A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour

Sander Stuijk[1], Marc Geilen[1], Twan Basten[1,2]
[1] Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands
[2]Embedded Systems Institute, Eindhoven, The Netherlands
{s.stuijk, m.c.w.geilen, a.a.basten}@tue.nl

***Abstract*—The design of new embedded systems is getting more and more complex as more functionality is integrated into these systems. To deal with the design complexity, a predictable design flow is needed. The result should be a system that guarantees that an application can perform its own tasks within strict timing deadlines, independent of other applications running on the system. Synchronous Dataflow Graphs (SDFGs) provide predictability and are often used to model time-constrained streaming applications that are mapped onto a multiprocessor platform. However, the model abstracts from the dynamic application behaviour which may lead to a large overestimation of its resource requirements. We present a design flow that takes the dynamic behaviour of applications into account when mapping them onto a multiprocessor platform. The design flow provides throughput guarantees for each application independent of the other applications while taking into account the available processing capacity, memory and communication bandwidth. The design flow generates a set of mappings that provide a trade-off in their resource usage. This trade-off can be used by a run-time mechanism to adapt the mapping in different use-cases to the available resource. The experimental results show that our design flow reduces the resource requirements of an MPEG-4 decoder by 66% compared to a state-of-the-art design flow based on SDFGs.**

I. Introduction

Modern embedded multimedia systems are often executing multiple applications concurrently. A user may for example use a mobile phone to listen to his favourite music using an MP3 decoder while at the same time he is watching an accompanying video that is being decoded using an MPEG-4 decoder. The typical user expects that both decoders have a robust behaviour and that their performance is guaranteed [4]. This requires that every application running on the system has a predictable timing behaviour which is independent of other applications running on the same system. The design flow that binds and schedules the applications onto the hardware platform should provide this predictability.

Several trends need to be considered when developing a predictable design flow. In the architecture domain, there is a trend to use heterogeneous multiprocessor platforms to meet the computational requirements of novel applications [16]. Furthermore, the number of use-cases (i.e., combinations of applications) that an embeded system has to support is growing rapidly. Different use-cases may require a different mapping of an application onto a multiprocessor platform. In some use-cases, an application could for example be allowed to use a lot of computational resources, but limited storage resources, whereas the situation may be exactly opposite in other use-cases. A design flow can support this by creating at design-time a number of different mappings of an application that provide a trade-off in their resource requirements. At run-time, the most suitable mapping can then be selected based on the resource usage of the applications which are already running on the platform [17], [23].

Modern streaming applications are becoming increasingly complex and dynamic. Existing design flows (e.g., [14], [20]) model these applications using relatively simple and static models, such as (homogeneous) synchronous dataflow graphs [11]. These models abstract from the dynamic behaviour of an application which may lead to a large overestimation of its resource requirements. The dynamic behaviour of an application can be taken into account in a design flow by using a scenario-based design approach [6]. In this approach, the dynamic behaviour of an application is viewed upon as a collection of different behaviours (*scenarios*) occuring in some arbitrary order, but each scenario by itself is fairly static and predictable in performance and resource usage. Therefore, resource allocation can be performed for each scenario using existing design flows. However, these design flows can only provide timing guarantees per scenario. They cannot guarantee the timing behaviour when switching between scenarios. A predictable design flow should however also guarantee the timing behaviour of an application when switching between scenarios.

This paper presents a design flow that maps throughput-constrained applications, whose behaviour can be captured with a set of scenarios, onto a multiprocessor platform while providing throughput guarantees. Applications are modeled using a so-called *scenario graph* that captures the behaviour of each scenario with a synchronous dataflow graph. The dataflow graphs of different scenarios may differ in all aspects (e.g., communication rates, execution times). The design flow generates a number of alternative mappings for each application. These mappings provide different trade-off's between the amount of compute, storage, and communication resources that are used from the platform.

This work was supported in part by the EC through FP7 IST project 216224, MNEMEE.

These can be used at run-time to adapt to different use-cases. Thus, the flow addresses both the dynamic behaviour within applications and the dynamic behaviour between applications. We show that this new design flow reduces the resource requirements of an MPEG-4 decoder by 66% when compared to an existing state-of-the-art design flow [20]. Resource savings are also shown for an MP3 decoder (up-to 21% less memory and up-to 23% less bandwidth).

The remainder of this paper is organized as follows. Sec. II discusses related work. Sec. III introduces the scenario graph model. Sec. IV and Sec. V discuss the application and architecture model. Sec. VI presents a method to perform throughput analysis for an application mapped onto a multiprocessor platform. Our predictable design flow is discussed in Sec. VII. Experimental results are presented in Sec. VIII. Sec. IX concludes this paper.

