Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip

P. Poplavko[1,2,*], T. Basten[1], M. Bekooij[2], J. van Meerbergen[1,2] and B. Mesman[1,2,*]

[1]Department of Electrical Engineering
Eindhoven University of Technology

P.O. Box 513, NL-5600 MB
Eindhoven, The Netherlands

[2]Embedded Systems and Architectures on Silicon
Philips Research Laboratories Eindhoven

WDC 31, Prof. Holstlaan, 4, NL-5656 AA
Eindhoven, The Netherlands

{p.poplavko,a.a.basten}@tue.nl, {marco.bekooij,jef.van.meerbergen,bart.mesman}@philips.com

ABSTRACT

We consider a dynamic application running on a multiprocessor network-on-chip as a set of independent jobs, each job possibly running on multiple processors. To provide guaranteed quality and performance, the scheduling of jobs, jobs themselves and the hardware must be amenable to *timing analysis*. For a certain class of applications and multiprocessor architectures, we propose *exact timing models* that effectively co-model both the computation and communication of a job. The models are based on interprocessor communication (IPC) graphs [4]. Our main contribution is a precise model of network-on-chip communication, including buffer models. We use a JPEG-decoder job as an example to demonstrate that our models can be used in practice to derive upper bounds on the job execution time and to reason about optimal buffer sizes.

Categories and Subject Descriptors

B.8 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Performance, Theory

Keywords

system-on-chip, network-on-chip, real-time, data flow graph, performance evaluation, buffer minimization

CASES '03, Oct.30–Nov.2, 2003, San Jose, California, USA.

*Supported in part by the IST-2000-30026 project, Ozone.

1. INTRODUCTION

In the design of embedded systems with video applications it is important to provide *guaranteed performance* to meet real-time constraints as often as possible in order to guarantee a good video quality. To achieve that goal, the applications should be characterized in terms of real-time scheduling theory.

Many embedded system applications can be seen as being object-based, i.e., they can be characterized by a dynamic set of objects that can dynamically appear or disappear depending on some rules and user actions. An example is a multi-window TV, where the objects are windows. We consider the set of objects as a set of real-time *jobs*. It is assumed that the jobs are started and stopped by a run-time resource manager. In video applications, each job will typically possess a lot of internal parallelism, so we allow different subtasks of one job to execute concurrently on multiple processors.

To provide predictable performance, the jobs, the scheduling and the hardware must be amenable to *timing analysis*. It would be advantageous to have formal definitions of different scheduling problems as *optimization problems*. For both timing analysis and optimization, a mathematical timing model is required. We aim in this paper at accurate *timing models* that effectively co-model the computation and communication of one job. In order to do that, some properties of the given application domain can be taken into account.

In the video application domain, the core part of algorithms often consists of *loops* repeating the same set of operations for certain number of *iterations*. In this paper, we assume that job is just a loop. To model the loop body and its parallelism, *synchronous data-flow graph* (SDF) model is commonly used in multiprocessor scheduling [13, 8]. In that model, the subtasks executed within the loop are encapsulated within multiple data flow *actors*, which are nodes of an SDF graph.

In our modeling approach we use SDF models called *IPC graphs* (IPC stands for interprocessor communication), studied in [4] and [17]. IPC graphs model both the computation and communication, not assuming global synchronization. Although execution times of the subtask actors may be variable, the *worst case timing* of multiple IPC graph iterations is theoretically proven to be

periodic, which makes *analytical approach* to the performance evaluation possible.

Our work can be positioned in the following 'design flow'. The starting point is a job specification in the form of an SDF graph that describes the computations of the job (*computation graph*). In Step 1, the graph is fed to a multiprocessor scheduling 'tool suite'. The tools partition the graph into *processes*, or the parts that will run on one processor. They also create *channels* for communication between processes. The tools produce at the output a structure we call a *configuration network*, consisting of processes and channels. We say that Step 1 actually determines the *intra-job scheduling*, because it manages the parallelism inside a single job.

In Step 2, the configuration network is converted into an IPC graph, and we use it to reason about the job throughput, execution time and buffer sizing. We focus particularly on Step 2, leaving Step 1 for future work. Our main contribution is the modeling of on-chip communication channels in IPC graphs, including *buffer models*. To make accurate modeling possible, we make a few assumptions.

In intra-job scheduling, we adopt currently the *static-order self-timed* scheduling [17]. In such a scheduling, all the data-flow actors (subtasks) assigned to a given processor are executed in a static cyclic order as soon as the input data is available.

In this paper, we also assume that jobs run *independently*, i.e., they do not interact with each other and the external world, or, if they do, that interaction does not influence the timing properties at the task level. The job independence assumption also necessitates that jobs do not experience resource conflicts with each other. This would be possible in a real-time environment where processing time budgets and communication services with *bandwidth guarantees* are given to jobs. Such services can be offered at a reasonable cost by on-chip router networks through independent point-to-point network connections [7, 15]. Packet-switched networks are an attractive way to implement the interconnect between the on-chip processors in case their number is large enough. Such a system-on-chip is referred to as a multiprocessor network-on-chip (MP-NoC).

We start the paper by introducing the required basics of the synchronous data-flow graphs in Section 2. In Section 3, an MP-NoC architecture template is introduced that ensures predictable communication delays and independent communication channels. Section 4 is the core of the paper, showing how jobs, partitioned into a network of processes that communicate via the network, can be modeled using IPC graphs. Section 5 demonstrates practical use of our models for the derivation of the worst-case performance and optimal buffer sizing, using a JPEG decoder job as an example. Finally, Section 6 concludes this paper.

