Scheduling Multiple Independent Hard-Real-Time Jobs on a Heterogeneous Multiprocessor

Orlando Moreira
NXP Semiconductors
Research
Eindhoven, Netherlands
orlando.moreira@nxp.com

Frederico Valente
Universidade de Aveiro
Aveiro, Portugal
fmvalente@ua.pt

Marco Bekooij
NXP Semiconductors
Research
Eindhoven, Netherlands
marco.bekooij@nxp.com

ABSTRACT

This paper proposes a scheduling strategy and an automatic scheduling flow that enable the simultaneous execution of multiple hard-real-time dataflow jobs. Each job has its own execution rate and starts and stops independently from other jobs, at instants unknown at compile-time, on a multiprocessor system-on-chip. We show how a combination of Time-Division Multiplex (TDM) and static-order scheduling can be modeled as additional nodes and edges on top of the dataflow representation of the job using Single-Rate Dataflow semantics to enable tight worst-case temporal analysis. We also propose algorithms to find combined TDM/static order schedules for jobs that guarantee a requested minimum throughput and maximum latency, while minimizing the usage of processing resources. We illustrate the usage of these techniques for a combination of Wireless LAN and TD-SCDMA radio jobs running on a prototype Software-Defined Radio platform.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems

General Terms

Algorithms, Design, Theory

Keywords

Real-Time, Multi-processor, Dataflow, Scheduling

1. PROBLEM STATEMENT

In order to deliver high quality output, streaming media applications have tight real-time requirements, which typically defined in terms of minimum guaranteed throughput and maximum latency.

EMSOFT'07 September 30–October 3, 2007, Salzburg, Austria

Embedded platforms for the streaming media domain are expected to handle several streams at the same time, each with its own rate. Typically, functionality can be divided in minimal groups of communicating tasks that are started and stopped independently by an external source. We refer to such groups of tasks as jobs. The number of use-cases (i.e. combinations of simultaneously executing job instances that the device must support) can be high.

This domain includes Software-Defined Radio [3], where an embedded multiprocessor system is used to do baseband processing of several radio standards. Typically, several radio baseband processing jobs may be active at the same time, and they may start/stop execution at different times, according to the demands of the user.

We assume that the radio jobs are implemented as dataflow graphs, be it Single-Rate, Multi-Rate or Cyclo-static dataflow [4]. Multi-Rate Dataflow was first introduced in [13] under the name of Synchronous Dataflow. All of these have comparable expressive power, and all of them can be analyzed for real-time behavior by using the same techniques, since it has been shown that both Multi-Rate and Cyclo-Static Dataflow can easily be converted to Single-Rate Dataflow [19], by using a pseudo-polynomial algorithm.

We assume a Multiprocessor Systems-On-Chip (MPSoC) hardware platform, since it can provide a good balance between cost, power efficiency and flexibility. Systems of this type are typically heterogeneous, as the usage of application-specific coprocessors can dramatically improve performance at low area cost.

On an MPSoC, jobs share computation, storage, and communication resources, in order to allow maximum flexibility at the lowest cost. This poses a particularly difficult problem for the scheduling of real-time applications: resource sharing leads to uncertainty about resource provision and, therefore, to difficulties in computing minimum throughput and maximum latencies.

In this paper, we assume an MPSoC designed to facilitate worst-case analysis by following the rules presented in [1]. These rules are not necessary but sufficient rules, that is, while predictable systems can be designed without following them, compliance guarantees that the resulting platform is amenable to temporal analysis techniques. According to these rules, every processor has its own local dedicated memory and caches are not used. Communication between processors is strictly done by posted writes. The inter-processor communication infrastructure provides guaranteed throughput connections. We also assume a heterogeneous system,

and that all the processing cores can support Time Division Multiplex (TDM) scheduling. In this paper, we assume a fully heterogeneous multiprocessor, where each task has a single implementation and can only run in one of the processors. This assumption fits the prototype SDR baseband chip which we use in our example section. The approach can be rather trivially extended to deal with multiple homogeneous processing elements by allowing the computed schedule to be allocated at configuration time to one of many identical processor elements, by using an online resource allocator such as the one proposed in [17].

The problem we are addressing in this paper is finding a scheduling strategy and schedules that allow a heterogeneous MPSoC to handle a dynamic mix of hard-real-time jobs. As the key contributions of this paper we show that for solving this problem a combination of run-time and static order scheduling is desirable and we present techniques for the computation of TDM scheduler settings and static-order of actors per job per processor, in such a way that timing requirements are met.

The rest of the paper is organized as follows. In section 2, we provide some information on our dataflow modeling and real-time analysis techniques. In section 3, we motivate and describe our choice of scheduling strategy. In section 4 we show our approach to the scheduling problem that arises from our chosen scheduling strategy. In section 5 we focus on the sub-problem of finding the TDM slices times. In section 6 we discuss phase coupling issues. In section 7 we show an example from a Software-Defined Radio application, and present results. Section 8 compares our approach and contributions to the state-of-the-art. Section 9 concludes the paper and discusses future work.

2. SINGLE RATE DATAFLOW GRAPHS

In this section we introduce the Single-Rate Dataflow model, its timing analysis properties and its usage in modeling streaming job implementations, with timing and scheduling information included. This is necessary since constraint-checking based on this model enables us to compute schedules that meet the real-time requirements of the jobs. This section includes mostly reference material that can be found elsewhere [13], [19], [16], except for the section on the modeling of combined TDM and static-order scheduling which improves on modeling techniques described in [2] and [16].

2.1 Notation and Relevant Properties

A Single-Rate Dataflow (SRDF) graph is a directed graph $G = (V, E)$. Nodes, referred to as actors, represent time consuming entities; edges represent FIFO queues that direct values from the output of an actor to the input of another. Data is transported in discrete chunks, referred to as **tokens**. When an actor is activated by data availability, it is said to be enabled. In SRDF an actor is enabled when the number of tokens in each of its input edge is one. An enabled actor can be **fired**. An actor fires by consuming a token in each input edge, and producing one token in each output edge. Between the begining and end of the firing, an arbitrary amount of time can elapse. During the execution of a dataflow graph, all the actors may fire an infinite amount of times. Each firing of an actor is also referred to as an iteration of the actor.

We are interested in applications that process data streams, which typically involve computations on indefinitely long data sequences. Therefore, we are only interested in SRDF graphs that can be executed in a non-terminating fashion. Consequently, we must be able to obtain schedules that can run infinitely using a finite amount of physical memory.

