Predictable Embedded Multiprocessor System Design

Marco Bekooij, Orlando Moreira, Peter Poplavko,
Bart Mesman, Milan Pastrnak, Jef van Meerbergen

Philips Research
Prof. Holstlaan 4
Eindhoven, The Netherlands

Abstract

Consumers have high expectations about the video and audio quality delivered by media processing devices like TV-sets, DVD-players and digital radios. Predictable heterogenous application domain specific multiprocessor systems, which are designed around networks-on-chip, can meet demanding performance, flexibility and power-efficiency requirements as well as stringent timing requirements. The timing requirements can be guaranteed by making use of resource management techniques and the analytical techniques that are described in this paper.

1. INTRODUCTION

Multimedia signal processing and channel decoding in consumer systems is, for performance and power-efficiency reasons, typically performed by more than one processor. The processing in these embedded systems has often stringent throughput and latency requirements. In order to meet these requirements the system must behave in a predictable manner such that it is possible to reason about its timing behavior. The use of analytical methods is desirable because simulation can only be used to demonstrate that the system meets its timing requirements given a particular set of input stimuli. During the design of a multiprocessor system these analytical methods are needed for the derivation of the minimal hardware such that the timing requirements can be met. Given an instance of the multiprocessor system, these methods are needed to program the system in such a way that the timing requirements are met.

A traditional implementation of a predictable multiprocessor system for channel decoding and video processing is a pipe of tighly coupled dedicated hardware blocks with some glue logic for the communication between the blocks. However, there are many reasons to consider this design style as becoming less viable.

First of all, we are currently witnessing the convergence of previously unrelated application domains. For example, ordinary TV-sets are gradually evolving from straightforward terminals to interactive multimedia terminals. Digital auto radios are being combined with navigation systems, and wireless telephone functionality which provides a low bandwidth uplink. DVD players are evolving into DVD writers with complex graphics pipelines that allow to include special effects in our home-brew videos. This convergence leads to heterogeneity and many options, which all must be supported in a robust and efficient way by the multiprocessor system. The flexibility that is needed in these systems is often more than can be provided with dedicated hardware blocks. The reason is that these blocks are typically designed for one particular function. Also the order of the hardware blocks is fixed in a dedicated hardware pipe while the same blocks could be reused in different applications if the order could be adapted.

Another important reason why a pipe of tighly coupled dedicated hardware blocks becomes a less viable solution, is that the applications become more dynamic as new algorithms seek to take advantage of the disparity between average and worst-case processing. Also the increased interaction with the environment makes systems more dynamic. Often tasks and complex data types are created and deleted at run-time based on non-deterministic events like pressing of a remote control button. This results in many irregularities in the control flow of the application which can be handled effectively by sequential processors but not with dedicated hardware. This dynamism requires more scheduling freedom than can be provided by tightly coupled and synchronized hardware blocks.

The so-called "design productivity gap" states that the increase in our ability to design embedded systems does not match with the exponential growth over time of the number of transistors that can be integrated in an IC as described by Moore's law. To close the gap, system design methods must harness exponential hardware resource growth. This requires a modular, scalable design style, which is definitely not the case for a pipe of dedicated hardware blocks. The design style must also be composable, which is the case if the correctness (including the timing) of a complete system can be established by using only the (independently proven) features of its components.

The rapid increase of the masks' cost makes it necessary to design a single System-on-Chip (SoC) for a complete product family. The required flexibility of these SoCs make programmable multiprocessor systems an attractive option. A tradeoff between flexibility and performance must be made due to the power dissipation limit of approximately one 1W of cheap plastic IC-packages and of approximately 100 mW for battery powered devices. So, for power-efficiency reasons these systems will typically contain a mix of dedicted hardware blocks, application domain specific processors, and general purpose microprocessors. It is therefore necessary that the timing analysis techniques are applicable for systems in which application domain specific processors and dedicated hardware blocks are applied that do not support preemption.

A Network-on-Chip (NoC), like the network proposed by the Æthereal project [1], seems a promising option to con-

nect processors with each other. A connection in such a network describes the communication between a master and a slave and such a network can offer differentiated services for these connections. Connections with a guaranteed throughput service which have a guaranteed minimum bandwidth, a maximum latency, a FIFO transaction ordering and end-to-end flow-control are essential for the design of predicatable systems as will be explained in the next sections.