II. Related Work

Resource allocation for time-constrained acyclic graphs has been studied in [8], [9], [10]. Hashemi et. al [8] propose an algorithm based on graph partitioning to compute a binding and schedule which maximizes the throughput of the application. The acyclic graph model can however not capture cyclic dependencies between subsequent executions of the task graph. Our application model can capture such dependencies and our design flow considers them during resource allocation. This improves pipelining of different executions of the same task. Hu et. al assume that every task can only be bound to a single processor type [9]. Their strategy only decides on which processor (i.e. location) to use. Our strategy has to decide on the processor type, its location, and the schedule of tasks on the processors. It also works for a larger class of models. In [10], 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.

Several design flows [2], [12], [15], [20] have been presented to map multiple applications onto a multiprocessor platform with the objective to minimize resource usage while meeting a throughput constraint. These flows assume that each application is modeled with a Cyclo-Static Dataflow Graph ([15]), or an SDFG ([2], [20]), or a homogeneous SDFG ([12]). These models abstract from (most of) the dynamic behaviour of an application. This may lead to a large overestimation of the resource requirements of the application. Our application model and design flow are able to capture and exploit the dynamic behaviour when allocating resources. The design flow from [7] can deal with applications that exhibit a dynamic behaviour. The design flow can however not provide timing guarantees which is needed to guarantee the robust behaviour of multimedia systems.

All aforementioned work only considers the binding and scheduling of an application onto a multiprocessor platform, our flow considers also these steps, but it also considers the dimensioning of the buffers between the tasks of an application. These buffers have a large impact on the throughput of the application and on its memory requirements. Our design flow considers this trade-off when allocating resources. Existing flows in contrast assume either infinite buffer space, which is unrealistic, or they assume that the assigned buffer space is input to the design flow.

Different use-cases may require a different mapping of an application on the platform. All aforementioned flows generate only a single mapping. The user must modify the mapping constraints to obtain different mappings for the same application. In contrast, our flow generates a set of mappings that provide a trade-off in their resource requirements. A run-time mechanism (e.g., [17], [23]) can then select the most suitable mapping based on the resource usage of all applications that are active in the use-case.

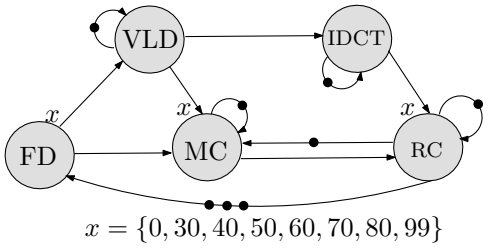
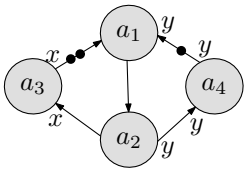


Figure 1. Scenario graph of an MPEG-4 SP decoder.

III. Scenario Graphs

Synchronous Dataflow Graphs (SDFGs) [11] are used to model time-constrained multimedia applications. They allow 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 [5], [21]. The graph in Fig. 1 is an SDFG when *x* is assigned a constant value (e.g. 99). This graph models an MPEG-4 Simple Profile decoder. The nodes, 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* (indicated next to edge ends; rates 1 are omitted for clarity). An actor can only fire if sufficient tokens are available on the edge from which it consumes. Tokens thus capture dependencies between actor firings. Such dependencies may originate from data dependencies, but also from dependencies on shared resources.

The rates determine how often actors have to fire wrt each other such that productions and consumptions are balanced. These rates are constant, which forces an SDFG to execute in a fixed repetitive pattern, called an *iteration*. An iteration



$rate$	s_1	s_2
x	1	2
y	0	1

Figure 2. Example application graph.

actor	$s_1 \times p_1$	$s_2 \times p_1$	$s_1 \times p_2$	$s_2 \times p_2$
a_1	(1,10)	(1,10)	(1,10)	(1,10)
a_2	(1,7)	(1,7)	(1,7)	(1,7)
a_3	(3,13)	(3,13)	(10,13)	(10,13)
a_4	(0,0)	(5,10)	(0,0)	(3,10)

Table I
PROPERTIES (EXECUTION TIME, MEMORY REQUIREMENT) OF THE EXAMPLE APPLICATION.

consists of a set of actor firings that have no net effect on the token distribution. These actor firings typically form a coherent collection of computations. An iteration could for example correspond to the processing of a frame in an audio or video stream. This makes iterations the natural granularity for defining scenarios, from the perspective of the application and from the perspective of the model. Note that subsequent iterations are allowed to overlap in time. Hence, different scenarios may be active simultaneously, typically in a pipelined fashion.