Edges have a valuation $d : E \to \mathbb{N}_0$; $d(a_i, a_j)$ is called the delay of edge (a_i, a_j) and represents the number of initial tokens in (a_i, a_j). In the graphical representation of an SRDF graph, the delay is represented by a number of dots on the edge equal to its value.

If an SRDF graph has any cycle where the sum of delays on its edges is 0, then the graph is said to be **deadlocked**. A deadlocked graph eventually gets to a state where it cannot make any progression if implemented within finite memory. A SRDF graph is *deadlock-free* if all its cycles traverse at least one initial token. A deadlock-free SRDF graph can be scheduled periodically using a finite amount of memory [19].

We represent the set of processors in the MPSoC as $\Pi = \{p_1, p_2 ... p_n\}$. In this paper, we assume actors are uniquely assigned to execute on a particular processor. This is represented by the valuation $\pi : V \to \Pi$, where $\pi(a)$ represents the processor to which actor a is assigned. Actors have a valuation $t : V \to \mathbb{N}$, where $t(a)$ is the execution time of actor a. To an SRDF graph extended with π and t valutation we refer as a timed SRDF.

The **cycle mean** of a cycle c in a timed SRDF graph is defined as

$$\mu_c = \frac{\sum_{a_i \in V(c)} t(a_i)}{\sum_{e \in E(c)} d(e)} \tag{1}$$

where $V(c)$ is the set of all nodes traversed by cycle c and $E(c)$ is the set of all edges traversed by cycle c.

The **Maximum Cycle Mean (MCM)** μ of a timed SRDF graph G is defined as:

$$\mu(G) = \max_{c \in C(G)} \frac{\sum_{a_i \in V(c)} t(a_i)}{\sum_{e \in E(c)} d(e)} \tag{2}$$

where $C(G)$ is the set of cycles in graph G.

The inverse of the MCM of a timed SRDF graph provides a fundamental upper bound to its minimum guaranteed throughput [18]. Many algorithms of polynomial complexity have been proposed to find the maximum cycle mean (see [9] for an overview).

In [16], we have shown how a latency constraint can be converted into a throughput constraint for a dataflow model, as long as the best-case temporal behavior of the source of the system is characterized. Because of this, for the rest of this paper we will assume, without loss of generality, that only throughput constraints need to be met.

2.2 Dataflow Model of a Job

The timed SRDF graph that serves as input to the scheduler is a description of the job where every actor corresponds to a data-triggered non-blocking segment of a computational task. The $t(a)$s of actors correspond to the worst-case execution times of these task segments executing on a dedicated processor (i.e., resource sharing is not taken into account). Resource constraints such as the latency and throughput of the communication channels and bounds on FIFO buffer sizes can be modeled by adding more actors and edges to this dataflow graph [14], [16]. In this paper, we will just

omit the constraints on inter-processor communication latency and throughput and buffer sizes that should be be added to the model for conservative throughput analysis, in order to simplify the explanations.

2.2.1 Modeling TDM Scheduling

In [2], it has been shown that the effect of TDM scheduling can be modeled by replacing the worst-case execution time of the actor by its worst-case response time under TDM scheduling. The response time of an actor a_i is the total time it takes to fire a_i , when resource arbitration effects (scheduling, preemption, etc) are taken into account. This is counted from the moment the actor meets its enabling conditions to the moment the firing is completed. Assuming that a TDM wheel period P is implemented on the processor and that a time slice with duration S is allocated for the firing of a_i, such that $S \leq P$, a time interval longer than $t(a_i)$ passes from the moment an actor is enabled by the availability of enough input tokens to the completion of its firing. This is due to what can be seen as two different effects of TDM arbitration. The first of this is the scheduling time, i.e. the time it takes until the TDM scheduler grants execution resources to the actor. In the worst-case, a_i gets enabled when its time slice has just ended, which means that the scheduling time is the time it takes for the slice of a_i to start again. This amounts to $P - S$ time units. The second effect has to do with the fact that the time slice may be too small for the firing of a_i to be executed in a single slice. The time a_i will take to fire, in the worst-case, is equal to $\lceil \frac{t(a_i)}{S} \rceil \cdot P - (t(a_i) \bmod S)$. The total worst-case response time of a_i is then given by the sum of these two values:

$$r(a_i) = (P - S) \cdot \lceil \frac{t(a_i)}{S} \rceil + t(a_i) \quad (3)$$

Furthermore, if the $t(a_i)$s of all actors are replaced by worst-case response times, $r(a_i)$s, which take into account the fact that the actual time from enabling of an actor iteration to finish is affected by TDM arbitration, MCM analysis of the timed SRDF graph thus obtained will yield the minimum guaranteed throughput of such an implementation, assuming all times are conservative and all resource constraints modeled.

2.2.2 Modeling Static-Order Scheduling

A static-order schedule of a set of actors $A = \{a_0, a_1, ..., a_n\}$ mapped to the same processor (i.e. $i, j \in A : \pi(i) = \pi(j)$), is a sequence of execution $so = [a_k, a_l ... a_m]$ that generates extra precedence constraints between the actors in A such that from the start of the execution of the graph, a_k must be the first to execute, followed by a_l and so on, up to a_m. After a_m executes, the sequence is reset, and execution order restarts from a_k for the next iteration of the graph.

Any static order imposed to a group of SRDF actors executing in the same processor can be represented by adding edges with no tokens between them. From the last to the first actor in the static order an edge is also added, with a single initial token. This construct reflects the fact that, the graph execution being iterative, when the static order finishes execution for a given iteration, it re-starts it from the first actor in the static order for the next iteration.

Notice that the new edges represent a series of sequence constraints enforced by the static order schedule and do not represent any real exchange of data between the actors. In the dataflow diagrams that follow, for ease of read, every time there is more than one edge from the same source to sink, only one edge with the lowest $d(i, j)$ is represented, as it imposes the tightest sequence constraint between the actors.

2.2.3 Mixing Static-Order and TDM

Lets assume that instead of attributing a TDM slice to each actor we attribute a TDM slice to a group of actors and we statically order these actors. The reasons why we may want to do this are exposed in section 3. For now we will focus only on how such a mixed schedule can be modeled for the purpose of worst-case analysis.

Consider a set of actors $\{a_1, a_2, a_3\}$, belonging to the same job, and mapped to the same processor p. Consider that a static order of execution $so = [a_1, a_2, a_3]$ has been imposed on them, and that this static order execution is allocated to a time slice of S time units within a time wheel with a period P.