The outline of this paper is as follows. In Section 2, the characteristics of the target application domain are described, and our model that captures the behavior of the application is introduced. The resource management techniques which are described in Section 3, bound the sources of uncertainty in the system such that reasoning about the timing behavior of the system becomes possible. A multiprocessor system architecture which allows resource budgets allocation and enforcement is described in Section 4. The timing analysis and scheduling techniques are described in more detail in respectively Section 5 and Section 6. Finally, in Section 7 we state conclusions and indicate future work.

2. APPLICATION DOMAIN MODEL

Applications like high quality multi-window television with pixel processing enhancement, graphics and MPEG-video decoders require a computational performance in the order of 10-100 giga operations per second. Such a performance can not be provided by a single processor and requires a multiprocessor system.

In such an application the user can start and stop *jobs* that process video streams which are displayed after scaling in a separate windows. A job consists of tasks which are created (deleted) as the job starts (stops). Figure 1 shows an application that consist of an MPEG2 video-decoder job, an MPEG1 video-decoder job, a mixer job, a contrast correction job and an audio decoder job. The MPEG2 job consists for example of an IDCT, a VLD, and some other tasks. The tasks communicate via FIFO channels. The tasks within a job are activated on data availability. In some cases, it is necessary to activate tasks in a job by an event like a clock signal, instead of the presence of new input data. For example, assume that the mixer job in Figure 1 must produce a video frame every 20 ms. In this case the mixer should redisplay the previous frame after a clock pulse of a 50 Hz clock in the case that the decoding of an MPEG2 frame takes more than 20 ms.

The MPEG1 and MPEG2 decoder jobs have a soft real-time requirement, i.e. it is desirable that these jobs produce a new video frame every 20 ms. The mixer and contrast correction job in Figure 1 have hard real-time requirements, i.e. these jobs must produce a new video frame every 20 ms as is required by the display. Also the audio decoder has hard-real time requirements because an uninterrupted audio stream must be produced which is synchronized with the video stream.

The jobs are described as Synchronous Data Flow (SDF) graphs [2] [3], such that the throughput and latency of the jobs can be derived with analytical techniques. Tradionally these SDF graphs are used because they make compile time scheduling possible which eliminates the run-time scheduling overhead. However, also run-time scheduling of SDF graphs can be an attractive option, as will be discussed in Section 5 of paper.

An example of an SDF is shown in Figure 2. The nodes

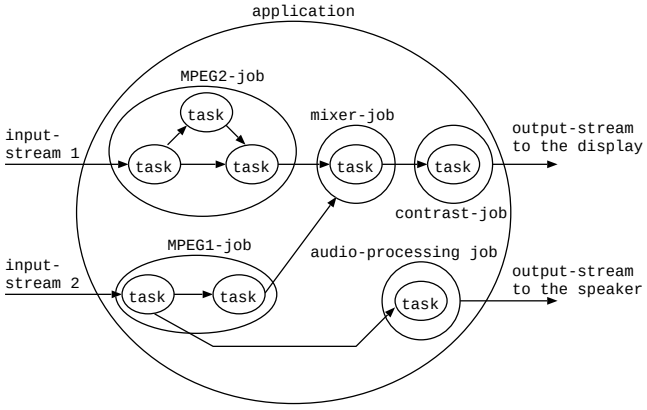

Figure 1: An application which consist of jobs. Jobs are started and stopped by the user. Jobs consist of tasks which communicated via FIFO channels.

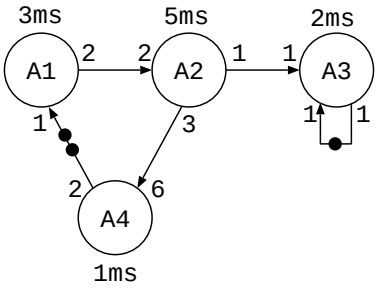

Figure 2: An SDF graph.

in an SDF are called actors. Actors are tasks with a well defined input/output behavior and a execution time which is less than equal to a specified worst-case execution time. The black dot in Figure 2 is a token. A token is a container in which a fixed amount of data can be stored. The edges in the SDF are called data edges and denote data dependencies. The number at the head of a data edge denotes the number of tokens that an actor consumes when it is fired. The number at the tail of an edge denotes the number of tokens an actor produces when it is fired. An actor is fired as soon as on every incomming edge of the actor at least the number of tokens are available as is specified at the head of the data edge. These tokens are consumed from the input edges of the actor before the execution of an actor finishes. The number of tokens specified at the tail of every output edge of the actor is produced before the execution of the actor finishes. Tasks with internal state are modeled in an SDF with a self edge, like the self edge of actor A3 in Figure 2. This self edge is given one initial token such that the next execution can not start before the previous execution is finished.