The dynamic behaviour of an application can be captured in a set of scenarios. Each scenario can be modeled with an SDFG. These SDFGs together form a model of the application. This set of SDFGs is called a *scenario graph*. This model is a restricted form of the Scenario-Aware Dataflow Model [22]. Consider, as an example, the MPEG-4 decoder of [22] shown in Fig. 1. The frame detector (FD) models the part of the application that determines the frame type and the number of macro blocks to decode. The modeled decoder supports two different types of frames (I or P). When a frame of type I is found, a total of 99 macro blocks must always be processed. This scenario is called *I*99. A frame of type P requires processing between 0 and 99 macro blocks. The workload varies considerably depending on the number of macro blocks that is processed. Therefore, a number of different scenarios *Px* are defined based on the number of macro blocks that must be processed. The graph contains different scenarios for the situations in which (up to) 0, 30, 40, 50, 60, 70, 80, or 99 macro blocks are processed for a single P frame. Within each scenario, the VLD and IDCT operations are performed for every individual block. The other operations are performed once per frame. Therefore, the communication rates vary with each scenario. As a consequence, x is set equal to the maximum number of macro blocks that may need to be processed in the scenario.

IV. Application Model

The structure of an application can be described with a scenario graph. A design flow needs also information on the resource requirements of the actors and edges in the graph. An application with its resource requirements is described by an *application graph*. Fig. 2 and Tab. I show an example application graph with two scenarios (s_1 and s_2). The table shows for each actor the execution time (in time units) and memory requirements (in bytes). These numbers may depend on the scenario s_i and processor type p_j to which the actor may be mapped. It shows, for example, that actor a_3 needs 3 time units to execute on processor type p_1 and 10 time units on type p_2. It is also interesting to note that in scenario s_1, the edges connected to actor a_4 have rate 0 and that a_4 has zero execution time and no memory requirements. Hence, no resources need to be allocated for a_4 in s_1 as this actor is not active in s_1. So, it is possible to model scenarios in which not all actors are active.

The execution time of an actor depends on the scenario in which the actor is fired. This makes it possible to reduce the over-allocation of resources when using a scenario graph instead of a single SDFG to model an application. Consider as an example the MPEG-4 SP decoder shown in Fig. 1. The worst-case execution time (WCET) of actor RC (350 time units) occurs in scenario *I*99. Actor MC has an execution time of 0 time units in *I*99. Its WCET (390 time units) occurs in scenario *P*99. Actor RC has an execution time of 320 time-units in *P*99. A scenario graph can capture the fact that there will never be a frame in which both actors reach their WCET. An SDFG that conservatively models the MPEG-4 SP decoder must on the other hand use these WCET as the execution times of the actors. As a result, a predictable design flow that uses this SDFG will always over-allocate resources in the multiprocessor platform.

V. Multiprocessor Platform

The architecture template in our work is similar to the tile-based multiprocessor of [3] 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. 3 shows an example multiprocessor platform with three connected tiles. Each tile contains a 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.

The processors in a multiprocessor platform may have to be shared between tasks (i.e., actors in case of a scenario graph) of one or more applications that are executing on the platform. A scheduler must be used to arbitrate the access to the processors. Similar to [15], [20], we use a time-division multiple-access (TDMA) scheduler to order the execution of different applications. This allows us to provide timing

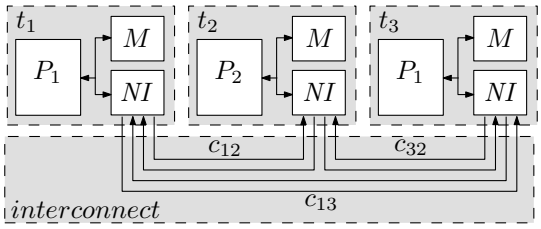


Figure 3. Example multiprocessor platform.

guarantees on the behaviour of individual applications as a TDMA scheduler virtualizes the processor resources. A static-order schedule is used on each processor to schedule the actors that belong to the same application. This schedule can be computed at design-time. So, there is no run-time scheduling overhead.

Tab. II gives the values of all elements in the multiprocessor platform of Fig. 3. It specifies for each tile its processor type (*pt*), the size of the processor's TDMA time wheel (*w*) (in time units), the memory size (*m*) (in bytes), the maximum number of incoming and outgoing connections supported by the NI (*ci*,*co*), and the maximum incoming and outgoing bandwidth (*bi*,*bo*) (in bytes/time-unit). The connections between tiles introduce a latency when data is sent between them. The column labelled *l* in Tab. II gives the latency for the connections in our example platform. 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 that can be sent per time-unit (i.e. bandwidth) is limited by the incoming, *bi*, and outgoing bandwidth, *bo*, of the tiles.