2. Background

2.1 HSDF Graphs

In the literature on multiprocessor scheduling for DSP applications, like [10] and [14], the whole application can be seen as a single job [19]. The job is represented by a dataflow graph model, where the nodes, called *actors*, correspond to the processor code segments that should be executed on a single processor (*computation actors*), and edges show data dependencies between them (*data edges*) or execution order

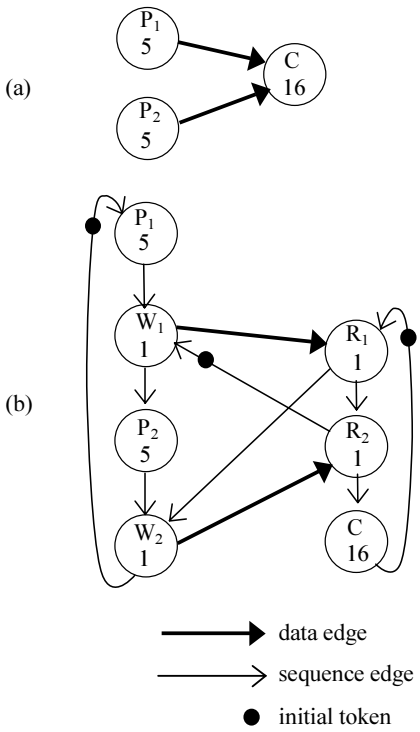

Figure 1: HSDF graphs of a producer-consumer example. (a) Computation graph. (b) IPC graph

constraints (*sequence edges*). Every edge can carry an infinite number of *tokens* between two actors, and can contain *initial tokens* (present on the edges at start time).

In this paper, we use synchronous dataflow (SDF) graphs [13] to model jobs. These models have been widely used to model multi-rate DSP applications [8]. More specifically, we use only homogeneous SDF (HSDF) models. Each actor in that model follows a simple *firing rule*: whenever there is at least one token on every input, a *firing* takes place, meaning that the actor consumes one token on every input and produces one token on every output. The rules of execution of the graph is usually restricted in such a way that actors consume and produce tokens in a first-in-first-out (FIFO) manner, as we will see later in this section. In the more general SDF model, actors are allowed to consume and produce more than one token per input and output edge per firing, but any SDF graph that satisfies some general properties can be converted into an equivalent HSDF graph [9].

Following a common practice, we extend HSDF models with *actor execution time* annotations (real numbers), representing the time span between the consumption of input tokens and the production of output tokens. These annotations can be either fixed or variable. In HSDF graphs, a key notion is the notion of an *iteration*, which is defined as a set of actor firings such that all actors in the graph fire exactly once.

HSDF graphs are reused in our work to model both computation only and computation together with communication. Let us briefly consider the former, before we proceed with the latter in the next subsection.

Computation graphs are HSDF graphs that express computations and data dependencies contained in the algorithm performed by a job. For example, Figure 1(a) shows a simple computation graph with three actors, namely, two 'producers' (P_1 and P_2) and one 'consumer' (C). In each iteration, first, each producer must produce a token, and, then, the consumer may fire and consume the tokens. In this example, we annotate the actors with some arbitrary computation time values. Computation graphs serve as an input for static-order intra-job scheduling, which, as already mentioned, is outside the scope of this paper. The output of the static-order scheduling can be transformed into an interprocessor communication (IPC) graph.

2.2 IPC Graphs

An IPC graph is an HSDF graph that models the execution of the job on a multiprocessor architecture [4, 17 - §7]. It can be seen as a transformation of the computation graph, where extra actors and edges have been added. So far, IPC graphs have been applied to multiprocessors with communication based on a bus and a global memory. In Section 4, we construct IPC graphs assuming network-on-chip communication.

In addition to the computation actors of the computation graph, IPC graphs for the aforementioned bus-based multiprocessors contain *data copy actors*, or '*write*' and '*read*' actors, that copy blocks of data (data tokens) from the local memory of a processor to the global memory and vice versa. In addition to data edges, *sequence edges* are introduced. Sequence edges do not imply any data communication between actors; they only enforce some sequencing of actor firings. Although they have a different purpose, they are not distinguished from data edges when analyzing the timing properties of the graph. For each processor and for the bus, sequence edges are used to create a distinct *cyclic path*, where the actors are put in a specific static order, and an initial token is placed on the sequence edge at the input of the first actor in the order. The static ordering eliminates resource conflicts between the actors *assigned* to the given resource. The ordering of actors on a processor or a bus can be decided by a static-order scheduling algorithm [14, 17].

For example, Figure 1(b) shows an IPC HSDF graph for the example of Figure 1(a) assuming a two-processor case, where the producers are assigned to one processor and the consumer is assigned to the other one. It includes write actors (W_1 and W_2), read actors (R_1 and R_2) and two data edges, derived from the computation graph, to pass data tokens from the producers to the consumer. The graph contains two processor cycles, namely, (P_1, W_1, P_2, W_2)* and (R_1, R_2, C)*, and one bus cycle (W_1, R_1, W_2, R_2)*. Note that in the latter case only two sequence edges are introduced to enforce that cyclic order, namely, (R_1,W_2) and (R_2,W_1). The other two sequence edges are not necessary due to the presence of the data edges.

2.3 IPC Graph Properties

2.3.1 IPC Graphs in General

Some major difficulties in the timing analysis of HSDF graphs appear when, for efficiency reasons, one allows multiple iterations of an HSDF graph to overlap and when the graph contains cycles. Both are true for IPC graphs. Fortunately, there exist theoretical studies on the timing properties of such models. It is, actually, the main reason why we apply IPC graphs in our work. In this section, we mention, without proofs, three useful properties of such HSDF graphs, namely, monotonicity, periodicity and boundedness. These properties are mainly based on known properties of the models called *event graphs* [3]. As it follows almost directly from [3 - §2.5.2.6], any HSDF graph can be easily translated into an equivalent event graph. We apply these properties in Section 5, where we demonstrate the usefulness of IPC graphs for practical problems.

Let us define *self-timed execution* of an HSDF graph as a sequence of firings of each HSDF-graph actor, where every firing happens immediately when there are tokens on the inputs. As mentioned in the introduction, this is exactly the kind of execution assumed in this paper for intra-job scheduling.

There are certain restrictions that any IPC graph should satisfy for the aforementioned properties to hold. An IPC graph has to be:

1) a *strongly-connected* HSDF graph, i.e., for any two actors there exists a directed cycle (not necessarily a processor or a bus cycle) that contains both actors;
2) a *first-in-first-out* (*FIFO*) HSDF graph, i.e., in the self-timed execution, the completion events of actor firings occur in the same order as the corresponding starting events. In other words, tokens cannot overtake each other in an actor ([3 - §2.5.2]).