Tokens are, in the SDF model, consumed in the same order as they are produced. Therefore, FIFOs can be used as buffers in the communication channels. The maximal number of tokens that can be stored in a FIFO is called the FIFO capacity. FIFOs with a fixed capacity can be modelled in an SDF graph with an additional data edge

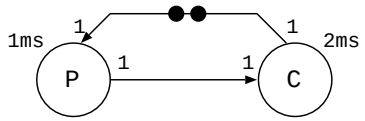

Figure 3: SDF model of a FIFO with a capacity of 2 tokens between a producing and consuming actor.

from a consumer (C) to the producer (P) as is shown in Figure 3. The number of initial tokens should be equal to the FIFO capacity.

Jobs are often described by Kahn process networks [4]. A major difference between an SDF description and a Kahn process network description is that the execution time of a Kahn process is not defined. It is sure that an actor will finish its execution within its worst-case execution time. However, depending on the values of the input data, a Kahn process can block and wait till it receives new input data via one of its input ports. Also the number of the tokens consumed by a Kahn processes is not defined. Without a specification of the process execution time and the number of the tokens that are consumed it is impossible to derive the throughput that will be obtained when the Kahn process network is executed. Therefore it is not possible to derive timing properties of an application given a Kahn processes network description of an application. It is even impossible to guarantee that the system does not deadlock. The reason is that the required capacity of the FIFOs in the communication channels of a Kahn process network can only be determined at run time. Therefore, is it possible that the memory needed during execution exceeds the memory capacity available in the system. A request for more memory than is available in the system can result in a deadlock situation. A similarity between Kahn process networks and SDF graphs is that the communication takes place via communication channels that maintain a FIFO order. Another similarity is that Kahn process networks and SDF graphs can be executed purely data driven, that is, without a clock signal or even a global notion of time. Kahn process networks as well as SDF graphs are deterministic i.e. these models will produce for the same stream of input values the same stream of output values.

To be able to design a fully predictable system that can meet the timing requirements, it is necessary to bound the uncertainty that can be introduced by the user, the application and the hardware. The uncertainty is bounded by the resource management techniques that are described in the next section. Given that the uncertainty is bounded, a guaranteed minimal throughput and maximal latency can be derived for each job, with the analysis techniques that are described in Section 5.

3. RESOURCE MANAGEMENT

The uncertainty introduced by the application, the hardware and the environment can make reasoning about the timing behavior of jobs difficult or even impossible. The resource management techniques described in this section bound the uncertainty such that the timing behavior of jobs can be derived.

We distinguish the uncertainty in the resource supply from the uncertainty in the resource demand. The uncertainty in the resource supply is due to resource arbitration in the hardware of the multiprocessor system. Resource arbitration of busses and the replacement policy of cache lines, are examples. The uncertainty in the resource demand is due to the data value dependent processing (e.g., conditional branches in the program code) and external events. External events are for example generated by the user to start and stop jobs.

The uncertainty in resource supply is bounded by making use of predictable hardware arbitration schemes. An arbitration scheme is predictable in the case it is known how long it maximally takes before a resource becomes available. The minimal time that the resource stays available must also be known. An example of a predictable arbitration scheme for a bus is Time Division Multiple Access (TDMA). In this scheme, it is guaranteed that the bus can be obtained after a fixed amount of time and that during a fixed amount of time the data can be transfered via the bus.

The uncertainty in the resource demand is bounded by making use of admission control and hardware resource budget enforcement. Admission control takes care that sufficient hardware resources are available when a job is started. If there are insufficient resources available in the system then the user is notified that the job is rejected. Resource budgeting guarantees that an actor of a job gets a certain amount of memory, bandwidth and processor cycles. By enforcing budgets it becomes impossible for an actor to claim more resources than its budget. Due to these enforced budgets the interference of actors of different jobs is bounded. With enforced resource budgets, it looks for a job as if it runs on its own private hardware. In other words, resource budget enforcement creates for each job its own virtual platform with private hardware resources.

