Monotonicity and Run-Time Scheduling [*]

Maarten H. Wiggers[1], Marco J. G. Bekooij[2,3], and Gerard J. M. Smit[3]
[1]Department of EE, Eindhoven University of Technology, Eindhoven, The Netherlands
[2]NXP Semiconductors, Eindhoven, The Netherlands
[3]Department of EEMCS, University of Twente, Enschede, The Netherlands
m.h.wiggers@tue.nl, marco.bekooij@nxp.com, g.j.m.smit@utwente.nl

ABSTRACT

Modern embedded multi-processors can execute several stream-processing applications concurrently. Typically, these applications are partitioned into tasks that communicate over buffers together forming a task graph. The fact that these applications are started and stopped by the user combined with the knowledge that not all applications are necessarily completely characterised makes it attractive to use run-time scheduling. We define and characterise a class of budget schedulers that by construction bound the interference from other applications. Furthermore, we will show that the worst-case effects of these schedulers can be included in dataflow process networks. The execution of the resulting dataflow process network is shown to result in tight and conservative bounds on the end-to-end temporal behaviour of the execution of the task graph on a cycle-true simulator. Given that the inter-task synchronisation of the application allows for a dataflow model that is functionally deterministic, this enables exploration of various buffer capacities and scheduler settings at a high level of abstraction.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms

Algorithms, Performance, Theory

Keywords

Dataflow, real-time, multi-processor, simulation

1. INTRODUCTION

Current consumer electronic devices, such as smart-phones, offer many applications, including audio/video playback and the baseband processing for the actual phone functionality. For such stream processing applications, often a distributed implementation on a multi-processor system is required for reasons of performance and power dissipation. In order to reduce costs, the processors and communication infrastructure are often shared between different applications. Typically, not all applications on such a system are completely characterised in terms of their execution times and activation rates. In combination with the fact that users start and stop applications, it is attractive to have run-time scheduling on the resources that are shared by multiple applications on such multi-processor systems.

Given an application that is partitioned into tasks that execute on different processors of such a multi-processor system, it is not necessarily straightforward to verify that the functional behaviour of the distributed implementation equals the functional behaviour of the reference implementation, which is typically a fully sequential program. However, if the distributed implementation is functionally deterministic, i.e. a given sequence of inputs always produces the same sequence of outputs, then this reduces the verification effort to a large extent.

For stream processing applications it is quite natural to have a distributed implementation that is a task graph, in which tasks communicate data over FIFO buffers. Such task graphs can often be intuitively modelled as dataflow process networks [15]. Sufficient conditions are known for dataflow process networks to be functionally deterministic [15].

Next to the functional behaviour also the temporal behaviour of these task graphs is of interest. While dataflow process networks have been extended with time [21], this was limited to task graphs that are executed on resources without run-time scheduling. In this paper, we will show that, for a class of run-time scheduling schemes, we can conservatively model the effect of run-time scheduling on the temporal behaviour of the task graph, given that the task graph can be modelled as a functionally deterministic dataflow process network. This is done by associating a suitable amount of time to the activations of the dataflow processes of this functionally deterministic dataflow graph. The resulting performance analysis is applicable for both single processor as well as multi-processor systems.

The proof of a conservative model as presented in this paper allows to model a (cyclic) sequence of execution times. This is not possible using the concepts used in the proof presented in [30]. This result not only widens the scope of dataflow modelling, it also has the following practical application. Dataflow process networks can be straightforwardly executed in a simulation environment. The fact that we show that such dataflow process networks can model the effects of run-time scheduling given a sequence of execution times means that we can include the effects of a real-time operating system at the communicating processes plus time (CP-T) abstraction level instead of the programmer's view plus time (PV-T)

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'09, October 12–16, 2009, Grenoble, France.
Copyright 2009 ACM 978-1-60558-627-4/09/10 ...$10.00.

abstraction level [8]. Furthermore, the results we obtain by simulation of the dataflow process network are valid, i.e. conservative, independent of the time at which applications are started and independent of other applications. This reduces the time required by a single simulation run, and the number of simulation runs required to characterise the performance of a task graph. Our model is tight, i.e. there are cases in which the actual behaviour equals the modelled behaviour, and typically the end-to-end temporal behaviour is accurately estimated.

The contribution of this paper is that we show that constraining oneself to functionally deterministic task graphs and a specific class of schedulers allows for a model at a high level of abstraction that provides conservative, i.e. pessimistic, estimates on end-to-end performance. The essential property, on which this contribution rests, is the monotonic temporal behaviour of our dataflow model.

Related work is discussed in Section 2. In Section 3, we will define functionally deterministic dataflow graphs. Then in Section 4, we will introduce our task model and from its relation with functionally deterministic dataflow graphs it will follow that the task graphs that adhere to the presented task model have time-invariant functional behaviour. Subsequently, in Section 5, we will show that functionally deterministic dataflow graphs have monotonic temporal behaviour. In Section 6, we discuss sufficient conditions for a dataflow graph to conservatively model the temporal behaviour of a task graph. A classification of different run-time scheduling schemes is given in Section 7. After which we show in Section 8 that a class of schedulers allow their worst-case effects to be included in a dataflow process. Implementation issues concerning inter-task synchronisation are discussed in Section 9. In Section 10 we evaluate the accuracy of our estimations on end-to-end temporal behaviour. After which we discuss some implications of this work in Section 11, and conclude in Section 12.

2. RELATED WORK

Dataflow process networks have been extended with time by Sriram and Bhattacharyya [21]. However, their work does not immediately extend to include the effects run-time scheduling. This is because if tasks are scheduled by a run-time scheduler, then you need to differentiate between the time that sufficient input data and output space is available for a task and the time that a task is started. This differentiation is not made in [21]. In [3, 30] the effects of run-time scheduling are included in the dataflow model. However, compared to [3] this paper specifies sufficient conditions for a task graph to be functionally deterministic and specifies the class of run-time schedulers whose effects can be modelled by dataflow processes. Compared to [30], this paper presents a proof that the model is conservative that does not rely on concepts from the Latency-Rate model [25]. The consequence is that instead of a model that uses one (worst-case) execution time, the model presented in this paper is valid for a (cyclic) sequence of execution times. The application of a (cyclic) sequence of execution times instead of a single execution time potentially improves the accuracy of the analysis, as for instance shown in [18].

Existing approaches, such as [9, 20], that present a simulation environment in which the effects of a real-time operating system are included all present a simulation environment at the PV-T abstraction level [8]. The approach presented in this paper is at the CP-T abstraction level. As shown in Section 10, this has major implications for the time required for a single simulation run. This is because at the CP-T level of abstraction fewer events need to be handled by the simulation kernel than at the PV-T level of abstraction. Furthermore, in our approach we have that the results obtained from a single simulation run are valid independent of the time at which applications are started and independent of other applications. These abstractions are difficult to achieve at the programmer's view plus time abstraction level.

