow Graphs (SDFGs) are used to model time-constrained multimedia applications. They allow modeling of cyclic, multirate dependencies between tasks. However, existing resource allocation techniques can only deal with acyclic and/or single-rate dependencies. Dependencies in an SDFG can be expressed in single-rate form, but then the problem size may increase exponentially making resource allocation infeasible. This paper presents a new resource allocation strategy which works directly on SDFGs, building on an efficient technique to calculate throughput of a bound and scheduled SDFG. Experimental results show that the strategy is effective in terms of run-time and allocated resources.

Categories and Subject Descriptors: C.3 [Special-purpose and Application-based Systems] Real-time and embedded systems

General Terms: Algorithms, Experimentation, Theory.

Keywords: Synchronous Dataflow, multi-processor, throughput, mapping.

1. INTRODUCTION

Modern embedded multimedia systems support many different applications. To meet the growing computational demands of these applications, multiprocessor systems-on-chip (MP-SoCs) are used. Consumers expect that the system is robust and its performance is guaranteed [9]. This requires that every application running on the system has a predictable timing behavior which is independent of other applications running on the same system. The resource allocation strategy, which binds tasks from an application to the resources and schedules the tasks and the inter-task communication on the assigned resources, should offer this predictability.

Synchronous Dataflow Graphs (SDFGs) [13] are used to model multimedia applications with timing constraints that must be bound to an MP-SoC [18]. It allows modeling of both pipelined streaming and cyclic dependencies between tasks. Furthermore, analysis techniques to study, for example, the throughput and storage requirements of an SDFG exist [10, 21]. An example of an SDFG is depicted in Fig. 1. The nodes of an SDFG, called *actors*, communicate with *tokens* sent from one actor to another over the edges. The actors typically model application tasks and the edges model data or control dependencies. An essential property of SDFGs is that every time an actor *fires* (executes) it consumes the same amount of tokens from its input edges and produces the same amount of tokens on its output edges. These amounts are called the *rates*. In an SDFG, the rate at which tokens are produced on an edge may differ from the rate at which tokens are consumed from the edge. An SDFG can always be converted to a *homogeneous* SDFG (HSDFG) in which all rates are equal to one [20]. However, this can lead to an exponential increase in the number of actors in the graph. For example, the HSDFG corresponding to the SDFG shown in Fig. 1 contains 4754 actors.

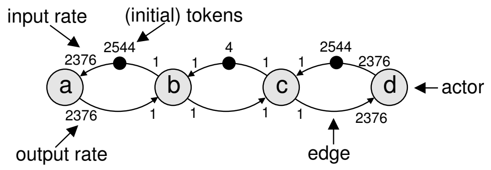

Figure 1: SDFG of an H.263 decoder.

Existing resource allocation strategies for time-constrained applications are based on HSDFGs or acyclic dependency graphs (acyclic HSDFGs). Techniques for acyclic graphs often do not take streaming (iterative, overlapping execution of the graph) into account, which makes them not very suitable for throughput-constrained multimedia applications. Moreover, an SDFG model of an application must be converted to an HSDFG to apply any of these techniques. The H.263 decoder example shows that this conversion drastically increases the problem size, rendering this approach often infeasible. The biggest problem with working on an HSDFG instead of on its corresponding SDFG is the time needed to compute the throughput. A resource allocation strategy must compute the throughput of an application bound to the system at least once in order to verify whether the throughput constraint is met. Throughput is determined by the cycles in a graph. The fastest method to compute the throughput of an HSDFG is the use of a maximum cycle ratio algorithm [20]. The fastest known variant has a run-time of 21 minutes on a P4 at 3.4GHz for the HSDFG of the H.263 decoder shown in Fig. 1. So any HSDFG-based resource allocation strategy runs for at least 21 minutes on the H.263 decoder. Typically, a resource allocation strategy performs a throughput computation more than once in order to get a notion of the critical cycles in the application and tune the resource allocation. This paper presents a novel technique for task binding and scheduling directly on SDFGs. This keeps the problem size typically much smaller and allows us to perform resource allocation for a larger class of applications within a limited run-time. For example, the proposed strategy has a run-time of less than 3 minutes on the H.263 decoder. It performs 8 throughput checks directly on the provisionally mapped SDFG during the trajectory to find the binding and scheduling.

Overview. Sec. 2 discusses related work in the field of resource allocation for dataflow graphs. Sec. 3 and Sec. 4 introduce respectively SDFGs and schedules for them. The architecture platform is introduced in Sec. 5 and the application model is formalized in Sec. 6. The resource allocation problem is defined in Sec. 7. Throughput computation for an SDFG bound to an MP-SoC is explained in Sec. 8. Our SDFG-based resource allocation strategy is described in Sec. 9. The experimental results are presented in Sec. 10.

*This work is supported by the Dutch Science Foundation NWO, project 612.064.206, PROMES.

2. RELATED WORK

An overview of traditional scheduling and binding techniques for dataflow graphs can be found in [20]. All mentioned techniques use acyclic graphs in which every task must be executed once. We assume that tasks are repeatedly executed and different tasks may be executed with different rates. We also allow cyclic dependencies between different (pipelined) executions of the same task.

