Predictable Dynamic Behaviour in NoC-based Multiprocessor Systems-on-Chip

R.J.H. Hoes
November 2004

Eindhoven University of Technology (TU/e)
Philips Research Labs Eindhoven

Professor:
Prof. dr. ir. J. van Meerbergen (Philips Research, TU/e)

Supervisors:
Dr. ir. A.A. Basten (TU/e)
Dr. ir. M. Bekooij (Philips Research)

Abstract

Modern streaming multimedia applications are becoming increasingly complex and dynamic. Still, designers want to give guarantees about the quality and performance of their applications. A combination of model-based design methods and predictable hardware architectures enables designers to develop individual application components (jobs) in a way that it is guaranteed that real-time constraints are satisfied, even when multiple jobs are concurrently running on a single hardware platform.

A multiprocessor Sytem-on-Chip is considered, which consists of a heterogeneous set of processing elements that are interconnected through a Network-on-Chip. Existing design methods are often based on the Synchronous Dataflow (SDF) model of computation, which offers the necessary design-time predictability. However, SDF can not adequately model jobs that contain dynamic behaviour, like conditionals and loops with data-dependent bounds. By making dynamic behaviour explicit in the specification model of a job, it is possible to estimate the needed amount of platform resources more accurately and thus arrive at more efficient job implementations.

This thesis proposes an extension to SDF, called Predictable Dynamic Dataflow (PDDF). In PDDF it is possible to incorporate dynamic behaviour in a job's specification by the use of special constructions, while it is still possible to guarantee bounds on timing and memory usage. PDDF introduces two dynamic constructs: the conditional and the data-dependent iteration. Both constructs behave like SDF towards their environment, so they can be plugged into any existing SDF graph. Moreover, they are fully modular, in the sense that they can be nested in any possible way. The semantics of these constructs are explained and it is shown how the use of PDDF can lead to more efficient implementations compared to the SDF-based approach. To illustrate this, an implementation of an H.263 video decoder is used as a test case.

Acknowledgements

I would like to express my gratitude to professor Jef van Meerbergen for giving me the opportunity to carry out the research for this Master's thesis partly within the Hijdra project in Philips Research Labs Eindhoven. Without the support and enthousiasm of my supervisors, Marco Bekooij and Twan Basten, who both spent many hours of their valuable time to have interesting discussions with me and to provide me with useful feedback, this result would not have been possible. Thanks a lot! I also like to thank Sander Stuijk, who kindly rewrote parts of his H.263 decoder (a tedious task) to provide me with a fully functional test application, and Marc Geilen for his input during the discussions with Twan and me. Finally, I would like to thank the other members of the Hijdra team and colleagues at Philips Research and the people of the Electronic Systems group at TU/e for the very good time I had at both places during the past nine months!

Table of Contents

Chapter 1

Introduction

1.1 Background

Modern multimedia applications are becoming increasingly complex, requiring a lot of processing power, with this power being used in a very dynamic way. Even as processor cores become more powerful each year, also their energy consumption increases exponentially. Therefore, in many cases it can be more efficient to combine several average-speed processor cores and divide the workload between them to achieve a high performance. A multiprocessor System-on-Chip (SoC) is one such approach that allows the integration of a possibly heterogeneous mix of processing and memory components on a single chip. The communication between processors is implemented by a Network-on-Chip (NoC), consisting of routers and links. NoCs provide a flexible, scalable, and predictable on-chip communication infrastructure.

To simplify the development of these complex systems, an application is split up into several independently operating parts called jobs. A job in such a multimedia application is generally a unit that processes streams of data and has real-time constraints. A data stream in this sense, is a long chain of data packets that arrive in a periodic fashion. The real-time constraints generally demand that the time it takes to process such a packet is bounded and that the throughput of the job – the number of output packets per second that the job produces – is at least a pre-defined minimum. When designing a job, the developer wants to be guaranteed that all real-time constraints are met in all possible cases, for all possible input streams. Consequently, he wants to predict the behaviour of the job, as it will eventually run on the multiprocessor SoC, to provide these guarantees. Therefore, he uses a model of computation (MoC) to design his job. Using the design rules of such a MoC, he constructs a model of a job, which he can analyse and simulate to obtain information about its eventual behaviour. The implementation of the job follows directly from the model. In this way, the behaviour of the job is guaranteed to be predictable.

The MoC that is currently used for this purpose, called Synchronous Dataflow (SDF), is very restricted in the sense that it can only represent relatively static jobs. When a job contains a lot of operations of which the behaviour heavily depends on the input data, the usage of an SDF model may lead to highly unsatisfactory results. This thesis proposes extensions to the existing MoC, to make it possible to incorporate knowledge about this dynamic behaviour in the model of a job. This eventually leads to job implementations that are better predictable and utilise the resources of the multiprocessor platform in a more efficient way. In the next section, the precise objectives are defined.

This work builds on existing NoC-based SoC simulators and SDF-based analysis tools and programming environments, developed in the Hijdra project of Philips Research Labs Eindhoven and at Eindhoven University of Technology (TU/e).

1.2 Objectives

As said above, a job in a real-time streaming application generally has demands on its throughput. For example, a job that decodes a video stream should produce a predefined number of images every second. When the job is mapped to a multiprocessor platform it will use a certain amount of the platform's resources (processor cycles, memory and communication bandwidth), depending on the input data. The challenge is to design and map the job in such a way that it meets its throughput constraints, while having a minimum resource utilisation. This leads to the following mapping problem statements, which are slightly different for two types of jobs, hard and soft real-time jobs:

- **Hard real-time** jobs have constraints on their throughput that need to be satisfied at all times: all deadline have to be met, but it is not necessary to perform better. In fact, it is often the best to get as close as possible to the constraint, to save resources. The mapping problem for hard real-time jobs is therefore an optimisation problem, in which throughput constraints have to be satisfied, while resource usage has to be minimised.
- **Soft real-time** jobs also have deadlines, but it is not necessary to meet them all. The objective here, is to miss as few deadlines as possible or at most an acceptable number. Thus, trade-offs between the number of deadline misses and resource utilisation can be made. Therefore, the mapping problem for soft real-time jobs comprises minimising both deadline misses and resource utilisation.

To solve these problems, a model is needed that represents a job that is mapped to a platform. It needs to be possible to analyse this model to derive (predict) run-time properties already before implementing the job. For timing properties, this analysis needs to be *conservative*, so the behaviour of the implementation can never be worse than the predicted behaviour. This is necessary for real-time systems, in which a certain quality needs to be guaranteed. This model also serves as a model for simulation, which can be performed to obtain average-case or more accurate worst-case results for a 4certain set of input streams. Finally, the model can be directly used for generating code for the platform and for deriving the necessary amounts of resources on the platform.

The two mentioned mapping problems are very complex, as there are many possible ways to split up the workload in a job and to map the parts to different resources on the platform. Many parts of a job may contain data-dependencies of various kinds: processing times of tasks within a job may be dependent on input data and even the activation of such tasks may be influenced by data streams within the job. These data-dependencies result in dynamic run-time behaviour in the system, which makes it even more difficult to predict what exactly will happen when the job is implemented.

This work is part of an approach to solve the mapping problems. A lot of work has already been done, but data-dependencies are hardly taken into account. This work builds on this existing knowledge. The objective is to introduce data-dependent behaviour explicitly in the models, to deal with dynamically behaving systems. Using more knowledge about data-dependencies may lead to more efficient implementations. A key point is the importance of

prediction and conservative analysis and simulation.

The main research question is formulated as follows:

> *"How can knowledge about dynamic behaviour in a job that is to be mapped to a multiprocessor System-on-Chip, be exploited in solutions to the mapping problems for both hard and soft real-time, to arrive at tighter conservative timing estimations and ultimately more efficient implementations?"*

Several goals are stated, to help finding an answer to the research question. The following results are desired:

1. A characterisation of the various kinds of dynamism that may be present in jobs that run on the multiprocessor platform.
2. A description of the existing model of computation for jobs and the way to incorporate platform information in it. This includes a design flow in which the MoC is used in order to solve the mapping problems for both hard and soft real-time jobs.
3. Extensions to the existing model of computation to explicitly include the various kinds of dynamism from point 1. In the original MoC, only upper-bounds on dynamic constructions can be modelled.
4. A proof of how using the extended model of computation can lead to tighter conservative timing estimations and more efficient implementations compared to the original approach.
5. A simulation method based on the new model of computation.
6. A test case to show the benefits of the new model of computation.

In this document, these goals are referred to as "Goal x" when needed.

1.3 Assumptions & restrictions

The following points are assumed throughout this thesis:

- This document only deals with the mapping of single jobs to the multiprocessor platform; the mapping of multiple jobs simultaneously is not considered. Also, interactions or dependencies between jobs are not taken into account: every job is assumed to be completely independent. This is possible because of the use of resource budgets for jobs, as explained in Chapter 2.
- The jobs that are discussed, can be specified by means of task graphs (see Chapter 3). Methods exist to adequately split a job into multiple, possibly parallel tasks of a certain grain size, so these task graphs can be used as input for the methods described in this document.
- For a job that is represented as a certain task graph, upper-bounds on the usage of platform resources can be derived, as well as upper-bounds on the processing time of the tasks.

- The state of a platform is available at the time of mapping, so it is known which resources are free to use.

1.4 Overview

This document is organised as follows. The next chapter starts explaining the relevant details about the multiprocessor platform template. Subsequently, the possible types of dynamism are discussed (Goal 1). Also, the concept of scheduling for a multiprocessor system is explained and several scheduling methods are shown. Furthermore, it is argued which methods are the most suitable to be used for the type of applications and platform that are discussed here.

Chapter 3 focusses on how to design a job and how to map it to the multiprocessor platform. A design flow that is based on the SDF model of computation is proposed; all necessary steps are explained (Goal 2). SDF and its use is explained in detail: it is shown how jobs are specified and how to link the job SDF model to the platform, so it can be used for timing analysis and simulation. It also explains in what sense the SDF model is restricted and what are the implications of these limitations.

Subsequently, Chapter 4 proposes an extension to SDF to incorporate dynamic behaviour in a well-defined way (Goal 3). It suggests constructs to include conditionals and data-dependent iterations. It also shows how to analyse the extended model of computation and why the use of this leads to potentially tighter timing estimations and more efficient implementations (Goal 4).

In Chapter 5 an existing simulator that is used for SDF is described and it is explained how it can be used with the new dynamic model (Goal 5). The simulator is extended with the possibility to choose different mappings and to obtain detailed timing information. The initial, "transient" phase of a job's execution appears to be hard to predict analytically. Simulation is used to gain insight in the behaviour of jobs in this phase and, especially for soft real-time jobs, to acquire average case timing information.

To verify the new models, analysis and simulation techniques, a case study is worked out in Chapter 6 (Goal 6). A functional model of an H.263 video decoder is used for the experiments.

Finally, in the last chapter, the work is summarised and conclusions are drawn. Also, recommendations for future work are given.

Chapter 2

Multiprocessor implementations

2.1 Introduction

This chapter serves as an introduction to the basic concepts and implementation issues that are relevant in this thesis. The first section explains the multiprocessor architecture and the type of applications that is considered. Section 2.3 states that jobs in an application can contain dynamic properties and presents a classification of jobs to position the contributions of the work in this thesis in the overall picture. Subsequently, Section 2.4 explains the concepts of reconfiguration and mode changes and how this work relates to these concepts. Finally, Section 2.5 explains and compares the various scheduling methods that are used to schedule tasks on processors. It also gives recommendations about which methods to use for which type of jobs.

2.2 Platform & applications

2.2.1 Platform template

The work in this thesis is based on the platform template that is developed in the Hijdra project at Philips Research Labs Eindhoven. The platform template (see Figure 2.1) is a certain multiprocessor architecture on a single chip (System-on-Chip, SoC). The SoC consists of multiple *tiles*, each containing a processor (CPU, DSP, ASIP) and/or memory and a communication assist (CA). The tiles are connected through a Network Interface (NI) to a packet-switched Network-on-Chip (NoC), which consists of routers and links, to provide predictable communication between tiles. An example of such a NoC, which is used in the Hijdra project, is Æthereal [7]. The communication assist takes care of arbitration for the memory and communication to and from the network. The platform template is described in more detail in e.g. [2] and [20]. An instantiation of this platform template (a platform) has a specific number of tiles and a certain NoC instance. The type of processor and amount of memory can be different for each tile and also the number of routers and connections in the network can be arbitrarily chosen.

To the application designer, a platform is considered as a set of resources. There are three types of resources: processing (clock cycles on processors), memory and communication (network bandwidth between tiles). Every resource is restricted in a practical situation. In Section 3.4 the properties of the platform are modelled more precisely in terms of these

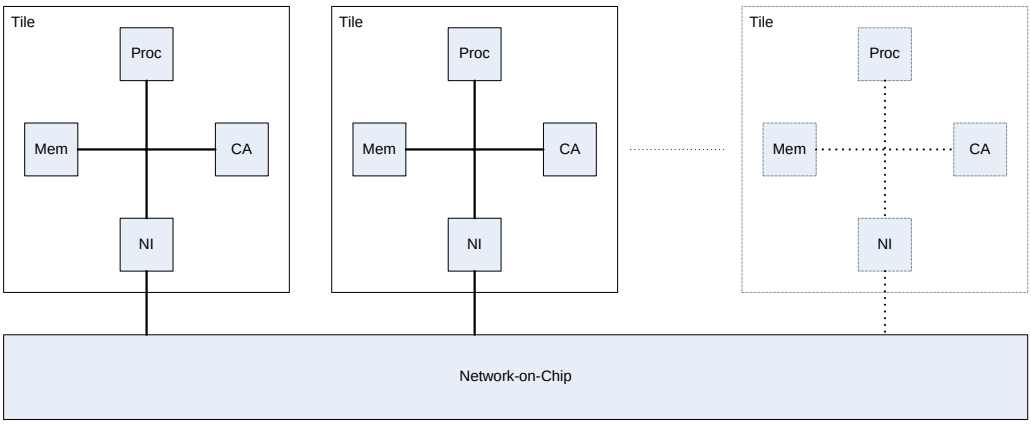


Figure 2.1: Platform template

resources.

An important property of the platform template is *predictability*: the throughput and latency of every part can be determined so the performance of the system can be guaranteed. This is very important for the domain of real-time applications, in which the platform template is used.

2.2.2 Applications, jobs, tasks

A platform is used to execute a certain multimedia *application*, for example for displaying multiple video windows on a television screen. An application consists of one or more *jobs* that are able to independently process streaming data. Each job consists of several *tasks*, which are the smallest functional units of the application. A task is considered to be an atomic operation that has a bounded processing time, which means that a task always finishes its execution once it is started, within a certain limited time span. This property is necessary to ensure predictability. A task can start its execution when it is *enabled*, i.e. when it has enough data available to complete its execution and there is enough space available to store its output.

A job can be represented by a directed (task) graph, in which the vertices are tasks and the edges are data-dependencies between tasks. Because the tasks of a job are operations that are bounded in time, it is also possible to bound the timing of the job as a whole and thus making it predicable. Another goal is to be able to design jobs independently from one another. Putting jobs together in an application, may not change the behaviour of the jobs. To ensure that a job can run independently, it is assigned resource *budgets* for all three resource types. Within its budget, a job can freely use the resources and it is guaranteed that they are always available. The platform resources that are assigned to a job together are called a *virtual platform*. A job can be mapped to the virtual platform; a *mapping* of a job to a (virtual) platform is an allocation of resources to tasks, buffers and communication between tasks. This also involves the *scheduling* of tasks on the processing resources, when multiple tasks are assigned to a single resource (more about scheduling methods in Section 2.5). The timing behaviour and resource utilisation of a job that is mapped to the virtual platform is equal to the timing behaviour and resource utilisation of this job, when it is mapped to the real platform, together with other jobs. Thus, every job on it self remains predictable at all

times.

Jobs may have periodic deadlines on the production of output data. For example, a certain audio decoder job must produce output samples with a frequency of 44.1 kHz. Two types of deadlines exist: *hard* and *soft* deadlines [22]. A hard deadline is a critical deadline that is either met or missed. A hard deadline is never allowed to be missed. A soft deadline is a non-critical deadline; an action is either more or less in time, depending on its completion time with respect to the deadline. Not meeting a soft deadline can demand the execution of a fall-back mechanism that makes sure that the user does not experience a strong quality reduction in the output of the application.

Jobs can be classified in three groups. Based on the presence and type of deadlines, a job can be hard real-time, soft real-time or best effort. A hard real-time job is a job containing at least some actions with a hard deadline that has to be met. A soft real-time job is a job containing some actions with a soft deadline, in which one tries to meet a certain optimality criterion, e.g. "minimise the number of missed deadlines". In a soft real-time job, the resources are not allocated for the worst case, but for a more average case. Resources are used more efficiently in this way. The price that has to be paid for this, is that it cannot be guaranteed that all deadlines are met and thus fall-back mechanisms have to be invoked that will probably reduce the quality of the service. The last job type, the best-effort job, is a job that has no deadlines to be taken into account. It does not matter much when the task is finished; it just uses the resources it can get to finish it as soon as possible. This thesis does not consider best-effort jobs; only the two real-time job types are taken into account.

The real-time constraints on a job are generally not defined as deadlines, but on the throughput of the job. Throughput is defined as follows:

Definition 2.1 (Throughput of a job) *The throughput of a job is the average number of output samples per time unit that are produced on a certain pre-selected output port of the job, over a certain period of time.* □

As throughput is defined as an average over a certain period of time, it is allowed for some samples to be late, as long as this is made up for before the end of the period. Buffering at the output can be used to even out these differences, so data samples are still available at the required rate, albeit with a longer latency.