In the communication by sampling method [27], tasks execute quasi-periodically and sample their input buffers. This has the advantage of fault containment, but the disadvantage that the functionality of the task graph depends on the execution times of the tasks. In the presented approach, we require that tasks use blocking communication primitives. Given this requirement and the requirement that the output values of a task execution are a function of its input values, we can show that the task graph is functionally deterministic. This means that independent of knowledge on worst-case execution times that the output values of the task graph are completely determined by the input values of the task graph. In our approach, the required type of run-time schedulers guarantees that faults are contained within a single application, i.e. the bounds on the temporal behaviour of one application are independent of other applications.

3. DETERMINISTIC DATAFLOW GRAPH

In this section, we state the sufficient conditions, from [15], for a dataflow graph to be functionally deterministic, i.e. output values are only determined by input values. This will enable us to show in Section 4 that YAPI task graphs [7] are functionally deterministic.

As defined in [15], a firing of a dataflow actor maps tokens from input queues into tokens on output queues. A set of firing rules specifies when an actor can fire. More specifically, a firing rule is a condition that specifies the tokens that need to be present on specific input queues before the actor can fire. A firing consumes, i.e. removes, input tokens and produces output tokens. A sequence of actor firings is called a dataflow process.

A sufficient condition for a dataflow process to be functionally deterministic is that each actor firing in this dataflow process is functional and that the set of firing rules is sequential. An actor firing is functional if it is side-effect free, i.e. the produced tokens in any firing are a function of the consumed tokens in that firing. A set of firing rules is sequential if there exists a pre-defined order in which the firing rules can be tested.

A network of dataflow processes interconnected by queues is called a dataflow graph. A dataflow graph is allowed to contain cycles. In general, initial tokens need to be placed on queues of these cycles in order to enable a non-terminating execution of the dataflow graph. We say that a dataflow graph is functionally deterministic if the tokens produced on the output queues of the graph are completely determined by tokens from the input queues and initial tokens from cycles in the dataflow graph. It is clear that if all dataflow processes in the dataflow graph are functionally deterministic, then also the dataflow graph is functionally deterministic. We define a functionally deterministic dataflow graph as a dataflow graph in which all dataflow processes only have actor firings that have (1) sequential firing rules and that are (2) functional.

4. TASK GRAPH

We require that an application is implemented as a task graph that consists of tasks that communicate over fixed capacity FIFO buffers. The capacity of a FIFO buffer is given by the number of containers it holds, where a container is a place-holder for data in which tasks, once they acquired the container, can do random-access. In order to write output data on a particular buffer, we require a task to first acquire a number of empty containers on this output buffer that is sufficient to store this data in. Similarly, a task first needs to acquire a number of full containers on its input buffers before it can read input data. On all input buffers, the number of full containers on which a task waits is only allowed to depend on the state of the task, i.e. the values of the local variables and the values in already acquired containers. Furthermore, we require that the produced values are a function of the values of the local variables and the values in already acquired containers. An application in which the tasks only communicate data using the YAPI [7] `read` and `write` primitives would adhere to these requirements, we call this a YAPI task graph. A code-fragment from an example YAPI task is shown in Listing 1. This example mimics the behaviour of a variable length decoding task, as found in audio or video decoders. For such a task, the input stream contains information concerning the number of bits with which the next packet or frame is encoded, i.e. the amount of input data that needs to be present for the second `read` primitive to succeed depends on earlier acquired data.

Listing 1 Example YAPI task.

```
// declare variables
...
while(1)
  {
   read(in_port, &header, HEADER_SIZE);
   // parse header
   ...
   read(in_port, &packet, this_packet_size);
   // process data
   ...
   write(out_port, &output, OUTPUT_SIZE);
  }
```

We will now show that a YAPI task graph is functionally deterministic. This is done by showing a one-to-one relation between YAPI task graphs and functionally deterministic dataflow graphs.

We associate with each task in the task graph a unique dataflow process in the dataflow graph. Then, we associate with each buffer in the task graph two queues in opposite direction connecting the corresponding dataflow processes. One of these two queues models the flow of full containers, i.e. data, while the other queue models the flow of empty containers, i.e. space. A full or empty container that is present when the task graph is started is reflected in the dataflow graph by an initial token on the corresponding queue. An example is shown in Figure 1, where dataflow process v_1 corresponds with task u_1, dataflow process v_2 corresponds with task u_2, and the two queues with d_1 initial tokens correspond with the buffer that contains d_1 initially empty containers.

Further, a task is partitioned in non-blocking code-segments by letting each acquisition of containers start a non-blocking code-segment. For a YAPI task graph, this means that each `read` and `write` starts a non-blocking code-segment, leading to three non-blocking code-segments in Listing 1. Execution of a task leads to a (possibly non-terminating) sequence of successive executions of this finite number of non-blocking code-segments. We required that the number of containers on which a task is waiting to arrive is only allowed to depend on the state of the task. Furthermore, we required that the values in the produced containers are a function of the values of the local variables and the values of already acquired containers.

We associate the state of a task, which is carried from one task execution to the next task execution, with one token on a queue from and to the corresponding dataflow process. If a set of firing rules of the actor firing is such that depending on the value of the token on this self-edge the number of tokens to be consumed from each other queue is completely determined, then the set of firing rules is sequential. It is clear that a set of firing rules can be constructed such that the number of tokens consumed from various queues depending on the value of this token has a one-to-one correspondence with the number of containers acquired from various buffers depending on the state of the task.

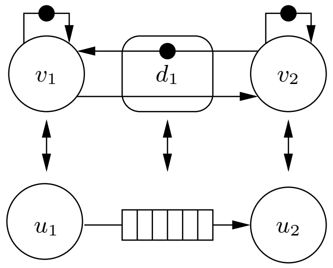

Figure 1: Example one-to-one correspondence between task graph and dataflow graph.

This implies that each task is associated with a functionally deterministic dataflow process. Since all these functionally deterministic dataflow processes are interconnected by queues the resulting dataflow graph is functionally deterministic. Because there is a one-to-one relation between task graph and dataflow graph, we can say that also the task graph is functionally deterministic.

5. TIMED DATAFLOW GRAPH

If a dataflow process is functionally deterministic, then the tokens produced by a dataflow process are only determined by the tokens arriving on the input queues. This implies that the produced tokens are independent of the arrival times of tokens on the input queues of dataflow processes.