Resource allocation for acyclic graphs with timing guarantees is studied in [11, 12]. Hu et. al assume that every task can only be bound to a single processor type [11]. The strategy only decides on which processor (i.e. location) to use. Our strategy has to decide on both the processor type and its location and works for a larger class of models, as explained. In [12], the resource allocation problem is formulated as a constraint satisfaction problem. Cyclic dependencies which determine, for example, the throughput of an application cannot be expressed in this framework.

A multi-objective evolutionary algorithm to bind an application described as a Kahn Process Network to a heterogeneous MP-SoC is presented in [8]. The approach can deal with cyclic task graphs, but no timing guarantees are provided on the resulting binding.

In [15], an approach is presented to perform resource allocation for a time-constrained HSDFG on a homogeneous MP-SoC. Binding is done using a multi-dimensional bin-packing algorithm that considers the same resources as our strategy does. However, our strategy can handle arbitrary SDFGs while targeting a heterogeneous MP-SoC.

A method to bind an application described as a Cyclo-Static Dataflow graph onto a heterogeneous MP-SoC is given in [6]. It tries to maximize the throughput which can be realized with the available resources. Only a single application can be mapped to the system. Resource allocation to multiple applications with throughput guarantees for each of them is not considered. Our strategy tries to minimize resource usage under given throughput constraints, thus maximizing the number of applications that can run concurrently on the system with throughput guarantees.

3. SYNCHRONOUS DATAFLOW GRAPHS

Let $\mathbb{N}$ denote the positive natural numbers, $\mathbb{N}_0$ the natural numbers including 0, and $\mathbb{N}_0^\infty$ the natural numbers including 0 and infinity (∞).

DEFINITION 1. (SDFG) *An SDFG is a tuple (A,D) consisting of a finite set A of actors and a finite set $D \subseteq A^2 \times \mathbb{N}^2$ of dependency edges. A dependency edge $d = (a,b,p,q)$ denotes a dependency of actor b on a. When a executes it produces p tokens on d and when b executes it removes q tokens from d. Edges may contain initial tokens defined by $Tok : D \to \mathbb{N}_0$.*

As mentioned, actor execution is defined in terms of *firings*. An essential property of SDFGs is that every time an actor fires it consumes the same amount of tokens from its input edges and produces the same amount of tokens on its output edges. These amounts are called the *rates*. The rates determine how often actors have to fire wrt each other such that the distribution of tokens over all edges is not changed. This property is captured in the *repetition vector*.

DEFINITION 2. (REPETITION VECTOR) *A repetition vector of an SDFG (A,D) is a function $\gamma : A \to \mathbb{N}_0$ such that for every edge $(a,b,p,q) \in D$ from $a \in A$ to $b \in A$, $p \cdot \gamma(a) = q \cdot \gamma(b)$. A repetition vector γ is called non-trivial if $\forall a \in A$, $\gamma(a) > 0$.*

An SDFG is called *consistent* if it has a non-trivial repetition vector. The smallest non-trivial repetition vector of a consistent SDFG is called *the* repetition vector. Consistency and absence of deadlock are two important properties for SDFGs which can be verified efficiently [5, 13]. Any SDFG which is not consistent requires unbounded memory to execute or deadlocks, meaning that no actor is able to fire. Such SDFGs are not useful in practice. Therefore, we focus on consistent and deadlock free SDFGs.

Throughput is an important design constraint for embedded multimedia systems. The throughput of an SDFG refers to how often an actor produces an output token. To compute throughput, a notion of time must be associated with the firing of actors and an execution scheme must be defined. We do so in the following sections.

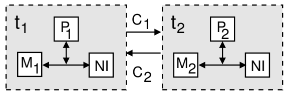

Figure 2: Example architecture platform.

Table 1: Properties of the example platform.

	pt	w	m	c	i	o		$\mathscr{L}(c)$
t_1	p_1	10	700	5	100	100	c_1	1
t_2	p_2	10	500	7	100	100	c_2	1

4. SCHEDULING

A resource allocation strategy can bind multiple actors, possibly of different applications, to the same processor. In a system with a predictable timing behavior, a scheduling mechanism must order the execution of the actors such that it is possible to provide a guarantee on the maximum amount of time between the moment that an actor is ready to fire and the completion of the firing (its *response time*). Time-division multiple-access (TDMA) scheduling offers these guarantees [4]. It uses a periodically rotating time wheel. An application reserves a time slice on the wheel during which it may fire its actors. A TDMA scheduler can also be used to schedule multiple actors of the same application on a single processor [4]. Time slices are then reserved for individual actors. This gives very conservative estimates on the worst case actor response time [20]. Therefore, we apply static order scheduling for actors within an application. Following [16], a static-order schedule S for a set A of actors is an infinite sequence $a_1 a_2 a_3 ...$ of actor firings, with $a_i \in A$. Practical static-order schedules consist of a (possible empty) sub-sequence which is seen once followed by a finite sub-sequence which is infinitely often repeated.

5. ARCHITECTURE PLATFORM

The architecture template in our work is similar to the tile-based multiprocessor described in [7] in which multiple tiles are connected by an interconnection network. We assume point-to-point connections with a fixed latency between tiles. These connections can, for example, be implemented through a network-on-chip with timing guarantees. Fig. 2 shows an example architecture platform with two connected tiles. Each tile contains one processor (P) and a local memory (M). A tile contains also a set of communication buffers, called the network interface (NI), that are accessed both by the local processor and the interconnect. Multiprocessor systems like Daytona [1], Eclipse [19], Hijdra [3], and StepNP [17] fit nicely into this template. The resources in a tile can be described as follows. Let PT be the set of all processor types.