2.3 Dynamism in a running job

The tasks inside a job are often pieces of software that are written in a programming language like C. Typically, such a program consists of statements that specify a certain computation and of statements that influence the flow of the program. The former type of statements can normally be compiled into a sequence of processor instructions that take a fixed number of clock cycles to execute. The latter type can be, for example, `if-then-else` or `switch` statements that make the execution of pieces of code conditional with respect to a certain control variable. It can also be loops in a program, constructed with a `for` or `while` statement, of which the body is executed a number of times, also depending on a certain control variable. Because of this, the number of clock cycles that it takes to execute a task on a certain processor, may depend on the value of some variables inside the task. The processing time

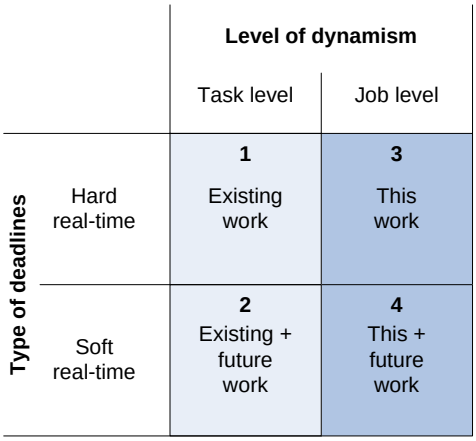


Figure 2.2: Classification of jobs

can be any number of clock cycles from zero up to a certain upper-bound. This is the first level of dynamism that can be present in a job.

Although this first level of dynamism causes the processing time of individual tasks to be variable, it does not change the number of times that tasks are executed. The above reasoning can also be raised to the level of the job, which suggests a second level of dynamism. In this second level of dynamism, the data that is transferred between tasks may influence the relative number of times that tasks are executed. A job can contain *conditionals*: certain parts in its task graph, in which tasks are only executed when a certain expression that is based on data in the job has the Boolean value TRUE. This is similar to an `if-then-else` construction in a conventional programming language. Tasks inside *data-dependent iterations*, are tasks that are executed a number of times that is also dependent on data. This construct is similar to a `for` loop with a data-dependent bound. Only when a job does not contain conditionals or data-dependent iterations, the relative number of executions of a task compared to other tasks in the job, is fixed. Thus, three forms of dynamism are considered, based on the above:

- Data-dependent task processing times (task level dynamism)
- Conditionals (job level dynamism)
- Data-dependent iterations (job level dynamism)

It is said argued that this covers all possible forms of dynamic behaviour that could occur in a job. For example, anything related to non-determinism (i.e. the behaviour of a job does not solely depend in its input data), is not included. However, based on the previous reasoning, it seems that the named three types are the most basic and that they apply to many jobs in practice.

The classification of jobs based on the type of deadlines (hard or soft), which was made in the previous section, can be extended with another orthogonal dimension: the level of dynamism (see Figure 2.2). Most of the existing work focusses on hard real-time jobs that can have data-dependent processing times, but do not display other kinds of dynamic behaviour (square 1 in the figure). For these jobs, all resources are allocated for the worst possible case: when all tasks use their maximum processing times. Because the relative number of

executions of the tasks is constant, it is always possible to predict the job's behaviour and to guarantee that all deadlines are met. Also the second class, soft real-time jobs with the same limited type of dynamism, can be treated with the existing techniques. However, the use of fall-back mechanisms in a predictable way, has not yet been studied very well. This remains to be future work. Chapter 3 talks about the design of jobs that fit in these two classes.

The main contribution of the work in this thesis lies in the classes 3 and 4 in Figure 2.2: hard and soft real-time jobs that contain all three types of dynamism. For these jobs, the existing techniques can be reused and extended, to allow conditionals and data-dependent iterations (see Chapter 4). Anything concerning fall-back mechanisms in soft real-time jobs remains as future work.

2.4 Reconfiguration & modes

During the execution of a job, it could happen that the demand for resources changes at a certain moment in time. This could be caused by the user, who, for example, enlarges a window that is displaying a movie. Another reason could be that in a soft real-time video job, the displayed scene changes drastically and more processing power is structurally needed to maintain the desired image quality.

The total distribution of resource budgets of a certain platform to jobs of an application is called a *configuration*. A *reconfiguration* is a redistribution of platform resources over jobs that is necessary when there is a structural demand for more resources in a running job. Also when new jobs are supposed to run on the platform, a reconfiguration could be needed. Since the key issue in this domain of streaming real-time applications is predictability, also a reconfiguration should be fully predictable. This means that the timing it takes to calculate a new configuration and, after that, to load it, must be bounded in all cases. How to take care of this issue is yet an unsolved problem.

Because the reconfiguration problem is still too difficult, a simpler version of the problem was solved first: *mode changes*. In this concept, only one job is considered, that can have several *modes*. A mode is defined as a state of the job that has its own task graph. The complete job's task graph can be seen as the union of all its modes' task graphs. Switching between modes is trigged by the data in the job. The elements of the complete job, including all modes, are mapped to the platform at design time: all necessary resources (processor cycles, network connections, memory) are allocated before starting the job, so reconfigurations are never needed.

The theory that is developed in this thesis can be applied to implement modes and mode changes. The various modes can simply be considered as conditional parts of a job's task graph. These conditionals can be treated in a predictable way with the model that is explained in Chapter 4.

2.5 Scheduling methods

Scheduling a job to run on a hardware platform generally involves three steps: assigning the tasks to processors, determining the order in which tasks may run and setting the start times at which the tasks will be executed. Each step can be performed either at run-time or at compile-time. Lee and Ha [15] specified a scheduling taxonomy based on these steps, covering

a range of scheduling methods from fully dynamic scheduling (all steps are performed at run-time) to fully static scheduling (the complete schedule is fixed at compile time), each method having its own pros and cons.

Scheduling methods can be compared on some important properties, which may be interrelated. This section considers five properties. Firstly, the calculation and enforcing of a schedule needs to be considered. If much work has to be done at run-time, this will cost significant resources. Thus, the amount of *run-time overhead* (property 1) should be limited when resource constraints are important. Next, especially for hard real-time jobs, the *worst-case timing and resource utilisation* (property 2) is important. As for these jobs deadlines always have to be met, resources for the worst-case situation have to be reserved. A good average case behaviour is not a main concern for hard real-time jobs. Soft real-time jobs, on the other hand, do benefit from good *average timing behaviour and resource utilisation* (property 3). These jobs do not attempt to guarantee that even the worst-case situations can be handled, but accept a slight performance degradation to substantially lower the amount of resources that is claimed. A closely related factor is the ability to handle the various kinds of *data-dependencies* (property 4): execution times of tasks may vary and conditionals and data-dependent iterations may be present in the graph, so the scheduler has to make certain run-time decisions. Taking these decisions cleverly may improve the average behaviour of the job. Finally, another important property of certain scheduling methods is the possibility to tightly estimate the behaviour of the running job by analysis during compile time. This *offline predictability* (property 5) can help to determine adequate resource budgets. A short comparison of the various scheduling methods on these points is presented below.

2.5.1 Fully-dynamic scheduling

In fully dynamic scheduling all scheduling decisions are made during run-time. This means that there has to be a scheduler in the system that has knowledge about the job's task graph and the available platform resources at any given point in time. In this way, any task's execution can be assigned to any available processor in the system, so it can potentially balance the load optimally: it has the best average-case timing and resource utilisation when neglecting overhead. Also, fully dynamic scheduling is able to adequately handle non-determinism and data-dependent behaviour. Its huge overhead of calculating the schedule and enforcing it (global resource management, moving actors from processor to processor, etc.) is obviously its greatest drawback.

2.5.2 Static-assignment scheduling

Static assignment (or static allocation) scheduling fixes the task-to-processor assignment before starting the job. At run-time, it remains to be decided when to execute which task. When the task processing times are varying, a flexible task execution order can potentially yield a better average timing behaviour compared to the more static methods, because any enabled task can possibly start, so also the one that leads to the best behaviour. Especially for soft real-time jobs, this is an advantage. However, when multiple tasks are enabled at a certain point in time, a choice has to be made about which task to execute first. Moreover, the worst-case performance of this method could be worse than the more static methods, because of bad run-time scheduling decisions. This may be a disadvantage for hard real-time jobs, as for these jobs the worst possible behaviour has to be taken into account. Also, the

run-time scheduling overhead is considerable, which affects the resource utilisation of the job. Lastly, static-assignment scheduling is very well capable of dealing with data-dependent task executions, as scheduling decisions can be made based on the availability of data. Still, it may be possible to derive bounds on the behaviour of the system, at design time.

Various methods have been developed to tackle the problem of making run-time choices. Examples are Round Robin and Time Division Multiple Access (TDMA). Round Robin scheduling is the easiest (and so least expensive) and most straightforward method. It checks each task consecutively; if a task is enabled, it is executed immediately. TDMA uses a fixed period (time wheel) of which each task gets a predefined time slot. During this time slot, a task has the opportunity to execute until the end of the slot. When the task is not enabled, because it has no data, the processor is idle for the remaining time in the slot. TDMA may be used in combination with pre-emption, to allow the execution of tasks to span over multiple time slots. The power of this method is that the execution of tasks (or jobs) can be decoupled, as if they were running on separate processors: if two tasks do not depend on each other's data they cannot influence each other's timing. Pre-emption, however, needs additional hardware and introduces task-switching overhead.

2.5.3 Static-order scheduling

The third type of scheduling, static-order scheduling, fixes not only the task-to-processor assignment, but also, per processor, the order in which tasks are executed. A static-order schedule is valid when it has an execution order for all tasks, possibly on different processors, so that all precedence constraints are met and one cycle of the schedule returns the job's task graph to its original state. A strong point is that this method requires little run-time overhead: since tasks are executed in a fixed order, they do not have to be compiled as separate pieces of code, but they can be compiled as one program to let the compiler handle the synchronisation. Also, its worst-case timing behaviour and resource utilisation are usually better than for the previously named methods: as the execution is restricted, bad extreme cases can be avoided. Especially for hard real-time jobs this is an advantage. For the same reasons, jobs that are scheduled in this way, are very well predictable at design time. However, compared to static-assignment scheduling, static-order scheduling is less well capable of dealing with data-dependencies. Varying task processing times can be dealt with, but it is not allowed to adapt the schedule at run-time, when another execution order seems to be better. For soft real-time jobs, this may be undesirable. Conditionals or data-dependent iterations are not at all possible, as they always require flexible execution orders.

2.5.4 Fully-static scheduling

When a job is fully-statically scheduled, this means that every task execution is fixed before running the job: every task is assigned to a processor and also the execution order and exact start times are predetermined. Because of this, the timing behaviour of such a job, as well as its resource usage, is very well predictable at design time. Also, the run-time overhead is minimal: the schedule is based on a table of task start times that is stored in the memory or the various tasks are compiled into sequential code per processor. The obvious drawback is the inflexibility of the method: no data-dependent behaviour at all is possible. The processing time of tasks may be shorter than defined in the schedule, but the time that becomes available cannot be used for other tasks. Also, the method is not robust, as a task that runs a little

longer than allowed, may ruin the schedule and mess up the jobs execution. Actually, in [8] it is argued that the performance of a static-order schedule that is derived from a fully-static schedule in which the fixed task start times are released, never performs worse than this fully static schedule. Because of this, and since static-order schedules are more robust, they are preferred over fully-static schedules.

2.5.5 Quasi-static scheduling

The last type of scheduling, the quasi-static method, is similar to fully-static scheduling, but some run-time decision making is possible to support conditionals or data-dependent iterations. This method is extensively discussed in e.g. [9] and [4]. Although these schedules can work with data-dependent constructions, the execution of jobs that contain very flexible processing times can be very poor, because of the fixed nature of the method. For very regular hard real-time jobs it may be a good options, but definitely not for soft real-time jobs.

2.5.6 Conclusion

A summary of the comparison of the various scheduling methods on the five properties is given in Table 2.1. An overall conclusion for the type of jobs that are considered in this work – soft and hard real-time jobs that may contain a lot of data-dependencies – is as follows. Fully dynamic scheduling is too unpredictable (there is a large difference between average-case and worst-case behaviour) and too expensive in terms of run-time overhead to be practically useful. The quasi and fully static methods are not flexible and robust enough for this type of jobs. Especially for soft real-time jobs, static-assignment scheduling seems the most appropriate, because of the possibly better average case timing behaviour. For hard real-time jobs that contain no dynamic constructions, static-order scheduling is a good option, since it requires very little run-time overhead and has the best worst-case behaviour. Even if a hard real-time job contains dynamic constructs and a static-assignment schedule has to be used, tasks within such a construct could be statically ordered. The use of these scheduling methods is worked out in more detail in Section 3.5.3.

Table 2.1: Comparison of scheduling methods

	Run-time overhead	**Average case**	**Worst case**	**Data-depen-dencies**	**Predicta-bility**
Fully dy-namic	−−	+	−−	++	−
Static assignment	−	+	−	+	+
Static order	+	+/−	+	+/−	++
Fully static	++	−	+	−−	++
Quasi static	++	−	+	+/−	++

As said earlier, if multiple jobs run on a single platform, they are required to be completely independent: their timing behaviour may not be influenced by other jobs. To enforce this, resource budgets are introduced for each job. A suitable way to enforce fixed budgets on processing resources (processors), is to use TDMA scheduling, a form of static-assignment

scheduling. When a job is given fixed time slots on the various processors, the job's tasks can be scheduled in any way inside these time slots, without affecting other jobs. As is argued later in this thesis, Round Robin (which is also static assignment) scheduling is generally more efficient than TDMA, so it is suggested that the tasks inside the job's time slots are Round Robin scheduled for soft real-time jobs, or Round Robin or static-order scheduled for hard real-time jobs. This combined TDMA/Round Robin/static-order approach is also worked out in Section 3.5.3.

2.6 Conclusions

The concepts of the multiprocessor platform template and applications are explained in this chapter. Two levels of dynamism are defined: dynamism within the boundaries of a task, and dynamism on the level of the job as a whole. This work considers three types of dynamism: data-dependent processing time of the task (a kind of task level dynamism), conditionals and data-dependent iterations (both job level dynamism). It is also shown what is the status of the existing work and which gaps are filled by the work in this thesis. As dynamic behaviour in jobs could not yet be dealt with in a predictable way, the approach that is introduced in this thesis can be a useful addition. Also, another application of this work is suggested: mode changes. Finally, an overview of scheduling methods for tasks on processors is presented. A combined TDMA/Round Robin/Static-order approach is suggested for the purpose of the enforcement of processing budgets and efficient and flexible scheduling of hard or soft real-time jobs.

Contributions of this chapter are:

- The identification of three types of dynamism that can occur in a job, based on the definition of two levels of dynamism. This result corresponds to Goal 1.
- A classification of jobs based on the type of deadline and the level of dynamism.
- A comparison of scheduling methods and recommendations of which methods to use.

Chapter 3

Job design

3.1 Introduction

3.1.1 Overview

The focus of this work is about the design of multimedia applications, or more specifically, the design of independently operating parts of an application, called jobs. In Chapter 2, a multiprocessor platform is described. This chapter shows how a specific model of computation, called Synchronous Dataflow (SDF), can be used to design jobs for this platform. Already at design time the behaviour of the job, as it will run on the platform, has to be fully predictable. This is firstly, because the job's throughput has to meet a certain real-time constraint. Secondly, the job's resource usage should be known, to arrive at an efficient mapping and a correct resource budget assignment.

This introduction continues with a short overview of models of computation that are often used in this field. Subsequently, Section 3.2 proposes a design flow, based on SDF, that is used as a framework for the development of a job that satisfies its throughput contraints, while its resource usage is minimised. Next, Section 3.3 gives the details of the SDF model; it explains the semantics of the model and ways to analyse it to predict the timing of the job that it describes. The succeeding section makes the link to the multiprocessor platform: the features of the platform are modelled, and it is explained how a job that is specified as an SDF graph, can be extended to accurately reflect the behaviour of the job as if it is running on the platform. Section 3.5 focusses on the details of these extended SDF graphs. In the final section, the results are summarised.

Existing work, mainly about the general SDF model and its analysis and about incorporating platform details in SDF graphs is used and extended at some points. Another contribution is the description of a design flow, which serves as a structured framework for the usage of SDF to reflect the behaviour of a job that runs on the platform. It is also explained what the shortcomings of the SDF model are; the next chapter proposes an improved model that can be used in the same design flow, but which potentially leads to more efficient implementations.

3.1.2 Models of Computation

A job can be represented by a task graph: a network of computational tasks (nodes) and data links (edges), which is especially suitable to express task-level parallelism in the job. Various models of computation exist that all use task graphs, but may have different semantics and

usage. Examples are Kahn Process Networks (KPN) and Dataflow Graphs (DFG).

The nodes in a KPN [11] are called processes and represent (continuous) functions on streams of symbols; output streams are produced, based on the complete input history. The edges represent FIFO connections with unbounded buffering. When a process has insufficient input data, it blocks until new data is available. Because of their definition as functions, KPN processes are fully compositional: they can be combined and nested in any way, without changing their behaviour. Timing information, however, is hard to incorporate in a KPN, because its processes are continuous functions on data streams and do not contain operations that can be assigned a duration.

General Dataflow Graphs (DFG), also known as Dynamic Dataflow (DDF) or Dataflow Process Networks [17], have so-called actors to represent tasks and arcs that model data dependencies between actors. Arcs are FIFO channels, which can hold unlimited amounts of data tokens. Actors can fire (execute) in several ways. Firing rules specify when an actor can fire and what happens in this case. A firing rule can become enabled, when a certain pattern of input data is available; if this happens, the actor fires as prescribed by this rule. An actor is idle, until enough data on its inputs is available to enable a firing rule. Every firing is an atomic operation, which means that it can never be interrupted. If more than one firing rule is enabled at the same time, it is unspecified which type of firing is executed. This is a way to express non-deterministic behaviour. A certain processing time can be associated with a firing, which enables reasoning about timing of the job that the graph describes.

A special, restricted case of a DFG is the Synchronous Dataflow (SDF) Graph [16], which is deterministic like KPN and offers design-time predictability of its timing. This makes SDF very suitable to model real-time applications. An SDF consists of dataflow actors ard arcs, but they can have only one firing rule: per firing, they produce and consume a fixed number of tokens. Because of this property, an SDF graph is said to have *fixed rates*. This makes SDF deterministic and predictable. SDF is a semantical descendant of DFG, but at the same time it is also a subset of KPN.