VI. Throughput Computation

In [20], a technique is presented to compute the throughput of an SDFG that is bound and scheduled on a multiprocessor platform. This technique starts with modeling all binding decisions in the graph. Next, the throughput is computed using the state-space exploration technique from [5]. This throughput computation technique can be extended to scenario graphs. Fig. 4 shows how a binding of our example application (Fig. 2) on our example architecture (Fig. 3) is modeled in a scenario graph. The actors a_1, a_2, and a_3 are in this example bound to tile t_1 and actor a_4 is bound to tile t_2. On a tile, only one instance of an actor can be executing at the same moment in time. This is modelled with the self-edges with one initial token that are connected to a_1, a_2, a_3, and a_4. Actor a_4 is bound to a different tile than a_1 and a_2. So, tokens that are communicated between a_4 and a_1 or a_2 must be sent via a connection in the platform. The actors $a_{c,1}$ and $a_{c,2}$ model the delay of sending tokens through a connection. Actor a_c is a very simple connection model. It can be replaced with a more detailed model, such as the network-on-chip connection model of [13]. We assume that the TDMA time wheels on tile t_1 and tile t_2 do not have to

tile	*pt*	*w*	*m*	*ci*	*bi*	*co*	*bo*	*connection*	*l*
t_1	p_1	100	4000	10	12	10	12	c_{31}, c_{13}	4
t_2	p_2	100	5600	10	12	10	12	others	3
t_3	p_1	100	4000	10	12	10	12		

Table II
Properties of the example platform.

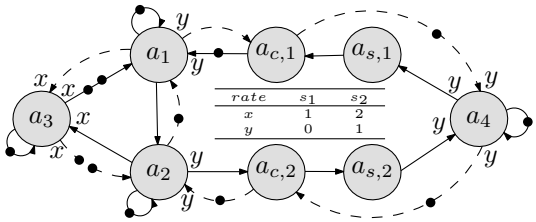


Figure 4. A binding-aware scenario graph.

be synchronized wrt each other. The actors $a_{s,1}$ and $a_{s,2}$ are used to decouple the TDMA time wheels on the processors. The execution times of these actors are chosen such that they model the worst-case position of the two time wheels wrt each other. This guarantees that the throughput analysis of the model is conservative wrt an implementation. The design flow, as explained in the next section, assigns a finite storage-space to each edge in the application graph. These storage-space constraints are modelled by the dashed edges in Fig. 4[1]. The edge from a_2 to a_4 and the edge from a_4 to a_1 are bound to the interconnect. Storage-space must be allocated for these edges in both tiles. Therefore, there are two dashed edges in Fig. 4 connecting a_1 and a_4 via actor $a_{c,1}$ and two dashed edges connecting a_2 and a_4 via $a_{c,2}$.

Once a resource allocation has been modeled into a scenario graph, its throughput can be computed using a state-space exploration similar to [20]. The notion of a state and self-timed execution are adapted to take scenarios and scenario transitions into account. The throughput analysis essentially explores all cycles in the state-space. The longest cycle determines the bound on the throughput that can be guaranteed. This scenario-aware state-space exploration can take arbitrary sequences of scenarios into account. The set of possible scenario transitions can also be restricted through a Finite State Machine (FSM). This would provide for a tighter analysis. Such an FSM can for example be used to limit the possible scenario sequences in an MPEG-4 decoder to those sequences in which an I frame is always followed by 11 P frames.

VII. Predictable Design Flow

This section introduces our predictable design flow. The first subsection gives an overview of the steps in the flow. The remaining subsections discuss the details of all steps.

[1]Dashed edges are only used for clarity. There is no difference between edges as far as dataflow theory is concerned.

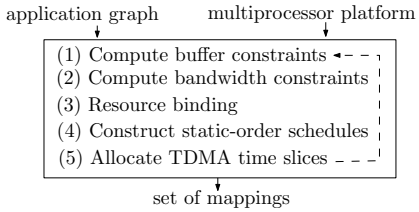


Figure 5. Predictable design flow.

A. Overview

Our predictable design flow maps a throughput-constrained application graph (see Sec. IV) to a multiprocessor platform (see Sec. V). The dynamic behaviour of the application is captured in the application graph with a set of SDFGs (i.e., one for each scenario). We assume that actors and edges in each SDFG are labelled with a name. Our flow maps actors or edges that appear in different SDFGs, but which have the same name, to the same resources (e.g. processor, memory, etc.). This gives a unified mapping of all SDFGs in the application graph. Using this unified mapping, we avoid that data items (i.e. tokens) need to be moved between different memories when switching between scenarios. Hence, there is no overhead when switching between scenarios.