DEFINITION 3. (TILE) *A tile is a 6-tuple (pt,w,m,c,i,o) with $pt \in PT$ the processor type, $w \in \mathbb{N}_0$ the size of the processor's TDMA time wheel (in time units), $m \in \mathbb{N}_0$ the memory size (in bits), $c \in \mathbb{N}_0$ the maximum number of connections supported by the NI, and $i,o \in \mathbb{N}_0$ the maximum incoming and outgoing bandwidth (in bits/time-unit).*

In practice, a time wheel may already be partially occupied when binding an application to a tile. The function $\Omega : T \to \mathbb{N}_0$, with T the set of tiles, gives for a tile the size of the time wheel which is already occupied. Other resources in a tile may also be (partially) occupied. For simplicity, we assume that all memory (m), connections (c), and incoming (i) and outgoing (o) bandwidth specified by a tile are available for an application. Resources that are not available (i.e., used by other applications) should not be specified.

DEFINITION 4. (ARCHITECTURE GRAPH) *An architecture graph $(T,C,\mathscr{L})$ consists of a set T of tiles, a set $C \subseteq T^2$ of connections and a latency function $\mathscr{L} : C \to \mathbb{N}$. A connection is a tuple $c = (u,v)$ through which data can be sent from a tile u to a tile v with a latency $\mathscr{L}(c)$ (in time units).*

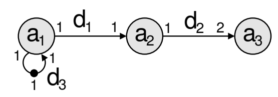

Figure 3: Example SDFG.

Table 2: Properties of the example application.

	$p_1(\tau,\mu)$	$p_2(\tau,\mu)$		sz	α_{tile}	α_{src}	α_{dst}	β
a_1	$(1,10)$	$(4,15)$	d_1	7	1	2	2	100
a_2	$(1,7)$	$(7,19)$	d_2	100	2	2	2	10
a_3	$(3,13)$	$(2,10)$	d_3	1	1	0	0	0

The connections between tiles introduce a latency when data is sent between them. Each connection can have a different latency. In this way, the latency of different connections through a network-on-chip or segmented bus can be taken into account. The amount of data which can be sent per time-unit (i.e. bandwidth) is limited by the incoming, i, and outgoing bandwidth, o, of the tiles. Tab. 1 gives the values of all elements in the architecture graph of Fig. 2.

6. APPLICATION MODEL

The structure of an application can be described with an SDFG. A resource allocation strategy needs also information on the resource requirements of the actors and edges in the graph. It must, for example, know to which processor types an actor can be bound and how many CPU cycles it requires on these processors. Furthermore, the application model must also provide a throughput constraint which must be satisfied when the application is bound to the architecture graph. An application with its resource requirements and throughput constraint is described by an application graph.

DEFINITION 5. (APPLICATION GRAPH) *An application graph is a 5-tuple $(A,D,\Gamma,\Theta,\lambda)$ of an SDFG (A,D), the functions $\Gamma : A \times PT \rightarrow \mathbb{N}^{\infty} \times \mathbb{N}_0^{\infty}$ and $\Theta : D \rightarrow \mathbb{N}_0^5$, and the throughput constraint $\lambda \in \mathbb{R}$. Function Γ gives for each actor $a \in A$ and each processor type $pt \in PT$ a tuple (τ,μ) with τ and μ respectively the execution time (in time units) and memory requirement (in bits) of a when assigned to a processor of type pt or ∞ if a cannot be assigned to a processor of type pt. Function Θ gives for each dependency edge $d = (a,b,p,q) \in D$ a 5-tuple $(sz,\alpha_{tile},\alpha_{src},\alpha_{dst},\beta)$ with sz the size of a token (in bits), α_{tile} the memory (in tokens) required when a and b are assigned to a single tile, α_{src} and α_{dst} the memory (in tokens) required in the source and destination tile when a and b are assigned to different tiles and β the bandwidth (in bits/time-unit) required when a and b are assigned to different tiles.*

Tab. 2 shows the values of the functions Γ and Θ for the actors and edges of the application graph shown in Fig. 3.

7. RESOURCE ALLOCATION PROBLEM

A resource allocation strategy must bind each actor from the application graph $(A,D,\Gamma,\Theta,\lambda)$ to a tile in the architecture graph $(T,C,\mathscr{L})$. As a consequence, also each dependency edge in the application graph is assigned to a connection between two tiles or to the memory inside a tile. The binding of actors to tiles is given by the *binding function*.

DEFINITION 6. (BINDING FUNCTION) *A binding function is a function $\mathscr{B} : A \rightarrow T$ which gives for every actor $a \in A$ the tile $t \in T$ to which it is bound.*

Multiple applications are scheduled on a tile using a TDMA scheduler (see Sec. 4). For each application graph, a time slice should be reserved on each tile which executes actors from the graph. A static order schedule, ordering the actor execution of an application graph on a processor, must also be constructed for each tile. Both the size of the time slice and the static order schedule are given for each tile by the *scheduling function*.