Resource budgets are determined by the resource manager at the start-up of a job. Hard real-time, soft real-time and best effort jobs are treated differently. For hard real-time jobs it is unacceptable that a single deadline is missed, and per definition these jobs do not support graceful degradation. Therefore, the worst-case amount of resources that could be needed during the execution of a hard real-time job must be allocated by the resource manager. For soft real-time jobs, less than the worst-case amount of resources can be allocated because missing of a deadline results in some diminished value for the end user. The allocated resource budgets are based on a prediction of the behavior of the job. If more resources are needed during the execution than are allocated, then the job will miss a deadline or the job can reduce the required amount of resources by reducing the quality of its end result. In case the quality must be reduced for a longer period of time, then the job can ask the resource manager for an increase of its budget. For best-effort jobs there are no timing constraints defined. A faster average execution time of a best-effort job is appreciated by the end user. These jobs can make use of the resources which are either not allocated to hard real-time and soft real-time jobs or are not used by the real-time jobs during their execution.

The architecture template of the proposed predictable multiprocessor system is described in the next section. In such a system resource budgets can be enforced and predictable arbitration mechanisms are applied.

4. MULTIPROCESSOR TEMPLATE

The architecture template of the proposed multiprocessor

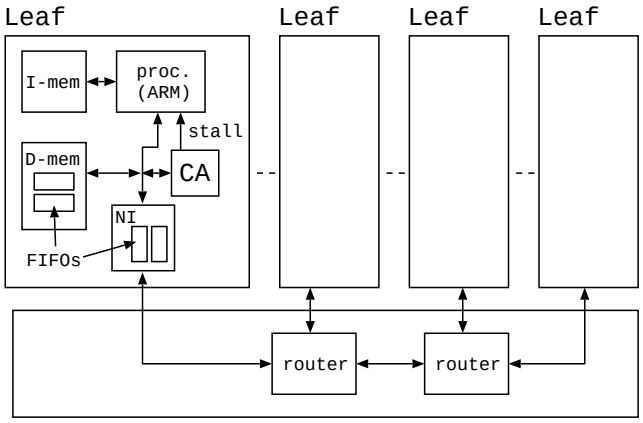

Figure 4: Multiprocessor template.

system is shown in Figure 4. The processors in this template are, together with their local data memory, connected to the Network Interface (NI) of a NoC. The transfer of data between a local memory and a network interface is performed by a Communication Assist (CA). A processor together with its local instruction and data memory, communication assist and network interfaces is grouped into a leaf. The leafs are connected to the routers of our network. Network links connect the routers in the desired network topology.

A processor in a leaf has a separate Instruction Memory (I-mem) and Data Memory (D-mem) such that instruction fetches and data load and store operation do not cause contention on the same memory port. A wide range of memory access time variation due to contention on a memory port is intolerable because this would result in an unpredictable execution time of the instructions of the processor. A processor can only access its local data memory. Given a 1 cycle access time of a local memory there is no reason to make it cacheable.

Every processor in this template has its own private memory which improves the scalability of the system. Such a multiprocessor system is intended for message passing which is a suitable programming paradigm for most signal processing applications. Because message passing is used instead of the shared address space communication model there is no memory bottleneck to a central shared memory. Coherency can't be a problem in a system which makes use of message passing in the case that there is only one producer and one consumer for every data item that is communicated between processors.

Communication between actors on different processors takes place via a virtual point to point connection of the NoC. The result of the producing actor is written in a logical FIFO in the local memory of the processor. Such a logical FIFO can be implemented with the C-HEAP [5] communication protocol, without use of semaphores. A communication assist polls at regular intervals whether there is data in this FIFO. As soon as the CA detects that there is data available it copies the data into a FIFO of the NI. There is one private FIFO per connection in the NI. Then the data is transported over the network to the NI of the receiving processor. As soon as the data arrives in this NI, it is copied by the CA into a logical FIFO in the memory of the processor that executes the consuming actor. The data is read from this FIFO after the consuming actor has detected that there is sufficient data in the FIFO. Flow control between the producing and consuming actor is achieved by making sure that data is not written into a FIFO before it is checked that there is space available in this FIFO.