Since the actors are guaranteed to be mutually exclusive because of the static order, it turns out that whenever one of them is activated none of the remaining others is. Thus, we can conservatively assume that the TDM scheduling will affect the response time of each actor in the same way as it would if the time slice were allocated exclusively to it. We can therefore model the mix of the two schedulers by adding to the scheduling analysis graph the edges that represent the imposed static order and replacing the execution times by the response times under TDM.

There are situations, however, where this approximation is too conservative. Consider, for instance, the case where the inputs of node a_2 are all local, i.e., are produced either by a_1 or a_3. Then, whenever a_1 finishes execution, a_2 can immediately start execution, because all its inputs must be available, since a_2 is mutually exclusive with all the actors on whose input it depends. It does not have to wait for the worst-case where the actor gets enabled in the instant when its time wheel is just over. In this case, we can subtract $P - S$ from its response time, as given by Equation 3.

In fact, we can go further than that, by observing that there is never a wait caused by scheduling time between two actors running on the same time slice with a pre-defined static order. We can separate the response time in two terms, one, a scheduling time related term $r_S(a_i) = P - S$ and another term which accounts for the time since the execution starts until it finishes, $r_x(a_i) = (P - S) \cdot (\lceil \frac{t(a_i)}{S} \rceil - 1) + t(a_i)$. An actor $a \in A$ in the original SRDF must now be represented by two actors: one, a_s, that has an execution time of $r_s(a)$ and receives inputs external to the processor and then forwards them to another actor, a_x, that has an execution time $r_x(a)$ and receives all inputs internal to the processor, plus the input from a_s. This is represented in Figure 1. One other way to look at it is to think that any path expressions involving an edge (a_i, a_j) across processors will "see" a response time $r(a_j) = r_x(a_j) + r_s(a_j)$, while if $\pi(a_i) = \pi(a_j)$, the expression $r(a_j) = r_x(a_j)$ is taken instead.

3. SCHEDULING STRATEGY

Although static-order scheduling is a popular strategy for scheduling dataflow graphs [19], it cannot singly solve our problem. Since jobs start and stop independently, a static-

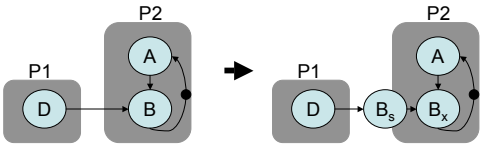
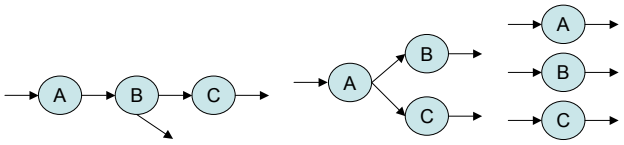
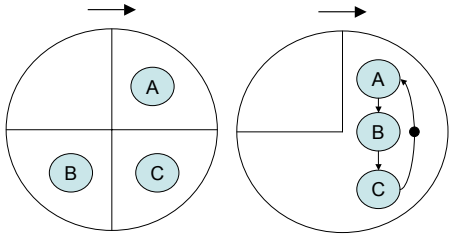


Figure 1: Splitting node response time to obtain tighter throughput analysis.



(a) (b) (c)

Figure 2: Three job fragments.



(a) TDM (b) Combined

Figure 3: Two schedules of three actors.

order schedule would have to be computed at design time for every combination of jobs that can be active simultaneously. This problem is aggravated by the fact that, when a transition occurs from a job-mix to another, remaining jobs must not experience any discontinuity caused by the change of configuration. Secondly, as different jobs can have very different rates, a static schedule could only be realized if the faster jobs where almost completely latency-insensitive, which is certainly not the case with jobs such as Wireless LAN and, even if this were the case, the length of the static order schedule and the amount of buffering of inputs required could easily become prohibitive.

Since we assume that task to processor assignment is given, fully dynamic global scheduling is not needed. One could think of simply using local TDM schedulers per processor and relying on FIFO communication for actor synchronization. This is actually the approach we suggested in [17] for a homogeneous MPSoC. The main problem with such a strategy is that the bounds on the worst-case response times of actors executing on independent TDM slices completely overlook the fact that, within a job, we have more information about the interdependence of actors. For instances, a set of actors may be mutually exclusive – in an SRDF graph this happens when the actors belong to the same single-delay cycle – and allocating a different slices to each of these task wastes resources, since, if all share the same slice, each task can use the whole slice when enabled. As we assume static processor allocation, we know that actors allocated to the same processor are already forced to execute in mutual exclusion.

To illustrate this, we compute the worst-case latency from input to output for three groups of three actors, where all three actors belong to the same job and are allocated to the same processor. These are shown in figure 2. In two of the cases, there are direct dependencies amongst them. In the third case no such dependencies exist. Edges without a source receive their input tokens from actors executing in other processors, which are not depicted.

Assume that each actor has an execution time of $t = 1$. Figure 3 shows two different schedules, both valid for all of the three cases, assuming a TDM wheel period $P = 4$. In schedule 3(a) each actor gets its own time slice of $S = 1$ duration. In schedule 3(b), the actors are statically ordered, and a slice of $S = 3$ duration is allocated to the statically scheduled group. Both schedules require exactly the same amount of processing resources (3/4 of the time wheel), but worst-case response times are much smaller for the combined scheduling strategy.

If the actors in Figure 2(a) are scheduled according to the schedule in Figure 3(a), the output of B will be produced, in the worst-case, after A and B have both executed once. A will take $r(A) = 4$ to execute, because in the worst-case A has to wait for 3/4 of the wheel to turn to get its slice, and then take 1/4 of the wheel to process. Now B has to execute once. That takes $r(B) = 4 - 1$, since B can execute sooner that waiting for a full turn after A has completed – we know that B is not allocated to the same slice as A. The output of B will therefore be generated after $r(A)+r(B) = 7$ units of time. The output of C will still have to wait for the response time of C, which will be again less than one period, since the slice allocated to C will certainly be reached before the slice of B, which just finished. Therefore $r(C) = 3$, and C will produce output after $r(A) + r(B) + r(C) = 10$ time units. On the other hand, the combined TDM/static-order schedule 3(b) takes as worst-case the time from the end of the slice allocated to the group to the next end of the slice allocated to the group, since the 3 actors all execute in sequence within one slice. Therefore, A ends execution at time 2, the output in B is produced at time 3 and the output of C has a response time of 4.