DEFINITION 7. (SCHEDULING FUNCTION) *A scheduling function is a function $\mathscr{S} : T \rightarrow \mathbb{N}_0 \times SO$, where SO is the set of all static order schedules. It gives for a tile $t \in T$ from the architecture graph a tuple (ω,S), where ω is the size of the TDMA time slice reserved for the application graph and S is a static order schedule for the actors from the application graph which are bound to t.*

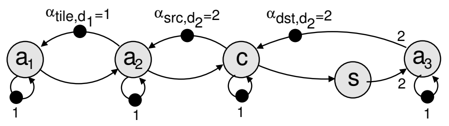

Figure 4: A binding-aware SDFG.

In the remainder, we use the following notations. For each tile $t \in T$, $t = (pt_t, w_t, m_t, c_t, i_t, o_t)$ and $\mathscr{S}(t) = (\omega_t, S_t)$; for each actor $a \in A$ and processor type $pt \in PT$, $\Gamma(a,pt) = (\tau_{a,pt}, \mu_{a,pt})$, and, for each dependency edge $d \in D$, $\Theta(d) = (sz_d, \alpha_{tile,d}, \alpha_{src,d}, \alpha_{dst,d}, \beta_d)$. We use A_t to denote the set of all actors $a \in A$ bound to $t \in T$. Using the set A_t, we define three sets of dependency edges. The first set $D_{t,tile}$ contains all dependency edges of which both the source and destination actor are bound to t. Set $D_{t,src}$ contains all dependency edges of which the source actor is bound to t and the destination actor is bound to a different tile; $D_{t,dst}$ contains all dependency edges of which the destination actor is bound to t and the source actor is bound to a different tile.

Binding and scheduling functions give a resource allocation for an application graph on an architecture graph. This allocation is called valid iff the throughput constraint is met and not more resources are allocated than available. The next section explains how throughput is computed. To guarantee that not more resources are allocated than available, the following must hold for each tile $t \in T$:

1. the allocated time slice is available: $\omega_t \leq w_t - \Omega(t)$,
2. not more memory is allocated than available: $\sum_{d \in D_{t,tile}} \alpha_{tile,d} \cdot sz_d + \sum_{d \in D_{t,src}} \alpha_{src,d} \cdot sz_d + \sum_{d \in D_{t,dst}} \alpha_{dst,d} \cdot sz_d + \sum_{a \in A_t} \mu_{a,pt} \leq m_t$,
3. not more connections are allocated than available: $|D_{t,src}| + |D_{t,dst}| \leq c_t$,
4. not more input and output bandwidth is allocated than available: $\sum_{d \in D_{t,dst}} \beta_d \leq i_t \wedge \sum_{d \in D_{t,src}} \beta_d \leq o_t$.

8. THROUGHPUT ANALYSIS

8.1 Modeling Resource Allocations in SDFGs

In [10], a technique is presented to compute the throughput of an SDFG for a platform with infinite resources. It uses a function $\Upsilon : A \rightarrow \mathbb{N}$ which assigns to every actor $a \in A$ the time it takes to execute the actor once. To compute the throughput of an application graph bound to an architecture graph, the resource allocation decisions must be taken into account. The effect of the binding function is modeled in a *binding-aware SDFG* (A_b, D_b, Υ) consisting of an SDFG (A_b, D_b) with a timing function Υ.

We use the application graph of Fig. 3 which is bound to the architecture graph of Fig. 2 to explain how binding decisions are modeled into the binding-aware SDFG shown in Fig. 4 (omitting rates 1 for clarity). Assume that the actors a_1 and a_2 are bound to tile t_1 and a_3 is bound to t_2. The execution time of a_1 and a_2 is then equal to 1 and the execution time of a_3 is equal to 2 (see Tab. 2). On a tile, only one instance of an actor can be executing at the same moment in time. This is modeled by adding a self-edge with rates one and one initial token to the actors a_2 and a_3. The binding of a_1 and a_2 to the same tile implies that the edge d_1 is also bound to t_1. The memory constraint imposed by this binding is modeled with an edge from a_2 to a_1 with α_{tile,d_1} initial tokens. This limits the storage space of edge d_1 to the memory requirement given by the application graph. Edge d_2 must use connection c_1 as its source and destination actor are bound to tiles t_1 and t_2. The delay for sending a token over the connection is modeled with the actor c.

Its execution time, $\Upsilon(c)$, is equal to $\mathscr{L}(c_1) + \lceil sz_{d_2}/\beta_{d_2} \rceil$. The self-edge on actor c enforces that the tokens are sent sequentially over the connection. Actor c is a very simple connection model. It can be replaced with a more detailed model if available, such as the network-on-chip connection model of [14]. The edges from a_3 to c and from c to a_2 enforce respectively the memory constraints of d_2 in the destination and source tile. No assumption is made on the position of two TDMA time wheels wrt each other when a token is sent over a connection. To guarantee that the throughput analysis is conservative wrt an implementation, we must assume that a token arrives at the destination tile exactly at the end of the slice which is reserved for the application. The token must then wait $w_{t_2} - \omega_{t_2}$ time steps before it can be used. This is modeled by s, which has an execution time $\Upsilon(s) = w_{t_2} - \omega_{t_2}$.