3.2 Design flow

In this section, a design flow is introduced that is used as a guideline in this thesis. This flow is visualised in Figure 3.1 and explained below. It is not argued that this is the best design flow possible in the sense that it is a very efficient algorithm and that it always leads to optimal results. It is merely introduced to show in what way dataflow models can be used in a design flow for the given type of platform. Better strategies can be developed in the future, based on these ideas. Another design flow, based on a related model of computation, called Cyclostatic Dataflow, is described in [13].

A job is **specified** using the SDF model. An SDF graph consists of several actors, which may have precedence restrictions (arcs) that constrain the order in which the actors can be executed (fired). Every task of the job is represented by an SDF actor. This model is useful to represent the task-level parallelism in a job and provide clear semantics about its execution and can be annotated with timing information. SDF is further discussed in the next sections. It is assumed in this work that a task graph of a job is given. Converting an arbitrary job description into SDF, however, is only possible under certain conditions. This is a complicated issue on its own and lies out of the scope of this work.

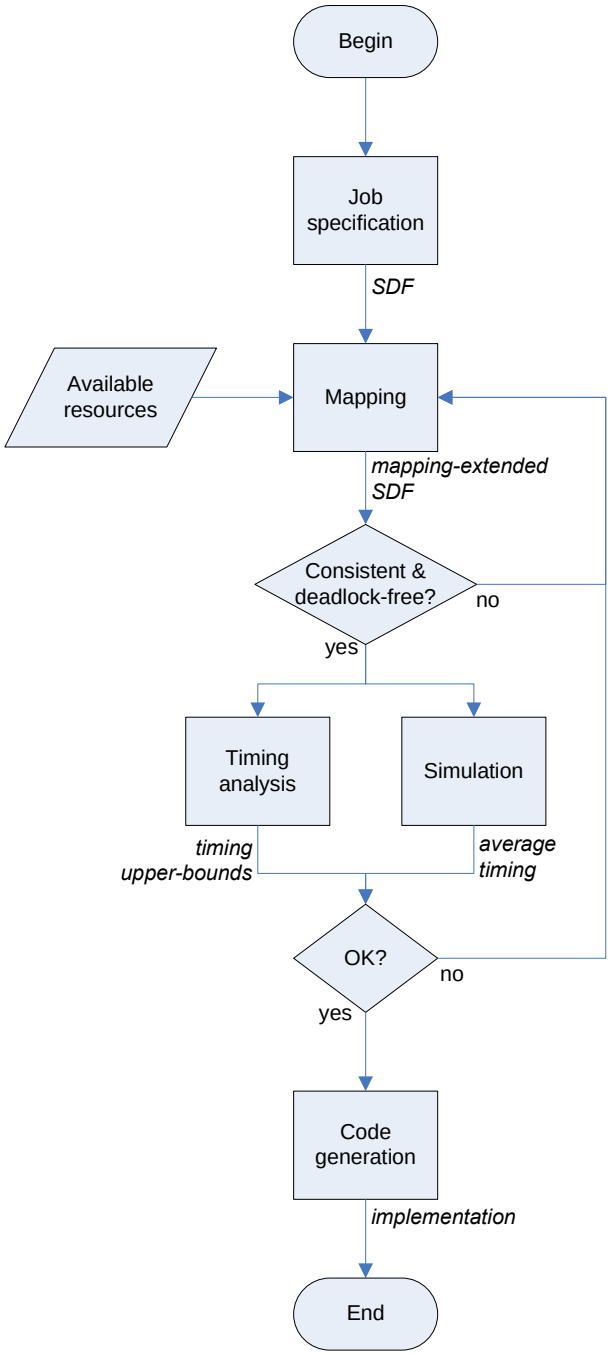


Figure 3.1: Design flow

Given an SDF graph and the platform resources that are available, a **mapping** has to be found. Because the main design goal is to meet the timing constraints and minimising the usage of resources only has the second priority, it is a good idea to see what the timing will be when the maximum, so least constraining, amount of resources that can be offered by the platform is used. This leads to an estimation of the best possible timing for the job. In further design iterations, various more efficient mappings can be tried. Determining a good new candidate mapping is also a very specific topic, which is not extensively investigated in this work.

When a job's SDF graph is annotated with worst-case actor response times and other mapping information (like FIFO sizes) is incorporated in the graph, the result is called a *mapping-extended graph*. This graph can easily be **checked** to guarantee that no memory leaks exist (**consistency**) in the eventual implementation and that the job does not **deadlock**.

Next, it can be **analysed** to acquire a conservative estimation of the minimum throughput that can be guaranteed. This is important information if the job has (hard) real-time constraints. Timing analysis means that, without actually executing or simulating the job, the desired characteristics can be derived by algorithms. Timing analysis generally does not use information about the job's input streams; its results are upper-bounds for any possible input, which makes it suitable to be used for hard real-time jobs. The main part of this chapter and Chapter 4 deals with analysis. In case of soft real-time jobs, it is possible to use the mapping-extended SDF graph for **simulation** to obtain average-case timing information and estimations of the number of deadline misses, for a pre-defined set of input streams. Simulation can also be used to derive the behaviour of the job in the initial, "transient" phase of a job's execution, which is hard to do analytically. Simulation generally takes more time than analysis and is only reliable when a possibly wide, representative range of input streams is used. A simulator has been implemented and is discussed in Chapter 5. An important demand for both analysis and simulation is that their timing results are *conservative*, which means that the consumption and production of data samples always occurs at the same time or later as in the real implementation, but never earlier. This means that the worst possible behaviour should be incorporated to give guarantees about the timing of the eventual implementation.

The results of analysis and simulation can be compared to the constraints. If it appears that the **timing requirements** are not met, a different mapping will have to be tried. If the constraints are met, the job could be implemented according to the derived resource budgets and mapping (**code generation**). However, another reason to try a different mapping is to see whether the constraints can also be met with less resources. Thus, the loop in the design flow attempts to satisfy the timing constraints, while minimising the allocated amount of resources. This is a possible approach to solve the optimisation problems described in Section 1.2.

When a job is implemented according to its SDF specification, i.e. the implemented job follows SDF semantics, some important properties are guaranteed: the job will have bounded timing (latency and throughput) and bounded memory usage. Note that the SDF model is not used to model a given implementation of a job. Rather, it is used as a specification scheme to arrive at a fully predictable implementation: if the building blocks are used, the named properties are guaranteed. It is possible, however, that the implementation is more efficient if it releases the strict SDF rules. But then the system is not predictable anymore and no real-time guarantees can be given. It depends on how appropriate the SDF model is

for a given job, how efficient its implementation will be. For this reason, an extension to the SDF model to incorporate dynamic behaviour is developed. This extension guarantees the same properties as the original SDF model, but it can express more.

This model-driven design method is used in a similar way in the Model Driven Architecture (MDA) framework [18]. The specification-level SDF graphs are similar to the so-called Platform Independent Models (PIM) in MDA and the mapping-extended SDF graphs remind of the Platform Specific Models (PSM), from which the implementation code is generated. This approach also uses formal models to specify applications, in order to ensure correctly behaving implementations and portability to various platforms.

The next section explains why the SDF model is suitable to be used in the described design flow. The next chapter dives into constructions that can serve as extensions to SDF, while the same design flow is still applicable.

3.3 Synchronous Dataflow (SDF)

3.3.1 Model requirements

When a job is designed, it is desired to be able to predict its behaviour when it will be running on the multiprocessor platform. First of all we want to be sure that it is functionally correct, but we also need to know how much platform resources the job will need to be able to satisfy its timing specifications. With this knowledge, we can accurately reserve resources (budgets) for this job on the platform. Thus, needed is a job specification model that not only represents the correct function of the job, but can also be analysed offline to acquire knowledge about the timing and resource utilisation of the job on the multiprocessor platform.

Upper-bounds for the following run-time properties should exist, when an implementation is made according to the semantics of the model:

- Timing: latency and throughput of the job
- Memory usage: for tasks and communication buffers
- Communication bandwidth: between tasks on different processors (inter-processor communication delay)

Furthermore, it should be possible to obtain these bounds for a given job, by analysis of the model. By using the Synchronous Dataflow (SDF) model of computation, it is possible to guarantee bounds on the three properties. This section explores how. This thesis mainly focusses on timing analysis; it is explained in detail how timing bounds can be obtained from an SDF graphs. It is possible to calculate bounds on memory and communication as well, but here it is only shown that these bounds exist and hints are given for how to calculate them.

The following section first explains the Homogeneous SDF (HSDF) model, for which the named properties have been proven in literature. Next, it is argued that any practically useful (i.e. consistent, explained below) SDF graphs can be transformed in an HSDF representative, so the derived properties also hold for SDF.

3.3.2 Homogeneous Synchronous Dataflow (HSDF)

An HSDF graph (see e.g. [23]) is a directed graph, consisting of *actors* that are connected by *arcs*. Actors represent processing tasks and arcs represent data-dependencies between actors. Actors communicate by means of data units called *tokens*. Arcs may hold an unlimited number of tokens. Actors are able to *fire* (execute), which means that they consume exactly one token from each of their input arcs and produce exactly one token on each output arc. An actor can only fire when enough input tokens are present; if so, it is said to be enabled. When input tokens for n firings of a certain actor are available, the actor can fire n times concurrently. Arcs may hold a number of *initial tokens* that are already present when the graph is created, in order to enable a first set of actors. Thus, an HSDF graph is defined as follows:

DEFINITION 3.1 (HSDF GRAPH) *An HSDF graph G is defined by the tuple (V, E, d), where*

- *V is the set of actors (graph nodes)*
- *$E \subseteq V \times V$ is the set of arcs (graph edges)*
- *$d : E \to \mathbb{N}$ is a function that assigns a number of initial tokens (delays) to an arc $(u, v) \in E$ ($\mathbb{N}$ includes 0)* □

HSDF graphs are drawn like other dataflow graphs by using circles for actors and arrows from producing to consuming actors to symbolise arcs. Initial tokens are denoted by dots on an arc. An example is given in Figure 3.2.

Executing an HSDF graph means continuously firing actors that are enabled. This may happen in any desired order, which offers a degree of scheduling freedom when implementing the graph as a real system. The so-called *self-timed* execution of an HSDF graph is an execution in which every actor is fired as soon as it is enabled. If too few initial tokens are present in an HSDF graph, it will *deadlock*, meaning that no actor is able to fire after some point in time during an execution. This usually implies an error in the design. In [16] a systematic method for checking for deadlock is given for SDF graph. As HSDF is a special case of SDF (see the next subsection), this method also works for HSDF.

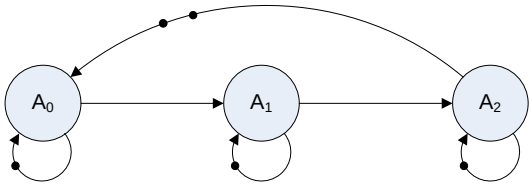


Figure 3.2: HSDF example

A deadlock-free HSDF graph can be used to model a job, such that it can be guaranteed that the timing and utilisation of all resources is bounded. The proof of these properties follows below. To relate HSDF to a job, actor firings are annotated with functions that have a response time (also see Section 3.4) and tokens are associated with values.

HSDF actors are *deterministic*, which means that the value of a token that is produced by an actor, is only defined by the value of tokens that are consumed by that actor during

the same firing. There is nothing else (like time or an actor's state) that influences an actor's output. As all actors of an HSDF graph are deterministic, every HSDF graph as a whole is also said to be deterministic.

One inconvenience of HSDF (and also SDF) is that it is not straightforward to model connections to the outside world. Only arcs between two actors are possible; arcs to or from outside are not allowed. For this reason, inputs and outputs to the job described by an HSDF graph are often just omitted. Still, some actors can be appointed as input and output actors of the job. When analysing or simulating the system's behaviour, it is assumed that there is always enough input data available for the input actors to fire and that there is always enough free space in the output buffers for output actors to store their produced tokens. The underlying assumption here is that it is always the job and not the environment that is the bottleneck, so this approach can be used to find out what the job's limits are. If the interaction of the job with the environment needs to be investigated, the environment could be modelled by special actors that are connected to the input and output actors of the job [2].

The most important timing-related property for streaming real-time applications is throughput. This property only makes sense in relation to the environment of the job. Below it is explained that the self-timed execution of an HSDF graph will always eventually enter a periodic phase. Given this fact and following Definition 2.1 on page 7, throughput for HSDF is defined as follows:

DEFINITION 3.2 (THROUGHPUT OF AN HSDF GRAPH) *The throughput of an HSDF graph is equal to the average number of tokens per time unit that a certain pre-selected output actor produces to the environment, when the self-timed execution of the graph has entered the periodic phase.* □

An HSDF graph has three timing-related properties that make HSDF suitable to model jobs that have to be predictable: *monotonicity*, *periodicity* and *boundedness* [20]. These properties have been proven for HSDF graphs that are *strongly connected*, meaning that for any two actors there exists a directed cycle that contains both actors. For an HSDF graph that as a whole is not strongly connected, but consists of multiple strongly connected components, the properties still hold per component. It is then possible to draw similar conclusions for the whole graph. Another restriction is that the HSDF graph has to be a FIFO graph: tokens cannot overtake each other in actors or arcs. Therefore, arcs have to be defined as FIFO buffers between actors. Actors can behave in a non-FIFO way, when their response times vary over different firings: multiple firings could overlap in this case. A sufficient condition to fix this, is to make sure that every actor is contained in a cycle with only one initial token on it. This could be a cycle that solely contains this one actor (a so-called self-loop). In such a cycle, none of the actors can fire (auto)concurrently. The HSDF graphs that are mentioned in the remainder of this section, are assumed to abide by these rules, unless stated otherwise.

The first property, monotonicity, says that in the self-timed execution of an HSDF graph, increasing actor response times can only result in non-decreasing start times of actor firings in the graph. This property implies that when worst-case response times of actors are used to annotate actors, the self-timed execution of an HSDF graph will lead to worst-case timing results. Consequently, since conservative timing results are desired (Section 3.2), worst-case response times, or at least upper-bounds, should be used.

Next, periodicity means, that the self-timed execution of an HSDF graph that is annotated with worst-case response times, will eventually enter a periodic regime. This implies that a

certain state of the graph returns periodically. The state of an HSDF graph is defined as a certain token distribution over the arcs of the graph; the value of tokens is ignored, only the number of tokens per arc is taken into account. Suppose that every actor $v \in V$ of a certain HSDF graph is annotated with a (fixed) response time $R(v)$, the time it takes to complete one particular firing. The periodicity is expressed in Equation 3.1, in which $s(v, k)$ denotes the start time of actor v in firing k.

$$s(v, k + N) = s(v, k) + \lambda \cdot N \tag{3.1}$$

For any HSDF graph, a certain natural number K exist, for which Equation 3.1 is valid for any firing $k \geq K$. The time-span of one period is equal to $\lambda \cdot N$; actor v fires N times within this period. The values of λ and N are equal for every actor in the graph. Thus, conform Definition 3.2, the factor λ is equal to the inverse of the average throughput of the graph in the given period, during the periodic phase of the self-timed execution of the graph. The value of K specifies the duration of the transient phase: the phase before the execution reaches the periodic phase. Unfortunately, K is not easily computable, so it is generally unknown how long it takes until the execution of an HSDF graph reaches its periodic phase. Simulation could be used to obtain timing information about the transient phase of the execution. This periodicity property together with the monotonicity, implies that for a graph that is annotated with upper-bounds on the actor response times, $\frac{1}{\lambda}$ is a lower-bound on the throughput of the job. The underlying assumption here is, as mentioned earlier, that the throughput of the job is not constrained by its environment: input data from outside is always available and there is always enough space to store output tokens.

The cycle mean (CM) of a simple cycle c (a cycle, in which no actor occurs multiple times) in a graph is defined by Equation 3.2: the sum of the response times $R(v)$ of the actors v on c divided by the total number of initial tokens $d(e)$ on the arcs e of c. The Maximum Cycle Mean (MCM) of a strongly-connected HSDF graph G, as given by Equation 3.3, is the maximum CM of all simple cycles c in G. The MCM is equal to λ and is therefore a measure for the throughput [23]. Suppose the graph consists of multiple strongly connected components, the MCM of the whole graph becomes the maximum over the maximum cycle means of all these components. A simple cycle that is responsible for the maximum cycle mean, is called a *critical cycle*. The critical cycles are the "weakest links", the bottlenecks in the job, that are constraining the throughput. Therefore, measures to improve the throughput should focus on critical cycles. Note that Equation 3.1 states that an actor fires N times during the period. The value of N can be derived from the number of initial tokens on the critical cycles [2]. To calculate the MCM of a graph, a polynomial algorithm exists [6].

$$\text{CM}(\text{simple cycle } c) = \frac{\sum_{\text{actors } v \text{ on } c} R(v)}{\sum_{\text{arcs } e \text{ on } c} d(e)} \tag{3.2}$$

$$\text{MCM}(G) = \max_{\text{simple cycle } c \text{ in } G} \text{CM}(c) \tag{3.3}$$

Suppose the following response times apply to the actors of the example in Figure 3.2: $R(A_0) = R(A_1) = R(A_2) = 1$. There are four simple cycles in this graph: the self-loops around each of the actors, each having a cycle mean of 1, and the cycle through all three actors, having a cycle mean of $\frac{3}{2}$. Therefore, the latter cycle is the critical cycle and the MCM equals $\frac{3}{2}$. The value of λ is equal to this MCM and the factor N is equal to 2 in

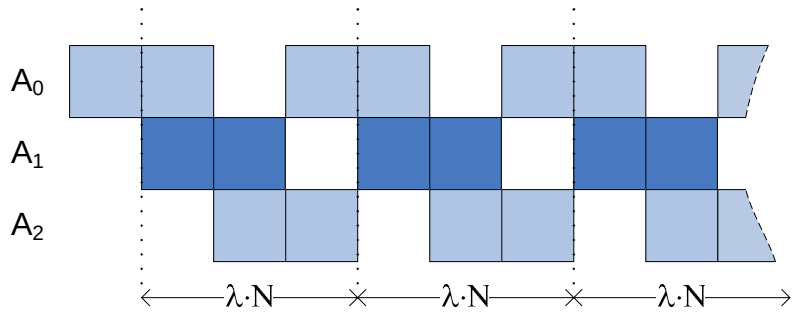


Figure 3.3: Self-timed execution of example in Figure 3.2

this case, a result from the two initial tokens on the critical cycle. Thus, the period $\lambda \cdot N$ equals 3. To illustrate what this means, the self-timed execution of this example is depicted in Figure 3.3. It is apparent from this figure that the periodic phase, with the derived period of 3, starts after a short transient phase of one firing of actor A_0 ($K = 1$ for this actor). Suppose actor A_2 is the output actor of the system. It is clear that this actor fires twice every three time units ($= \frac{1}{\lambda}$) and hence, the throughput of the job according to Definition 3.2 equals $\frac{2}{3}$ tokens per time unit.