Fig. 5 shows the main steps in our design flow. The flow starts with refining the resource requirements of the application. The application graph only specifies the resource requirements of the actors. Estimating the resource requirements of the edges (i.e., storage space, bandwidth) is performed in the first two steps of the flow. In the third step, the flow binds each actor from the application graph to the resources in the multiprocessor platform. This step generates a set of alternative bindings that provide a trade-off in their resource requirements. The last two steps of the flow are performed for each of these bindings individually. Step 4 creates a static-order schedule for each tile that contains actors of the application graph. Finally, TDMA time slices are allocated on these tiles. The result of the flow is a set of mappings of the application graph on the multiprocessor platform. A mapping is called valid iff the throughput constraint is met (according to the model) and not more resources are allocated than available. It is possible that no valid mapping is found when the storage space assigned to the edges is too constrained to meet the throughput constraint. In that case, the design flow iterates back to step 2 and increases the storage space assigned to the edges of the graph. The design process ends as soon as a non-empty set of valid mappings has been found or when the storage space assigned to the edges can no longer be increased. In the latter case, the design flow is not able to find a mapping that satisfies the throughput constraint. More resources should be added to the platform or the application and its constraint should be modified in order to find a mapping that meets the throughput constraint.

Several steps of the design flow address NP complete problems (e.g., finding the optimal resource binding and scheduling). It is therefore not possible to compute the optimal mapping for realistic applications within an acceptable run-time. Our design flow uses therefore heuristics to prune the design space. The experimental results show that these heuristics allow our design flow to generate mappings of real applications on a multiprocessor platform while requiring a run-time of only a few minutes.

B. Buffer constraints

Tokens that are communicated over the edges of an application graph must be stored in memory. The amount of storage space that is allocated to these edges has a large impact on the achieved throughput of the application. Allocating space for more than one token to an edge might increase throughput because it may increase pipelining opportunities. The size of the storage space must be chosen such that the throughput requirement is met, while minimizing the required storage space. The exact throughput constraint for parts of the system is however not known at the start of the flow. Therefore, a trade-off must be made between the realizable throughput and the storage requirements of the application graph. Each scenario in the application graph corresponds to an SDFG. Using the technique from [21], the throughput-storage space trade-off space of individual scenarios (SDFGs) can be found. Next, these trade-off spaces are combined into a single trade-off space for the application graph. This is done by unifying these spaces under the assumption that edges from different SDFGs, but with the same name, must be assigned the same storage space in memory. In this way, the flow avoids that tokens (i.e., data) must be moved between different memory locations when switching between scenarios. The obtained trade-off space contains all distributions of storage space that achieve the maximal throughput under a given total storage size constraint. The flow selects the smallest storage distribution that meets the throughput constraint of the application graph to limit the storage space of the edges. The source and destination actor of an edge might in the end be bound to different tiles. In that case, the storage space allocated to an edge has to be split over both tiles. The flow assigns in this case sufficient storage-space to the source and destination side to complete one firing of the source and destination actor. This allows the actors to perform a complete firing without the need to communicate data through the interconnect, which simplifies the execution time analysis of actors. Any storage-space that is left is divided evenly over both sides. In this way, both sides get an equal amount of storage-space to use as slack. It is not possible to make a better distribution at this point in the flow as the binding of the actors to the processors is not known yet. Therefore, the flow has no knowledge on the processor load which will eventually determine the available slack.

C. Bandwidth constraints

The dataflow model assumes that edges have an infinite bandwidth and no latency. So, communication of tokens over an edge takes no time. Edges whose source and destination actor are bound to different tiles will be bound to a connection in the interconnect. This connection has a latency and finite bandwidth. The latency is specified in the multiprocessor platform. The bandwidth is constrained by the incoming and outgoing bandwidth offered by the tiles. An edge will typically only require a fraction of this bandwidth. Step 2 of our flow estimates the amount of bandwidth that must be assigned to the edges of the application graph. To avoid reconfiguration, an equal amount of bandwidth must be allocated to an edge in all scenarios. Step 2 estimates for each edge d the bandwidth requirement in all scenarios. The maximum bandwidth which is required for edge d across all scenarios is then taken as the bandwidth constraint of the edge.

The bandwidth requirement of an edge d depends on the amount of data that must be communicated and on the throughput constraint of the application. The latter constrains the amount of time that is available for the communication. The former depends on the size of the tokens and on the number of tokens that are communicated in one iteration. Note that the number of communicated tokens is scenario dependent. Multiplying the token size, the number of tokens communicated per iteration, and the throughput constraint gives the average number of bytes that must be communicated per time-unit to meet the throughput constraint. This number is taken as the bandwidth requirement of the edge. This bandwidth would be too small if all edges are mapped to an interprocessor connection. In that case, no time slack would be left for the actual computation. In practice, only a fraction of the edges is mapped to an interprocessor connection and the bandwidth constraint is sufficient. In this way, the flow saves resources as it does not allocate the worst-case bandwidth.