As in [21], we can extend a functionally deterministic dataflow model to include time by separating the token consumption and token production of each actor firing. Instead of defining an actor firing as an atomic action in which tokens are consumed and produced, we define an actor firing as two atomic actions. One action consumes tokens, while the other action produces tokens. Since the produced tokens are a function of the consumed tokens, we require that the action that produces tokens is not before the action that consumes tokens. It is clear that this does not change the functionality.

Let $t(i) \in \mathbb{R}^+$ be the difference in time between the token production action and token consumption action of actor firing i. In a self-timed schedule, actor firings start as soon as the firing rule is satisfied. Let $\sigma(G, t)$ provide the start times of the self-timed schedule of dataflow graph G, given that actor firing i requires $t(i)$ time. In a self-timed schedule of a dataflow graph, every actor firing starts as soon as its firing rule is satisfied.

DEFINITION 1. *A dataflow graph has monotonic temporal behaviour if we have that*

$$(\forall i \bullet t'(i) \leq t(i)) \Rightarrow \sigma(G, t') \leq \sigma(G, t) \qquad (1)$$

THEOREM 1. *Functionally deterministic dataflow graphs have monotonic temporal behaviour.*

PROOF. All dataflow processes in a functionally deterministic dataflow graph G are functionally deterministic. A functionally deterministic dataflow process only has actor firings that have sequential firing rules and where the produced tokens are a function of the consumed tokens. This implies that the firing rules and produced tokens are independent of the arrival times. Given schedule

$\sigma(G,t)$ that is the self-timed schedule of G if actor firing i takes $t(i)$ time. With $t'(i) \leq t(i)$, we have that any actor firing i can only take less time, which implies that this actor firing can only produce tokens earlier. The firing rules and the number of tokens produced by actor firings are independent of arrival times of tokens. This means that, with a self-timed schedule, any earlier production of tokens can only lead to earlier start times of other actor firings. Earlier start times in turn can again only lead to earlier token productions. □

In the next section, we will apply the fact that functionally deterministic dataflow graphs have monotonic temporal behaviour to introduce sufficient conditions on the dataflow graph such that upper bounds on container arrival times are given by the arrival times of the corresponding tokens.

6. CONSERVATIVE MODEL

In this section we will present sufficient conditions on the relation between the dataflow model and the task graph such that the dataflow model allows us to derive conservative times at which sufficient containers are available for non-blocking code-segments to start. The first condition is that there is the following one-to-one correspondence between containers and tokens.

PROPERTY 1. *For each buffer in the task graph, there are two unique queues in the dataflow graph. Furthermore, for each container in the task graph, there is one token in the dataflow graph.*

Let $a(c)$ be the arrival time of container c, and let $\hat{a}(c)$ be the arrival time of the token that corresponds to container c. In the next definition, consumptions destroy containers and tokens and productions create containers and tokens. Given that Property 1 holds, the following definition says that a dataflow graph is temporally conservative to a task graph if the fact that container arrival times on input buffers are bounded from above by token arrival times on the corresponding queues implies that container arrival times on all buffers are bounded from above by token arrival times on their corresponding queues.

DEFINITION 2. *Given that Property 1 holds for dataflow graph G and task graph T. This dataflow graph G is temporally conservative to T if*

$$(\forall c_i \in C_I \bullet a(c_i) \leq \hat{a}(c_i)) \implies (\forall c \in C \bullet a(c) \leq \hat{a}(c)) \quad (2)$$

where C_I is the set of containers that are either initially present or arrive on input buffers of T and C is the set of all containers.

By constructing the same dataflow graph as in Section 4, we can derive a requirement in terms of non-blocking code-segments and actor firings that is more straightforward to verify than the more implicit requirement given by Equation (2). The required one-to-one correspondence between actor firings and non-blocking code-segments is more precisely described in the following property.

PROPERTY 2. *Property 1 holds. Furthermore, for non-blocking code-segment m there is a unique firing rule in the set of firing rules that is only satisfied if (1) the tokens that correspond with the containers consumed by m are present, and (2) the token that signals the finish of the previous actor firing is present. Furthermore, satisfaction of the firing rule that corresponds with non-blocking code-segment m enables an actor firing that computes the same function as non-blocking code-segment m except that tokens are produced in an atomic action on queues instead of containers being produced on buffers.*

The required one-to-one correspondence between tasks and dataflow processes is specified in Property 3.

PROPERTY 3. *For each task u in the task graph there is a unique dataflow process v in the dataflow graph, such that for each non-blocking code-segment of u, the dataflow process v has a firing rule such that Property 2 holds.*

Let $e(m,i)$ be the time at which execution i of non-blocking code-segment m is externally enabled, which means that sufficient containers are available on all adjacent buffers. Let $\hat{e}(m,i)$ be the external enabling time of the corresponding actor firing, which is the earliest time at which the tokens are present that correspond with the required containers of modelled non-blocking code-segment m. This means that the external enabling time is independent of the presence of the token signalling the finish of the previous actor firing. Furthermore, let $f(m,i)$ be the finish time of execution i of non-blocking code-segment m and let $\hat{f}(m,i)$ be the finish time of the corresponding actor firing.

The following theorem provides a more straightforward check on the dataflow graph, then the requirement specified in Definition 2. This is because this check is on external enabling and finish times of actor firings instead of token arrival times.

THEOREM 2. *Given that Property 3 holds for task graph T and a dataflow graph G. If Equation (3) holds for any execution i of any non-blocking code-segment m, then G is temporally conservative to T.*

$$e(m,i) \leq \hat{e}(m,i) \Rightarrow f(m,i) \leq \hat{f}(m,i) \quad (3)$$

PROOF. Dataflow graph G is temporally conservative to task graph T, if given a starting situation in which all token arrival times are conservative no actor firing can lead to token arrival times that are not conservative. In G, actor firings consume and produce the same amount of tokens as their corresponding non-blocking code-segments consume and produce containers. Furthermore, we have that non-blocking code-segments consume containers not before their start and produce containers not after their finish, while actor firings consume tokens at their start and produce tokens at their finish. This implies that if Equation (3) holds, then no actor firing produces its tokens earlier than the corresponding non-blocking code-segment produces its containers. This implies that given token arrival times that are conservative every actor firing leads to token arrival times that are again conservative, which implies that G is temporally conservative to T. □

We can now create a performance evaluation set-up for the case that the task graph executes on resources without resource sharing. We first determine the execution time of each non-blocking code-segment, which is the time required by this non-blocking code-segment when run in isolation. Subsequently, we execute the task graph in a discrete time simulation environment, by letting each non-blocking code-segment consume all containers at the start, wait until the simulation time has advanced by the execution time of this non-blocking code-segment and produce all containers. In this way, we basically create the corresponding dataflow graph while executing the task graph. Theorem 2 tells us that the arrival times observed in this simulation are conservative. We have now obtained conservative arrival times for the case that the task graph executes on resources without run-time scheduling. In the next sections, we will show that for a class of schedulers this simulation set-up can be extended to obtain conservative arrival times for the case that the task graph does execute on resources with run-time scheduling.