Data is stored in the local memory of the processor before it is transfered accross the network. This done for a number of reasons. First of all, the bandwidth of a connection is set by configuring tables in the network for a longer period of time. The bandwidth reserved for a connection will typically be less than the peak data rate generated by the producing actor. Therefore a buffer is needed between the processor and the network to average out the data rate such that the bandwidth provided by the network is well utillized. The size of this buffer is significant given the assumption that the actors produce large chunks of data at large intervals. On the other hand the network will transfer very small chunks of data (3 words of 32 bits) at very small intervals (2 ns). Given that large memories are inherently slow it is desirable to split the large logical FIFO between the processor and the network, in a small (32 word) dedicated FIFO per connection in the network interface and a large logical FIFO in the local memory of the processor. The task of the CA is then copying of the data between FIFOs in the NI and FIFOs in local memory of the processor.

Another important reason to store data in a local memory and not via the network in a remote memory is that the latency of the load and store operations that are executed on the processor should not depend on the latency of the network. A longer latency of load and store operations whould result in a longer worst-case execution time of the actors and would effectively decrease the performance of the processor. By making use of a FIFO in the local memory, the computation performed by the processor can be overlapped by the communication that is performed by the CA.

Finally, there is no need to send addresses over the network because the remote memories are not accessed via the network by a processor nor by a communication assist. This will save network bandwidth and will improve the power efficiency of the multiprocessor system. Only data can be sent over the network because the configuration of the network and the communication assists determine between which FIFOs data is transported.

The CA is also responsible for the arbitration of the data memory bus. The applied arbitration scheme [6] is such that a low worst-case latency of memory store and load operations is obtained and that a minimal throughput and maximal latency per connection is guaranteed. In this scheme the time is divided in intervals P of p cycles. Each interval P is subdivided in intervals Q of q cycles. During each interval Q it is guaranteed that the processor can access the memory at least once. This guarantees that a load/store operation of the processor takes at most q cycles. If the processor does not access the memory during an interval Q, then this access can take place in a successive interval Q that belongs to the same interval P. This way the processor can access the memory at most p/q cycles per interval P. The processor is stalled by the CA till the next period P if it issues more than p/q load and store operations per interval P. Therefore there are at least $p - p/q$ cycles per interval P for the communication assist to copy data between the NI and the local memory. A predefined portion r of the $p - p/q$ cycles is al-

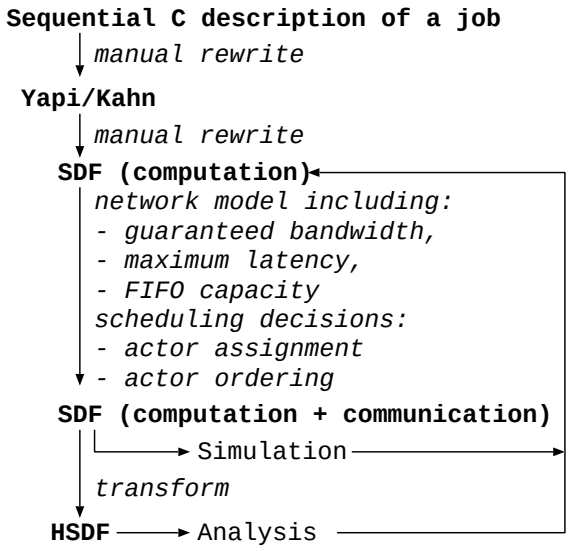

Figure 5: Analysis and simulation flow.

located per connection for copying of data between the CA and the NI. The minimal throughput and maximal latency for a connection is guaranteed because data can be copied between the local memory and the CA for at least r cycles per p cycles.

In the proposed architecture the communication between actors that run on different processors has a guaranteed minimal throughput and a maximal latency. Given these characteristics, the communication can be modeled as if it takes place through completely independent virtual point-to-point connections. These connections are modeled together with the actors of a job in one SDF graph. Given this SDF graph, the guaranteed minimal throughput of the job can be determined with the analysis techniques that are described in the next section.

5. ANALYSIS TECHNIQUES

It is assumed that initially a sequential C-description of a job is provided, which is the input of our analysis and simulation flow (see Figure 5). This sequential C description is manually rewritten into another C-description in which the task level parallelism in the job is made explicit. In a successive rewriting step care is taken that all tasks have actor semantics, i.e. they do not start before there are tokens on all inputs, have a bounded execution time and consume tokens of a fixed size. Also the WCET of the actors is determined by analysis of the program flow [7]. The SDF graph that is obtained represents a job.