8.2 Throughput Computation

A self-timed execution of an SDFG is used in [10] to compute the throughput of the graph. In this type of execution, an actor fires as soon as sufficient tokens are present on all its inputs. The firing ends when time has advanced with the execution time of the actor. At that moment, the actor produces tokens on all of its outputs. The state of the SDFG is described by the distribution of tokens over the channels and the remaining execution time of all active actor firings. To compute the throughput, states visited during the self-timed execution are examined, and a small subset is stored, till a recurrent state, which always exists, is found. At that moment, all reachable states are found and the throughput can be computed from the periodic part of the state-space. Fig. 5(a) shows the state-space of our example SDFG of Fig. 3. States are represented by black dots and state transitions are indicated by edges. The label with a transition indicates which actors start their firing in this transition and the elapsed time till the next state is reached. Actor a_3 executes once every 2 time-units (i.e. its throughput is $1/2$). This is the maximal achievable throughput taking into account only the dependencies inherent in the SDFG. Fig. 5(b) shows the self-timed state-space of the binding-aware SDFG. The limited storage space of d_1 causes a_1 and a_2 to fire in sequence and communication and synchronization is taken into account via firings of actors c and s. Actor a_3 executes once every 29 time-units, which is the maximal achievable throughput taking the binding into account.

One option to model static order schedules in dataflow graphs is proposed in [2]. However, the SDFG must then be converted to an HSDFG. This conversion leads to an increase in the time needed for the throughput computation (see Sec. 1). To avoid this issue, the scheduling function, i.e., the time wheel allocations and static order schedules, is not modeled into the binding-aware SDFG. Instead, those are used to constrain the execution of the SDFG when constructing its state-space. This is done by extending the state of the SDFG used in [10] with information on the position of the static order schedule of each tile and the position of the TDMA time wheels. Additionally, before an actor is allowed to fire, it should not only have sufficient tokens on its inputs, but it should also be the actor which should be fired in the current position of the static order schedule. Furthermore, the remaining execution time of an actor which is firing is only reduced when the position of the TDMA time wheel of the tile to which the actor is bound indicates that the current time unit is reserved for the application to which the actor belongs. The explored state-space which considers these constraints on the execution of the SDFG is shown in Fig. 5(c). The chosen static-order schedules $(a_1a_2)^*$ and a_3^* are in line with the self-timed schedule, so they do not affect the result. 50% of the TDMA time wheels are allocated to the application. These time slot allocations cause actor a_1 and a_3 to post-pone their firings for respectively 5 and 1 time-unit (see boxes in Fig. 5(c)). As a result, actor a_3 fires only once every 30 time-units.

In [4], TDMA time slice allocations are modeled by increasing the execution time of every actor firing with the fraction of the TDMA time wheel which is not reserved by the application. This increases the execution time of actor a_3 with 5 time units. This is the maximum time our technique will post-pone the firing of an actor. In many situations, the time with which a firing is post-poned is less. Hence, our technique gives a more accurate throughput result. This reduces the resource requirements of the application while guaranteeing its timing behavior.

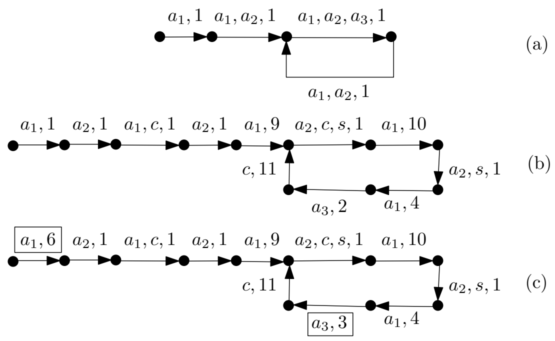

Figure 5: State-space application (a), binding-aware SDFG (b) and constrained execution (c).

9. RESOURCE ALLOCATION STRATEGY

The resource allocation strategy consists of three main steps which are each executed once. First, an actor binding is constructed, then a static order schedule for each tile containing actors of the application graph, and finally time slices are allocated.

9.1 Resource binding

The resource binding step must bind every actor from the application graph to a tile in the architecture graph. An important objective in the resource allocation strategy is to meet the throughput constraint specified by the application graph. For this reason, it is important that actors whose execution time has a large impact on the throughput of the application are considered first. The throughput of an SDFG is known to be limited by its critical cycle [20]. This is a cycle in the corresponding HSDFG with the maximal ratio between the execution time of the actors on the cycle and the number of tokens on the edges of the cycle. The conversion of an SDFG to an HSDFG can lead to an exponential increase in the number of actors in the graph [10]. This makes it infeasible to analyze the HSDFG of an application graph to identify the actors on its critical cycle. Therefore, the binding step tries to estimate the criticality of all cycles in the graph (and the actors on them) directly on the SDFG. This is done with the cost function given by Eqn. 1, with $a \in A$ an actor, C the set of cycles through a, γ the repetition vector of the application SDFG and $Tok(d)$ the number of initial tokens of edge d.

$$cost(a) = \max_{c \in C} \frac{\sum_{actors\ b \in c} \gamma(b) \cdot \operatorname*{avg}_{\{pt \in PT \mid \tau_{b,pt} \neq \infty\}} \tau_{b,pt}}{\sum_{edges\ d=(u,v,p,q) \in c} Tok(d)/q}, \qquad (1)$$