For job 2(b), the TDM schedule in figure 3(a) will yield a response time of $4 + 3 = 7$ for both external outputs, while the combined schedule will yield the same values as for job 2(a).

For job 2(c), the combined schedule imposes an arbitrary order between 3 independent actors, and thus creates extra scheduling dependencies. However, the worst-case production times, assuming all external inputs are available, are still 2,3,4, for the outputs of A, B and C, respectively, while for the TDM schedule, the worst-case production times are 4, 4 , 4. Note however, that in this case the combined schedule imposes an extra dependency on the arrival of the input of A to the production of B's output, which did not exist before. If the input of B is ready before the input of A, this particular schedule may be quite inefficient. However, this is a problem of choosing the "right" static-order. The results of this example are summarized in Table 3.

	TDM			Comb.
	Job a	Job b	Job c	All
Out A	4	4	4	2
Out B	7	7	4	3
Out C	10	7	4	4

Table 1: TDM vs Combined scheduling.

The point we want to make is that the combination of TDM slicing with static order schedule allows, when applicable, the determination of much tighter bounds on worst-case response times for the same amount of allocated resources than TDM. Since at compile-time we know only about one single job, it is at this level that static-order can be used to shorten worst-case bounds on response times.

Because of this, we use different scheduling methods to schedule among tasks within a job (intra-job scheduling) and to reserve resources among jobs (inter-job scheduling) such that each running job can meet its timing requirements independently of any starts/stops of other jobs.

Intra-job scheduling is handled by means of static order, i.e., per job and per processor, a static ordering of actors is found that respects the Real-Time requirements while trying to minimize processor usage, while inter-job scheduling is handled by means of local TDM schedulers: per job and per processor a slice time duration S is defined.

An admission controller is needed to start jobs upon the arrival of a start request. It is similar to the one we suggested in [17], in that it must check if enough memory, communication and computing resources (enough time in the TDM wheel of the processor) are available for the requirements of the job. If not, the start of the job is refused.

4. THE SCHEDULING PROBLEM

Our objective is to schedule each job in such a way that we guarantee real-time constraints and minimize processing resource usage, such that resources can be shared with other running jobs, both increasing the probability of starting the job on an already running system, and the probability that jobs requested to start later will find enough resources to start. We need to define the period $P(p)$ of the time wheel for each processor p in the multiprocessor, the slice time S(p,j) attributed to job j on processor p and, a static-order schedule $so(p, G)$ per processor p and job G.

4.1 Temporal Requirements

We define **maximum production period** μ_D as the maximum acceptable period between consecutive executions of an SRDF actor for a given job graph G.

We wish to find a schedule such that, if $G = (V, E)$ is the input job graph, $d(i, j)$ the delay for all edges $(i, j) \in E$, $\pi(v)$ and $t(v)$, respectively, the processor assignment and the execution time, for all actors in V, the graph $G' = (V', E')$, obtained by modeling all scheduling decisions represented by $P(p), S(p, j)$ and $so(p, j)$ in G should be such that $\mu(G') \leq \mu_D$ — this is equivalent to defining μ_D^{-1} as the minimum average throughput of the system.

4.2 Optimization Criterion

For each job, the schedule should use as few cycles per period as possible, as this will increase the percentage of the processor available to other jobs. A good measure of the amount of processing resources used by a job G on a processor p is the processor utilization $U(p, G)$, which can be defined, for TDM, as the ratio between the time reserved on the processor for the job and the total amount of cycles per time wheel period:

$$U(p, G) = \frac{S(p, G)}{P(p)}. \quad (4)$$

The variable $P(p)$ is set per processor as a fixed system parameter as will be described in section 4.4. For a heterogeneous multiprocessor, we try to reduce the utilization of all processors. This is done in a weighted way, since it is possible that certain processors are more required than others, and thus their utilization should be kept lower. We introduce a processor cost coefficient $c(p)$. Our optimization criterion is the minimization of the weighted sum of the utilizations:

$$\text{minimize} \sum_{p \in \Pi} U(p, G) \cdot c(p) \quad (5)$$

This function still disregards the fact that, although the resources of a processor may be more scarce than of another, if the "cheaper" processor is fully occupied, several jobs may become impossible to start – e.g. a second instance of an already running job. This could be accounted for by an optimization criterion that makes the cost of a processor increase with utilization, but that would yield a complex, non-linear objective function. Instead, we solve the problem by adding a constraint per processor on the maximum utilization per job, $\hat{U}(p)$, and we enforce that $U(p, G) \leq \hat{U}(p)$.

4.3 Phase decoupling

The problem of trying to find $P(p)$, $S(p, G)$ and $so(p, G)$, subject to the afore-mentioned constraints and optimization criterion, all at once, is complex. Besides, $P(p)$ should be a system-wide parameter: since jobs must be able to execute together, the same time wheel period must be shared by all.

Per job, we split the problem in two: finding the slice times $S(p, G)$, and finding the static order schedules $so(p, G)$.

4.4 Setting Time Wheel Periods

Assuming a constant processor utilization, it becomes evident, when one inspects Equation 3, that, when the time wheel period decreases, the worst-case response time of an actor becomes closer to its execution time. This seems to imply that the time wheel period should be as small as possible, while still allowing partitioning among the maximum amount of job instances one wishes the platform to be able to run simultaneously. However, this neglects the overhead caused by the context-switching time at the change of time slice: since the context-switching time is constant, a smaller time wheel period implies a higher absolute number of context switches and, thus, a lower utilization of the processors. We can chose $P(p)$ in such a way that we guarantee that the overhead of context-switching does not exceed an arbitrary percentage $pcc(p)$ of the processing cycles of p. If $cc(p)$ is the cost of a context-switch for processor p and $nj(p)$ is the maximum number of job instances we wish to run simultaneously, and since per each period each job gets one time slice, it comes that the number of cyles per period spent on context switch is $cc(p) \cdot nj(p)$ and the value of $P(p)$ should be set such that:

$$P(p) \geq \frac{cc(p) \cdot nj(p)}{pcc(p)} \quad (6)$$

On the other hand, since any actor in a timed SRDF graph must be able to execute once per μ_D of its job to meet the temporal constraints, the time wheel period must be such that allows this for all jobs, and therefore $P(p) \leq \mu_D(G)$ for any job G the system must run. If the two inequalities are not compatible, one must accept a higher percentage of context-switching overhead.