Note that an HSDF graph can be non-FIFO only if actors have variable execution times. Tokens can overtake each other because multiple actor firings can overlap in time. A sufficient condition that excludes overlapped firings is the following: for any actor, there is a cycle in the graph that contains that actor and has only one initial token. Obviously, this property holds for the IPC graphs discussed in the previous subsection, because each actor there belongs to a processor cycle and every processor cycle has only one initial token.

Lemma 1 (Monotonicity) The self-timed execution of any FIFO HSDF graph is *monotonic* on the actor execution times in the sense that increasing actor execution times result in non-decreasing starting times of firings.

This lemma is applicable for IPC graphs with *variable* execution times of the actors. If the execution time of any actor increases, the starting times of any actor depending on it can only be postponed to the future.

This lemma can be proven by constructing evolution equations as described in [3 - §2.5.2] for the corresponding event graph and observing that these equations are monotonic on the firing times.

The monotonicity property is needed in practice to prove that, in order to derive the worst-case execution time of a dynamic job, one can use the worst-case execution times of the actors as the fixed execution time annotations.

2.3.2 IPC Graphs with Fixed Execution Times

When we use fixed (worst-case) actor execution times, we can obtain the worst-case throughput of the job using the periodicity property and bound the job's execution time by a simple analytical expression using the boundedness property. Both properties are introduced in this subsection.

Now suppose an IPC graph with fixed execution times is given. Let us consider a *simple cycle* in the graph, i.e., a cyclic path that cannot contain any actor more than once. The cycle may contain both data edges and sequence edges. In the remainder, we refer to

simple cycles just as cycles. Let us define the *cycle weight* as the sum of execution times of the actors in the cycle. Because the execution times are fixed, the cycle weight is a constant value.

We define the *cycle mean* as the cycle weight divided by the total number of initial tokens on the edges of the cycle. A cycle with the maximum value of a cycle mean among all cycles in the graph is called a *critical cycle*.

The *maximum cycle mean* (MCM) of the graph can be calculated in polynomial time [6]. Intuitively, it signifies the average time distance any initial token in the critical cycle has to travel, until it comes to the next starting position of another initial token (or itself). If the same would happen with every initial token in every cycle of the graph, the graph would come into the same state as where it started (a period is completed). The tokens of the critical cycle are the "slowest" ones in this sense, thus, they constrain the speed of the whole graph.

The following theorem, known in the domain of event graphs ([3 - §3.7]) and also applied for HSDF graphs, e.g., in [17 - §7], states that after some limited number of iterations (N) an IPC graph will indeed return into a state in which it has been before:

Theorem 2 (Periodicity) For the self-timed execution of a strongly-connected HSDF graph $G(V,E)$ with fixed actor execution times, there exist integers K and N, such that for all $v \in V$, $k \geq K$,

$$s(v, k+N) = s(v,k) + p \cdot N \qquad (2.1)$$

where the 'iteration interval' p is equal to the MCM; k is the *iteration index*; $s(v, k)$ is the firing start time of actor v in a given iteration of the self-timed execution; K is the number of iterations before the execution enters a periodic regime; N is the number of iterations in one period.

Note that, by convention, we assign index 0 the first firings of actors, in other words, we assume that an actor v starts initially at time $s(v, 0)$.

Theorem 2 implies that the execution of graph G eventually enters a periodic regime, where every actor is guaranteed to start exactly N firings within any half-closed time interval of length $p \cdot N$. This actually means that the MCM value p is the *average iteration interval* of graph G. The actual iteration intervals may vary, despite the fact that the actor execution times are fixed. Nevertheless, these variations will periodically repeat every N iterations.

To characterize a periodic execution, one only needs to find actor start times $s(v,k)$ during one period (N iterations). Suppose, that this information is available. Then, we can derive a characteristic of the graph that we call *lateness,* denoted σ :

$$\sigma = \max_{v \in V} \left(\max_{k=n..n+N-1} \left(s(v,k) - p \cdot k + t(v) \right) \right) \qquad (2.2)$$

where $t(v)$ is the fixed execution time of actor v and n is an arbitrary integer such that $n \geq K$ (since it doesn't matter which particular period we take); to apply this formula in practice, $n = K$ can be chosen (the starting point of the periodic regime).

The lateness shows how late the iteration with index k can possibly finish with respect to the reference point of time $p \cdot k$.

Note that when I iterations of the graph complete, the index k of the last iteration is equal to $I - 1$. The following corollary follows immediately from Theorem 2 and the definition of lateness:

Corollary 3 (Boundedness) With the preconditions of Theorem 2, and for $I > K$, an upper bound on the completion time of I iterations of HSDF graph $G(V,E)$ is given by:

$$\text{HSDF-BOUND}(I) = p \cdot (I - 1) + \sigma \qquad (2.3)$$

Note that, whereas we can obtain the value of p by computing the MCM of the graph G, to compute σ, we need a sample of starting times of all actors during one period, which implies that we also need to know K and N. In Section 5, where Corollary 3 is actually applied, we describe a method to compute these values in practice. We also explain how to compute HSDF-BOUND in case $I \leq K$.

3. Architectural Issues

3.1 Background

As already mentioned, the IPC-graphs we propose in this paper co-model computation and communication. In Section 2.2, we used related work involving *bus-based* communication to introduce IPC graphs. However, we assume architectures with *network-based communication*, as described in this section.

[5 - §1] describes a general template for multiprocessors. It consists of multiple *processing tiles* connected with each other by an interconnection network. Each tile contains a few processor cores and local memories. The tile may contain communication *buffers,* accessed both by the local processors and the network. The tile has a small controller, called *communication assist*, that performs buffer accesses on behalf of the network. Many embedded MP-SoCs implemented on silicon, e.g., Daytona [1], AxPe [16] and Prophid [18], fit nicely into this general template.