The last HSDF property that was stated in [20], is boundedness. This property ensures that for any $I(> K)$ firings of the actors in an HSDF graph, the time until all actors finish executing is bounded. This is actually the timing property that was desired in the previous subsection: it has now been proven that timing bounds on the execution of HSDF graphs exist and it has been shown how the minimum throughput can be calculated.

Boundedness of memory also follows from the periodicity property. In the periodic phase, every actor in a certain strongly-connected component of the graph, fires N times per period, so exactly N tokens are produced and consumed on every arc in this component. This means that the memory usage of a strongly-connected component is always bounded. Graphs that are not strongly connected, can easily be modified by added extra edges, in such a way that they become strongly connected and thus bounded in memory. This should be done in such a way that the graph is still consistent and deadlock free. This leads to the following straightforward approach. An arc between two actors, can be turned into a bounded FIFO buffer by adding an extra arc (containing initial tokens) back from the consuming actor to the producing actor (see Section 3.4). Using this construction to model every FIFO buffer in the job, turns the job's HSDF graph into one big strongly connected component that is bounded in memory usage. The total number of initial tokens in the graph then represents the total FIFO capacity that is needed in the memory.

Also inter-processor communication can explicitly be in corporated in HSDF graphs, by adding extra actors and arcs [1, 19, 20]. Since the timing of any HSDF graph is bounded, as explained above, also the inter-processor communication delays are bounded, when incorporated in an HSDF graph.

3.3.3 Synchronous Dataflow (SDF)

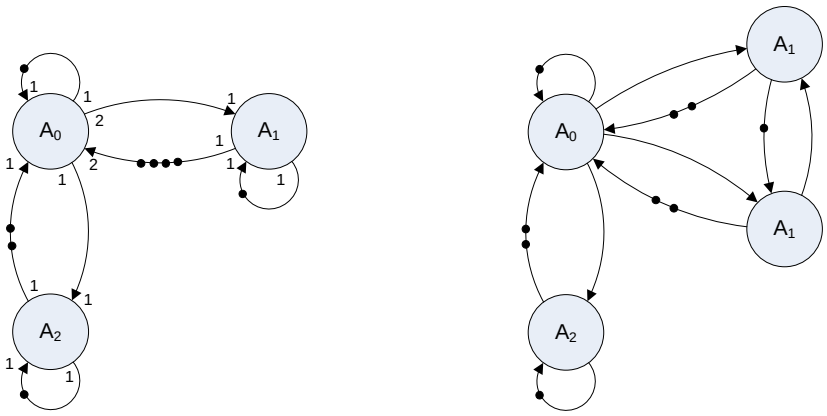
The SDF model was introduced by Lee and Messerschmitt in [16]. SDF is a generalisation of HSDF: it is defined in the same way, with one exception: actors can transfer multiple tokens to and from incident arcs, per firing. However, these *rates* have to be fixed numbers. Because

of this, SDF graphs are more suitable to represent multi-rate jobs in a compact manner. As explained below, any practically useful SDF graph can be converted to an HSDF equivalent, so all necessary properties that were desired in Section 3.3.1 are also valid for SDF. An SDF graph is now defined as follows:

DEFINITION 3.3 (SDF GRAPH) *An SDF graph G is defined by the tuple (V, E, d, I, O), where*

- *V is the set of actors (graph nodes)*
- *$E \subseteq V \times V$ is the set of arcs (graph edges)*
- *$d : E \to \mathbb{N}$ is a function that assigns a number of initial tokens (delays) to an arc $(u, v) \in E$*
- *$I : E \to \mathbb{N}^+$ is a function that defines the number of tokens that actor v consumes from arc $(u, v) \in E$ in one firing*
- *$O : E \to \mathbb{N}^+$ is a function that defines the number of tokens that actor u produces on arc $(u, v) \in E$ in one firing* □

The numbers that the functions I and O assign to arcs are the rates. Rates are written at the points where arcs enter or leave the actors. An example is given in Figure 3.4(a). Note that an arc can have only two actors connected, but, as a shorthand, in SDF diagrams arcs are split into multiple arcs when convenient. Implicit copy-actors are present at these split points, which simply copy incoming tokens to all outgoing arcs.



(a) SDF (b) HSDF

Figure 3.4: SDF example (a) and its HSDF representative (b)

For an SDF graph to be useful, it needs to have *consistent* rates. When this is the case, the graph can be converted into an equivalent HSDF graph [23], so it has the same properties of boundedness of timing and resource utilisation. The self-timed execution of an SDF graph with consistent rates, will eventually reach an equilibrium state, in which all actors are fired periodically and on each arc the same number of tokens is consumed as produced (see e.g. [14]). Inconsistencies can lead to accumulation of tokens on arcs, so it is not possible anymore to give memory or timing bounds. An SDF graph with consistent rates is said to be a consistent SDF graph. To check whether an SDF graph is consistent, one can try to solve

the *balance equations*, which impose the consistency rule (production equals consumption) on every arc. If the balance equations have a nontrivial solution (a solution other than a vector of zeros), the SDF graph is consistent. The possible solutions to the balance equations are so-called *repetition vectors*. Such a vector specifies for each actor the relative number of firings it has per period, in the equilibrium state. For the example in Figure 3.4(a) the repetition vectors $\vec{\rho}$ are as follows:

$$\vec{\rho} = [\rho_{A_0} \quad \rho_{A_1} \quad \rho_{A_2}]^T = k\,[1 \quad 2 \quad 1]^T, \quad k \in \mathbb{N} \tag{3.4}$$

Converting the SDF example from Figure 3.4(a) to HSDF leads to the graph in Figure 3.4(b). It can be seen in this example that actor A_1 has two instances in the HSDF graph; this is a result of the rates of 2 at the arcs (A_0, A_1) and (A_1, A_0). The number of instances of a certain actor in the HSDF representative of an SDF graph, always coincides with the repetition rate it has in the SDF graph's repetition vector.

The throughput of an SDF graph is defined exactly like for HSDF graphs (see Definition 3.2). If λ (see Equation 3.1) and the repetition vector of an SDF graph is known, the throughput τ can be easily calculated by multiplying the repetition rate of the graph's output actor o (ρ_o) with its output rate (called O_{env}) and dividing this by λ:

$$\tau = \frac{\rho_o \cdot O_{env}(o)}{\lambda} \tag{3.5}$$

To calculate λ, the SDF graph can be converted into its HSDF equivalent, of which the MCM (which is equal to λ) can be calculated. Suppose all actors in the example in Figure 3.4 have response times $R = 1$. The critical cycle in the HSDF graph is the cycle comprising the two A_1 actors; the MCM, so also λ, is equal to 2. This may be unexpected, as in the SDF graph it seems that the simple cycle around A_1 only has a cycle mean of 1. The MCM analysis, however, is only valid for HSDF graphs, so this would be a false conclusion. The value of λ that is calculated from the HSDF graph, can be used to calculate the throughput for the original SDF graph using Equation 3.5. Suppose actor A_1 is the output actor of the job, having an output rate to the environment of 2 tokens per firing. In this case, the throughput τ of the job equals 2 (the rate $O_{env}(A_1)$) times 2 (the number of A_1 repetitions ρ_{A_1}) divided by 2 (λ), being 2 tokens per time unit.

3.4 Platform & application

3.4.1 Basics definitions

The multiprocessor platform template and applications, as introduced in Section 2.2, are now related to the SDF model of computation. To do this, the properties of a platform and an application are first defined more precisely.

A platform is defined in terms of its *resources*, of which there are three types: processing, memory and communication. This leads to the following characterisation. A platform is considered to be a possibly heterogeneous set Π of processing elements. All *processing elements* $\pi \in \Pi$ are processors of a certain type, running at a certain clock speed. Processor type and speed are not explicitly modelled; rather, this is reflected in the processing times of tasks, as described below. Every processing element has a certain amount M_π of memory at its disposal and can communicate with every other element through a network. Communication

through the network happens through channels with a certain guaranteed minimum throughput that connect two processors. It depends on the existing network configuration whether it is possible to create a new channel and, if that is possible, how much bandwidth may be reserved. This is not specified in more detail in this work, as it is out of the scope of this work.

A platform is designed for a certain *application*. An application is a set of independently operating *jobs*, each having a fixed resource *budget* on the platform. A certain allocation of resource budgets to jobs is called a *configuration*. TDMA arbitration is used to enforce a job's processing budget on a processor. The processing budget of a job is defined as the assignment of a fraction $0 \leq f_\pi \leq 1$ of the TDMA period on every processor $\pi \in \Pi$ ($\Pi \rightarrow [0,1]$). The memory budget is the amount of memory that is available for the job for each processor: $m_\pi \leq M_\pi$. Finally, the communication budget is a set of channels $C \subseteq 2^\Pi$ each having a predefined bandwidth. Also this is not modelled in more detail, as it is not used in this work. This leads to the following definition of a configuration:

Definition 3.4 (Configuration) *A configuration is an allocation of platform resources to jobs. As long as the configuration does not change, a job is guaranteed to have the following:*

- *A processing budget: a fraction* $0 \leq f_\pi \leq 1$ *on each processor* $\pi \in \Pi$
- *A memory budget: a quantity* $m_\pi \leq M_\pi$ *to be used at each processor* $\pi \in \Pi$
- *A communication budget: a set of channels* $C \subseteq 2^\Pi$ □

A job consists of a set of tasks $\mathcal{T}$ that may have data-dependencies that impose precedence restrictions. A task is an operation that takes a certain time to execute on a processor. A single task cannot be divided over multiple processors; if this is desired, the task needs to be split into multiple separate tasks. A task $t \in \mathcal{T}$ that is scheduled on a processor π has a processing time $p_\pi(t, \vec{d})$ that is dependent on its input data $\vec{d}$. The processing time is measured as the total time it takes to execute the task when it has a full processor at its disposal, and hence is never interrupted, and can start executing immediately. Every task should also have a known upper-bound $P_\pi(t)$ on the processing time, such that $p_\pi(t, \vec{d}) \leq P_\pi(t)$ for every possible input $\vec{d}$. The subscript π is dropped in the remainder of this document, when all processors are identical (homogeneous processing elements) or when it is clear which processor is meant.

Task-to-task communication happens via data packets called *tokens* that are transferred through FIFO buffers (FIFOs), which are mapped in a memory. The set of all *FIFOs* is called $B \subseteq \mathcal{T} \times \mathcal{T}$. The token-size is fixed per FIFO and every FIFO has a fixed capacity: the maximum number of tokens it can hold at the same time, equivalent to the amount of memory that has been reserved.

A job can be mapped to the platform, within the constraints of the resource budgets it has. A mapping is defined as follows:

Definition 3.5 (Mapping) *A mapping of a job to the platform involves:*

- *assignment of the job's tasks to processors by means of the function* $mapT : \mathcal{T} \rightarrow \Pi$
- *selection of the capacity of each FIFO (buffer sizing) by means of the function* $mapF : B \rightarrow \mathbb{N}$*, such that the total needed memory per processor is at most* m_π

- *assignment of channels/bandwidth to FIFOs that cross processor boundaries by means of the function mapC* : $B \rightarrow C$ □

On every processor, within the time slice that is reserved for the job, a certain type of scheduling is imposed, which is also considered to be part of the mapping. To incorporate this in the model, the notion of *response time* (or *latency*) is defined for the tasks of a job:

DEFINITION 3.6 (RESPONSE TIME) *The data-dependent response time* $r_\pi(t, \vec{d})$ *of a task* t *that is mapped to the processor* π *and receives input data* $\vec{d}$ *is defined as the time it takes from when the task is enabled (i.e. when it has all necessary data to execute) until it is finished executing.* □

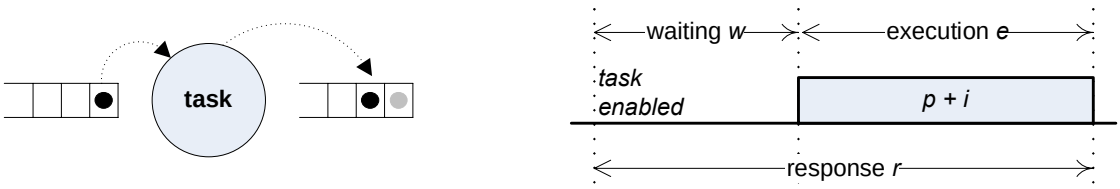


Figure 3.5: Task response time

The subscripts π will again be dropped here, if no confusion can arise. The task may have to wait for the processor to get free before it can start and it can be interrupted several times during its response. This waiting time depends on which actors can be enabled at the same time. Not only the task's processing time, but also the waiting time ($w(t)$) and interruption time ($i(t)$) are included in the response time. Consequently, the response time is highly dependent on the mapping. The response time excluding the waiting time is called the execution time $e(t)$ of a task. All times $r(t, \vec{d})$, $w(t)$, $i(t)$ and $e(t, \vec{d})$ have upper-bounds denoted by capital letters: $R(t)$, $W(t)$, $I(t)$ and $E(t)$. The relations between the various symbols are summarised in Equation 3.6 and visualised in Figure 3.5.

$$\begin{cases} r(t, \vec{d}) = w(t) + p(t, \vec{d}) + i(t) \\ e(t, \vec{d}) = p(t, \vec{d}) + i(t) \end{cases} \tag{3.6}$$

Note that this notion of response time is only defined for tasks and does *not* incorporate latencies in FIFOs. So if the end-to-end latency from input to output is asked, it is not sufficient to simply add the latencies of the tasks. However, because this work is focussing mainly on streaming systems where throughput is the most important constraint, this is not a major limitation.

3.4.2 Mapping-extended SDF

Now, the design method that was described in Section 3.2 can be applied. For this purpose, a job should be specified as an SDF graph: every task should be represented by an actor, data-dependencies between tasks should lead to arcs in the graph and correct rates have to be applied to the ports of all actors.

An SDF graph can be used to analyse the timing of a job and it can serve as a simulation model (also see Section 3.2). In both cases the job's SDF graph has to be extended with

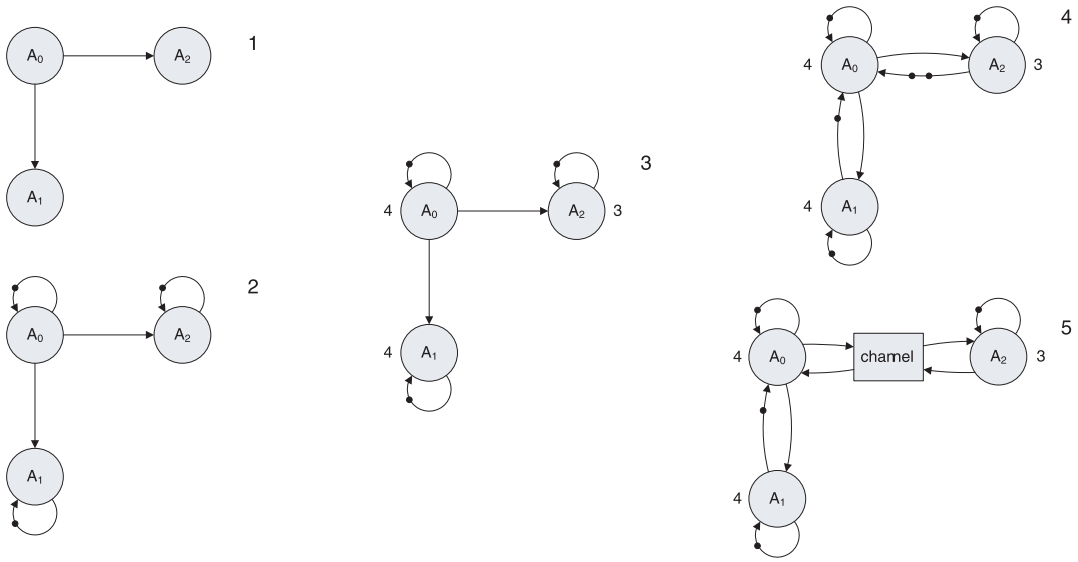


Figure 3.6: Five steps to create a mapping-extended SDF graph

timing and mapping information. The following things have to be taken into account here. The SDF model says that an actor can be fired as soon as all its input data is present (in which case the actor is said to be *enabled*), but also later. The model also implies, due to its inherent timelessness, that an actor can be fired multiple times concurrently, when enough data is available to be processed. The MCM analysis, which is used to compute throughput bounds for timing-annotated SDF graphs, assumes that an actor will be fired immediately upon its enabling – so called *self-timed* execution of the graph – and that multiple instances of an actor can run concurrently. Also simulators may assume a self-timed execution. However, when an SDF graph is used to describe a job that is mapped to the platform this is not necessarily correct, as the platform may impose constraints on the points in time the task can execute.

The following five steps have to be carried out to create an SDF graph that is annotated with timing information and conservatively reflects the behaviour of a job that is mapped to the platform. These steps are illustrated in the example of Figure 3.6.