4.5 Finding Static-Order Schedules

The first thing to notice about the static order schedules is that the only precedence constraints that need to be respected are the ones inside one iteration. This is because the static order imposes an execution order with no iteration overlap per processor (but notice that between two different processors there may still be iteration overlap). Therefore, for the purpose of the static-order schedule, only edges in the input graph that have the delay equal to 0 need to be taken into account.

To give an idea of how the static order of actors on a processor influences the temporal behavior, we will give an example. Figure 4 depicts the timed SRDF model of a job. Assume that $\pi(A) = \pi(B) = \pi(C) = p_1$ and $\pi(D) = p_2$. Assume also that $t(A) = t(B) = t(C) = t(D) = 1$ and that $\mu_D = 3$. The MCM of the job is $\mu = 3$, because of the ACD cycle.

Now inspect the two static-order schedules in 4.5. Both of them respect the precedence constraints defined by the 0-delay edges, but while the schedule depicted in 5(a) has an MCM of 3, from cycles ACD and ACB, the schedule depicted in 5(b) exceeds μ_D since the cycle $ABCD$ has $\mu_c = 4$, due to the extra dependency caused by the static order on p_1.

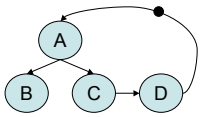


Figure 4: A job.

This reflects the fact that our particular scheduling problem is a constraint satisfaction problem: after each scheduling decision, a set of constraints that includes the original data dependencies of the input graph and the constraints imposed by the partial schedule can be checked by running an MCM algorithm to see if the MCM is beneath μ_D.

Our static-order scheduler performs full-backtracking on top of a multiprocessor-wide ready-list scheduler. Every time a scheduling decision is taken, an MCM algorithm is used to do constraint checking on the graph obtained by modeling the scheduling decisions on top of the input graph. For MCM computation, we implemented the Howard algorithm [6], since it is one of the fastest, according to the benchmark provided in [9] and, being a policy-improvement algorithm, it is easy to write an incremental version of it: at each step we start the search of the MCM for the new model graph by the MCM cycle which was the solution in the previous step.

The static order scheduler can be used before or after Slice Time attribution. If it is used before, then the original $t(i)$s are used for MCM computation. If it is used after, or if a maximum value for the slice time is required, the response time model, computed as described in Section 2.2.1, is used instead.

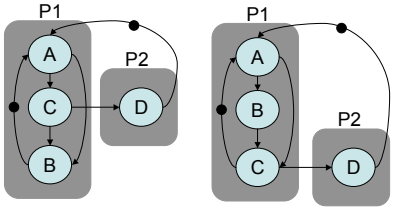


Figure 5: Two schedules for the same job.

5. FINDING THE SLICE TIMES

In order to show how to solve the problem of choosing an adequate slice time for each group of actors belonging to the same job running on the same processor, we will first show how to compute the amount of time slack that is available to each actor, with respect to the timing requirements. We encapsulate this knowledge in the concepts of deadline extension and deadline extension pool, that we describe in some detail below.

5.1 Deadline Extension

The deadline extension of an actor is an amount by which its $t(i)$ can be increased, while still guaranteeing that the MCM of the job is kept below or equal to a maximum desired production period μ_D.

5.1.1 Deadline Extension Pools

Given a desired production period μ_D and a timed SRDF job G, with $\mu(G) \leq \mu_D$, we define the *deadline extension pool* $\Delta D(c)$ of cycle $c \in \text{Cycles}(G)$ as the maximum time that can be added to the sum of execution times of c that keeps $\mu(G) \leq \mu_D$. This is given by:

$$\Delta D(c) = \max x : \mu_D \geq (\sum_c t(i) + x) / \sum_c d(i,j). \quad (7)$$

We can see that x is maximum in the upper limit of the inequality. Thus it must be that:

$$\mu_D = (\sum_c t(i) + \Delta D(c)) / \sum_c d(i,j). \quad (8)$$

Therefore, $\Delta D(c)$ is

$$\Delta D(c) = (\mu_D - \sum_c t(i)) / \sum_c d(i,j)) \sum_c d(i,j), \quad (9)$$

and if we now substitute (1) in (9):

$$\Delta D(c) = (\mu_D - \mu(c)) \sum_c d(i,j). \quad (10)$$

What the deadline extension pool of a cycle tells us is how much the execution of a cycle as a whole can be delayed while

still guaranteeing the desired throughput, if other cycles are not delayed. We are more interested in computing the maximum deadline extension per actor as it can help us finding scheduler settings.

5.1.2 Maximum Deadline Extension of an Actor

We define the *maximum deadline extension* $\delta\hat{(}i)$ of actor i as the maximum amount by which the execution of i can be delayed while still guaranteeing the desired throughput, if we assume that all other nodes are not delayed.

If an actor i belongs to a single cycle c, then it is clear that its $\delta\hat{(}i)$ will be equal to the $\Delta D(c)$ of that cycle. However, in general, i will belong to multiple cycles, and the maximum deadline extension available to i will depend on the "slowest" cycle to which it belongs. Therefore, the deadline extension pool $\delta\hat{(}i)$ of actor i is given by

$$\hat{\delta}(i) = \min_{c \in C(i)} \Delta D(c) = \min_{c \in C(i)} ((\mu_D - \mu(c)) \sum_c d(i,j)), \quad (11)$$

where $C(i)$ is the set of cycles in G to which actor i belongs.

Equation (11) seems to imply that we need to know all the cycles in the graph to compute the $\hat{\delta}(i)$'s. This would cause a severe algorithmic complexity problem. If i has a self-cycle, which we assume to be normally true for actors that represent tasks, then we know that its maximum execution time that keeps the MCM of the graph below μ_D is equal to μ_D, while the lowest value is 0. Therefore, we may use binary search with an MCM computation algorithm to find the Maximum Deadline Extension for each actor in the graph.

We will show that there is a simpler way to compute these values. If we substitute the cycle mean expression 2 in expression 11, we obtain:

$$\hat{\delta}(i) = \min_{c \in C(i)} ((\mu_D - \frac{\sum_c t(i)}{\sum_c d(i,j)}) \sum_c d(i,j)), \quad (12)$$

and by manipulating this we obtain

$$\hat{\delta}(i) = \min_{c \in C(i)} (\mu_D \sum_c d(i,j) - \sum_c t(i)), \quad (13)$$

and as we can factorize the sum operator

$$\hat{\delta}(i) = \min_{c \in C(i)} \sum_c (\mu_D d(i,j) - t(i)). \quad (14)$$

If we create a valuation w for edges $(i,j) \in E$ such that:

$$w(i,j) = \mu_D \cdot d(i,j) - t(i) \quad (15)$$

then equation (14) is the definition of the shortest path from i to i over a cycle.