Among these architectures, Prophid is the most interesting one for us, because it uses a simple packet-switched network for communication and provides performance guarantees for hard real-time tasks. Prophid contains application-domain-specific processors communicating through a switch, based on a time-division multiple access (TDMA) scheme, enabling guaranteed-bandwidth communication. Different tasks running on the multiprocessor communicate with each other using *asynchronous message passing*, meaning that processors synchronize based on the availability of data in buffers. Message passing introduces the buffer overflow issue, which is solved in Prophid using a kind of end-to-end *flow control*. We reuse some ideas of the Prophid architecture in our architecture template.

3.2 Architecture Template

The MP-NoC architecture template adopted in our work is shown in Figure 2. We focus on defining the issues important for the timing models introduced in Section 4.

The interconnection network in our template is a network-on-chip (NoC). Each processing tile is 'plugged' into the network through an *input link* and an *output link*. The NoC offers unidirectional point-to-point connections. The connections must provide guaranteed bandwidth, and a tightly bounded propagation delay per connection. The connections must also preserve the ordering of the communicated data. Other details of the NoC are hidden. An example of a NoC providing these properties is ÆTHEREAL

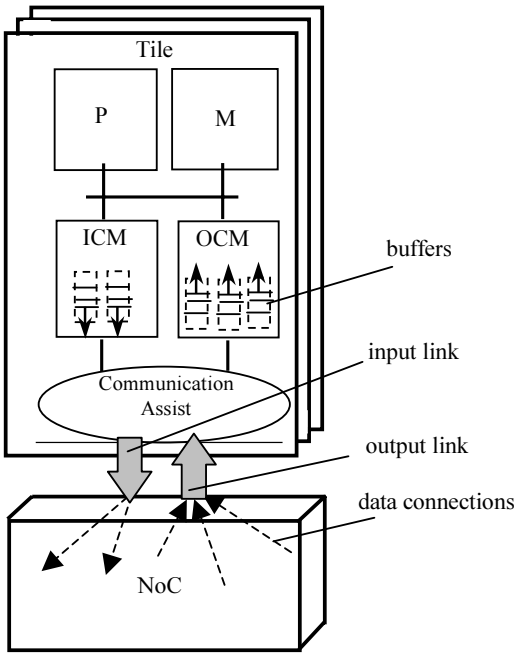

Figure 2 Architecture template

[15]. It uses the TDMA scheme, for which reasonably cheap implementations of the network routers are possible. Note that several other schemes ensuring guaranteed performance in the multi-hop networks exist in the literature, but we do not know whether any of them has been implemented so far in the context of NoCs. The choice of the particular scheme does not influence the main idea and the structure of the timing models we present in this paper.

To keep the processing tile simple, we assume only one processing core (denoted 'P' in Figure 2) per tile. The local memory layout contains three blocks: the general-purpose memory ('M') for processor instructions and data, the input communication memory (ICM) and the output communication memory (OCM). ICM and OCM have ports for the processor and for the communication assist.

Connections as introduced above are the key components of *channels*, which are the basic communication primitives for jobs. Channels are managed by the communication assists. Each channel connects two different processing tiles and transfers data in one direction. A channel contains an *input buffer*, a *data connection*, a *flow-control connection* and an *output buffer*. The buffers are located in the communication memories (ICM and OCM) at different sides of the channel. Every channel pumps data from its input buffer to its output buffer through the data connection. Every communication assist can run multiple channels concurrently. The example tile in Figure 2 has three incoming and two outgoing channels.

For our timing models to be valid, we require that the following memory accesses be independent:

1) the processor and the communication assist accesses;
2) the communication assist accesses to ICM and OCM;
3) the accesses initiated by different channels through the same ICM/OCM port.

In this context, with 'independence' we mean that the variation of the access time due to contention (if any) is sufficiently small. Requirements 1) and 2) are achieved by using two dual-ported memories for ICM and OCM. However, we do not actually demand that the memory layout be exactly the same as in the figure, as long as the independence requirements are satisfied. Requirement 3) can be satisfied if an appropriate arbitration scheme is used for the ICM/OCM ports. We believe that this can be achieved at a reasonable cost, and it is a subject of future work.

The communication of a data token through the channel takes place as follows. First, the processor at the input side puts a token into the input buffer, where it waits for the tokens in front of it (in that buffer) to depart. Then, the local communication assist transfers the token into the input link. We call the time interval between the start of the transfer and the departure of the token a *transfer delay* (t). The exact value of t depends on the NoC implementation. In an ideal network $t = s_{token}/B$, where B is the bandwidth of the connection and s_{token} is the size of the token. In a real network, it also includes the medium access delay, which, in case of TDMA, would depend on the TDMA frame size. The time elapsed from the departure to the complete arrival of the token at the output buffer is the *propagation delay* of the data connection (δ_D). Summed with t it gives the total channel delay.

An end-to-end *flow control mechanism* helps avoiding output-buffer overflow. The communication assist at the input side keeps a (pessimistic) counter of the number of free places (*credits*) in the buffer at the output side. The channel is blocked when that counter reaches zero. The counter is decremented whenever a new token departs into the network. Every time the processor at the receiving side frees one or more places in the output buffer, it triggers a credit packet to be sent back to the sender through the flow-control connection of the channel, and the credit counter is incremented accordingly. We assume that a triggered credit packet takes time δ_C to reach the input side of the channel.

4. Timing Models

To co-model the computation and communication of a job, it has to be mapped to a subset of the processors of the target multiprocessor. In order to do that, starting from the job computation graph, a set of intra-job scheduling decisions should be taken. The computation actors should be assigned to processors and a static order of actors should be set on every processor. The data edges should be assigned to channels, and the buffer size and bandwidth values should be set for the channels. The details are outside of the scope of this paper. Some appropriate methods are presented, e.g., in [14], [19], and [8]. We represent the final result of those design decisions in the form of a job *configuration network* which can be seen as a refined form of the YAPI *process networks* of [12]. The latter has been proposed as a functional model of DSP multiprocessor applications. On top of YAPI process networks, we have introduced timing and enforced some restrictions on the internal structure of the processes and channels. In this section, we first present the notion of a job configuration network, and then explain in detail how it is *translated* into an IPC graph.