1. Every task in the job should be represented by an SDF actor; arcs have to be inserted between actors when the corresponding tasks have data-dependencies and correct rates should be applied to the ports of all actors.

2. To make an actor fire sequentially like a task on a processor, it is necessary to add a self-edge with one initial token to every actor. This token may be used to represent the state of the otherwise stateless actor that is transferred to the next firing.

These first two steps coincide with the "Job specification" task in the design flow of Section 3.2. The following three steps are involved in the "Mapping" task. They use additional knowledge about the resources that are available.

3. All actors should be annotated with conservative response times. As explained before, the response time does not only include the actor's processing time, but potentially

also waiting and interruption times to signify static-assignment scheduling of multiple actors on a processor. If static-order scheduling is desired, the graph can be extended with extra arcs to indicate the firing sequence of actors. If the graph is used for timing analysis purposes, the response time annotations have to be worst-case or at least upper-bounds, as generally no knowledge about the data is available. If the model is used in a simulator, the response times can be data-dependent, as long as they are conservative for the given input data.

The resulting SDF graph is a representation of the job that takes into account the assignment of tasks to processing resources and the scheduling of these tasks. It still assumes, however, that the FIFOs have unlimited capacity, as that is the assumption for general SDF. Also inter-processor communication is neglected, so the bandwidth is considered to be unlimited. To also take the memory and communication constraints into account the graph can be further extended in the following way:

4. FIFOs with bounded capacity are modelled by adding an edge from the consuming actor back to the producing actor, with a number of initial tokens equal to the FIFO capacity.

5. Inter-processor communication can be modelled by replacing the simple FIFO model (introduced in the previous step) by a special SDF sub-graph that models the communication over the network. See [19] for more information.

The next section focusses in more depth on the way SDF is used.

3.5 The use of SDF

3.5.1 Modularity and granularity

The SDF-model can be considered *modular* in the sense that any actor in an SDF graph can be replaced by a complete SDF sub-graph, if this does not violate the consistency rules. If this is the case, then the behaviour of the sub-graph and the original actor is, to the rest of the graph, semantically equal. When looking at timing behaviour, however, things may be different, as the sub-graph may include buffering and introduce cycles that influence the throughput. Nevertheless, this property of modularity is useful when creating a specification of a job, because actors can be split and joined to create an optimal structure.

When a job is specified as an SDF graph (the first step in the design flow of Section 3.2), it is very important to cleverly split the job into tasks/actors, to exploit parallelism. That is, it is crucial to choose a suiting actor granularity (actor size). Splitting a job into many small actors offers a lot of flexibility in the assignment of actors to processors and more details about the structure of the job are expressed when the actors are small. This may lead to more efficient resource usage, e.g. smaller FIFOs. However, a large number of small actors has the disadvantage that it may introduce a large scheduling overhead. It is a difficult problem to find the optimal division of a job into actors.

Not only the granularity of actors is relevant, also the size of tokens is of importance. The size of a token can be set per each arc. The reasoning is similar to the one for actors. Splitting a large token into multiple smaller ones may increase the amount of parallelism in the execution (pipelining) and therefore improve the throughput. This may also lead to smaller buffers. Tokens that are very small, however, may also create extra overhead.

3.5.2 Timing of production and consumption

In the original SDF model, timing is not explicit and the model assumes that actor firings are atomic: an actor firing is an instance at which tokens are consumed and produced by the actor. To reason about timing, an actor is annotated with its response time: the time span from enabling until completion. In the implementation, the actor may consume and produce tokens and perform computations at any time during its response. At design time, it is not known in which order or at which exact points in time these events occur. A demand was, however, that the results from timing analysis and simulation should be conservative, so consumption and production of tokens in the implementation happens never later than seen in the analysis or simulation. To reflect this in the model that is used for timing analysis and simulation, it has to be assumed that the consumption and production of tokens always happens as late as possible: at the end of the response of an actor. This is illustrated in Figure 3.7.

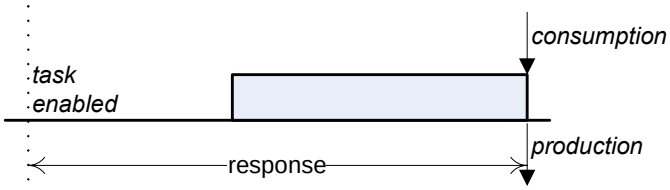


Figure 3.7: Consumption & production

3.5.3 Scheduling models

As described in the previous section, the third step in the process of creating a dataflow graph that represents a job that is mapped to the platform, is adding the characteristics of the scheduling. In this section, it is explained how to make scheduling explicit in the graph. The problem of *finding* a schedule is not discussed here. A main demand is, as said before, that the model is *conservative* when it is used for timing analysis and simulation.

In Chapter 2, several scheduling strategies have been illustrated. Of these methods, *static-assignment scheduling* and *static-order scheduling* are used in the models in this chapter, as they appear to be the most relevant. In fact, when using the dynamic constructs that are described in Chapter 4, static-order and fully-static scheduling cannot be applied anymore and static-assignment schedules become the best choice, as they can cope with the necessary dynamism.

Static-order scheduling is easily incorporated in an SDF model by adding extra arcs that impose the sequence in which actors are fired. Once an actor-to-processor assignment has been made, all actors that go to the same processor need to be in a cycle with a single initial token on it to constrain the order in which they can fire in the model. To find an order, the self-timed order could be used by reusing the existing data-dependencies and adding extra sequence edges to complete the cycles.

Static-assignment scheduling needs another approach, as the firing order of actors on a processor is not fixed, but determined by an arbitration scheme that may take run-time information into account. The scheduling is now incorporated into the graph by introducing upper-bounds for the waiting and interruption times in the graph – depending on the type

of arbitration – which increase the response times of actors. Two types of arbitration are discussed and modelled here: Round Robin and Time Division Multiple Access (TDMA). For TDMA, two different approaches are given. After that, a combined scheme is derived, in which Round Robin arbitration is used inside a TDMA schedule. For all types, it is possible to derive upper-bounds on the response times per actor. The Round Robin and first TDMA scheme appeared earlier in [19]; the second TDMA scheme and the combined method are new.

Round Robin

Round Robin arbitration of actors on a processor is performed by simply making a list of all actors that are executed on this processor and checking them for being enabled in this order. If an actor appears to be enabled it will be fired right away. If not, the next actor in the sequence will get the chance to fire. An actor's execution is never interrupted by other actors. After the last actor in the list has been treated, the sequence is repeated again from the beginning. If the relative frequency of the actors differs, as could be derived from the repetition vector of an SDF graph, it could be advantageous to include multiple entries of the same actor in the schedule list in order to decrease waiting times. Another way is to first convert the SDF graph into an HSDF graph to make all repetitions explicit and schedule this graph.

The waiting time of an actor a that is scheduled on processor π can never exceed the sum of the processing times upper-bounds of the other actors in the Round Robin list, as given in Equation 3.7. The constants k_a indicate the number of times an actor a appears in the list; the other symbols are introduced in Section 3.4. This $W(a)$ is clearly an upper-bound for the waiting time of actor a.

$$W(a) = \sum_{\substack{mapT(\alpha)=\pi \\ \alpha \neq a}} k_\alpha P(\alpha) \tag{3.7}$$

$$r(a, \vec{d}) \leq p(a, \vec{d}) + W(a) \leq P(a) + W(a) \tag{3.8}$$

This implies that if the response time for every actor a that is scheduled on π in the model is taken as the sum of the processing time of a and the waiting time upper-bound $W(a)$, like in Equation 3.8, timing analysis and simulation based on this model will be conservative, even when data-dependent processing times $p(a, \vec{d})$ are used in the simulation. The timing analysis will be based on an SDF model that uses the processing time upper-bounds $P(a)$, because no knowledge of input data is used here. The actors are then annotated with the response time upper-bounds $R(a) = P(a) + W(a)$.

Suppose, for example, that two actors are scheduled on processor π: A_0 with $P(A_0) = 3$ and A_1 with $P(A_1) = 2$. Both actors are scheduled with Round Robin arbitration; actor A_0 appears once in the list, actor A_1 twice. In the worst case, an enabled actor A_0 has to wait for two full executions of actor A_1, before it may fire. Therefore, its waiting time upper-bound $W(A_0) = 4$. A conservative approximation of its response time is then this waiting time plus its own processing time. For the timing analysis, when using response time upper-bounds, this equals 7. Likewise, for A_1's response time $3 + 2 = 5$ is taken.

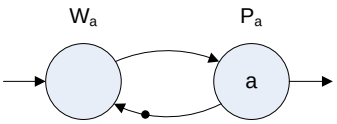


Figure 3.8: Round Robin

Round Robin scheduling can be incorporated in an SDF graph by annotation the actors with the response times specified by Equation 3.8. In [3], a two-actor model is used, as shown in Figure 3.8: one actor for the waiting time and one actor for the processing time. This is equivalent to using only one actor, annotated with the total response time. In [3] it is formally proven that this model, for the case that all k_a are equal to one, leads to conservative timing analysis and simulation, when it is used to extend an (H)SDF graph with scheduling information. This proof can be easily adapted for $k_a > 1$.

Time Division Multiple Access (TDMA)

The TDMA arbitration scheme makes use of a "time wheel" per processor, which spins at a constant pace with a fixed period. For a certain processor π, this period is T. This period is divided into time slots; the position of the time wheel determines which slot is active. Every actor a that is scheduled on processor π gets a fraction $f(a) \in [0, 1]$ of the period, giving it a time slot $T_a = f(a) \cdot T$ in which it can perform its computations. If the actor is still processing at the end of the time slot, it is pre-empted and it has to wait for the new period to resume.

When for every actor an upper-bound $P(a)$ on the processing time is known, one can choose to make the time slot for every actor equal to this $P(a)$ – so the fraction $f(a)$ becomes $(P(a)/T)$ – and prescribe that an actor can only start firing at the beginning of its time slot. In this way, pre-emption is never needed. A model for this is derived in [19]. This model says that an actor a has a waiting time that is always less than the period T: if an actor becomes enabled just after the beginning of its time slot, it will have to wait for (almost) the complete period T before it can fire. This leads to the waiting time upper-bound $W(a) = T$ (Equation 3.9). A conservative approximation for the response time is then given by Equation 3.10. Note that this TDMA response time is exactly a term $P(a)$ larger than the Round Robin case (Equation 3.8) with all $k_a = 1$, as it is included *twice* in the formula. For some $k_a > 1$, it will be even worse.

$$W(a) = T = \sum_{mapT(\alpha)=\pi} P(\alpha) \tag{3.9}$$

$$r(a, \vec{d}) \leq p(a, \vec{d}) + T \leq P(a) + T \tag{3.10}$$

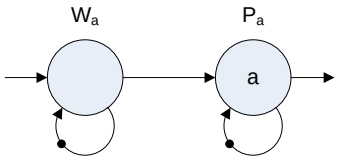


Figure 3.9: TDMA model

The actors in the SDF model for timing analysis and simulation could be annotated with these response times, like in the Round Robin case. This would lead, however, to an underestimation of the throughput. The throughput of an actor that is TDMA arbitrated, is clearly bounded by $\frac{1}{T}$: when sufficient tokens are available in the input FIFOs, this actor can fire every period T. This can be expressed in a pipelined two-actor model like in Figure 3.9. In this model the throughput for this small sub-graph is bounded by the actor that is annotated with the largest duration: the waiting time actor, which has a duration of $T \geq P(a)$ for any a.

A reason to use this limited TDMA scheme, in which every actor's execution "fits" in its time slot, may be to avoid the overhead of pre-emption. In the worst-case, the throughput of an actor is equal to the throughput for Round Robin arbitration. The response time bound, however, is a term P_a longer. Also the average throughput of the system with Round Robin

would probably be better than when this TDMA scheme is used. Both is the case, because when an actor in the TDMA case is not enabled, its time slot is wasted, while in the Round Robin case the next actor can already start firing. Therefore, it seems that this use of TDMA is not very useful in the context of this thesis.

Loosening the limitations to allow an actor to start anywhere in its time slot and span its processing over multiple periods, leads to a different model. The response time is now defined by a more complicated term comprising a waiting time $w(a)$ and interruption time $i(a)$. As shown below, $w(a)$ and $i(a)$ may have different values in certain cases, but that their sum is bounded, as given by Equation 3.11. This leads to the conservative estimation in Equation 3.12 for the total response time.

$$w(a) + i(a) \leq (T - f(a) \cdot T) \left\lceil \frac{p(a, \vec{d})}{f(a) \cdot T} \right\rceil \tag{3.11}$$

$$r(a, \vec{d}) = p(a, \vec{d}) + w(a) + i(a) \leq p(a, \vec{d}) + (T - f(a) \cdot T) \left\lceil \frac{p(a, \vec{d})}{f(a) \cdot T} \right\rceil \tag{3.12}$$

The SDF model for timing analysis and simulation is similar to the one for Round Robin in just one actor, having the response time as given in Equation 3.12. For conservative timing analysis, again $P(a)$ will be substituted for $p(a, \vec{d})$, as no knowledge about the input data is assumed. The correctness of this model can be derived as follows. Assuming that the time-wheel position is unknown and actor a becomes enabled, for any possible firing, there are two possible cases:

1. Time-wheel positioned *outside* time slot of actor a
2. Time-wheel positioned *inside* time slot of actor a

In the first case, the waiting time is at most $(T - f(a) \cdot T)$, the total time in the TDMA period outside actor a's time slot. After this waiting time, the wheel is positioned at the beginning of the slot. The number of wheel periods that are necessary for a to complete its firing is now equal to its processing time divided by the length of its time slot and rounded up:

$$\left\lceil \frac{p(a, \vec{d})}{f(a) \cdot T} \right\rceil .$$

The number of interruptions is exactly one less. The interruption time is then equal to the number of interruptions times the total time in the period outside the time slot:

$$(T - f(a) \cdot T) \left(\left\lceil \frac{p(a, \vec{d})}{f(a) \cdot T} \right\rceil - 1 \right) .$$

This implies that the total time $w(a) + i(a)$ in this case is at most

$$(T - f(a) \cdot T) + (T - f(a) \cdot T) \left(\left\lceil \frac{p(a, \vec{d})}{f(a) \cdot T} \right\rceil - 1 \right)$$

which is equal to the right-hand side of Equation 3.11.

In the second case, the wheel is already positioned inside the correct time slot, so the waiting time is equal to zero. But because it is not known where exactly inside the slot the actor starts executing, it may need an extra wheel period, compared to the former case, to finish its firing. Surely this is the case, if the firing starts just before the end of the time slot. The sum $w(a) + i(a)$ is then bounded by

$$0 + (T - f(a) \cdot T) \left(\left\lceil \frac{p(a, \vec{d})}{f(a) \cdot T} \right\rceil + 1 - 1 \right)$$

which is also equal to the right-hand side of Equation 3.11.

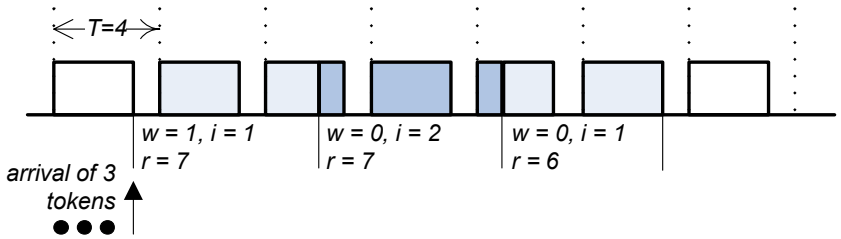


Figure 3.10: TDMA example: $T = 4, f(a) = 0.75, p(a, \vec{d}) = 5$

This result is illustrated by the time shape in Figure 3.10 of the firing of an example actor a that is scheduled with this TDMA scheme. At the point indicated by the arrow, three tokens arrive. The processing time is equal for every token: $p(a, \vec{d}) = 5$. The period T is equal to 4 and the actor is appointed 75% of it. In this example, it becomes clear that the waiting and interruption times vary, even though the processing time remains the same. However, the sum of $w(a)$ and $i(a)$ never exceeds $(T - f(a) \cdot T) \left\lceil \frac{p(a,\vec{d})}{f(a) \cdot T} \right\rceil = (4 - 3) \left\lceil \frac{5}{3} \right\rceil = 2$, and thus the response time is bounded by $R(a) = 5 + 2 = 7$.

The time it takes to handle the pre-emptions is considered to fall outside the actor's time slot. Consequently, an extra time slot needs to be reserved in the time wheel just after the actor's time slot. This will influence the period T that needs to be used. Hence, the pre-emption time is incorporated in the response time equation by means of the period T and the choice of the fraction $f(a)$. Using a larger T, while keeping the fraction $f(a)$ the same, means that a relatively smaller time slot for the pre-emption time has to be reserved, leaving a larger fraction for other actors. Equivalently, this means that fewer pre-emptions are needed. On the other hand, a smaller T generally has a positive effect on the worst-case waiting and interruption times. Consequently, it seems there is an optimum period T that yields the highest throughput. This is, however, highly dependent on the run-time values of $p(a, \vec{d})$, so simulations may be used to arrive at the best period T for certain typical input streams.

Combined TDMA and Round Robin

In the previous chapter the idea was raised to combine TDMA and Round Robin. TDMA can be used to logically separate jobs and to enforce processing budgets: every job gets its own fixed time slots on the processors so their timings cannot influence each other. It appears as if every job runs on its own private set of processors (a virtual platform). Round Robin

(or static order scheduling, if the dataflow is not data-dependent) can be used to efficiently schedule the actors of a job inside their time slots.