7. SCHEDULERS

A non-blocking code-segment can start if sufficient containers are present on the input and output buffers of this task. The difference between the time at which sufficient containers are available and the time at which the non-blocking code-segment finishes is determined by the execution time of the non-blocking code-segment and the interference caused by other tasks sharing the same resource.

We see three classes of run-time scheduling that differ in the type of information used to bound the interference from other tasks. Interference can be bounded by knowing (i) how often other tasks are started, and (ii) what execution time is associated with these starts. Often (i) is characterised by a period or a minimum inter-arrival time, while (ii) can be characterised by e.g. a worst-case execution time. This leads to the following three classes of run-time scheduling

1. interference depends on (i) and (ii)
2. interference is independent of (i) but depends on (ii)
3. interference is independent of (i) and (ii)

The first class includes non-starvation free scheduling schemes, such as static priority pre-emptive. For any task, the interference caused by tasks with higher priorities in a certain time interval can only be bounded, if the number of activations of higher priority tasks together with their execution times are known within that interval. The second class, which is a subclass of the first, encompasses the starvation-free scheduling schemes for which a latency-rate characterisation [25] can be derived. This class of schedulers for instance includes round-robin, where the interference of other tasks is independent of the start frequency of other tasks, but does depend on the execution time of other tasks. We call the third class of scheduling schemes the class of budget schedulers as defined in Definition 3. This third class of schedulers is a subclass of the second class.

DEFINITION 3. *A budget scheduler guarantees every task a minimum amount of time B in every interval of time with length P.*

We call B the budget in time interval P. Budget schedulers are the subclass of a-periodic servers [5] that satisfy Definition 3. The class of budget schedulers includes, but is by far not limited to, time-division multiplex, priority-based budget scheduling [24], polling server [22] and constant bandwidth server [2]. This class excludes the total bandwidth server [23], because it cannot provide a budget to tasks that is guaranteed independent of the execution times.

For budget schedulers, the budget guaranteed in a specific interval of time is independent of the execution times of this task. In the next section we will model the effect of budget and time interval selection on the responsiveness of a task. In Section 10, we will investigate trade-offs that can be made in processor and buffer utilisation by various selections of budgets and intervals on the one hand and buffer capacities on the other hand.

8. MODELLING BUDGET SCHEDULERS

In this section, we show that budget schedulers allow for an upper bound on the finish times of task executions. This bound requires that budget B and interval P are known, and, furthermore, a conservative enabling time of this execution, a conservative finish time of the previous execution, and the execution time are known. If, instead of the execution time, an upper bound on the execution time is known, then this upper bound on the execution time can be used to compute a conservative finish time.

This section contains the main technical contribution of this paper, and generalises [30] by providing a proof of the presented upper bound on the finish times that is independent of concepts from the Latency-Rate model [25]. This independence allows to use a sequence of execution times instead of a single (worst-case) execution time. Using a sequence of execution times instead of a single execution time allows for a more accurate analysis.

In this section, we focus on a task u, which is further omitted from the discussion for reasons of clarity. Let $e(i)$ be the external enabling time of execution i, i.e. the time at which execution i is enabled by sufficient containers on all adjacent buffers, let $f(i)$ be the finish time of execution i, while $x(i)$ is the execution time of execution i. Let $i-1$ denote the previous execution. The execution time $x(i)$ is defined as the time interval $f(i) - f(i-1)$, when the task is executed in isolation on this resource and $e(i) \leq f(i-1)$, i.e. the task is executed without interruption. We will show that Equation (5) holds for all schedulers in the just defined class. This equation specifies an upper bound on the finish time that holds for all schedulers in this class, a tighter upper bound can be found for specific schedulers.

DEFINITION 4. *Execution i of task u is part of a consecutive execution that starts with execution k of u if for all executions j of u, with $k < j \leq i$, we have that $e(j) \leq f(j-1)$.*

THEOREM 3. *Given that execution i is part of a consecutive execution that starts with execution k. Then for every scheduler that guarantees a task a minimum amount of time B in every interval of time P, an upper bound on the finish time of execution i is given by*

$$f(i) \leq f^w(i) = e(k) + \sum_{j=k}^{i} x(j) + (P-B) \left\lceil \frac{\sum_{j=k}^{i} x(j)}{B} \right\rceil \quad (4)$$

PROOF. If $e(k) > f(k-1)$, then at time $e(k)$ execution k has sufficient containers present on all adjacent buffers and execution $k-1$ has finished, which implies that from $e(k)$ execution k can start its execution. The worst-case finish time of execution k occurs if previous executions have already depleted the allocated time budget. In this case, execution k needs to wait for maximally $P-B$ time before it can start its execution. This results in a worst-case finish time of $g(k) = e(k) + x(k) + (P-B)\lceil x(k)/B \rceil$. If for each execution j, with $k < j \leq i$, we have that $e(j) \leq f(j-1)$, then we can see executions k through i as a single execution. In this case, we have that an upper bound on the finish time of execution i is given by $f^w(i)$ as defined in Equation (4). $\square$

Bound f^w is a tight bound. This is because bound f^w equals the actual finish time f if $e(k)$ occurs just after the budget B is depleted, where k is the first execution of a consecutive execution, and the budget is always replenished with a quantum that equals B.

Bound f^w can be applied if the starts of consecutive executions can be determined. However, the condition that determines when a consecutive execution starts depends on the actual enabling and finish times, while bound f^w computes upper bounds on finish times and container arrival times. Determining the starts of consecutive executions based on the application of bound f^w is therefore problematic. The following theorem presents a bound that is an upper bound on f^w that does not depend on the knowledge of consecutive executions.

THEOREM 4. *For every scheduler that guarantees a task a minimum amount of time B in every interval of time P, we have that an upper bound on the finish time of execution i is given by*

$$f(i) \leq f^w(i) \leq \max(e(i) + P - B, f^w(i-1)) + \frac{P \cdot x(i)}{B} \quad (5)$$

PROOF. We conservatively bound Equation (4).

$$f^w(j) \leq f^l(j) = e(k) + P - B + \frac{P \sum_{i=k}^{j} x(i)}{B} \qquad (6)$$

For any execution i, we can have two cases. Either $e(i) > f(i-1)$ and i is the first execution in a sequence of consecutive executions, or $e(i) \leq f(i-1)$. In case $e(i) > f(i-1)$, then $f^l(i) = e(i) + P - B + {}^{P \cdot x(i)}/_{B}$. In case $e(i) \leq f(i-1)$, then we have that $f^l(i) - f^l(i-1) = {}^{P \cdot x(i)}/_{B}$. Therefore for any i we have that Equation (5) holds. □

The upper bound on finish times as given by Equation (5) is still a tight bound. This is because $\lceil x \rceil$ was bounded by $x+1$. Depending on the value of x this is a tight bound.