D. Resource binding

The resource binding step (step 3) must bind every actor from the application graph to a tile in the multiprocessor platform. The application graph contains an SDFG for each scenario. Because we assume a unified mapping, the flow must bind actors that appear in different SDFGs, but which have the same name, to the same resources. For simplicity, we assume that when we talk in this section about an actor (or edge) of the application graph we refer to the set of actors (or edges) from all SDFGs which have the same name.

An important objective of the design flow is to meet the throughput constraint of the application. For this reason, it is important that actors whose execution time has a large impact on the throughput of the application are considered first. Actors are therefore ordered based on the average execution time needed to complete one iteration of all scenarios. After sorting the actors in decreasing order, the resource binding step tries to bind the actors in the given order to the tiles. It is possible that an actor can be mapped to a number of different tiles. The algorithm constructs all possible (partial) bindings. In each binding, it assigns the actor to a processor inside the tile and it allocates the required memory space of the actor (see Tab. I) in the memory of the tile. When binding an actor, the algorithm may also implicitly bind an edge to the platform. This happens when the other actor that is connected to the edge is already bound to the platform. Therefore, the algorithm checks which edges have been bound implicitly due to the actor binding. It then allocates resources (i.e., storage space in the memories and bandwidth in the network interfaces) for these edges. As mentioned before, the binding of one actor to the platform may result in a number of different (partial) bindings. When no partial binding is found, the algorithm terminates and the problem is considered infeasible within the current constraints. Otherwise, the algorithm continues with the next actor. It tries to add this actor to all partial bindings which have been constructed in the previous iteration of the algorithm. This is done using the same procedure as described above. Note that this may lead to an explosion in the number of bindings that is considered. Therefore, the number of partial bindings that is considered when binding the next actor is limited by a user-defined number X. After each iteration of the algorithm, the number of partial bindings that is carried over to the next iteration is limited to at most X. When more than X partial bindings have been found, a selection procedure is used to reduce the number of partial bindings. This procedure works as follows. Each binding can be seen as a point in an N-dimensional space where each resource in the platform forms its own dimension. The procedure selects those X points which have the largest distance wrt to each other in the space. This gives a spread in the resource requirements of the different mappings. As a first point, the procedure selects a mapping that uses the smallest amount of processors. This mapping is typically the least resource demanding solution and it is therefore interesting to keep. The selection procedure limits the total number of partial bindings that is evaluated by the flow to X times the number of actors in the graph times the number of processors in the platform. Through X, the user of the design flow can make a trade-off between the run-time of the flow and the number of partial bindings that is evaluated. Note that the evaluation of a single binding requires typically less than 1ms. So, in practice it is possible to evaluate many different bindings within a limited run-time.

E. Static-order scheduling

When a resource is shared between different actors from the same application, a static-order schedule must be constructed that orders the accesses to the resource. Step 4 of the flow must construct such a schedule for each tile in the platform. The application graph contains an SDFG for each scenario. The number of actor firings that must be scheduled depends

on the scenario that is executed. Actor a_1 in our example application (Fig. 2) must for example be fired once per iteration in scenario s_1 and twice per iteration in scenario s_2. Actor a_3 on the other hand needs to be executed only once, independent of which scenario is executed. This example shows that the relative actor firing counts are not constant between all scenarios. As a result, it is not possible to construct a static-order schedule that can be used for all scenarios. Instead, a static-order schedule must be constructed for each scenario individually. When switching between scenarios, the processors must switch between static-order schedules. To avoid a complex switching mechanism, we constrain the number of actor firings in a static-order schedule to the number of actor firings needed to complete one iteration. Consider as an example our example application and assume that the actors a_1, a_2, and a_3 are mapped to the same tile. For tile t_1, the scheduler in step 4 must construct two schedules (one for scenario s_1 and one for scenario s_2). The schedule for s_1 contains one firing of each actor. The schedule for s_2 contains two firings of a_1 and a_2 and one firing of a_3. Since scenario switches can only occur at the end of an iteration and a schedule contains exactly one iteration, we do not require any mechanism to switch between static-order schedules of different scenarios. Processors only need to decide once per iteration which static-order schedule needs to be executed. This schedule can then be executed till completion.