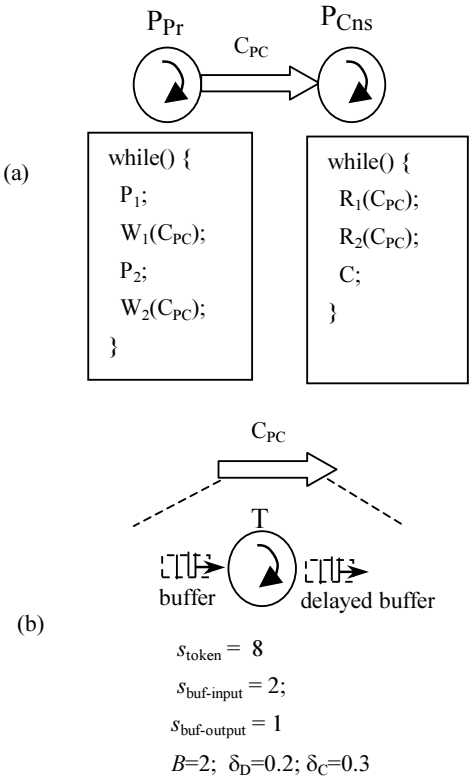

Figure 3 Output of intra-job scheduling (a) Configuration network (b) Channel

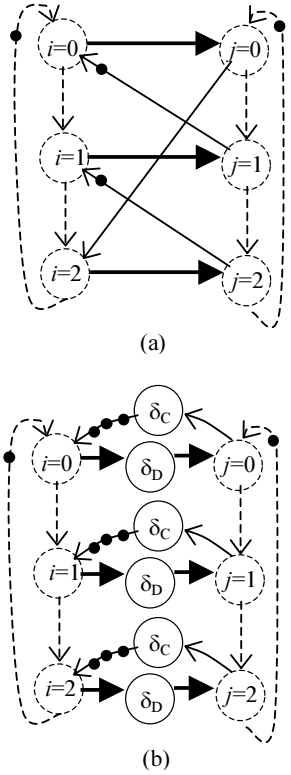

Figure 4 Buffer model examples

(a) buffer, size = 2 (b) delayed buffer, size = 9

4.1 Configuration Networks

The structure of a configuration network can be described by a directed graph. The nodes of the graph represent *processes*. The edges of the graph represent *channels*, in line with Section 3. We assume that every process has a sequence of computation and data-copy actors *assigned* to it. Each process runs on its own processing tile and executes its own sequence of actors in a static order within a software loop.

Figure 3(a) shows a configuration network for the "producer-consumer" job example of Section 2. It consists of two processes, P_{Pr} and P_{Cns}, and one channel, C_{PC}. Also, the list of actors is shown for each process, coming in the same static order as in the processor cycles in the example of Figure 1(b). These actors copy the data tokens from the local general-purpose memory to a communication memory and vice versa.

We assume the channel implementation as presented in the previous section. All the channels in a configuration network have the same structure, but, possibly, different parameter values. Figure 3(b) zooms into the structure and parameters of channel C_{PC}. We introduce a *transfer subproces* (denoted 'T' in Figure 3) to model the transfer of tokens from the input buffer to the output buffer. The output is a *delayed* buffer, because of the network propagation delay. Every buffer has an integer parameter s_{buf}, or the *buffer size*, specifying the maximum number of tokens fitting in that buffer. Figure 3(b) assumes some arbitrary values for all the parameters of channel C_{PC}.

4.2 HSDF Models of Components

For the translation of a configuration network into an IPC graph, first, for every process, buffer, transfer subprocess and delayed buffer we introduce an HSDF model. In a second step, the models are assembled together to form one IPC graph.

The processes are modeled by processor cycles, described in Section 2.2. The transfer subprocess is modeled in a similar way. Let n be the number of 'write' or 'read' actors that access the given channel (they should be equal). The transfer cycle that models the transfer subprocess consists of n identical actors, each one annotated with the transfer delay t (see Section 3.2). Note that different channels, even those running on the same communication assist, have different independent transfer cycles; the correctness of this assumption rests on the independence of the accesses of different channels to OCM/ICM ports.

The models for the (delayed) buffers join different (sub)processes together. Figure 4(a) shows an example of a buffer model. The model itself includes only the primitives that are shown with solid lines. The dashed actors come from the processes or subprocesses that access the buffer. The actors drawn on the left, or *'put' actors*, put the tokens into the buffer. The actors on the right, or *'get' actors*, retrieve the tokens from the buffer. The dashed arcs in Figure 4(a) are also not part of the model; they show a

requirement on the ordering of the actors, that the process and subprocess models attached to this buffer should satisfy. A dashed arc requires that a path exist between the source and the sink of the arc, with the same total number of initial tokens as on the arc. The buffer model includes data edges (drawn from left to right), which model data dependencies, and sequence edges, or *backward* edges, which model the blocking of put actors when the buffer is full. The data edges join the 'put' and 'get' actors that get the same number in the ordering of buffer accesses. The positions of the backward edges and the number of initial tokens on each edge depend on the buffer size s_{buf}. Figure 4(a) assumes $s_{buf} = 2$. The reader can verify that in any sequence of actor firings there cannot be more than 2 tokens on the data edges in that model.

We formulate here, without a proof, the general rule to position the backward edges. Let us denote the number of actors on one side with n. Let us index the 'put' actors with variable i, and the 'get' actors with variable j, both variables getting the values $0..n$-1 in correspondence with the static order. The set of backward sequence edges is defined as follows:

- $E_{BACK} = \{(j, i) : 0 \le j < n \ \wedge \ \ i = (j + s_{buf}) \bmod n\}$;
- every backward edge (j, i) contains: $\lfloor (j + s_{buf})/n \rfloor$ initial tokens.

In case of output buffers (the delayed buffers), we first construct the same model as for the buffer, and then split each data edge and backward edge into two, introducing an actor in between with a (propagation) delay annotation δ_D on the data edges and δ_C on the backward edges (see Figure 4(b)). The backward edges on the left side get all the initial tokens, because the number of tokens on these edges models the value in the credit counter, which is initialized with the size of the output buffer.