The behaviour of such a schedule is easily modelled by combining the expressions for the response times in both cases. The outer level of the new response time upper-bound of a scheduled actor a remains the same as in the TDMA case (Equation 3.13). But instead of using the processing time $p(a, \vec{d})$, the complete Round Robin based response time $r^{RR}(a, \vec{d})$ (Equation 3.14) has to be substituted. This time $r^{RR}(a, \vec{d})$ includes $p(a, \vec{d})$ and the worst-case waiting time for actor a inside the job: the worst-case processing times of all other actors in the Round Robin list. The fraction f of the TDMA period T now denotes the time slot for all actors of the job that run on the currently investigated processor. If $f = 1$ is substituted (the job gets the whole processor), the whole right-hand term in Equation 3.13 becomes zero, so the arbitration is purely Round Robin.

$$r(a, \vec{d}) \leq r^{RR}(a, \vec{d}) + (T - f \cdot T) \left\lceil \frac{r^{RR}(a, \vec{d})}{f \cdot T} \right\rceil \tag{3.13}$$

$$r^{RR}(a, \vec{d}) = p(a, \vec{d}) + \sum_{\substack{mapT(\alpha)=\pi \\ \alpha \neq a}} k_\alpha \cdot P(\alpha) \tag{3.14}$$

These response times $r(a, \vec{d})$ can immediately be used to annotate the actors in the job's SDF graph.

3.5.4 Better worst-case estimations of waiting times

Waiting times are a part of the actors' response times, so they may influence the throughput of the system (if and only if the actor is in a critical cycle). A better estimation of the waiting times may lead to a more efficient resource allocation. In this subsection, it is argued that the worst-case waiting times introduced by static assignment scheduling, can often be estimated more precisely than suggested in the previous subsection. The given examples only use Round Robin scheduling, but similar reasoning also applies to the other methods.

In the worst-case, Round Robin scheduling implies that an enabled actor a may have to wait for the firings of all other actors to complete, before it can fire itself. In other words: the worst-case waiting time for each actor on a certain processor could be equal to the sum of the worst-case processing times of all other actors on this processor. However, because of the precedence relationships between actors (the arcs in the dataflow graph) and the order in which actors are checked in the Round Robin scheme, it can be shown for some actors that they can never be firing when actor a is enabled. The processing times of these actors do not have to be included in the waiting time of actor a.

Take for example Figure 3.11, which shows the part of an SDF graph that is assigned to a certain processor. Assume that these three actors are Round Robin scheduled and checked in the following order: $A_0 \rightarrow A_1 \rightarrow A_2$. To calculate the waiting time of an actor, it needs to be derived which others actors can be firing when this actor becomes enabled. Actor A_0 has an incident arc from outside the processor, so it can be enabled at any time. It could be possible that actor A_1 is firing when A_0 becomes enabled, so $P(A_1)$ has to be included in $W(A_0)$. Actor A_2, however, is in a loop with one initial token together with A_0, so they can never be enabled at the same time. Therefore, $P(A_2)$ does not have to be included in the waiting time, so $W(A_0) = P(A_1)$. Next, actor A_1 becomes enabled always immediately

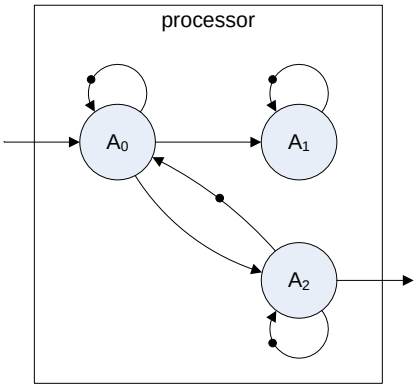


Figure 3.11: Example

after A_0 finishes its execution. According to the Round Robin list, A_1 gets the opportunity to fire immediately after A_0, and so it will. Hence, the waiting time $W(A_1) = 0$. Finally, also A_2 becomes enabled immediately after a firing of A_0, but it will always have to give A_1 the opportunity to fire first. It has already been said that A_0 and A_2 can never be enabled at the same time. Therefore, the waiting time $W(A_2) = P(A_1)$.

This example clearly shows that large savings can be obtained, when following the above mentioned reasoning to determine the waiting times. However, no structured approach that covers all cases has been developed yet. This will be a good topic for further research.

3.5.5 Data-dependencies

The SDF-model does allow one type of dynamic behaviour: data-dependent processing times. However, conditional constructs and data-dependent iterations cannot be explicitly expressed in SDF. It is possible to construct "conditional" paths in an SDF graph, but it is always required that all branches are executed: tokens will flow through all branches at the same time, because the rates are fixed. These branches should all end at one specific actor that passes on the tokens from only one branch and discards the others.

In order to use explicit dynamic constructs, the SDF model needs to be extended. The next chapter discusses another model, which can be seen as an extention to SDF. This model keeps SDF's nice analytical properties and can also express more data-dependent behaviour, which may lead to tighter timing estimations and more efficient implementations.

3.6 Conclusions

This chapter introduces a design flow for the mapping of jobs to a certain multiprocessor platform. This design flow is based on the SDF model and serves as a preliminary framework to apply the analytical techniques that are available for SDF, to jobs of the given platform. The basic SDF model is explained and it is described how the model can be extended for timing analysis. Also, an abstraction of the multiprocessor platform is given, which is used to incoporate platform information in the model. With this knowledge, an SDF graph can be made that reflects the (timing) behaviour of a job that runs on the platform.

However, it seems that, when SDF is used as a specification model for job, as suggested in this chapter, the resulting implementation is not efficient when the job contains a lot of

data-dependent behaviour. Therefore, the SDF model of computation is extended in the next chapter. This is done is such a way, that the design flow and all modelling and analysis methods still apply.

Contributions of this chapter, corresponding to Goal 2, are:

- An SDF-based design flow for mapping a job to the multiprocessor platform, based on the optimisation problems in Chapter 1.2.
- Guidelines for how to construct an SDF graph that reflects the behaviour of a job that is mapped to the platform.
- A new model for TDMA arbitration.
- A new model for combined TDMA and Round Robin arbitration.

Chapter 4

Predictable Dynamic Dataflow

4.1 Introduction

In the previous chapter, a design flow is presented that can be used to find a mapping of a job to the multiprocessor platform. The SDF model is used to specify a job and to express mapping details. However, it is not possible to make dynamic constructions in SDF: data-dependent execution times can be dealt with, but conditionals and data-dependent iterations are impossible to specify. This potentially leads to bad timing estimations and inefficient implementations.

Is this chapter, an extension to the SDF model is proposed, which is capable of handling all types of dynamism, as introduced in Section 2.3, in a guaranteed predictable way. A restricted version of the Boolean Dataflow (BDF) model of computation is used as a basis for two dynamic constructs: the *conditional* and the *data-dependent iteration*. These new elements, together with rules of how to construct graphs with them, form a new model of computation called *Predictable Dynamic Dataflow* (PDDF). Details of the BDF model and how it can be used in a predictable way, forming the basis of PDDF, are given in Section 4.2. Sections 4.3 and 4.4 respectively introduce the conditional and data-dependent iteration constructs and show how they can be analysed to obtain timing information. It is also made clear when and how the use of these constructs becomes beneficial. In Section 4.5, it is argued that it can be useful to represent the internals of large grain actors with a graph of finer grain actors including the new dynamic constructs. The chapter is concluded with a summary.

4.2 PDDF basics

As the PDDF model of computation uses theory from BDF, this model is first explained below. It is made clear how certain BDF constructions can be used in a graph, while the timing and memory usage remains bounded. After that, the basic rules of how to create PDDF graphs are presented.

4.2.1 Boolean Dataflow (BDF)

Lee [14] and Buck [5] described a dataflow model that can be considered as an extension to SDF with the possibility to introduce dynamic behaviour: Boolean Dataflow (BDF). In this model, the actors are like in SDF, but the number of tokens that is produced or consumed

on an arc can be a two-valued function of a control token. Control tokens are produced and consumed just like regular tokens. The BDF model satisfies the Kahn condition, which means that all data streams produced by actors are determinate.

One can define two specific BDF-actors, called SWITCH and SELECT, as follows (see Figure 4.1). The SWITCH has one data input, one control input and two data outputs, labelled TRUE and FALSE. Each firing, it consumes one data token and one control token and forwards this data token to one of its two output ports, depending on the value of the control token. Similarly, the SELECT has two data inputs, one control input and one data output. It consumes one control token and depending on its value it consumes a token from one of its two data inputs, which is forwarded to the data output. The rates of the conditional ports are not constant like in SDF, but dependent on p, the chance that the Boolean control token is TRUE, or $(1-p)$, for the TRUE and FALSE ports respectively.

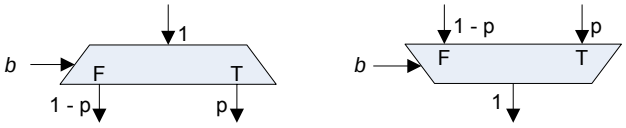


Figure 4.1: BDF SWITCH *(left) and* SELECT *(right) actors*

Lee [14] provides an extended notion of consistency for BDF graphs:

- Graphs, in which nontrivial solutions to the balance equations exist that are not dependent on the rate of the conditional ports (gathered in the vector $\vec{p}$), are called *strongly consistent*.
- Graphs for which solutions exist only for particular values of $\vec{p}$ are called *weakly consistent*.

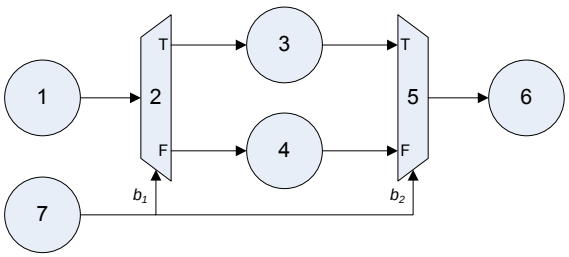


Figure 4.2: Strongly consistent BDF graph: if-then-else construct (all non-conditional rates are equal to 1)

An example of a strongly consistent BDF graph is the if-then-else construct given in Figure 4.2. The rates of the TRUE ports of both the SWITCH and SELECT are equal (p), since the Boolean input streams b_1 and b_2 are equal. The repetition vector has the form as given in Equation 4.1. The actors are ordered in the vector according to the numbering in Figure 4.2.

$$\vec{\rho}\,(p) = k\,[1 \quad 1 \quad p \quad (1-p) \quad 1 \quad 1 \quad 1]^T \tag{4.1}$$

This solution holds for any choice of p and thus the graph is *strongly* consistent. When speaking of consistency in the remainder of this document, strong consistency is meant, unless stated otherwise.

Two desirable properties for task graphs are the existence of a *bounded length schedule* and the possibility to schedule the graph with *bounded memory*. Having a bounded length schedule implies that during execution a certain state will always return within a given period of time. The state in this sense is the distribution of tokens over the arcs, taking into account the number of data tokens or the number and values of control tokens for each arc. An upper bound on timing can only be given if a graph has this property. Bounded memory means that for each arc a certain upper bound on the number of tokens can be given that holds for any execution.

Proving these properties for an arbitrary SDF graph simply requires checking for consistency: provided that no deadlock occurs, any consistent SDF graph has bounded timing and memory. For BDF this is not the case. It is for instance easy to give examples of BDF graphs that are strongly consistent, but may require an unbounded amount of memory.

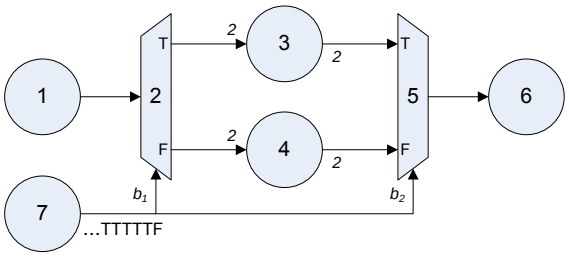


Figure 4.3: Modified if-then-else construct without a guaranteed bound on memory usage [5]. Actors 3 and 4 have in- and output rates of two tokens; all others are homogeneous (rate is one).

One example is presented in Figure 4.3. The same conditional construct as shown earlier is given again, but now the rates of actors 3 and 4 are two instead of one. This graph is strongly consistent with repetition vector $\vec{\rho}(p) = k\,[2 \quad 2 \quad p \quad (1-p) \quad 2 \quad 2 \quad 2]^T$. Suppose the Boolean stream produced by actor 7 starts with a single FALSE token and is followed by an indefinite number of TRUE tokens. The SWITCH actor will first produce a single token on its FALSE output and after that only tokens on its TRUE output. The SELECT will first need to read a token from its FALSE input before it starts reading token from its TRUE input. However, as actor 3 in the FALSE branch needs two tokens to fire, it will not fire and produce tokens until a second FALSE token is present in the Boolean stream. Consequently, also the SELECT cannot fire and an indefinite number of tokens will accumulate on its TRUE input, so no bound on the buffer size exists.

In accordance with Buck's work [5], BDF graphs are classified hierarchically in the following way (also see Figure 4.4 and Table 4.1). The lowest class is the class of SDF graphs. Graphs in this class can not express data-dependencies, but their memory usage and timing can be precisely analysed. The second class of BDF graphs has the property of bounded length schedules and therefore also bounded memory usage: as during execution a certain token distribution always returns within a limited amount of time, the number of tokens on all arcs will be bounded. This class includes all SDF graphs and also graphs with specific

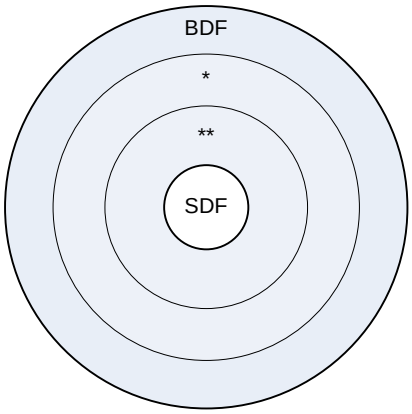


*Figure 4.4: BDF-hierarchy; * = memory bounded, ** = timing bounded*

conditional constructs and data-dependent iterations with a fixed upper bound. A superset of this class contains BDF graphs that can be scheduled in bounded memory, but have no bounded length schedule, so no timing bounds can be given. These graphs may express conditionals and data-dependent iterations. The top-level class of BDF graphs has no restrictions and is Turing complete. In general no bounds on memory or schedule length can be given anymore, in this case. It can be proven that the BDF-actors SWITCH and SELECT together with some SDF actors are enough to build a universal Turing machine [5].

Note that this (non-constructive) classification is based on the analytical properties of graphs and not on the expressiveness. By choosing graphs in a certain class of BDF, expressiveness and analysability can be interchanged.

Deadlock detection in BDF can be done, in a similar way to SDF, by construction of an (annotated) acyclic precedence graph [5]. This can be done in polynomial time for an arbitrary BDF graph.

Table 4.1: Overview of BDF-hierarchy

Model	**Properties**	**Expressiveness**
BDF	*No* memory upper bound *No* bounded length schedule	Turing complete
Memory-bounded BDF	Memory upper bound *No* bounded length schedule	Restricted, but certain *data-dependent iterations* & *conditionals* are still possible
Timing-bounded BDF	Memory upper bound Bounded length schedule	Restricted, but certain *data-dependent iterations with maximum* & *conditionals* are still possible
SDF	Precise memory & timing analysis	Restricted: only task level dynamism

A problem is that it is not possible to compute the boundaries between the three upper BDF classes. The problems of deciding whether an arbitrary BDF graph is strongly consistent (necessary for a bounded length schedule) and can be scheduled with bounded memory are both undecidable. Both problems can be reduced to the Halting problem for Turing machines.

Buck presented techniques to check whether a BDF graph can be scheduled in bounded memory, but they do not work for all graphs. It is not possible to give a general structure for all BDF graphs that can be scheduled in bounded memory. Only by using rules that restrict the construction of a graph, one can be sure that a graph is contained in one of the restricted classes. However, it is impossible to describe a *whole* class with construction rules, because of the previous.

4.2.2 PDDF construction

Clearly, a restricted, timing- and memory-bounded version of BDF could be used as a basis for dynamic elements in PDDF. Construction rules are necessary to ensure that a graph belongs to this class. In Section 4.3 and Section 4.4 two PDDF constructs are introduced: the conditional and the data-dependent iteration. Both constructs are fully modular, in the sense of modularity in SDF (see Section 3.5): an actor in an SDF or PDDF graph can be replaced by a PDDF construct, without making the graph invalid and keeping the desired timing and memory properties, when the consistency rules are respected. If the SDF graph that is obtained from a correctly constructed PDDF graph by replacing all variable rates by their upper-bounds is a consistent SDF graph, also the original PDDF graph is consistent. Moreover, it is possible to nest the new constructs in any way.

For the construction rules, the graph notation is extended with a new element. The values of tokens on certain arcs can influence the rates, but the graph still needs to be consistent. To prove consistency, token values are made explicit by adding a variable between angular brackets to the rates on such arcs. This notation is borrowed from [24]. When applying this to the BDF SWITCH and SELECT actors (see Figure 4.5), the rate at the control ports is denoted "1[b]", to indicate that, per firing, a single control token is read, having the value b. For the SWITCH and SELECT, the value of b can be either 1 (TRUE) or 0 (FALSE). The rates of the conditional ports are no longer probabilities, but they are annotated with b (for the TRUE port) and $1 - b$ (for the FALSE port). For each firing, these expressions are evaluated using the current data, so they directly represent the rates (0 or 1). An upper-bound on the rates of all ports of the SWITCH and SELECT actors is clearly 1. This convention is used is the remainder of this document, also for other constructs.

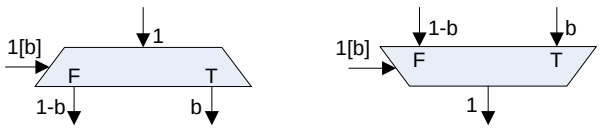


Figure 4.5: PDDF SWITCH *and* SELECT*, explicit token values*

4.3 Conditionals

4.3.1 Construction rules

The Boolean Dataflow (BDF) model is very useful to express conditionals, as already explained in the previous section. This section describes construction rules for graphs containing these conditionals.