As mentioned before, step 4 of our flow must construct for each scenario in the application graph a set of static-order schedules (i.e. one for each tile used in the platform). An earliest deadline first (EDF) scheduler is used to construct for a given scenario the static-order schedules for all tiles at once. This scheduler is executed once for each scenario. The scheduler starts with modeling the binding, which was created in the previous step of the flow, into a binding-aware graph using the technique presented in Sec. VI. To compute the execution time of the TDMA time wheel synchronization actors (e.g. $a_{s,1}$ and $a_{s,2}$ in Fig. 4), it is assumed that 50% of the available time wheel will be allocated to the application graph. Since the scheduler works with a single scenario at a time, the binding-aware graph is in fact an SDFG. Using the technique from [18], this SDFG is transformed to an acyclic precedence graph. This graph is then scheduled using the EDF algorithm from [1]. Applying this scheduler to our example application and assuming the mapping of a_1, a_2, and a_3 to tile t_1 and a_4 to t_2, we find the following static-order schedules for tile t_1 when executing respectively scenario s_1 and s_2: (a_1, a_2, a_3) and $(a_1, a_1, a_2, a_2, a_3)$. On tile t_2, the schedule (a_4) is executed in scenario s_2 and no schedule is executed in s_1 since a_4 is not active in this scenario.

F. Time slice allocation

The last step of the flow allocates TDMA 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 algorithm takes the context switching overhead when an actor firing cannot be finished within the allocated time slice into account. The search between the initial bounds of 1 time slice and the entire (unoccupied) time wheel continues until the throughput of the graph constrained by the current slice allocation is at most 10% larger than the throughput constraint. When the allocation of the entire unoccupied time wheels is insufficient to meet the throughput constraint, the flow returns to step 1 and enlarges the buffer constraints.

VIII. Experimental Results

The design flow has been implemented as an extension of the publicly available SDF^3 tool set [19]. We have used this implementation to map the MPEG-4 decoder, which has been discussed earlier, and an MP3 decoder onto a multiprocessor platform with three generic processors. We limit the number of partial bindings that are considered in step 4 of the flow (see Sec. VII-D) to 10 bindings. We compare the mappings obtained with our flow to the mappings found using the design flow of [20] that is available in SDF^3. This flow requires that the resource requirements of the edges are specified in the graph. Our flow computes these requirements in step 1 and 2. We extended the flow of [20] with similar steps in order to make a fair comparison between both flows.

A. MPEG-4 SP decoder

The scenario graph model of an MPEG-4 decoder is shown in Fig. 1. A conservative SDFG model, without scenarios, has the same graph structure (with x equal to 99). As explained in Sec. IV, the execution times of an actor in this SDFG are equal to the maximal execution time across all scenarios of the corresponding actor in the scenario graph. When mapping the model to the platform, we require that the decoder can produce 20 frames per second.

Our flow finds 10 feasible mappings when mapping the scenario graph model of the MPEG-4 decoder onto the platform. These mappings are generated by SDF^3 within less than one minute when running SDF^3 on an Intel Core 2 at 2.2GHz. The generated mappings differ in the number of processors used (1 to 3), the amount of memory used (201kBytes to 304kBytes), and the amount of communication bandwidth used (0 to 38MBytes/sec). Each of these mappings is at least better in one aspect (i.e., number of processors used, amount of memory or bandwidth used) than any of the other mappings (i.e., they are Pareto optimal). This shows that our flow is able to find a set of mappings that provide a trade-off in their resource requirements. A system that must support multiple use-cases can use this set of mappings to find the most suitable mapping for a particular use-case while considering the resource requirements of other applications that are active in the same use-case. In this way, the inter-application dynamism can be exploited in the system.

The flow from [20] is able to find one feasible mapping of the MPEG-4 decoder. This mapping binds all actors to one processor. To meet the timing constraint, the application must allocate 58% of the processor's TDMA time wheel. Amongst the 10 mappings generated by our flow, there is one mapping with the same binding as found by the flow from [20]. Both mappings require the same amount of memory and communication bandwidth. However, the mapping found with our flow, which considers scenarios, requires only 17% of the processor's TDMA time wheel. Hence, it reduces the requirements on the processor's time wheel with 66% compared to the flow from [20]. This shows the advantage of using a design flow which considers the dynamism within an application.

B. MP3 decoder

In [20], a conservative SDFG model of an MP3 decoder is mapped onto a multiprocessor platform. This SDFG model abstracts from the dynamic behaviour of the application. An MP3 decoder divides an audio stream into frames of 26ms. The decoder may employ five different coding schemes depending on the audio content. This dynamic behaviour can be captured with five scenarios in a scenario graph. To test our design flow, we constructed a scenario graph of an MP3 decoder using the SDFG model from [20].