We can now derive upper bounds on container arrival times for the case that the task graph executes on resources with resource sharing. We again first determine the execution time of each non-blocking code-segment. Subsequently, we execute the task graph in a discrete-event simulation environment, where we now use Equation (7) to determine the finish time of the actor firing.

$$\hat{f}(i) = \max(\hat{e}(i) + P - B, \hat{f}(i-1)) + \frac{P \cdot x(i)}{B} \qquad (7)$$

It is clear that Equation (3) is satisfied with these finish times. We have that Property 3 holds for the dataflow graph constructed during the simulation. This implies that conservative container arrival times are given by the arrival times of the corresponding tokens in the simulation. While this model associates an execution time with every task execution, the model from [30] only allows to associate a single (worst-case) execution time with a task, i.e. has an execution time that is constant over all task executions. This is an important extension that enables the presented simulation approach.

If the finish time of firing i is computed with Equation (7), then it is clear that an earlier external enabling time of firing i does not lead to a later finish time of firing i. Since the bounds do not affect the behaviour of any other actor firing, we have that the resulting dataflow model has monotonic temporal behaviour. This result is quite peculiar, since it is well known that schedulers can have non-monotonic behaviour, i.e. scheduling anomalies [10]. Also a budget scheduler can have non-monotonic behaviour, i.e. an earlier enabling of one task can lead to a later start of another task. However, in our model we have taken into account the latest time at which every task obtains its budget. In this way, we have bounded the non-monotonic effects of the scheduler. Furthermore, because the dataflow graph has a self-timed schedule, the scheduling of the dataflow graph is no longer a multiprocessor scheduling problem.

For the case without resource sharing, conservatism was introduced by letting consumptions be at the start and productions be at the finish. In case of resource sharing, we have that the arrival times are conservative for all initial states of the budget scheduler, e.g. independent of the current position in the time-division multiplex period when we start this application. This is because in case of a multiprocessor system in which the processors each have their individual clock, which are not synchronised with each other, then inherent – unknown – variation in the clocks leads to inherent – unknown – variation in the alignment of the time-division multiplex schedules, i.e. the alignment of the time-division multiplex schedules varies over time. In case the applied budget scheduler is a time-division multiplex scheduler, then with our dataflow model, we compute the worst-case arrival times for every possible alignment of the time-division multiplex schedules.

9. SCHEDULING OVERHEAD

We defined budget schedulers as the class of schedulers that can guarantee a minimum budget B in every interval of length P. However, if the scheduling overhead is difficult to bound, then it is also difficult to guarantee such a budget. If the notification of the arrival of containers is based on interrupts, then guaranteeing a budget is far from trivial. This is because the maximum number of interrupts that can occur in any interval depends on the best-case behaviour of the task graph. The best-case behaviour of the task graph depends on the best-case response times, which in turn depend on the best-case execution times. Since determining best-case execution times is, just like determining worst-case execution times, a difficult problem [16], we do not use interrupts to notify the arrival of containers.

In case of time-division multiple access (TDMA) scheduling, as applied in the experiments discussed in Section 10, we apply the inter-task synchronisation scheme as presented in [19]. In this scheme, the task that produces a container updates the buffer administration. A task that is waiting for a container to arrive polls the buffer administration for the availability of the container. Since all tasks are guaranteed a minimum budget this does not negatively affect the responsiveness of any of the other tasks. This set-up has the advantage that it is easier to bound the scheduling overhead, because accounting for the cost of polling is easier than accounting for the cost of interrupt processing.

The work presented in [24] shows that there is a sub-class of budget schedulers for which the polling-based inter-task synchronisation scheme from [19] is applicable.

10. EXPERIMENTAL RESULTS

In this section, we first discuss three experiments that are set-up in such a way that the results can be intuitively understood. A subsequent fourth experiment includes more complex real-life software. In the first experiment, we have an application with as little as possible variation, i.e. jitter, in its temporal behaviour. In this experiment, the execution times and the number of communicated containers is kept constant over task executions and the run-time scheduling can be seen as the only source of variation, i.e. jitter. In the second experiment, we introduce variation by letting the execution time of one task alternate between two values. In the third experiment, we introduce another source of variation, which is a task that communicates a variable number of containers. Communication of a variable number of containers mimics behaviour as for instance found in video decoders, where a to be decoded frame contains a variable number of blocks. The fourth experiment is an MP3 playback application that includes real-life software, more tasks, a periodic sink, and tasks with data-dependent consumption behaviour.

We have implemented a small scheduling kernel that implements time-division multiple access (TDMA) scheduling. With TDMA scheduling there is a fixed sequence in which tasks are allocated their time slices within a period. This implies that the interval of time over which tasks are guaranteed a minimum budget is the same for all tasks and equals the TDMA period. A typical sequence of events is shown in Figure 2. At time t_1, the scheduler sets the timer to the size of slice S_a of the next task, i.e. task a. From time t_1 to time t_2 the scheduler restores the context of task a after which task a can continue execution from time t_2. At time t_3, the processor receives an interrupt that signals that the time slice S_a is completed. From time t_3 to time t'_1 the scheduler stores the context of task a, after which at time t'_1 the timer is set with the slice S_b of task b.

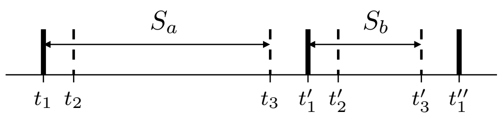

Figure 2: Typical sequence of actions in TDMA scheduler.

This sequence is repeated for all slices in a period and is the same in every period.

In all experiments, we consider an architecture with 2 ARM7 processors [1] that are directly connected to a double-ported memory. All instructions and (shared) data are in this memory. The advantage of this reduced set-up is that the limited number of sources of variation allows for a clean discussion of the observed differences between the simulations at different levels of abstraction. At the cost of a more elaborate model, the effects on the temporal behaviour of the application as for instance caused by resource sharing in the memory hierarchy can be included in the dataflow model [13]. On this architecture, we have observed an upper bound on $t_2 - t_1$ of 98 cycles and an upper bound on $t'_1 - t_3$ of 249 cycles. This implies that with n time slices the TDMA period equals the sum of the slices plus n times 249 cycles, while the budget allocated to a task equals the slice size minus 98 cycles. This is really a guaranteed budget since apart from the timer interrupts no other interrupts are received by this processor.