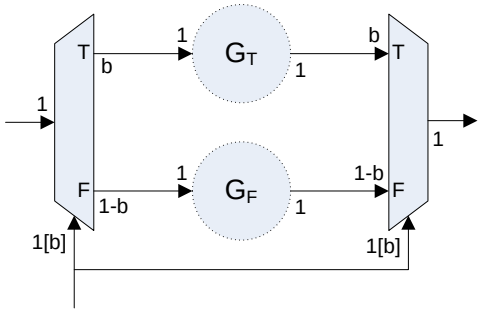


Figure 4.6: PDDF conditional construct

The basic conditional construct (often simply called "conditional" after this) is given in Figure 4.6. The structure is opened by a BDF SWITCH actor and closed by a SELECT actor. The two dotted circles G_T and G_F are arbitrary, consistent PDDF sub-graphs that need to have homogeneous input and output rates. This means that the rates on the arcs in the sub-graph, to which the SWITCH and SELECT are connected, have to be one. Of course the rates can always be converted to any other rate inside the sub-graphs. No arcs are permitted between G_T and G_F to prevent deadlock and no arcs are allowed to enter either of the two sub-graphs from outside the construct. The tokens that enter at the control inputs on the SWITCH and SELECT actors are Boolean tokens that determine whether tokens from the data input are led through G_T or G_F. The construct has only one (homogeneous) data input and one data output. If data from multiple tokens is needed to be processed in one pass of the conditional, it should all be multiplexed to one single large token somewhere outside the construct.

The definition of this conditional construct is clearly modular, meaning that the construct as a whole can be seen as an SDF sub-graph. Hence it can be used anywhere in any SDF graph to express conditional behaviour.

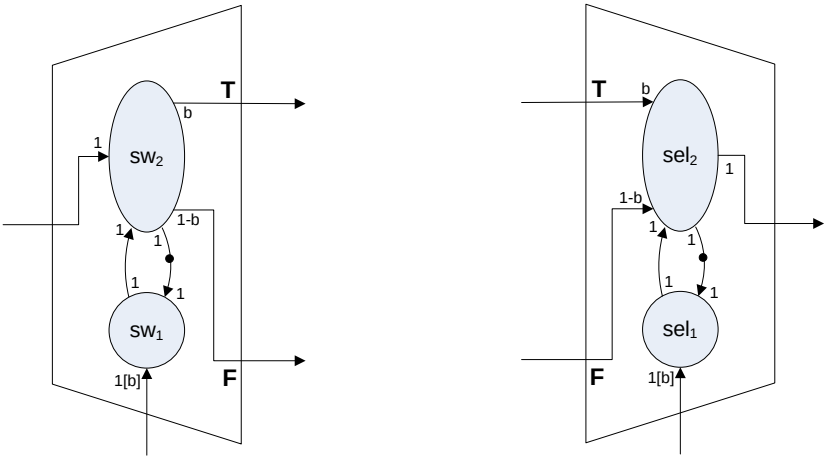
The semantics of the conditional construct are as follows. The SWITCH is enabled as soon as there is a data token and a control token available. The control stream consists of Boolean control tokens: they can only have the value TRUE or FALSE. Both the data and control stream have to be produced by another part of the job outside the conditional construct. When it fires, it consumes both tokens and transfers the data token to its TRUE output (towards G_T) when the value of the control token is TRUE and to its FALSE output (towards G_F) otherwise. The data tokens will enter either G_T or G_F and depart again towards the appropriate SWITCH's data input. The SELECT is enabled when there is a control token present and, depending on the value of the control token, a token on its TRUE or FALSE data input. When it fires it consumes both tokens and transfers the data token to its output. The same control stream is needed by both the SWITCH and the SELECT actors to maintain the correct token order from data input to output and to ensure boundedness of timing and memory.

The repetition vector for this construct is given by Equation 4.2.

$$\vec{\rho}(b) = [\rho_{Sw} \quad \rho_{G_T} \quad \rho_{G_F} \quad \rho_{Sel}]^T = k\,[1 \quad b \quad (1-b) \quad 1]^T \tag{4.2}$$

One of the main goals of using these PDDF models, is to predict the timing behaviour of jobs. When special BDF actors, like SWITCH and SELECT are used, they have to be defined

more precisely, in order to correctly represent the timing of actor enabling and the reads and writes. This is because the semantics of these actors are defined in two stages: first, the value of the control token has to be known and only then it can be checked whether the actor is enabled. For the SELECT this is obvious, but even for the SWITCH this is the case, when bounded FIFOs are used and the SWITCH has to check for free space on either the TRUE or the FALSE output. The internals of the SWITCH and SELECT are presented in Figure 4.7. Both actors are split into two actors that share a memory. The first one reads the control token and the second one transfers the data token from the correct input to the correct output, when it is enabled. The two actors are in a loop with a single initial token, so they can never fire concurrently.



(a) SWITCH (b) SELECT

Figure 4.7: SWITCH *&* SELECT *actor internals*

When performing timing analysis or simulations to obtain timing results, the SWITCH and SELECT should be broken down into these smaller actors.

4.3.2 Timing analysis

The MCM analysis, which is, as discussed before, used to derive a lower-bound on the throughput of a job, searches for worst-case cycles in the graph. If there is no information available about the control streams, all conditionals have to be considered independent and all paths through these conditionals are possible and have to be taken into account in the throughput analysis. The BDF SWITCH and SELECT actors can be treated like normal SDF actors that produce and consume data on every edge for all firings: their conditional rates b and $1 - b$ should be replaced by 1. The MCM analysis can then be performed as usual.

However, conditionals offer extra information about the flow of data and about when actors are enabled. Because of this, the waiting times for actors inside the branches of a PDDF conditional can be shown to be smaller than in the SDF case. This may lead to sharper bounds on the throughput (lower cycle means) and as a result of this a potentially more efficient mapping. This is worked out in the remainder of this subsection.

Consider a job that can be expressed as an SDF-graph as in Figure 4.8 and a homogeneous platform with processors of the same type and the same clock frequency. Suppose the actors

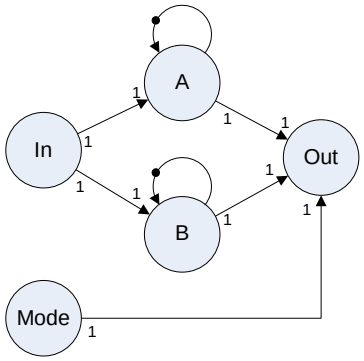


Figure 4.8: Example 1, job specification (SDF)

A and B both have a worst-case processing time of T, while the processing time of the others is negligible and considered to be zero. All actors also have self-edges with one token to make their operation sequential, but they are left out for the actors with zero processing time. Also suppose there is a throughput constraint that imposes a maximum of T on the MCM. This constraint is considered to be hard real-time, so resources have to be allocated to cope with every possible situation, while still meeting the deadlines. If the job were a soft real-time job, one may decide to reserve less resources than derived below, accepting occasional throughput violations.

The objective is to map this job to the multiprocessor platform. Two cases are investigated:

1. mapping A and B to two separate processors

2. mapping A and B to a single processor

In this example, Round Robin static-assignment scheduling is used when multiple actors are mapped to a single processor. The order of actors in the Round Robin list can be chosen in any desirable way. The SDF graph needs to be adapted to reflect the behaviour of the system when it is mapped to the platform. Only the impact of processing-resource budgets and scheduling is considered here; memory and inter-processor communication are assumed to have no influence. Inter-processor communication delays are assumed to be zero.

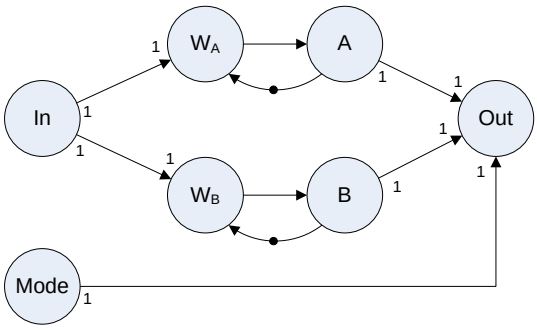


Figure 4.9: Example 1, mapped to one processor (SDF)

If A and B are mapped to separate processors, the graph remains as it is in Figure 4.8. The MCM in this case is T, due to the self-loops around both A and B, and thus satisfies the throughput constraint. The two processors will be fully used. However, if we try to map the actors A and B together to a single processor to save processing resources, the throughput will be lower, since A and B cannot be fired simultaneously. In the graph this can be expressed by adding waiting times, like in Figure 4.9. The worst-case waiting time for actor A is T. This is equal to the worst-case processing time of actor B, since A may have to wait for B to complete. Similarly, the waiting time for B is also T: the worst-case processing time of A. The MCM in this case is $2T$, which violates the throughput constraint. This implies that A and B cannot be mapped to a single processor.

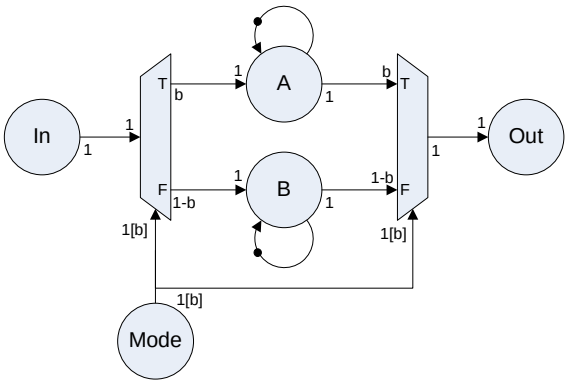


Figure 4.10: Example 1 (PDDF)

However, suppose we know that actors A and B are conditional in the sense that the output actor *Out* needs tokens from only one of the two incoming data branches to produce its output. So, for each produced output token either A or B has to be fired, but not both as previously was the case. The total needed processing time is halved. This knowledge can be made explicit by transforming the original SDF graph into a PDDF graph, see Figure 4.10. It is assumed that no knowledge is available about the Boolean stream that is produced by the *Mode* actor; any sequence of TRUE and FALSE tokens is possible. The SWITCH and SELECT are not split up into their two-stage actors in this example for clarity, and because their processing times are considered to be zero.

Consider again the two mapping cases: A and B mapped to separate processors, or mapped to a single processor. Using separate processors, the job's timing behaviour depends on the control stream. If, for example, the control stream produced by *Mode* only consists of alternating TRUE and FALSE tokens (101010...), then the actors A and B are able to run at the same time continuously and SELECT can fire twice per period T. In this best-case example the throughput will be twice as high as required. This implies that even if only 50% of both processors was reserved for this job with this kind of control stream, the required throughput would still be reached. The other extreme would be when the control stream contains only TRUE tokens, so every input token has to go through A, or only FALSE tokens, so every input token has to go through B. This implies that one processor is not used at all. The timing behaviour in this worst-case will be the same as for the SDF counterpart: the throughput is determined by the MCM, in this case T. If there is no information about the control stream at all, the throughput will be somewhere in between 1 and 2 tokens per period

T. Since the job has hard real-time constraints, then it is necessary to assume the worst-case situation for timing. This leads to the conclusion that when mapping A and B to separate processors, both processors need to be completely reserved. The throughput will then satisfy the constraint: the period equals T. This is the same result as when using the SDF model.

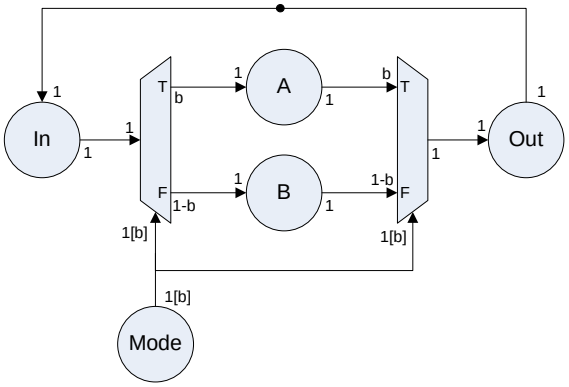


Figure 4.11: Example 1, mapped to one processor (PDDF)

When mapping A and B to a single processor it becomes more interesting. As for every input token either A or B has to be fired, the processor is always fully used. Still, the required throughput can always be met, regardless the values in the control stream. The conditional construct as a whole can now be seen as a single atomic SDF actor, having two distinct modes that are *mutually exclusive*. It is just like an actor with a data-dependent processing time. This can be enforced in the model with a feedback edge from the actor right behind the SELECT back to the actor right before the SWITCH. Self-edges on A and B are then no longer needed: they can be considered to have moved to the level of the large-grain actor (see Figure 4.11). The worst-case processing time of the large actor is equal to the maximum processing time of A and B. The conclusion is that for this job it is possible to map A and B to a single processor, while the required throughput is still met in all cases.

The next question is: considering jobs with a pipeline of multiple actors per conditional branch, how to exploit the knowledge about the conditional behaviour? There is no pipelining within the branches of the job that was studied above, as there is only one actor in each branch.

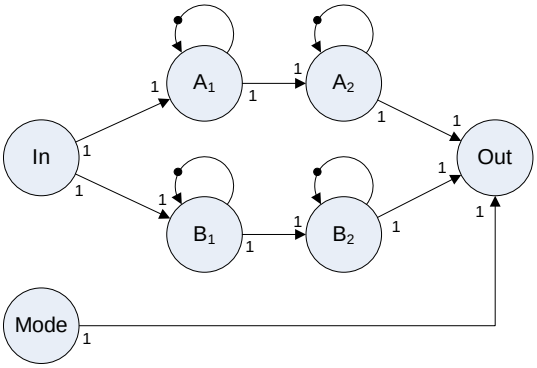


Figure 4.12: Example 2 (SDF)

The previous example's job is now adapted to have two actors on each branch: A is

replaced by A_1 and A_2; B is replaced by B_1 and B_2 (Figure 4.12). These four actors have a worst-case processing time of T and the throughput constraint is again characterised by a period of T. We have two processors available. Several different mappings are now possible of which the following are studied here (see Figure 4.13):

1. All four actors on a separate processor
2. A_1 and A_2 together and B_1 and B_2 together
3. A_1 and B_1 together and A_2 and B_2 together
4. A_1 and B_2 together and A_2 and B_1 together
5. All actors together on a single processor

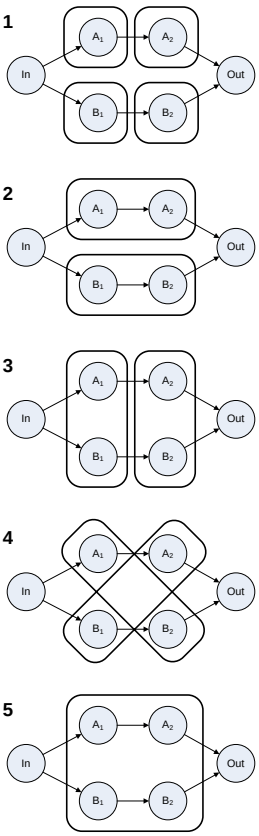


Figure 4.13: Example 2, mapping options

In the SDF case, so when ignoring the fact that the branches are conditional, it is quickly seen that only the first mapping option will meet the throughput constraint: sharing any two or more actors on a processor will introduce a waiting time in a cycle, so the cycle mean will be larger than the allowed value T. Using the conditional structure this first option of course works as well, but we can do more.

The fifth mapping option is no solution though. Similar to the previous example, the graph in this option can be seen as a single actor with two mutually exclusive modes, both having a worst-case processing time of $2T$. Also the second option will not work. This option can be reduced to the case with only one actor per branch, but now having a worst-case processing time of $2T$, which violates the throughput constraint.

The third option, however, offers a possibility. Here the conditional can be split up into two parts, one for each processor (Figure 4.14). As before, a conditional construct on a single processor can be seen as a single actor with two mutually exclusive modes, having a worst-case processing time of T. Pipelining between the two processors is still possible. The MCM of this job is T, which meets the throughput constraint. Another observation is that only one FIFO instead of two is needed between the processors. This is possible, because the conditional streams to both conditionals are identical.

This last method will only work when the conditional construct is split in parts and each part as a whole goes to a processor. In the fourth mapping option, for example, this will not work, as the conditional cannot be split so that A_1 and B_2 are in the same part and A_2 and B_1 as well. In this case it is necessary to add waiting times, like when using SDF, and there is no benefit anymore from the conditional. Option 4 will therefore not satisfy the throughput constraint.

The conclusion is that when a conditional is supposed to be mapped to multiple processors, it can be split up into smaller conditionals, one for each processor. The two branches inside each conditional can be made mutually exclusive, so there is no need to add waiting times for actors in the other branch to the model. In this way conditionals can be used to arrive at sharper worst-case timing estimations.

If for some reason mapping of actors on a processors like in mapping option 4 in Figure 4.13 is desired, another trick might lead to better results: making the whole conditional mutually exclusive. In this way, no waiting times for actors in the other branch have to be added, but an extra feedback with only one initial token has to be added around the whole conditional. This feedback is an extra cycle through the entire conditional, which decreases the possible amount of pipelining. Especially for long pipelines spanning multiple processors this may lead to a lower throughput. It depends on the sub-graphs in the branches which option is the best. For option 4 in the previous example, adding a feedback arc with one token does not help: it leads to the same result as when just adding waiting times. For the example in Figure 4.15, however, adding the feedback does lead to a higher throughput, when the three A actors share processor π_A and the B actors share processor π_B. Suppose all actors A and B have a processing time 1 and Round Robin scheduling is applied. Consider processor π_A. Without the loop, the waiting times would be: $W(A_1) = 1, W(A_2) = 0, W(A_3) = 2$ (this result makes use of the optimisations suggested in Section 3.5.4). For π_B this goes analogously. The MCM would be 3 in this case. But when the feedback loop is applied, no waiting times are needed anymore, as all actors are mutually exclusive: the MCM equals 2.

There are two more points that can be said about conditionals, which may improve the results:

- When multiple conditional constructs have the same control stream, like when a large conditional is split up, this restricts the number of possible paths through the graph. In Figure 4.14, for example, the paths from In to Out via A_1 and B_2 and via A_2 and B_1 are not valid. This can be taken into account when calculating the MCM of the graph to arrive at potentially sharper worst-case estimations. The algorithm to compute the MCM should be adapted to reach this.
- The construction rules for the conditional say that no arcs are allowed from outside the conditional to one of the sub-graphs. However, there is one exception to this rule: when multiple conditionals are fed exactly the same control stream, there are arcs allowed between the TRUE and between the FALSE branches of different conditionals. An example of this is present in the H.263 case study, as presented in Chapter 6.