LEMMA 1. *The maximum deadline extension $\hat{\delta}(i)$ of actor i is given by $\sigma(i,i)$ when all edges $(i,j) \in E$ have a valuation $w(i,j) = \mu_D d(i,j) - t(i)$ and*

$$\sigma(i,j) = \min_{p \in P(i,j)} \sum_{(m,n) \in p} w(m,n), \quad (16)$$

where $P(i,j)$ is the set of paths from node i to node j in G.

The proof of this lemma is left out for lack of space. It is available in [15].

We can therefore calculate the maximum deadline extension for all actors by running the Floyd-Warshal all-pairs shortest-path algorithm [7], which has a polynomial complexity of $O(|V|^3)$.

Deadline extension pools provide an excellent way of studying time slack, and can be used by the designer directly to estimate possible values for slice times.

5.2 Optimizing Sums of Deadline Extensions

One way to distribute time slack amongst actors is to find a sum of feasible deadline extensions that is optimal in some sense, which is equivalent to decreasing the weights given by Equation 15 while keeping all cycles at positive length. It is an instance of the inverse shortest-path problem, which has been studied in detail in [5].

The problem of maximizing the sum of deadline extensions is an inverse shortest-path problem where the actors' deadline extensions have to be chosen in such a way that no cycle has a negative shortest-path for the weights given in Equation 15, i.e., such that the MCM of the graph doesn't become higher that the desired production period μ_D. We present the formulation of the problem as a linear program, based on the observation that, for every actor $v \in V$ and every edge (u,y), it must hold, by the definition of shortest path that $\sigma(v,y) \leq \sigma(v,u) + w(u,t)$. If the weights $w(u,y) = \mu_D \cdot d(u,y) - t(u) - \delta(u)$, the linear program is:

Maximize	$\sum_{\forall v \in V} \delta(v)$
subject to	
$\forall (u,y) \in E, \forall v \in V,$	$\sigma(v,y) \leq \sigma(v,u) + \mu_D \cdot d(u,y) - t(u) - \delta(u)$
$\forall v \in V,$	$\sigma(v,v) \geq 0$
$\forall v \in V,$	$\delta(v) \geq 0$

This is a linear programming problem with a polynomial bound on the number of linear constraints, which is known to have polynomial complexity [12]. Extra linear constraints can be added that force two actors to have the same deadline or to give different weights to each deadline extension in the objective function. It is also possible to set the deadline extension of a node to 0, or add a weight to each $\delta(v)$ in the objective function.

5.3 Finding Slice Times

The linear program presented in the previous section gives a general way to distribute slack amongst the actors in an SRDF, such that timing requirements can be met. It cannot be used directly to compute time slices. The problem is that if we replace $t(v) + \delta(v)$ by the $r(v)$ expressions as given in section 2.2.1, we obtain a non-linear program. This is in part due to the ceil function in the $r_x(t)$ expression. We can linearize the problem by making a conservative (meaning overestimating) approximation of the response time by replacing $\lceil \frac{t(v)}{S(p,G)} \rceil$ for $\frac{t(v)}{S(p,G)} + 1$. We thus obtain the formula $r_x(v) \leq \frac{t(v)}{S(p,G)} \cdot P(p)$, which is linear.

In the case of an edge across different processors, the response time expression includes both v_x and v_s nodes, and it becomes $r(v) = r_x(v) + r_s(v)$. With our approximation, this becomes $r(v) \leq \frac{t(v)}{S(p,G)} \cdot P(p) - P(p) + S(p,G)$, which is still non-linear as our variable $S(p,j)$ appears both in the numerator and the denominator. We can however, make another conservative approximation. In the inequality, as $S(p,G)$

becomes smaller, the $\frac{t(v)}{S(p,G)} \cdot P(p)$ becomes bigger. We can linearize the expression by dropping the $+S(p,j)$ term. We also make a variable substitution: $N(p) = P(p)/S(p,G)$ (we drop the job parameter G, since when solving the slice time problem this can be kept implicit). An extra constraint is that $N(p) \geq U(p)^{-1}$. Instead of maximizing the total amount of used time slack, we maximize the sum of $N(i)$, weighted by the processor costs as defined in section 4.2. We obtain the following linear program:

Maximize	$\sum_{\forall v \in V} N(\pi(v)) \cdot c^{-1}(\pi(v))$
subject to	
$\forall (u,y) \in E : \pi(u) = \pi(y)$,	
$\forall v \in V$,	$\sigma(v,y) \leq \sigma(v,u) + \mu_D \cdot d(u,y)$ $-N(\pi(u)) \cdot t(u)$
$\forall (u,y) \in E : \pi(u) \neq \pi(y)$,	
$\forall v \in V$,	$\sigma(v,y) \leq \sigma(v,u) + \mu_D \cdot d(u,y)$ $-N(\pi(u)) \cdot t(u) + P(\pi(y))$
$\forall v \in V$,	$\sigma(v,v) \geq 0$
$\forall p \in \Pi$,	$N(p) \geq \hat{U}(p)^{-1}$

Notice that $N(p) = U^{-1}(p,j)$.

6. PHASE ORDERING

It is not trivial to chose in which order to perform the determination of the slice times and the computation of the static-order schedule, since these two steps are strongly interdependent. If one determines the slice times first, the determination is based on partial constraints since the schedule is yet to be derived. It may be that there is no valid static-order schedule that meets the tighter scheduling constraints caused by replacing the original execution times by the larger response times due to the chosen time slicing.

We prefer to determine the static-order schedule first, using response times based on the maximum allowed utilization per processor, $\hat{U}(p)$, and only afterwards try to find slice times that are compatible with that static-order schedule. This has the advantage that, provided there is a feasible static-order schedule, there is always a solution. However, it may be that the static-order schedule chosen is not optimal in the sense that does not allow for the lowest possible utilization that meets all the constraints.

We thus approach the problem first as a constraint satisfaction problem, to which the static order scheduler delivers a feasible solution, if there is one, and then try to optimize this solution by reducing utilization.