4.3 IPC Graph

Figure 5 shows the result of the expansion of the configuration network into an IPC graph for the example of Figure 3(a). In the middle, we see a subgraph that models channel C_{PC}. Inside this subgraph we see a transfer cycle (T_1T_2)*, both actors are annotated with a delay $s_{token}/B = 8/2 = 4$. The transfer actors serve as 'get' actors for the buffer of size 2 on the left and, simultaneously, as 'put' actors for the delayed buffer of size 1. The processor cycles in that graph are identical to the processor cycles in Figure 1(b).

At this point, we have a model that is amenable to timing analysis. To determine, for example, the average iteration interval p (see Theorem 2), we find the critical cycle in the graph. Cycle $(R_1\,\delta_{C1}\,T_2\,\delta_{D2}\,R_2\,C)$* is critical, yielding an iteration interval of 22.5 time units. Note that by introducing two more places in the output buffer (thus having size 3), we keep the same structure of the graph, but the edge (R_1,T_2) would include one extra initial token. Then, the number of initial tokens in the aforementioned cycle would become 2, thus, making this cycle no longer critical. The new critical cycle would be $(R_1\,R_2\,C)$*, resulting in an iteration interval of 18 time units. Thus, adapting a buffer size can improve the performance. We will see a practical example of buffer sizing in the next section.

The techniques explained and illustrated in this section provide a general means to transform any configuration network of a job to an IPC graph amenable for timing analysis. In the next section, we apply these techniques to a case study. An observation can be made about our IPC graphs that including δ_D and δ_C actors in the model could potentially violate the requirement that an IPC graph be a FIFO graph (see Section 2.3). However, we use fixed (worst-case) propagation delay values for them, so that the tokens cannot overtake each other in these actors. Instead, one can also use a sequence of variable propagation delay times, taken from the measurements, as long as the condition is satisfied that connections preserve the ordering of the data that is communicated through them.

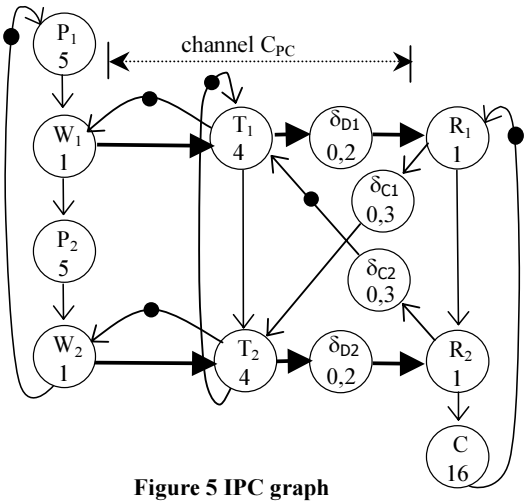

Figure 5 IPC graph

5. Case Study for Timing Models

5.1 Prerequisites

In this section, we consider a timing model for a decoder of an image in the JPEG File Interchange Format (JFIF), here referred to as the *JPEG job*. We use some realistic estimations of the communication and computation costs based on existing processor (ARM [2]) and NoC (ÆTHEREAL [15]) designs. The purpose of this case study is to show some methods for using our IPC graph models for 'real-life' problems. In Section 5.2, a timing analysis problem is solved. In Section 5.3, we demonstrate the relevance of the buffer-size minimization problem and show that it can be formally defined using our IPC graphs.

The computation graph of the JPEG decoder job as defined in this paper is shown in Figure 6. It is a task-level description of the body of the main loop of the program for the case when the input file is a 4:1:1 coded image. One iteration of the graph decodes one minimum coded unit (MCU, 16x16 pixels). All the *data edges* assume the same data-token type, an 8×8 pixel *block*. For clarity, the actors in Figure 6 are organized in columns and rows. The columns correspond to processing stages and the rows correspond to the six blocks of an MCU. The blocks undergo three processing stages: variable length decoding (VLD), inverse discrete cosine transform (IDCT), and scaling (SCALE), before they are fed into a color conversion stage (COLOR).

To measure the execution times of the actors, we have run the JPEG software from [11] on the ARM7TDMI core instruction set simulator (ISS) from ARM Ltd [2]. Flat memory model with single-cycle access and no caches has been assumed (which will be close to reality in case all data and instructions fit in the local

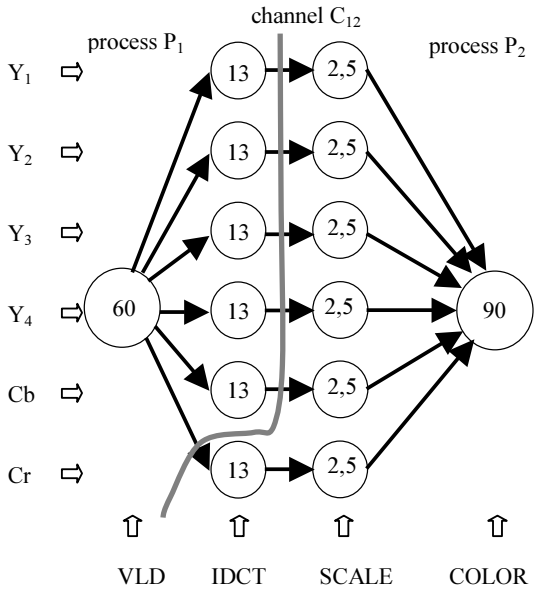

Figure 6 Computation graph of the JPEG decoder with a partitioning into two processes

memory). In future work we want to deal with remote memory accesses explicitly in our model.

One image, our '*reference image*', has been fed as an input to the program. It takes 20 iterations of the graph to decode the image. We have provided in the figure the average execution time values over all iterations (in kilocycles).

We have created for the JPEG job a configuration network, containing two processes, P_1 and P_2, and one channel, C_{12}. The computation graph has been partitioned between the processes, as shown in Figure 6. We chose to specify the static order of the actors per process by parsing the graph from left to right and from top to bottom (wrt to the columns and rows). For each of the six data edges crossing the splitting border, we have introduced a write actor in the actor list of P_1 and a read actor in the actor list of P_2. All these actors access channel C_{12}.