We mapped the scenario graph model onto the multiprocessor platform. Our flow finds 10 different mappings that provide a trade-off in their resource requirements. SDF^3 needs a run-time of less than 5 minutes to generate these mappings. They require between 1 and 3 processors, between 6.3kBytes and 14kBytes of memory, and between 0 and 19MBytes/sec of bandwidth. The SDFG model of the MP3 decoder was mapped to the same platform using the flow from [20]. This flow finds one mapping which uses all three processors. Our flow has found several mappings that use all three processors. When comparing these mappings to the mapping obtained with [20], we see that all mappings require the same fraction of the processor's TDMA time wheels (17% on each processor). Our mappings require however less memory (up-to 21%) and less bandwidth (up-to 23%). This shows again that our flow is able to reduce the resource requirements of an application by considering the dynamic behaviour of the application in the mapping flow.

IX. Conclusions

We have presented the first design flow that fully takes the dynamic behaviour of applications into account when mapping them onto a multiprocessor platform. The design flow considers both inter- and intra-application dynamism and provides timing guarantees for each application independent of the other applications while taking into account the available processor space, memory and communication bandwidth. The design flow produces at design-time a number of different mappings of an application. These mappings provide a trade-off in their resource requirements. At run-time, the most suitable mapping can then be selected based on the resource usage of the applications which are already running on the platform. The experimental results show that our design flow is able to reduce the resource requirements of an MPEG-4 decoder by 66% compared to a state-of-the-art design flow. The results also show that our design flow reduces the resource requirements of an MP3 decoder (up-to 21% less memory and up-to 23% less bandwidth). In future work, we want to alleviate the assumption of a unified mapping across scenarios and allow task migration between scenarios. To do this, we must extend the design flow such that it takes the cost of scenario switches into account.

References

[1] J. Blazewicz. *Modeling and Performance Evaluation of Computer Systems*. North-Holland, 1976, ch. Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines.

[2] A. Bonfietti, et al. Throughput constraint for synchronous data flow graphs. In *CPAIOR 09, Proc.* (2009), Springer-Verlag, p. 26–40.

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

[4] O. Gangwal, et al. *Dynamic and Robust Streaming In and Between Connected Consumer-Electronics Devices*, vol. 3 of *Philips Research Book Series*. Springer, 2005, ch. Building Predictable Systems on Chip: An Analysis of Guaranteed Communication in the AEthereal Network on Chip, p. 1–36.

[5] A. Ghamarian, et al. Throughput analysis of synchronous data flow graphs. In *ACSD 06, Proc.* (2006), IEEE, p. 25–36.

[6] S. Gheorghita, et al. System-scenario-based design of dynamic embedded systems. *ACM ToDAES 14*, 1 (2009), p. 1–45.

[7] S. Ha, et al. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM ToDAES 12*, 3 (2007), p. 1–25.

[8] M. Hashemi and S. Ghiasi. Throughput-driven synthesis of embedded software for pipelined execution on multicore architectures. *ACM TECS 8*, 2 (2009), p. 1–35.

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

[10] K. Kuchcinski. Constraint-driven scheduling and resource assignment. *ACM ToDAES 8*, 3 (2003), p. 355–383.

[11] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Comp. 36*, 1, p. 24–35.

[12] W. Liu, et al. Efficient SAT-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization. In *RTSS 08, Proc.* (2008), IEEE, p. 492–504.

[13] A. Moonen, et al. Timing analysis model for network based multiprocessor systems. In *ProRISC 04, Proc.* (2004), STW, p. 91–99.

[14] O. Moreira, et al. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *RTAS 05, Proc.* , IEEE, p. 332–341.

[15] O. Moreira, et al. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *EMSOFT 07, Proc.* , ACM, p. 57–66.

[16] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers 18*, 6 (2001), p. 23–33.

[17] H. Shojaei, A.H. Ghamarian, T. Basten, M.C.W. Geilen, S. Stuijk and R. Hoes. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management. In *DAC 09, Proc.* (2009), ACM, p. 917–922.

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

[19] S. Stuijk, M.C.W. Geilen and T. Basten. SDF^3: SDF For Free. In *ACSD, Proc.*, IEEE, p. 276–278. SDF^3 is available via www.es.ele.tue.nl/sdf3.

[20] S. Stuijk, T. Basten, M.C.W. Geilen and H. Corporaal. Multiprocessor resource allocation for throughput- constrained synchronous dataflow graphs. In *DAC 07, Proc.* , ACM, p. 777–782.

[21] S. Stuijk, M.C.W. Geilen and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[22] B. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *MEMOCODE 06, Proc.* (2006), IEEE, p. 185–194.

[23] Ch. Ykman-Couvreur, et al. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. In *Int. Symp. on SoC, Proc.* (2006), IEEE, p. 1–4.