10.1 Experiment 1

In this first experiment we have a task graph consisting of a data producing task and a data consuming task. The data producing task produces one container in every execution and the data consuming task consumes one container in every execution. Both tasks iterate through a loop in which they first block on the arrival of a container, then do some processing, subsequently copy the result of this processing in the container, and at the end of the iteration release the container. The tasks wait on a container by polling the buffer administration. As soon as the consuming task releases a container, i.e. finishes an execution, we trigger a monitor in the simulator that prints the current time. The execution time of an execution of a task is the time between successive finishes in case the polls always succeed and this task is the only task on the processor, i.e. no TDMA scheduling overhead is included. We have constructed this experiment such that we have a minimal variation in the execution times of these tasks, an upper bound on the execution time of the data producing task is 360794 cycles and an upper bound on the execution time of the data consuming task is 360796 cycles. These two tasks have a time slice of 2 Mcycles and execute on different processors. On both processors, there is one additional task that also has a time slice of 2 Mcycles.

In Figure 3, the first 20 finish times of the data consuming task as observed in our cycle-true simulator are shown for a buffer capacity of 3 containers. The simulation results named 'tdm-1' have been obtained by placing the data producing task as the second task that is allocated its slice by its TDMA scheduler, and placing the data consuming task as the first task that is allocated its slice by its TDMA scheduler. The results named 'tdm-2' have been obtained by placing the data producing task as the first task that is allocated its slice by its TDMA scheduler, and placing the data consuming task as the second task that is allocated its slice by its TDMA scheduler. The results named 'tdm-3' have been obtained by placing the data producing task as the first task that is allocated its slice by its TDMA scheduler, and placing the data consuming task as the first

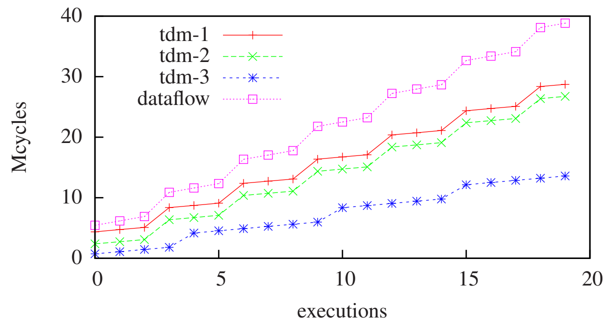

Figure 3: Finish times of consumer for buffer capacity of 3.

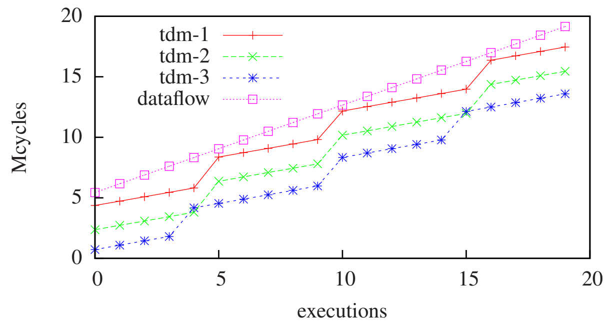

Figure 4: Finish times of consumer for buffer capacity of 8.

task that is allocated its slice by its TDMA scheduler. The results named 'dataflow' have been obtained in our dataflow simulator that together with execution of the tasks, evaluates Equation (7) in order to determine the finish time of the corresponding actor firing, which equals the token production time.

The bound on finish times and the actual finish times diverge. The reason is that with these time slices and this buffer capacity the throughput of this task graph is limited by the buffer capacity, the tasks can execute at least five times in their slice, while the buffer has a capacity of three. The consequence is that the conservative production times by the data consuming task in our dataflow simulator lead to conservative start times of the data producing task which again lead to conservative production times of the data producing task, etc. In short, since the buffer capacity determines the throughput an over-estimation of the task finish times results in a lower throughput estimate in our dataflow simulations. Furthermore, the finish times of the different TDMA configurations diverge. This is because, in situation 'tdm-3', the slices of the two tasks occur at the same time, allowing for more than 3 executions per slice.

In Figure 4, the first 20 finish times of the data consuming task are shown for the same set-up except that now the buffer has a capacity of 8 containers. In this case the rate of the start times in the dataflow simulations closely follows the actual rate. Note that typically a strictly periodically executing sink or source task determines the throughput of stream processing applications, which implies that typically the situation depicted in Figure 4 occurs.

When comparing the results for these two buffer capacities it becomes clear that the application of polling instead of interrupts does not necessarily lead to low processor utilisation. While the data

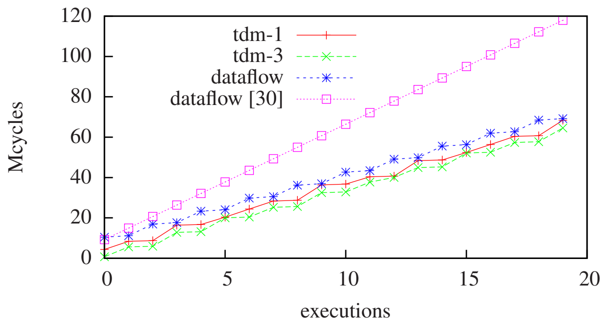

Figure 5: Finish times for cyclo-static execution times.

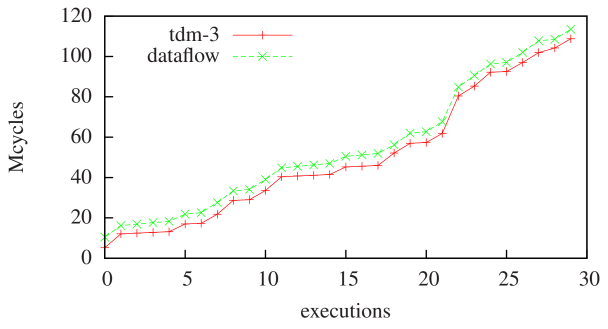

Figure 6: Finish times for variable production quanta.

consuming task only used 50% of its budget for the trace shown in Figure 3, where we had a buffer capacity of three, virtually the complete budget was used by the data consuming task for the trace shown in Figure 4, where we increased the buffer capacity to eight. Since the budgets of both tasks are equal, the budget not used by the data consuming task, in case of a buffer capacity of eight, is due to the difference in execution time with the data producing task. Alternatively, but not shown, we could have decreased the time slices instead of increasing the buffer capacity to increase the processor utilisation. This means that a suitable selection of buffer capacities and scheduler settings can result in an external enabling time of the next execution of a task that is before the finish time of the current execution. In this situation, only a single polling action per task execution is required and a high processor utilisation can be obtained.