7. EXAMPLE

We will now show how to apply these scheduling techniques to an actual application, in the domain of Software-Defined Radio. Assume a multiprocessor system designed for baseband decoding. It includes a general-purpose core, an ARM, to handle control and generic functionality, a vector-processor core, the EVP [3], to handle detection, synchronization and demodulation, and an application-specific Software Codec processor that takes care of the baseband coding and decoding functions. All these processors are interconnected via an Æthereal Network-On-Chip [11]. The platform is used to handle several radio standards (Wireless LAN, TD-SCDMA, UMTS, DVP-H, DRM). In our example we will assume that we want to derive scheduler settings such that we are able to run Wireless LAN (WLAN) 802.11a and TD-SCDMA simultaneously, with independent start and stop, and allow for up to 2 job instances to be active at a time, including configurations with two WLAN instances and two TD-SCDMA instances.

Figure 6 depicts the timed data-flow model of a WLAN 802.11a job. Execution times (indicated under the actor names) are given in nanoseconds. The different shading of nodes indicates the different cores to which the actors are assigned. Nodes with names starting by "Src" model the source (inputs from an external RF unit), the nodes "LatencyHeader" and "LatencyPayload" and their adjacent edges are used to convert latency into throughput constraints, as described in [16]. For space reasons, the Synchronization step is represented in Multi-Rate Dataflow syntax: 5 "CFESync" nodes process the output of 5 "Src" nodes in a chain of sources and synchronization nodes. The number of "PayloadDemode" actors and respective sources may vary between 1 and 255. We only depict the case where there is only one "PayloadDemode" actor. Source and Latency actors are not scheduled. This graph has a required maximum production period of $\mu_D(\text{WLAN}) = 40000ns$.

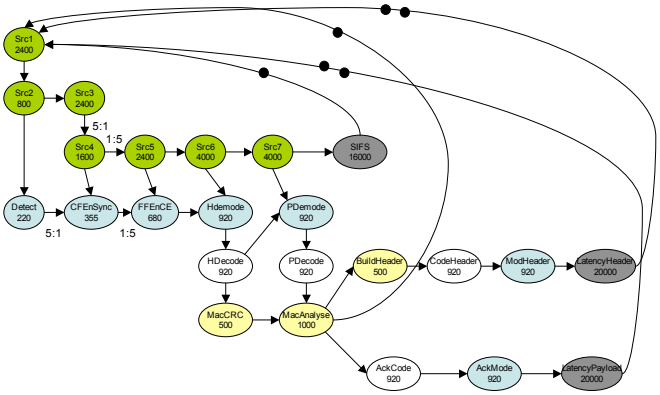


Figure 6: A Wireless LAN 802.11a job.

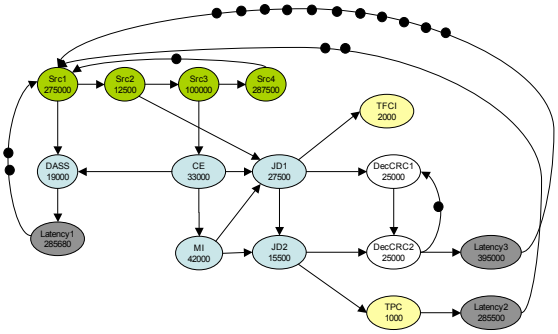


Figure 7: A TD-SCDMA job.

Figure 7 depicts the timed data-flow graph model of a TD-SCDMA job. The "Rx" nodes represent the source, and

Weights			Utilization (%)		
EVP	SwC	ARM	EVP	SwC	ARM
1	1	1	44.9	44.9	20.9
2	1	1	27.6	44.9	44.9
2	2	1	27.6	44.9	44.9
1	2	1	44.9	27.6	44.9

Table 2: WLAN scheduling results.

Weights			Utilization (%)		
EVP	SwC	ARM	EVP	SwC	ARM
1	1	1	44.9	15	13
100	1	1	42.4	15	13
1000	1	1	32.8	15	44.9

Table 3: TD-SCDMA scheduling results.

the "Latency" nodes and adjacent edges are used to convert latency into throughput requirements. The required maximum production period is $\mu_D(\text{TD-SCDMA}) = 675000ns$.

We start by choosing time wheel periods. For the EVP and the Software codec, we assume a worst-case context-switch time of 50ns. We want to support two jobs at a time and we would like to spend less than 10% of our time context-switching, so we set our wheel time to 1000ns. For the ARM, context switching takes more time. We assume a worst-case of 100ns, yield a period of 2000ns. Therefore, for $P(EVP) = P(SWC) = 1000$ and $P(ARM) = 2000$.

As we want to be able to run two WLAN or two TD-SCDMA jobs simultaneously, we set the maximum utilization per processor, $\hat{U}(p)$, at 0.45 (half processor minus context-switching).

For both jobs, we first computed static-order schedules and then time slice extensions. In both cases, a static-order schedule that meets the timing constraints (including the $\hat{U}(p)$ limitation on usage per processor) was found. These schedules were used to calculate slice time optimizations. We varied the values of the costs of the three processors to search for trade-offs. It turns out that the results tend heavily towards allocating all slack to a particular processor. A change in weights typically forces a drastic change from allocating slack to one processor to another.

In the case of the WLAN – the results are shown in Table 2 – the slice allocation allowed us to decrease the utilization of one processors from 45% to 27.6%. The table pretty much exhausts the possible trade-offs. It suggests that it makes sense to keep a table with several configurations and allow the admission controller to chose different TDM settings for a job instance taking into account the current level of resource utilization.

The results for the TD-SCDMA are shown in Table 3. One thing to notice is that the optimal slice times for the ARM and the Software Codec are obtained for equal weights. Increasing the weights of either ARM or Software Codec by any amount, didn't further decrease their slice times. By comparing the results in the first and the second entries of this table, we observe one of the problems with our linear programming formulation: the optimized value of $N(ARM)$ in the first entry does not allow a smaller slice time than the much lower value obtained in the second entry; it does however prevent the slice time for the EVP to be decreased from 449 to 424. This example does not allow many trade-offs: to allow a decrease from 449 to 329 on the EVP, we had to increase its weight to 1000. Moreover, this came at the cost of changing the utilization of the ARM from 1.3% to 44.9%. The bottom line is that, although not perfect, our slice time optimization allows a decrease of the utilization to 1.5% for the Software Codec and 1.3% for the ARM, from the imposed maximum 45%.

8. RELATED WORK

Much work has been published on the scheduling of dataflow graphs with real-time requirements. The level of dynamicity we allow in the start and stop of jobs is what basically differentiates our problem from other multiprocessor real-time scheduling problems. We will review proposed approaches that could be adapted to solve our problem.