The actors in the SDF are assigned to processors and ordered by a scheduler (see Section 6). The actor assignment and order is modeled with additional edges and actors in the same SDF graph. In the case that actors communicate via the network then a model of a connection in the network replaces edges in the SDF graph. This network connection model consist of SDF actors and edges. This model includes the FIFOs between the processor, the CA and the network. A description of such a network connection model is out of the scope of this paper.

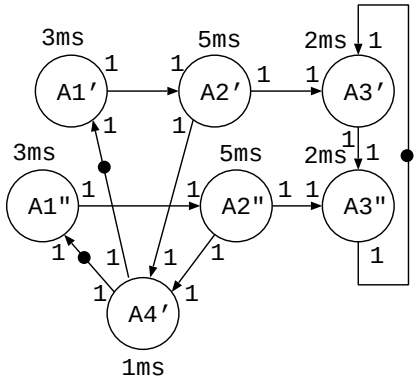

Figure 6: An HSDF graph which is equivalent to the SDF graph in Figure 2.

The SDF graph in which the job, the network as well as the actor assignment and actor order is modeled is transformed into a Homogeneous Synchronous Data Flow (HSDF) graph on which the timing analysis is performed. An algorithm which can transform any SDF graph into a HSDF graph is described in [3]. An HSDF graph is a special kind of an SDF graph in which the execution of an actor results in the consumption of one token from every incoming data edge of the actor and the production of one token on every outgoing data edge of the actor. The HSDF graph obtained after transformantion of the SDF graph in Figure 2 is shown in Figure 6.

An HSDF graph can be executed in a self-timed fashion, which is defined as a sequence of firings of HSDF actors in which the actors start immediately when there are tokens on all their inputs. If the HSDF graph is a strongly connected graph and a FIFO ordering is maintained for the tokens, then the self-timed execution of the HSDF graph has some important properties. A FIFO ordering is maintained if the completion events of firings of the same actor occur in the same order as the corresponding start-events. This is the case if an actor in the HSDF graph belongs to a cycle with only one token otherwise the actor must have a constant execution time. In [8] the properties of the self-timed execution of an HSDF graph are derived with max-plus algebra.

The most important property of the self-timed execution of an HSDF graph is, that it is deadlock-free if there is on every cycle in the HSDF graph at least one initial token. Secondly, the execution of the HSDF graph is monotonic, i.e. decreasing actor execution times result in non-increasing actor start times. Third, an HSDF graph $G(V, E)$ will always enter a periodic regime. More precisely, there exist integers K,N and λ, such that for all $v \in V$, $k > K$ the start time $s(v, k + N)$ of actor v in iteration $k + N$ is described by:

$$s(v, k + N) = s(v, k) + \lambda \cdot N \quad (1)$$

Equation 1 states that the execution enters a periodic regime after K executions of an actor in the HSDF graph. The time one period spans is $\lambda \cdot N$. So, λ is equal to the inverse of the average throughput. The number of iterations in one period is denoted by N.

The Maximum Cycle Mean (MCM) of the HSDF, which is equal to λ, is given by equation 2. In this equation is CM

the Cycle Mean of a simple cycle $c \in G$ (see equation 3). In this equation denotes $tokens(c)$ the number of tokens on the edges in a cycle c. The Worst Case Execution Time (WCET) of actor v is denoted by WCET(v).

$$\mathrm{MCM}(G) = \max_{c \in G} \mathrm{CM}(c) \tag{2}$$

$$\mathrm{CM}(c) = \sum_{v \ on \ c} \mathrm{WCET}(v)/tokens(c) \tag{3}$$

The number of iterations in one period is denoted by N. The value of N can be derived with the following procedure. First, all critical circuits are derived which are cycles $c \in G$ with a cycle mean equal to the MCM. Then, a critical graph $G^c(V^c, E^c)$ is derived which consists of all nodes and edges in G which belong to critical circuits. For each maximal strongly connected subgraph G^s in G^c the number of tokens t_s^p on every simple path p in the subgraph is derived. Then a value $d_s = gcd(t_s^{p1}, t_s^{p2}, ..., t_s^{pn})$ is derived for each subgraph G^s, in which gcd denotes the greatest common divisor. The number of iterations N in one period is equal to $lcm(d_{s1}, d_{s2}, ..., d_{sm})$ in which lcm denotes the least common multiple.