10.2 Experiment 2

In this experiment, we introduced variation in the execution time of the data producing task. In an alternating fashion, subsequent executions of the data producing task have an upper bound on their execution time of 2860779 and 360803 cycles. The buffer capacity in this experiment is eight containers. In Figure 5, the first 20 finish times of the data consuming task are plotted as observed in our cycle-true and dataflow simulation environments. In the dataflow simulator, we have used the just described sequence of worst-case execution times when computing the finish times denoted by 'dataflow' with Equation (7). The finish times denoted in Figure 5 that are denoted by 'dataflow [30]' are computed using the model from [30] that can only include a single (worst-case) execution time. Basically, the resulting dataflow simulator computes finish times of a cyclo-static dataflow model [4]. It is known that the self-timed execution of a cyclo-static dataflow model results in a periodic regime as again confirmed by the results from our dataflow simulation. These finish times could have also been computed in an analytic fashion [30, 29, 26].

10.3 Experiment 3

In this third experiment, we again changed the data producing task. Instead of producing one container in every execution, in this experiment the data producing task produced between zero and five containers in every execution. We increased the execution time of the data producing task to have an upper bound of 2861527 cycles, while still having very little variation. Dataflow analysis [31] told us that the chosen buffer capacity of eight containers was sufficient for deadlock-free execution.

In Figure 6, the first 30 finish times of the data consuming task are shown. The simulation results named 'tdm-3' are for the case that the slices of both tasks occur at the same time. Again upper bounds on the finish times of the data consuming task are computed in our dataflow simulator using Equation (7) and shown in this figure under the name 'dataflow'. The finish times for the cases 'tdm-1' and 'tdm-2' are not shown but these are upper bounded by the simulation results named 'dataflow'.

10.4 Experiment 4

In this fourth experiment, we add a number of interesting, more realistic, aspects. We use real-life software, increase the number of tasks, have a periodic sink, and have a task with data-dependent consumption behaviour. This experiment shows that our analysis can provide accurate conservative bounds on the finish times of tasks in case of a strictly periodically executing sink and aperiodic start times and finish times of tasks. This aperiodicity results from variation in execution times, scheduler state, and data-dependent inter-task communication behaviour. To prevent buffer overflow in this task graph with aperiodic starts and finishes, tasks first wait on sufficient empty buffer space before they write their data. This flow-control mechanism results in so-called back-pressure from the periodic sink, which is correctly taken into account in our analysis.

The task graph includes a file reader, an MP3 decoder and a digital-to-analog converter. The file reader reads the input data from a file and produces 2048 bytes of data in every execution. The MP3 decoder is the MAD [28] MP3 decoder. In this experiment, we used a 48kHz variable-bitrate mono MP3 file. For this input stream, the decoder produces 1152 samples in every execution. The MAD decoder has an internal input buffer of 3000 bytes, and replenishes this internal buffer if there are not enough remaining bytes in this internal buffer to decode the next frame. The number of bytes required to replenish this buffer varies from replenishment to replenishment. Furthermore, the number of replenishments relative to the number of times 1152 output samples are produced is data-dependent and varies while processing the input stream.

The file reader has an execution time of 33247 cycles for its first execution and a constant execution time of 2058 cycles for the subsequent executions. The execution times of the first three executions of the MP3 decoder are around 10^6 cycles, while subsequent executions have an execution time of $1.5 \cdot 10^6$ cycles with a variation of 5%. We bound the file reader and MP3 decoder to different SWARM processors on which they both share the processor with another task. On both processors these other tasks are allocated a slice of 500000 cycles, while the file reader is allocated a slice of 50000 cycles and the MP3 decoder is allocated a slice of 500000 cycles. This results in a budget of 49902 cycles in an interval of

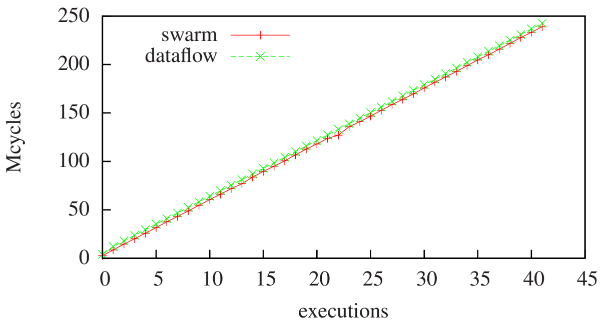

Figure 7: Finish times of MP3 decoder in cycle-true and dataflow simulator. Plot emphasises that throughput is accurately modelled. Dataflow simulator is conservative with a maximum difference of $6 \cdot 10^6$ **cycles.**

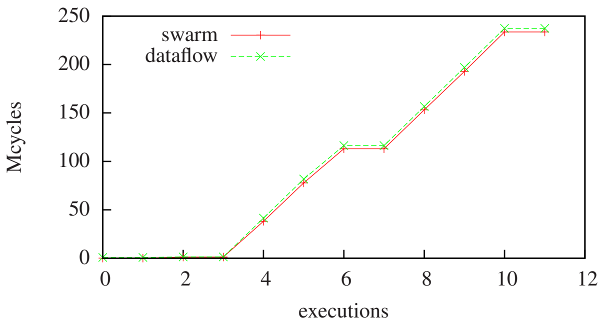

Figure 8: Finish times of file reader in cycle-true and dataflow simulator. Plot emphasises that throughput is accurately modelled. Dataflow simulator is conservative with a maximum difference of $4 \cdot 10^6$ **cycles.**

550498 cycles for the file reader and a budget of 499902 cycles in an interval of 1000498 cycles for the MP3 decoder. We select a buffer capacity of 5500 bytes for the buffer from the file reader to the MP3 decoder and a buffer capacity of 2000 samples for the buffer from the MP3 decoder to the digital-to-analog converter. The digital-to-analog converter samples at 48 kHz, which in this experiment corresponds to 5000 cycles.

In Figure 7, the first 42 finish times of the MP3 decoder are shown. The MP3 decoder produces 1152 samples in every execution and the digital-to-analog converter consumes a single sample in every execution. Because we selected settings such that data arrives in time at the digital-to-analog converter, the sampling frequency of the digital-to-analog converter determines the distance between subsequent enablings of the MP3 decoder. The variation in execution times and the variation in the state of the TDMA scheduler at the enabling time result in aperiodic start and finish times of the MP3 decoder.

For our input stream, we have that 42 executions of the MP3 decoder allow 12 executions of the file reader. The first 12 finish times of the file reader are shown in Figure 8. Even though the variation in execution times and scheduler state result in aperiodic start and finish times of the MP3 decoder, the dominant factor causing these aperiodic finish times of the file reader is that the MP3 decoder has a consumption behaviour that depends on the processed data-stream and varies while processing the stream.