We have assumed the ARM processor running at 133MHz [2] and the ÆTHEREAL NoC [15] with a link width of 16 bits and the clock speed set at 400MHz. We have allocated 1/256th of the link bandwidth to the channel C_{12}. For the given bandwidth and token size, we have assessed the execution time of write and read actors (roughly, 1 kilocycle on the ARM) and channel transfer actors (being in worst case 5.6 kilocycles and having a 5% variation). The data propagation delay and the credit transmission delay are both negligible, we assumed $\delta_D = 0.02$ kilocycles and $\delta_C = 0.3$ kilocycles.

Using the aforementioned assumptions and parameter values, we can build an IPC graph from the configuration network. The picture of the graph is omitted because it is too large, but it has a structure similar to Figure 5. It contains 12 and 14 actors in processor cycles P_1 and P_2, respectively, and 6 actors in the transfer cycle. By default, we use the minimum possible buffer size s_{buf}=1 for both input and output buffer, but we also experiment with other buffer sizes in Section 5.3.

For the experiments in this section, we have developed a simulation tool that computes the sequence of start times of all actors: $s(v,k)$. This tool simulates the self-timed execution of IPC graphs. It can be used:

1) to obtain the 'real' execution time of I iterations of the job, by feeding the traces of possibly variable execution times of the computation actors into the IPC graph (actor execution times are measured on ARM simulator).
2) to obtain a sample of start times of actor firings within one period of execution, assuming fixed execution times in each iteration. This sample can be used to compute the lateness σ (see Formula 2.2). The simulation is run until K and N are detected and then a sample of N iterations is extracted.
3) to obtain the HSDF-BOUND(I) in case $I < K$ by simulating I iterations.

5.2 Upper Bound on Execution Time

In job-level scheduling, a job may be required to produce I data items (e.g., blocks, lines, frames) within a certain time interval. To know apriori whether the job can meet its deadlines, it may be necessary to calculate an upper bound on the time the job needs to execute I iterations, or the *job execution time*.

If we model a 'real-life' execution of the job, due to data dependency of the execution time and network jitter, each actor may take a different time to execute its firing in every iteration. Nevertheless, due to the monotonicity of IPC graph execution (see Lemma 1), one can be sure that if one feeds upper bounds on the execution time for each actor into the IPC graph, the 'real' actor firings will always start earlier than the firings according to the IPC graph. Corollary 3 gives an upper bound on the completion time of the IPC graph execution. Consequently, Corollary 3 can provide a reliable upper bound also for a real job execution time.

We apply Corollary 3 to the JPEG job for the decoding of the reference image ($I = 20$), and see how tight the obtained bound is by comparing it to the real job execution time.

Depending on the result, we draw a conclusion on how useful our timing models are in this particular case. In fact, execution time can often be formulated in terms of some application-specific parameters. This is also true for JPEG. Thus we do two experiments. In the first one, we use knowledge about the worst-case values of some application-specific parameters of the reference image to obtain a *parametrical bound*. In the second experiment, we obtain a real *worst-case bound* valid for any input image.

For most computation actors of the JPEG job, the variations of the execution time are very small, so we neglect them in this case study. The same holds for all the communication actors (read, write, transfer and δ actors). An exception among the computation actors is the variable-length decoder (VLD). To reason about the execution time of VLD, we have analyzed its source code and observed that its upper bound execution time is a *linear function* in three parameters, specific for the VLD algorithm, with positive architecture-dependent coefficients. We have calculated these coefficients for the ARM processor from a set of measurements

made with the ARM simulator. This technique can be strongly related to known techniques of worst-case execution time estimation for embedded software, but it yields a parametrical expression rather than a worst-case constant value.

To derive the *parametrical* upper bound, we used the set of the maximum values of every parameter in all 20 executions of the VLD actor for the reference image. The linear function applied to this set of parameters yields an upper bound for the actor execution time. We fed that bound into the IPC graph and computed the lateness σ of the graph using our simulation tool: $\sigma = 282$ kilocycles. We have also computed MCM of the graph: $p = 188$ kilocycles. Applying Corollary 3, we get roughly 3854 kilocycles for the parametrical bound.

To derive the *worst-case* bound, we have found the maximum value each parameter can possibly have in any JPEG image. Again, first we obtain the actor execution time bound, by calculating the aforementioned linear function on this set of parameters. Applying Corollary 3 yields 19560 kilocycles for this bound.

The (simulated) real execution time of the JPEG job on the reference image is 3203 kilocycles. We see that, for the given input image, the parametrical upper bound yields only 20% overestimation, which is small compared to the overestimation from the worst-case bound, being equal to 511%. The main conclusion of this subsection is that we have shown a practical application of our method and timing models to derive an upper bound on the job execution time on an MP-NoC. In case of potentially large variations in actor execution times, the timing models can considerably profit from some information on the input data parameters. For video applications, such information can be provided relatively easy, e.g., in video frame headers.

5.3 Buffer Minimization

In Section 4.3, we have seen that the backward edges of the buffer models may be involved in the critical cycles of an IPC graph. Such a critical cycle can always be removed by increasing the size of the buffer from which a backward edge involved in the critical cycle originates. We define a *buffer sizing* as a vector of the buffer sizes of all buffers. A buffer sizing is called *rate-optimal*, if there is a critical cycle in the corresponding IPC graph that does not include any backward edge of any buffer. The name 'rate-optimal' refers to the fact that increasing the size of any buffer cannot reduce the iteration interval p of the IPC graph anymore. The *buffer minimization problem* can be defined as finding a rate-optimal buffer sizing that minimizes the sum of the buffer sizes of all the buffers. Thus, the models proposed in this paper allow us to define the buffer minimization problem in the context of on-chip networks.

Based on this problem formulation, we have found a feasible solution for it for the JPEG case study. In this exercise, we have followed the following method:

1) derive an IPC graph annotated with average execution times.
2) determine a rate-optimal buffer sizing by increasing buffers and changing IPC graph correspondingly. This is repeated to the point when iteration interval p does no longer change, where p is computed analytically as the MCM of the IPC graph.
3) to test the quality of the result, we measure real execution time for various buffer sizing options

Table 1 illustrates the results of this exercise.