The value K can be derived by simulating the HSDF graph in a self-timed fashion given that all actors have an execution time equal to their worst-case execution time. The value K is the first iteration in which the start-times of the actors correspond to equation 1.

The worst-case start-times of the actors during the transient state as well as the steady state can be derived by simulation. During this simulation, all actors must have an execution time equal to their worst-case excecution time. Due to the monotonicity of the HSDF are the start-times observed during this simulation are worst-case start times. Given equation 1 there is no need to simulate beyond the first period of the periodic regime.

So far it was assumed 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 violate the timing contraints of the job. However, in many signal processing system the input data is provided by an external source which provides a new input sample every clock period. An example of such a source is an A/D converter. A similar situation occurs often at the output of the system where an external sink consumes an output sample every clock period. An example of such a sink is a D/A converter. Given such an external source and/or sink it should be guaranteed that the samples produced by the source can always be stored in the FIFO between the source and the system. It should also be guaranteed that there are always sufficient samples in the FIFO between the system and the sink.

The source and sink are modeled as actors in the HSDF graph as is shown in Figure 7 in order to verify whether the FIFOs at the input and output of the system do not overflow or underflow. The source and sink actors are given a WCET of $1/f_{clock}$, which is the length of 1 clock period. The self edge with one initial token garantees that the next execution of the source and sink actor can not start before the previous execution is finished. Thus, the self-edge in combination with the WCET of the source and sink actors enforce a maximal execution frequency of these actors during the self-timed execution of the HSDF graph in the simulator. During simulation it should be verified that the time between succesive executions of the source as well as the sink actor is exactly one clock period. If this is the case, then it is guaranteed that in an implementation of the system, FIFO1 between source and the rest system never overflows and FIFO2 between the system and the sink never underflows. The reason is that actors can not start later than they start during a simulation run in which all actors have an execution time equal to their worst-case execution time. If the actors are started earlier then a token in FIFO1 can never be consumed later than during the simulation run, which results in the same or less tokens in FIFO1. If the actors starts earlier, then a token is never produced later in FIFO2 than during the simulation run, which results in a greater or equal number of tokens in FIFO2 in the implementation.

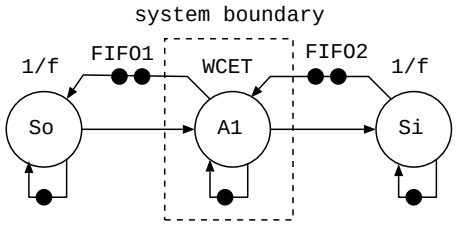

Figure 7: HSDF graph which is used to prove that, given a strict periodic source and sink, the FIFOs at the input and at the output of the system have sufficient capacity.

It depends on the applied scheduling strategy whether the actor assignment as well as actor ordering decisions are made during the execution of a job, or before a job is started. In the case that a decision is taken after the job is started, then the decision, out of all possible decisions, which would lead to as late as possible actor start-times is modeled in the SDF. The pros and cons of the different scheduling strategies are the topic of the next section.

6. SCHEDULING STRATEGIES

Scheduling decisions can be taken before a job is started or during the execution of the job. Taking scheduling decisions during the execution of the jobs (at run-time) implies some run-time overhead. This run-time overhead can be reduced if these schedule decisions are made before the job is started (at compile-time). Schedule decisions that are taken at compile-time imply that some freedom in the schedule is removed in order to reduce the run-time overhead. Therefore, the scheduling strategy that should be selected depends on the knowledge that is available at compile-time of the job and the amount of run-time scheduling overhead that is introduced when scheduling decisions are taken at run-time. A comparison of the scheduling strategies is given in Table 1. These scheduling strategies differ in whether the actor to processor assignment, the execution order of the actors on a processor and the start-times of the actors are determined at compile or at run-time. For all these scheduling strategies techniques exist to guarantee the timing. In the next paragraphs the characteristics of these scheduling strategies are described in more detail.