The third and fourth experiments show that the combination of budget scheduler and polling can immediately handle tasks that have aperiodic activations. Furthermore, we are still able to give accurate conservative bounds on the finish times of a task that executes aperiodically. Note that the bounds provided in this paper are only valid for the given input stream or for input streams that result in both (1) the same inter-task communication behaviour and (2) smaller than or equal execution times. The other streams are allowed to result in smaller than or equal execution times, because of the monotonic temporal behaviour of the dataflow model and the required one-to-one correspondence between task graph and dataflow model.

In these experiments, the dataflow simulator finished in a fraction of a second, while the cycle-accurate model required a couple of minutes to finish. A run of a thousand executions of the tasks of experiment 1 with a buffer capacity of eight required tens of milliseconds in our dataflow simulator and about an hour on the cycle-accurate model. However, note that we first need to determine execution times in a cycle-accurate simulator before we can use the dataflow simulator to explore various buffer capacities and scheduler settings.

11. DISCUSSION

We constrained applications to be task graphs as defined in Section 4, and required schedulers that guarantee a minimum budget B every interval of time P. These constraints lead to the fact that the functional behaviour of our applications is time-invariant, i.e. schedule independent, and that the bounds on temporal behaviour we obtain from our analysis are independent of the container arrival times and execution times of other applications. Approaches like [11, 14, 6] have weaker restrictions on the application model and support a larger class of schedulers. However, these approaches both have a more difficult analysis problem and stronger conditions under which the analysis results hold. For instance, cyclic resource dependencies [6] can occur, for which the analysis requires that worst-case execution times and typically also best-case execution times of all tasks, i.e. including tasks of other applications, need to be known.

These alternative approaches advocate the use of traffic shapers in order to obtain tight bounds on end-to-end behaviour [12, 17]. However, the presented approach, which is completely data-driven, obtains higher throughput and lower latency, while still being amenable to performance analysis. We see as the main reasons that budget schedulers by construction bound the interference from other tasks and that blocking inter-task communication by construction bounds the jitter in the application.

12. CONCLUSION

In this work, we have defined constraints on the implementation of applications such that the implementation is functionally deterministic. This implies that the functional behaviour is time-invariant. Subsequently, we defined the class of budget schedulers and presented an upper bound on the finish times of task executions for schedulers from this class. A budget scheduler guarantees every task a minimum time budget in an interval of time. We showed that dataflow graphs can be annotated with time and that the self-timed execution of a dataflow graph has monotonic temporal behaviour, which implies that an earlier finish time cannot lead to a later start time in the dataflow graph. This lead to the conclusion that upper bounds on the finish times of task executions can be observed in a simulation of the corresponding dataflow graph, given that the implementation is functionally deterministic and only budget schedulers are applied.

We showed that simulation of this dataflow graph which is at the communicating processes plus time abstraction level results in tight bounds on the behaviour observed in a cycle-accurate simulator, while at the same time resulting in significantly reduced run-times of the simulations. Therefore, placing constraints on the implementations of applications and on the applied schedulers enables efficient exploration of various system settings such as scheduler configurations and buffer capacities.

13. REFERENCES

[1] SoftWare ARM. www.cl.cam.ac.uk/ mwd24/phd/swarm.html.

[2] L. Abeni and G. Butazzo. Resource Reservation in Dynamic Real-Time Systems. *Real-Time Systems*, (27):123–167, 2004.

[3] M. J. G. Bekooij, S. Parmar, and J. van Meerbergen. Performance Guarantees by Simulation of Process Networks. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, September 2005.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[5] G. C. Butazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[6] R. L. Cruz. A Calculus for Network Delay, Part II: Network Analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.

[7] E. A. de Kock, W. Smits, P. van der Wolf, J. Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. Vissers, and G. Essink. YAPI: Application Modeling for Signal Processing Systems. In *Proc. DAC*, 2000.

[8] A. Donlin. Transaction level modeling: Flows and use models. In *Proc. ACM/IEEE Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004.

[9] A. Gerstlauer, H. Yu, and D. D. Gajski. RTOS Modeling for System Level Design. In *Proc. Design, Automation and Test in Europe (DATE)*, 2003.

[10] R. L. Graham. *Computer and Job Scheduling Theory*, chapter Bounds on the Performance of Scheduling Alogrithms. John Wiley & Sons, 1976.

[11] W. Haid and L. Thiele. Complex Task Activation Schemes in System Level Performance Analysis. In *Proc. ACM/IEEE Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.

[12] A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design Space Exploration and System Optimization with SymTA/S - Symbolic Timing Analysis for Systems. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2004.

[13] A. Hansson, M. H. Wiggers, A. J. M. Moonen, K. G. W. Goossens, and M. J. G. Bekooij. Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis. *IET Computers & Digital Techniques*, 2009. to appear.

[14] M. Jersak, K. Richter, and R. Ernst. Performance Analysis of Complex Embedded Systems. *Int'l Journal of Embedded Systems*, 1(1-2):33–49, 2005.

[15] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[16] Y.-T. Li and S. Malik. *Performance analysis of real-time embedded software*. Kluwer academic publishers, 1999.

[17] Z. Lu, M. Millberg, A. Jantsch, A. Bruce, P. van der Wolf, and T. Henriksson. Flow Regulation for On-Chip Communication. In *Proc. Design, Automation and Test in Europe (DATE)*, 2009.

[18] A. K. Mok and D. Chen. A Multiframe Model for Real-Time Tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, October 1997.

[19] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busa, K. Goossens, and R. Peset Llopis. C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.

[20] S. Park, W. Olds, K. G. Shin, and S. Wang. Integrating Virtual Execution Platform for Accurate Analysis in Distributed Real-Time Control System Development. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2007.

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

[22] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1(1):27–60, 1989.

[23] M. Spuri and G. C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems*, 10(2), 1996.

[24] M. Steine, M. J. G. Bekooij, and M. H. Wiggers. A Priority-Based Budget Scheduler with Conservative Dataflow Model. In *Proc. Euromicro Conference on Digital System Design (DSD)*, 2009.

[25] D. Stiliadis and A. Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, October 1998.

[26] S. Stuijk, M. Geilen, and T. Basten. Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs. In *Proc. Design Automation Conference (DAC)*, 2007.

[27] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale. Implementing Synchronous Models on Loosely Time Triggered Architectures. *IEEE Transactions on Computers*, 2008.

[28] Underbit Technologies. Mad: Mpeg Audio Decoder. http://www.underbit.com/products/mad/, 2009.

[29] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Proc. Design Automation Conference (DAC)*, June 2007.

[30] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Modelling Run-Time Arbitration by Latency-Rate Servers in Dataflow Graphs. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2007.

[31] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Buffer Capacity Computation for Throughput Constrained Streaming Applications with Data-Dependent Inter-Task Communication. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.