The most viable alternative to our solution is to use precompiled resource allocation configurations, such as in CPA [20]. For each job-mix, a separate optimal static schedule is derived at compile-time and stored in a look-up table. During operation, when there is a request to start a job, the run-time system checks which jobs are active and selects the appropriate configuration. This approach is not without problems. First, a different configuration has to be stored for each combination of jobs, which means that the number of configurations grows exponentially with the number of jobs. Second, if a job is not known at design time, it will force a whole new set of configurations to be compiled later. Third, it is difficult to assure continuity of execution of already running jobs during reconfiguration. For continuity, a configuration should be generated for each transition from a job-mix to another which easily becomes infeasible.

We calculate scheduling budgets per job at compile-time. During run-time two distinct temporal phases alternate: configuration phase and steady-state execution phase. During configuration phases, resources are allocated to jobs; during steady-state resource allocation is fixed. In this, our strategy can be compared with semi-static techniques [8], where system execution is divided in phases and resource allocation is redone at the beginning of each phase. But while in semi-static systems phases are periodical, in our case reconfiguration phases are triggered by a request to start or stop a job.

Our strategy has similarities with other time-multiplexing strategies such as gang scheduling [10]. Our jobs correspond roughly to a gang, i.e. a group of tasks (actors in our case) with data dependencies. There is, however, an important difference: in gang scheduling, time-multiplexing is global, i.e., the temporal slots are uniform across all processors, and synchronized context-switching is required. In our case, time-multiplexing is local to each individual processing element. Our strategy has several advantages over gang scheduling: it does not require global synchronization of context switches, which hinders design scalability; it leaves less unused resources because its time-sharing is more fine-grained; and it also allows for a different scheduling mechanism per processor.

In previous work, [17], we have shown how an online resource manager can be built to reserve resources per job at start time, while keeping already running jobs unaffected. In [16], we proposed the combination of TDM and static-order scheduling to address dynamic job-mixes but did not specify any flow. A recent paper [21] describes a method similar to ours, that mixes static-order and TDM scheduling. It does

not address different start/stop times. To account for the effect of TDM on the response times, symbolic simulation of the dataflow graph is performed, while keeping track of the state of the TDM wheels. For this to yield conservative response times, all possible states for a worst-case, self-timed schedule need to be simulated, which seems to imply that conservative resultes require that simulation continues until the same token positions *and* the same position on all TDM arbiters is reached simultaneously, which can lead to an exponential blow-up. This work also differs from ours in that the static-order schedule is determined by a ready-list scheduler with no backtracking and no timing constraints, and slice minimization is done using binary search on top of the symbolic simulation.

9. CONCLUSION

We have presented a scheduling strategy and a scheduling flow to solve the problem of running multiple jobs with different rates and different start/stop times. The scheduling strategy involves a combination of static-order scheduling per job per processor, and TDM scheduling to arbitrate between different jobs in each processor. We have shown how the combination of TDM with static-order scheduling is desirable and can be modeled for the purpose of temporal analysis. We have shown how the temporal analysis model can serve as a basis for a scheduling flow. We have proposed algorithms to estimate the temporal slack that can be allowed per actor for a timed SRDF, and how to exploit this time slack by decreasing slice times. This flow solves a practical problem arising in real-time streaming platforms (Software-Defined Radio, Car-Radio, Digital TV) and is unique in that it is able to find both TDM settings and static order schedules per job per processor, to handle a dynamic job-mix on a multiple processor and provide hard real-time guarantees for all admitted jobs.

Although we show this flow exclusively using Time-Division Multiplex for inter-job scheduling, this does not present an essential limitation of the techniques: any budget scheduler can be used, although changes must be made to the dataflow model and to the computation of scheduler settings.

There are several open issues we wish to address in future work. One is optimizing the static-order schedule for lower slice times; another is improving the slice minimization: the linearization in the current approach becomes an issue when the execution times are of the same order of magnitude as P, as the overhead of approximating $P - S$ by P becomes too high. The current approach allows communication and buffer capacity constraints to be taken into account, but it is also interesting to search for scheduling settings that optimize for low communication requirements and/or small buffer sizes.

10. REFERENCES

[1] M. Bekooij et al. Predictable embedded multiprocessor system design. In *Proc. Int'l Workshop SCOPES*, LNCS 3199. Springer, Sept. 2004.

[2] M. Bekooij et al. Dataflow analysis for real-time embedded multiprocessor system design. In *Dynamic and Robust Streaming in and between Connected Consumer Electronic Devices*, volume 3, pages 81–108. Springer, 2005.

[3] K. Berkel et al. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal on Applied Signal Processing*, (16), 2005.

[4] G. Blisen et al. Cyclo-static dataflow. In *IEEE Transactions on Signal Processing*, volume 44, pages 397–408, 1996.

[5] D. Burton. the inverse shortest path problem, 1993.

[6] J. Cochet-Terrasson et al. Numerical computation of spectral elements in max-plus algebra. In *Proc. IFAC Conf. on Syst. Structure and Control*, 1998.

[7] T. Corman et al. *Introduction to Algorithms*. McGraw-Hill, 2001.

[8] D. Culler et al. *Parallel Computer Architecture: a hardware/software approach*. Morgan Kaufmann, 1999.

[9] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):385–418, Oct. 2004.

[10] D. Feitelson. Job scheduling in multiprogrammed parallel systems. Technical report, IBM Research Report RC, 1994.

[11] K. Goossens et al. *Guaranteeing the quality of service in networks on chip*.

[12] L. Kachyian. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20:53–72, 1980.

[13] E. Lee and D. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, 1987.

[14] A. Moonen, M. Bekooij, and J. van Meerbergen. Timing analysis model for network based multiprocessor systems. In *Proc. Workshop of Circuits, System and Signal Processing (ProRISC)*, pages 91–99, Veldhoven, The Netherlands, 2004.

[15] O. Moreira. Self-timed scheduling analysis for real-time applications. Technical report, Technical University of Eindhoven, 2006.

[16] O. Moreira and M. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.

[17] O. Moreira, M. Bekooij, and J. Mol. Online resource management for a multiprocessor with a network-on-chip. In *Proc. ACM Symposium on Applied Computing*, March 2007.

[18] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, October 1968.

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

[20] M. Strik et al. Heterogeneous multiprocessor for the management of real-time video and graphics streams. *IEEE Journal of Solid-State Circuits*, 35(11):1722–1731, 2000.

[21] S. Stuijk et al. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proc. Design Automation Conference (DAC)*, 2007.