After sorting the actors in decreasing order, the resource allocation strategy tries to bind the actors in the given order to the tiles. For each actor $a \in A$ it may have to choose from a number of different tiles $T' \subseteq T$. The objective of the resource allocation strategy is to balance the load of the application equally over all tiles. The load of a tile is estimated by the relative processing performed on its processor, the fraction of memory used and the average fraction of occupied connections and bandwidth. Given a (partial) binding, and the corresponding sets A_t, $D_{t,tile}$, $D_{t,src}$, $D_{t,dst}$ (see Sec. 7), these aspects are captured in the following definitions, with t a tile and γ the repetition vector of the application SDFG.

$$l_p(t) = \frac{\sum_{a \in A_t} \gamma(a) \cdot \tau_{a,pt}}{\sum_{a \in A} \gamma(a) \cdot \max_{\{pt \in PT \mid \tau_{a,pt} \neq \infty\}} \tau_{a,pt}}$$

$$l_m(t) = (\sum_{a\in A_t} \mu_{a,pt} + \sum_{d\in D_{t,tile}} \alpha_{tile,d} \cdot sz_d + \sum_{d\in D_{t,src}} \alpha_{src,d} \cdot sz_d + \sum_{d\in D_{t,dst}} \alpha_{dst,d} \cdot sz_d)/m_t$$

$$l_c(t) = avg\left(\sum_{d\in D_{t,src}} \frac{\beta_d}{o_t}, \sum_{d\in D_{t,dst}} \frac{\beta_d}{i_t}, \frac{|D_{t,src}|+|D_{t,dst}|}{c_t}\right)$$

Eqn. 2 combines these aspects in a single cost function for a tile t.

$$cost(t) = c_1 \cdot l_p(t) + c_2 \cdot l_m(t) + c_3 \cdot l_c(t) \qquad (2)$$

The constants in the function are specified by the user of the binding step. This enables the user to trade-off how the various loads of the tile are weighted wrt each other. The algorithm tries to bind actor a to a tile $t \in T'$ in the increasing order given by the tile cost function based on the current partial binding with a bound to t. When a tile $t \in T'$ is found for which it holds that the binding of a to t does not conflict with the constraints given in Sec. 7 it binds a to t and the algorithm continues with the next actor. When all tiles are tried and no valid binding is found, the problem is considered infeasible. Tab. 3 shows the resulting binding of actors of the example for various settings of the constants.

After binding all actors to a tile, an optimization is performed to improve the load balance of the tiles. This is done by considering all the actors in reverse order. When reconsidering the binding of an actor $a \in A$ which is bound to a tile $t \in T$, its binding is first removed. Next, all tiles T' to which a can be bound are sorted using Eqn. 2, considering the load of all tiles when the whole application graph except actor a is bound. The algorithm then tries to bind a to a tile $t \in T'$ in the increasing order given by the cost function. Note that it will always be possible to find a valid binding as the original binding is one of the bindings which is tried.

9.2 Constructing static-order schedules

For each tile, a static-order schedule must be constructed that orders the firings of all actors bound to it. A list-scheduler is used to construct these static-order schedules for all tiles at once. The schedules are constructed via an execution of the binding-aware SDFG, assuming that for each tile 50% of the available time wheel is allocated to the application graph. Through actors like s in Fig. 4 the delay for tokens sent between tiles is taken into account in the schedule construction. When an actor becomes enabled in the execution of the binding-aware SDFG it does not start its firing immediately. Instead the actor is added to the ready list of the tile it is bound to. When no actor is firing on the tile, the first actor is removed from the list and its firing is started. At this moment, the actor is added to the schedule of the tile. The execution ends as soon as a recurrent state is found. At this point, a finite-length schedule has been constructed for each tile. For our example, the scheduler constructs for tile t_1 a schedule with 17 states - $a_1a_2a_1a_2a_1a_2a_1a_2a_1(a_2a_1a_2a_1a_2a_1a_2a_1)^*$. After constructing the schedule, an optimization is performed to remove all recurrent occurrences of the same scheduling sequence. In this way, the schedule on t_1 is reduced to $(a_1a_2)^*$.

9.3 Time slice allocation

The final step of the resource allocation strategy involves the allocation of time slices for all tiles. A binary search algorithm is used, which guarantees that a time slice allocation satisfying the throughput constraint is found if it exists. The search between the initial bounds of 1 time slice and the entire (remaining) time wheel continues until the throughput of the binding-aware SDFG constrained by the current slice allocation is at most 10% larger than the throughput constraint. It ends unsuccessfully if the allocation of the entire remaining time wheels is insufficient to meet the throughput constraint.

If successful, the slice allocation step so far allocates equal fractions of the remaining time wheel for each tile to which at least one actor is bound. This is based on the assumption that the processing load is perfectly balanced over the tiles. However, in case of an imperfect load balance it may be possible to reduce the allocated time slices using another binary search. The upper bound for every tile $t \in T$ is equal to the slice found in the previous step (i.e. ω_t) and the lower bound for every tile t is $\lfloor l_p(t) \cdot \omega_t / \max_{t\in T} l_p(t) \rfloor$ which takes into account the relative load of each tile. The binary search is continued till the slices can no longer be reduced without violating the throughput constraint.