Table 1. Arriving at a rate-optimal buffer sizing

$s_{\text{BUF-IN}}$, $s_{\text{BUF-OUT}}$	p	Real Exec. Time
1,1	152	3203
1,2	138	3066
2,2	131	2978
2,3	no change	2914
3,3		2870
3,4		no change

From this table, we see that the result of step 2, sizing (2,2), is quite good although not really optimal (form the execution time point of view), because that would be sizing (3,3). This can be explained from the fact that step 2 assumes that the execution time of the VLD actor is always the same (average), whereas in reality it changes at every iteration.

Buffer minimization has been considered in [8] and other research work as a linear programming problem, where the buffers are modeled with a set of linear constraints. However, that work does not involve an assignment of the actors to processors, nor has it any direct link with networks. It is interesting to investigate the relation between our approach with backward edges and the linear constraints.

6. Conclusions

In this paper, we have sketched an approach to implementation of the video applications targeting on-chip multiprocessors and providing guaranteed performance. In this approach, the main criterion is to support accurate reasoning about the timing. We have focused on the models enabling us to reason mathematically about the system requirements and guarantees. We assume that the basic components of the application are jobs that can be dynamically activated depending on some rules and user actions. Each job may run on multiple processors. We have restricted ourselves to jobs that can be modeled by executing a number iterations of a synchronous data flow graph (SDF) [13]. We can reason about each job in isolation, using guaranteed bandwidth services offered by the on-chip network for the communication and processing time budgets for the computation. We consider a multiprocessor mapping of one separate job and show that we can reason about the iteration interval, execution time and optimal buffer sizes of the job by constructing an interprocessor communication (IPC) graph and applying a timing analysis on it. The main contribution of this work are the new models for *network-based* communication, that we have introduced in the form of IPC graphs; in the past, IPC graphs have been used only for multiprocessors with *bus-based* communication. In a JPEG decoder case study, we have demonstrated some opportunities to use the proposed models in practice for timing analysis and buffer size minimization. The first results are encouraging, but more experiments are needed. In our future work, we plan more advanced video applications, like the natural video part of

MPEG-4. Also, an example of communication assist hardware design providing an interface between an embedded processor core and the ÆTHEREAL communication network [15] will be worked out in detail. Another important issue for our future work is handling the accesses to large data structures in remote memories. We also plan to extend our work to communicating jobs, to further develop tool support, and to apply our approach in a run-time resource manager for MP-NoCs.

7. References

[1] Ackland, B., et al., A Single-Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP. In IEEE Journal of Solid-State Circuits, vol. 35, no. 3, 412-424, March 2000.

[2] http://www.arm.com

[3] Baccelli, F., Cohen, G., Olsder, G.J., and Quadrat, J.-P., Synchronization and Linearity. New York: Wiley, 1992.

[4] Bambha, N., Kianzad, V., Khandelia, M., and Bhattacharrya, S.S., Intermediate Representations for Design Automation of Multiprocessor DSP Systems. In Design Automation for Embedded Systems, vol. 7, 307-323, Kluwer Academic Publishers, 2002.

[5] Culler, D.E., and Singh, J.P., Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Publishers, 1999.

[6] Dasdan, A., and Gupta, R.K., Faster Maximum and Minimum Cycle Algorithms for System-Performance Analysis. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 10, 889-899, Oct. 1998.

[7] Goossens, K.G.W, et al., Guaranteeing the Quality of Services in Networks on Chip. In Networks on Chip, ed. by A. Jantsch and H. Tenhunen, 61-82, Kluwer Academic Publishers, 2003.

[8] Govindarajan, R., Gao, G.R., and Desai, P., Minimizing Buffer Requirements under Rate-Optimal Schedule in Regular Dataflow Networks. In Journal of VLSI Signal Processing, vol. 31, 207-229, Kluwer Academic Publishers, 2002.

[9] Govindarajan, R., and Gao, G.R., Rate-Optimal Schedule for Multi-Rate DSP Applications. In Journal of VLSI Signal Processing, vol. 9, no. 3, 211-232, 1995.

[10] Hoang, P., and Rabaey, J., Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. In IEEE Transactions on Signal Processing, vol. 41, no.6, June 1993.

[11] Kock, E.A. de, Practical Experiences: Multiprocessor Mapping of Process Networks: a JPEG Decoding Case Study. In Proceedings 15th International Symposium on System Synthesis, 68-73, ACM, 2002.

[12] Kock, E.A. de., et al., YAPI: Application Modeling for Signal Processing Systems. In Proceedings 37th Design Automation Conference, 402-405, 2000.

[13] Lee, E.A., and Messerschmitt, D.G., Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. In IEEE Transactions on Computers, vol. 36, no. 1, 24-35, 1987.

[14] Lauwereins, R., Engels, M., Ade M., and Peperstraete, J.A., Grape-II: A System-Level Prototyping Environment for DSP Applications. In IEEE Computer, vol. 28, no. 2, 35-43, Feb. 1995.

[15] Rijpkema, E., Goossens, K.G.W., and Radulescu, A., Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip. In Proceedings of DATE'03, 350-355, ACM, 2003.

[16] Rudack, M., Redeker, M., Hilgenstock, J., and Moch, S., A Large-Area Integrated Multiprocessor System for Video Applications. In IEEE Design & Test of Computers, vol. 19, no. 1, 6-17, 2002.

[17] Sriram, S., and Bhattacharyya, S.S., Embedded Multiprocessors: Scheduling and Synchronization, Marcel Dekker, Inc., 2002.

[18] Strik, M.T.J., Timmer, A.H., Meerbergen, J.L.van, and Rootselaar, G.-J. van, Heterogeneous Multiprocessor for the Management of Real-time Video and Graphics Streams. In IEEE Journal of Solid-State Circuits, vol. 35, no. 11, 1722 –1731, Nov. 2000.

[19] Yang, M.-T., Kasturi, R., and Sivasubramaniam, A.A., Pipeline-Based Approach for Scheduling Video Processing Algorithms on NOW. In IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 2, 119-130, Feb. 2003.