In the fully-static scheduling strategy, the assignment of actors to processors as well as the order in which the ac-

Scheduling strategy	Assignment	Ordering	Invocation time
Fully-static	compile-time	compile-time	compile-time
Static order	compile-time	compile-time	run-time
Static assignment	compile-time	run-time	run-time
Fully dynamic	run-time	run-time	run-time

Table 1: Comparison of scheduling strategies.

tors are executed and the start-times of the actors, are determined by a scheduler before the job is started. This scheduling strategy requires that the processors have the same global time reference such that at run-time the actors can be started at the right moment in time. A disadvantage of this scheduling strategy is that a global notation of time is difficult to realize due to deep sub-micron effects in a large system with many processors. Another disadvantage of the fully static scheduling strategy is that the length of the schedule is fixed and does not decrease if the execution times of the actors decrease. Decreasing execution times of actors result in more idle processor cycles. The main advantage of the fully static scheduling strategy is that explicit synchronization between actors is not required because actors are started after it is guaranteed that their input data is available and that there is space to store their results.

In the static order scheduling strategy the assignment of actors to processors as well as the order in which the actors are executed are determined by a scheduler before the job is started. An actor is started as soon as its input tokens are available and it is the next actor to be executed on a processor according to the predefined cyclic execution order. Checking whether there is input data implies run-time overhead. A decrease in the execution time of the actors results in a shorter schedule. The remaining processor cycles can be used by another job. An important property of the static order scheduling strategy is that only the throughput of the system is decreased if the execution time of an actor is larger than the execution time assumed by the scheduler. It can not result in a deadlock of the system. This opens the possibility to determine the assignment and order of actors of soft real-time jobs by the scheduler based on a predicted execution time of the actors instead of on the worst-case execution time of the actors.

In the static assignment scheduling strategy the assignment of actors is determined before the SDF graph is started. After the SDF graph is started the activation of an actor occurs as soon as its input tokens become available. An important advantage of this approach is that the assignment of the actors to processors does not require a compile-time ordering step. The actor assignment problem is basically a bin packing problem where the total load on a processor may not exceed 100%. Currently we investigate whether the assignment of actors can be performed by an on-line algorithm just before the SDF graph is started. An advantage of the static assignment scheduling strategy is that the pipeline is automatically filled when the SDF graph of a job is started while an explicit encoded pre- and post-amble is needed in the static order scheduling strategy. The main disadvantage of the static assignment scheduling strategy is that the minimal required FIFO capacity will typically be larger than is needed for a static order schedule with the same throughput. Another related disadavantage is that the worst-case latency of the system is larger than in the case the fully static or the static order scheduling strategy is applied.

In the fully dynamic scheduling strategy the assignment of actors to processors as well as the execution order and start-times of the actors are determined at run-time. The actor to processor assignment is adapted at run-time such that the computational load of the processors is kept balanced. Load balancing can result in a shorter execution time of the jobs. Moving an actor at run-time from one processor to another implies, in our architecture, that data must be copied from one memory into another memory. We expect that for the multi-media signal processing domain, the gain in processor cycles obtained by balancing the processor load does typically not outweigh the cost of moving data. The reason is that, for this application domain, the difference between the worst-case and the average-case execution time of the actors in the jobs is often not large enough.

7. CONCLUSION & FUTURE WORK

The processing in many consumer systems is performed by embedded multiprocessor systems for performance and power-efficiency reasons. The processing in these systems has usually stringent throughput and latency requirements. This paper presents a multiprocessor architecture and an SDF-model of the jobs in an application which enables reasoning about the timing behavior of the system. A scalable multiprocessor system is obtained by making use of a network on chip for the communication between processors. The network provides virtual point-to-point connections with a guaranteed throughput and maximal latency such that the timing of the system can be derived with the presented analysis techniques.

8. REFERENCES

[1] E. Rijpkema, K.G.W. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip", Design, Automation and Test in Europe Conference (DATE), 2003, pp. 350–355.

[2] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow", Proceedings of the IEEE, 1987.

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

[4] G. Kahn, "The semantics of a simple language for parallel programming", Proceedings IFIP Congress, 1974, pp. 471–475.

[5] O.P. Gangwal, A. Nieuwland, and P. Lippens, "A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems", International Symposium on System Synthesis, 2001, pp. 1–6, ACM.

[6] S. Hosseini-Khayat and A.D. Bovopouplos, "A simple and efficient bus management scheme that supports continous streams", ACM Transactions on Computer Systems, 1995, vol. 13, pp. 122–140.

[7] K. Chen, S. Malik, and D.I. August, "Retargetable static timing analysis for embedded software",

International Symposium on System Synthesis, 2001, pp. 39–44, ACM.

[8] F Bacelli, G Cohen, G.J. Olsder, and J-P. Quadrat, *Synchronization and Linearity*, John Wiley & Sons, Inc., 1992.