Table 3: Binding of actors to tiles.

c_1, c_2, c_3	a_1	a_2	a_3
1,0,0	t_1	t_1	t_2
0,1,0	t_1	t_2	t_2
0,0,1	t_1	t_1	t_1
1,1,1	t_1	t_1	t_2

10. EXPERIMENTAL RESULTS

10.1 Experimental setup

A benchmark is needed to evaluate the run-time and quality of the resource allocation strategy and explore the impact of different parameter values in the tile-cost function. A benchmark of four ordered sets of application graphs was generated using SDF^3 [22]. The first set contains processing intensive graphs that have large execution times, do not communicate too often and have small token sizes and states. The second and third set are memory and communication intensive. The fourth set contains both SDFGs which are balanced wrt their processing, memory and communication requirements and graphs which are dominated by one or two of these aspects. For each set, 3 different sequences of graphs were generated to eliminate effects from the random generator.

Three different architecture graphs are used in the experiments. Each architecture graph is a 3x3 mesh-based architecture with 3 different types of processors. The graphs differ in the memory size and number of supported connections. All processors have an equally sized time wheel. The connections between the tiles are assigned a latency which is small compared to the execution time of the actors. This is realistic as the latency of an interconnect in an MP-SoC is typically much smaller than the execution time of the executed tasks.

Each set of graphs from our benchmark has been tested with five different settings for the tile-cost function (see first column Tab. 4). For a given tile-cost function, architecture graph, and sequence of application graphs, resources are allocated to application graphs till no valid resource allocation is found for a graph. This gives a conservative estimate on the number of applications for which resources can be allocated on the platform. A design-time preprocessing step that orders the applications to optimize the order in which they are handled, a (run-time) mechanism that rejects an application and continues with the next one or another implementation version of the rejected application, and/or a platform dimensioning step may improve the results.

10.2 Experiments on the benchmark

Tab. 4 shows the number of application graphs which could be bound for each tile-cost function and set of graphs from the benchmark. It averages over the 3 sequences of graphs contained in each set and the 3 architecture graphs used in the experiments. The average run-time of the strategy for a single application graph on a P4 at 3.4GHz is 5 seconds. On average, each run of the algorithm invokes 16.1 times the throughput computation technique described in Sec. 8 which would make a trajectory based on a conversion to HSDFG very expensive. The result of the set with computation intensive tasks (set 1) shows that it is important to consider not only the processing (1st tile-cost function), but also the communication (3rd tile-cost function). The reason for this is that when the processing load is balanced, many dependency edges are bound to a connection, requiring synchronization between tiles. As a result, larger time slices need to be allocated on the tiles to meet the throughput constraint than when more actors of a single application are bound to the same tile. The latter effect is achieved by the 3rd tile-cost function. As expected, the 2nd tile-cost function, which considers

Table 4: Average number of application graphs bound.

	c_1, c_2, c_3	set 1	set 2	set 3	set 4
1:	$1,0,0$	20.22	5.22	7.56	18.56
2:	$0,1,0$	18.78	8.00	11.33	23.33
3:	$0,0,1$	29.22	7.56	12.89	25.00
4:	$1,1,1$	18.44	6.50	10.33	23.56
5:	$0,1,2$	24.56	8.00	12.89	30.11

Table 5: Resource efficiency for set 4.

	timewheel	memory	connections	input bw	output bw
1:	0.71	0.82	0.88	0.83	0.70
2:	0.85	0.93	1.00	1.00	1.00
3:	0.72	0.82	0.67	0.47	0.67
4:	0.96	0.98	1.00	0.94	0.79
5:	1.00	1.00	0.94	0.72	0.92

the memory resources, performs best on the memory constrained graphs (set 2). Similarly, the 3rd tile-cost function, which considers the connection and bandwidth resources, performs best on the communication intensive graphs (set 3). The results show further that the 4th tile-cost function, which considers all resources, gives an average result for all sets. This is to be expected as it balances all resources and as such does not give priority to the most constrained resource in any of the sets. The results show that it is important to minimize the number of connections in order to limit the synchronization overhead. They also show that balancing the memory usage is an important secondary objective as the 2nd tile-cost function gives good results for most sets. Based on these observations, we devised a 5th tile-cost function $(0,1,2)$ emphasizing minimization of the number of connections while balancing memory usage. Using this cost function, the largest number of application graphs is allocated onto the architecture for the set with mixed resource requirements (set 4). This shows that it is possible to guide the resource allocation through our tile-cost function.

The objective of the resource allocation strategy is to perform resource allocation for as many graphs as possible while keeping the total amount of resources used as low as possible. Tab. 5 shows the resource usage after resource allocation for the graphs from the 4th set. For comparison, the resource usage of each resource is normalized wrt the largest usage of this resource when using any of the 5 tile-cost functions. The results show that the 3rd tile-cost function achieves a good result by allocating a large number of applications (see Tab. 4) to the smallest amount of resources. It also shows that the 5th tile-cost function, which allocates the largest number of applications to the architecture, effectively uses the available resources. These results confirm that communication has a major impact on resource usage. This is as expected because we do not assume any synchronization between time wheels, meaning that communication has a large impact on the guaranteed throughput that can be obtained. Communication needs to be balanced by allocation of large time slices on the communicating processors. The table also shows that for all tile-cost functions the resource occupancy of the various resources is similar, indicating that all cost functions give a balanced resource utilization.

When doing resource allocation using the 5th tile-cost function on the graphs from the 4th set of our benchmark, we found that on average 73% of the resources in the architecture graphs are used. This result is reasonable because it is achieved without any optimizations. Resource utilization can be increased when doing system dimensioning, re-ordering applications before allocation and/or applying mechanisms to transform applications or to continue allocating applications after one application fails to be bound.

10.3 Experiments on a multimedia system

Besides the synthetic graphs, a multimedia system consisting of three H.263 decoders (each 4 actors) and an MP3 decoder (13 actors) is used. The four application graphs are bound and scheduled on a 2x2 mesh with 2 generic processors and 2 accelerators. The used tile-cost function $(2,0,1)$ focuses on balancing the processing load and it tries to limit the communication. The memory usage is ignored as all potential bindings have similar memory requirements. The strategy finds a resource allocation with a balanced resource utilization. The run-time of the strategy is 8 minutes of which approx. 90% is spent on the time slice allocation. The time slice allocation step performs 34 times a throughput computation in order to minimize the slices used by the applications. Resource allocation techniques that convert the SDFG to an HSDFG and compute throughput on the HSDFG would take several hours when performing a similar amount of throughput checks. (Recall that one throughput computation for the H.263 decoder takes in that case 21 minutes.) This experiment shows that our resource allocation strategy can handle SDFGs whose corresponding HSDFGs are large (14275 actors) within a limited run-time. It also shows that through a combination of modeling resource allocation decisions in the SDFG (as proposed e.g. in [2, 6]) and by constraining the execution of the graph it becomes feasible to analyze the throughput of realistic applications when bound to a heterogeneous MP-SoC.

11. CONCLUSIONS

We have presented the first resource allocation strategy that can bind multiple SDFGs to a heterogeneous multi-processor system while giving throughput guarantees to each individual application, also in a context of resource sharing. The technique can deal with multi-rate and cyclic dependencies between actors (tasks) without converting it to a homogeneous SDFG. The strategy uses generic cost-functions to steer the binding of the application to the architecture and incorporates an efficient technique to compute the throughput of a bound and scheduled SDFG. The experiments show that this enables a balanced resource allocation of time-constrained applications bound to a multi-processor system-on-chip.

12. REFERENCES

[1] B. Ackland, et al. A single chip 1.6 billion 16-b MAC/s multiprocessor DSP. *IEEE Journal of Solid-State Circuits 35*, 3 (2000), p. 412–424.

[2] N. Bambha, et al. Intermediate representations for design automation of multiprocessor DSP systems. *Design Automation for Embedded Systems 7*, 4 (2002), p. 307–323.

[3] M. Bekooij, et al. Predictable multiprocessor system design. In *SCOPES'04, Proc.* (2004), Springer, p. 77–91.

[4] M. Bekooij, et al. *Dynamic and Robust Streaming In and Between Connected Consumer-Electronics Devices*. Springer, 2005, ch. Dataflow Analysis for Real-Time Embedded Multiprocessor System Design, p. 81–108.

[5] S. Bhattacharyya, et al. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[6] G. Bilsen, et al. Cyclo-static dataflow. *IEEE Trans. on signal processing 44*, 2 (1996), p. 397–408.

[7] D. Culler, et al. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[8] C. Erbas, et al. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Trans. on Evolutionary Computation 10*, 3 (2006), p. 358–374.

[9] O. Gangwal, et al. *Dynamic and Robust Streaming In and Between Connected Consumer-Electronics Devices*. Springer, 2005, ch. Building Predictable Systems on Chip: An Analysis of Guaranteed Communication in the AEthereal Network on Chip, p. 1–36.

[10] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *ACSD'06, Proc.* (2006), IEEE, p. 25–34.

[11] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular noc architectures. *IEEE Trans. on CAD 24*, 4 (2005), p. 551–562.

[12] K. Kuchcinski. Constraint-driven scheduling and resource assignment. *ACM Trans. on Design Automation of Electronic Systems 8*, 3 (2003), p. 355–383.

[13] E. Lee, et al. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers 36*, 1 (1987), p. 24–35.

[14] A. Moonen, et al. Timing analysis model for network based multiprocessor systems. In *Progress'04, Proc.* (2004), STW, p. 122–130.

[15] O. Moreira, et al. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *RTAS, Proc.* (2005), IEEE, p. 332–341.

[16] P. Murthy. *Scheduling Techniques for Synchronous Multidimensional Synchronous Dataflow*. PhD thesis, UC Berkeley, 1996.

[17] P. Paulin, et al. Application of a multi-processor SoC platform to high-speed packet forwarding. In *DATE'04, Proc.* (2004), IEEE, p. 58–63.

[18] P. Poplavko, et al. Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip. In *CASES, Proc.* (2003), ACM, p. 63–72.

[19] M. Rutten, et al. A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design & Test of Computers 19*, 4 (2002), p. 39–50.

[20] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, 2000.

[21] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *DAC'06, Proc.* (2006), ACM, p. 899–904.

[22] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *ACSD'06, Proc.* (2006), IEEE, p. 276–278.