This concludes the theory about conditionals. The next sections deals with the other type

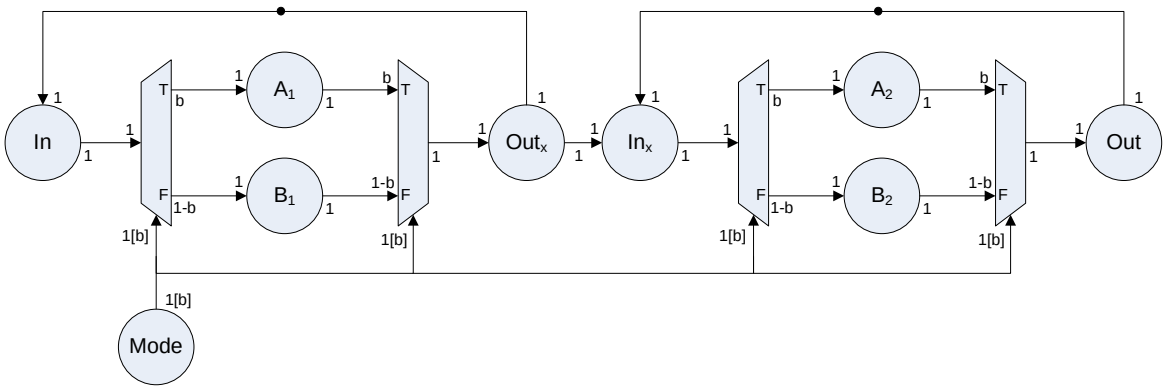


Figure 4.14: Example 2, mapping option 3 (PDDF)

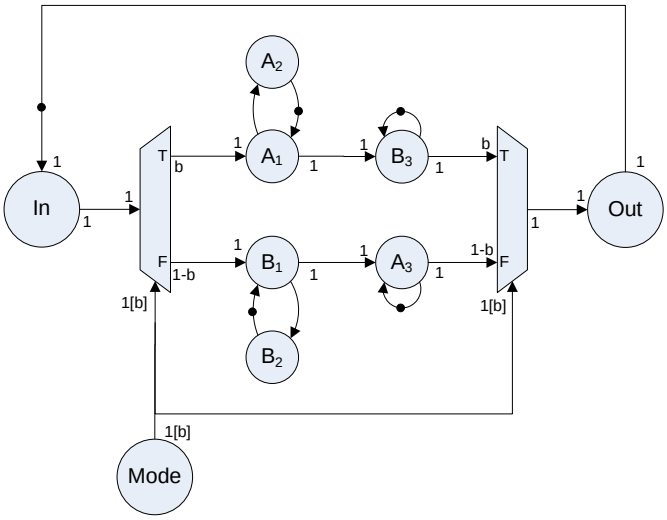


Figure 4.15: Example 3 (PDDF)

of dynamic PDDF construct: the data-dependent iteration.

4.4 Data-dependent iterations

4.4.1 Construction rules

One of the restrictions of SDF graphs is that all rates have to be fixed. Because of that, the execution of a job that is described by SDF will always enter a periodic regime. This means that per period every actor fires a fixed number of times. This knowledge can be used to construct static schedules at compile time, one of the main reasons for which SDF was developed.

Some jobs, however, require that certain actors do not fire a fixed number of times, but have a relative frequency that is dependent on the input data. An example is the H.263 decoder case that is worked out in Chapter 6. Some frames in video sequences that this decoder handles contain a variable number of blocks that also demand a variable number of firings of the IQ and IDCT actors. The construct that is described in this section is able to deal with this kind of behaviour, while the necessary properties of bounded memory usage and timing are kept.

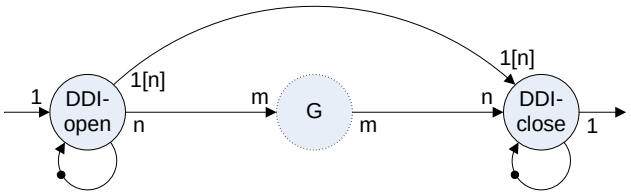


Figure 4.16: Data-dependent iteration construct

The basic construction for data-dependent iterations is given in Figure 4.16. The rates

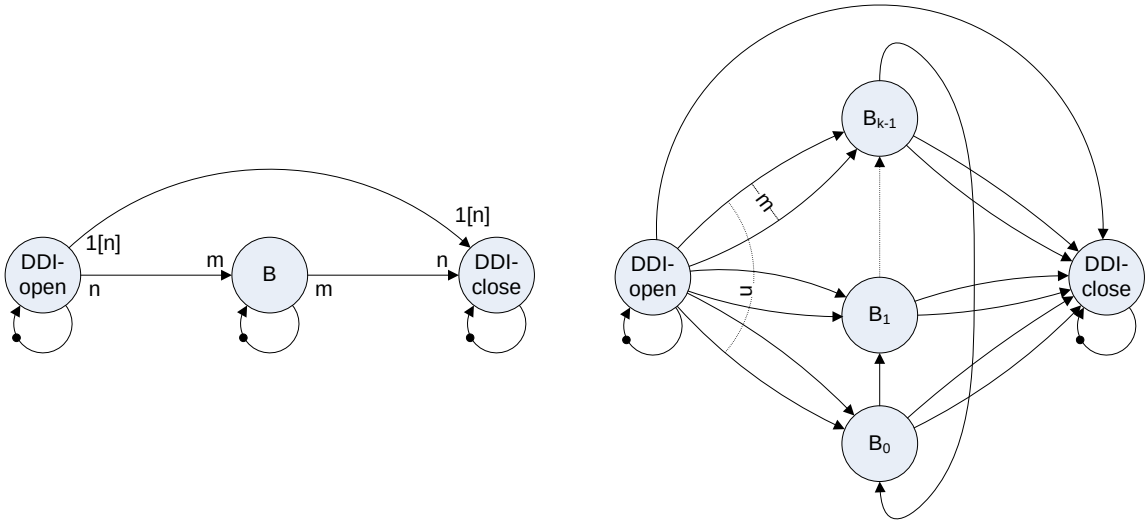


Figure 4.17: Data-dependent iteration, example (SDF and HSDF)

marked with n are flexible; the others are fixed. The dotted circle in the middle represents an arbitrary sub-graph G that can take any form, as long as the input and output rates are constant (m). No arcs from outside the construct are permitted to enter G. As stated in Equation 4.3, the value n has to be a multiple of m, so that the sub-graph will be executed $k = \frac{n}{m}$ times more often than the opening and closing actors DDI-open and DDI-close. The value of n is not allowed to exceed a fixed maximum N.DDI-open can have any number of inputs and DDI-close can have any number of outputs. The value of n can be set based on the input data of DDI-open, thus making the number of iterations of the actors in G data-dependent. This can be easily verified in the HSDF expansion of the example graph with only one actor B as sub-graph G in Figure 4.17. The HSDF graph contains $k = \frac{n}{m}$ actors B, which indicates that actor B is fired k times for every firing of DDI-open and DDI-close.

$$\begin{cases} n \leq N \\ n = k \cdot m \quad k \in \mathbb{N}^+ \end{cases} \tag{4.3}$$

The semantics of this construct are as follows. The DDI-open actor is enabled like a normal SDF actor when sufficient input tokens (one in this case) are present. When it fires, it consumes data from its input and determines the new value of n according to the input data. Then, it writes one token with the value of n to the arc towards DDI-close. This is represented in the model by the addition "[n]" to the rate "1" at this port. This value is made explicit to prove that the construct is consistent: the DDI-close actor has to consume as many tokens as DDI-open produces, otherwise tokens will accumulate or deadlock will occur. When DDI-close fires, it first consumes the token with the value n that comes from DDI-open. Subsequently it consumes n tokens from G, performs its computations and produces a token on its output. The sub-graph G behaves just like a regular SDF graph that needs a constant number of m tokens to execute and produces also m tokens. The variable n has a fixed upper bound N. Because of this, also the number of iterations of the actors in G and also the timing are bounded, a necessity for real-time jobs.

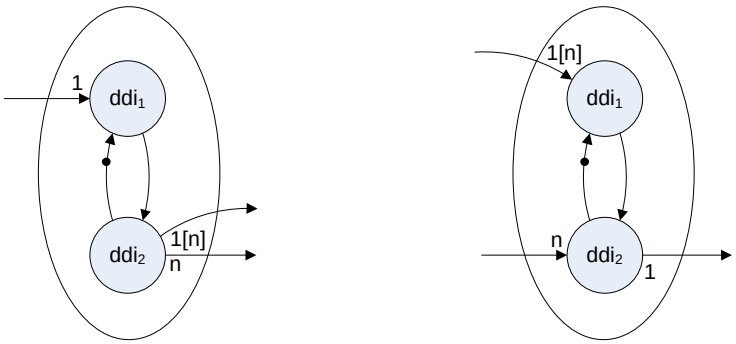
The repetition vector for this construct is given by Equation 4.4. This makes clear that

G is executed a variable number of times, as n is a variable number.

$$\vec{\rho}(n) = [\rho_{A_0} \quad \rho_G \quad \rho_{A_1}]^T = c\left[1 \quad \frac{n}{m} \quad 1\right]^T, \quad c, m \in \mathbb{N} \tag{4.4}$$

The construct as a whole has pure SDF semantics from outside and its definition is clearly modular. Hence it can be used as a sub-graph within any other SDF or PDDF graph.

Just like for the conditional SWITCH and SELECT actors, also the DDI-open and DDI-close actors should be split up into two sequential actors for timing analysis and simulation. How this is done is shown in Figure 4.18.



(a) DDI-open (b) DDI-close

Figure 4.18: Internals of DDI actors

4.4.2 Timing analysis

We consider a hard real-time job containing a data-dependent iteration construct. In this job, the sub-graph G will receive a data-dependent number of tokens per firing of A_0 and A_1, but never more than N. For a conservative worst-case analysis and without knowledge about the input data, it is necessary to assume that G always receives the maximum N tokens per period. This intuitively provides the largest possible amount of work for G, which leads to the worst-case timing. Regular MCM analysis can then be done to obtain the worst-case throughput. The MCM analysis searches for the cycle with the largest cycle mean in an HSDF graph (see Section 3.3.2). In the example of Figure 4.17, it can be seen what is the influence of variations in n on the MCM. If the construct is part of a larger SDF graph, it is possible that some cycles go through DDI-open, all instances of B and DDI-close. The number of instances of B is proportional to n, so also the cycle mean of this cycle is proportional to n. Hence, this cycle will be maximum for the largest possible value of n, being N. The conclusion is that substituting N for n leads to the worst-case MCM, so also to the worst-case throughput. This implies that using this PDDF construct does not lead to better timing results than when modelling the iteration as an SDF graph with fixed rates N. Unlike is the case for conditionals no further optimisations can be done for iterations. This worst-case result cannot be improved without additional knowledge of the input data. For soft real-time jobs, however, it can be very beneficial, because the average-case timing behaviour is probably better. This is especially the case when n is often smaller than N.

4.5 Data-dependent processing times

The previous two sections described ways to incorporate conditionals and data-dependent iterations inside dataflow graphs, while maintaining the necessary properties of boundedness of timing and memory and analysability. Because both structures can be used in a modular manner – from outside they behave like an ordinary SDF sub-graph – they can be used anywhere in an SDF-graph and they can also be nested.

It has already been known that data-dependent processing times of actors can be handled by SDF-graphs. However, when analysing hard real-time jobs in which even the worst situation needs to be dealt with, the worst-case processing times for all actors need to be used. The new dynamic constructs can be used to describe the internals of actors more precisely. This may lead to sharper worst-case bounds in analysis (when conditionals are used), but even more in simulations when real input data is used. Note that, when the internals of an actor can be represented with finer grain actors in this way, the structure will still be mapped as one single actor. This exercise is done purely to better estimate the timing of the actor.

This approach is an alternative to the use of scenarios as proposed in [21]. In this article, the processing time of an actor is defined by a function that has certain values from the data as its input parameters. Using input streams that are representative for the job in a simulation both approaches will improve the worst-case results.

4.6 Using data knowledge

All timing analysis that is performed in this document, assumes that no information about the data streams is available; only at the time an actor fires, the values of tokens become known. If it is possible to characterise streams of tokens, it is possible to arrive at better timing estimations. In [10] such ideas are used on similar systems. Consider a certain PDDF conditional construct. Suppose that the response time on one branch of the conditional is 2 and on the other branch this is 4. Without data knowledge, the conclusion would have been that the throughput of the conditional that can be guaranteed is $\frac{1}{4}$, as the worst-case branch determines the throughput. Now suppose that the Boolean token stream that controls the conditional construct, is always alternating (01010101...). This implies that the the throughput, when measured over two firings, is always $\frac{2}{4+2} = \frac{1}{3}$, which is an improvement.

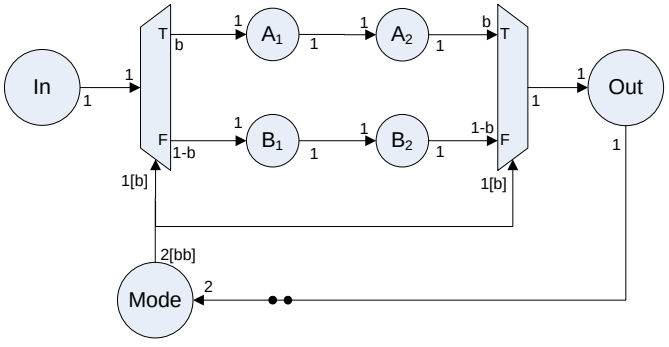


Figure 4.19: Example 4 (PDDF)

Another case, involving the graph with conditional as given in Figure 4.19, is a Boolean stream in which the values 0 or 1 always appear in pairs (e.g. 1100001100...). Then, a

feedback loop can be applied that contains two initial tokens and involves the actors that produce the Booleans (the *Mode* actor). This *Mode* actors produces two Boolean tokens at the same time, of equal value, and also waits for two tokens to return from the output of the conditional. In this case, the two branches of the conditional are also mutually exclusive and, because there are now two initial tokens on the loop, the throughput can be higher. However, it cannot be concluded that the throughput will be twice as good, because the graph first has to be transformed into an HSDF graph before timing analysis can be done. Still, it can be an improvement, depending on the sub-graphs in the conditional branches.

It seems interesting to further investigate this modelling of data characteristics.

4.7 Conclusions

This chapter introduces a new model called PDDF, that is derived from BDF, which can be used as an extension to SDF, while it remains possible to bound the timing and memory usage of the job. Two dynamic constructs are introduced, to incorporate dynamic behaviour on the job level in the job's task graph. The use of these constructs is explained and it is shown how timing estimations can be tighter and implementations can be more efficient when these constructs are used. The constructs are fully modular: they have SDF semantics on the outer level, so they can be used anywhere inside an SDF graph and they can be nested in any way. Furthermore, it is argued that these new constructs can also be used to describe the internals of large grain actors, to arrive at better results.

Contributions of this chapter are:

- The introduction of PDDF, a modular SDF extension that supports dynamic behaviour on the job level by two dynamic constructs (Goal 3):
  - A conditional construct.
  - A data-dependent iteration (with a maximum number of iterations).
- Ways to use the new dynamic constructs to potentially arrive at tighter timing estimations and more efficient mappings (Goal 4).
- A way to describe the internals of complex actors to obtain better results.

Chapter 5

Simulation

5.1 Introduction

In Chapter 3 a design flow is introduced, in which a job is specified as an SDF-graph and mapped to the multiprocessor platform. To find an efficient mapping of the job that meets the throughput demands, the job model can be analysed and simulated. If the job has hard real-time constraints, analysis can be used to obtain bounds for timing and resource usage, for every possible input stream. However, these timing bounds are only valid in the periodic phase of the execution of the job (Section 3.3.2) and it is not known from analysis at which point in time this phase starts. Thus, the behaviour of the job in the transient phase from the start of the execution until an unknown point in time, could be unacceptable, while this is not seen in the analysis of the job.

Moreover, for soft real-time jobs, timing upper-bounds are less important. For this type of jobs, it is attempted to maintain a certain user-perceived quality: it is not necessary to meet all timing deadlines, but preferably as many as possible. The number of deadline misses is dependent on the input streams. When the new, dynamic extensions to SDF are used, which are introduced in Chapter 4, this is even more the case. As, in timing analysis, the input data is not taken into account, analysis is not very useful for soft real-time jobs.

Simulations can be done to fill in the gaps. Simulations can give insight in timing behaviour during the transient phase, for both hard and soft real-time jobs. Furthermore, for soft real-time jobs, information about average timing behaviour can be obtained through simulation. Also, for hard real-time jobs that are only used with a specific, known set of input streams, simulations could be done to obtain worst-case results.

In Philips Research Labs Eindhoven a simulator has been developed for the given purpose. Section 5.2 gives an overview of the original version of this simulator. This simulator was extended for the simulation of jobs that contain data-dependencies, to conveniently incorporate mapping schemes and to extract timing information. Section 5.3 explains what is new and Section 5.4 shows how to use dynamic constructs in the simulator. In the last section conclusions are drawn.

5.2 HAPI simulator

In this section the HAPI simulator [19] is described, as it was developed at Philips Research Labs Eindhoven. HAPI is able to simulate the execution of SDF graphs, the model of computation that is used to specify jobs (See Section 3.3).

5.2.1 YAPI: the basis

This HAPI simulator is built on YAPI (Y-chart Application Programmer's Interface [12]), also developed by Philips Research, which is used to simulate Kahn Process Networks. A process in YAPI may have several input and output ports that are connected to FIFOs. All communication between processes is done through these FIFOs, which all have a certain capacity. A process can read and write data tokens to/from a FIFO through its ports.

YAPI is a C++ library on top of SystemC. It has classes for processes (`Process`) and FIFOs (`Fifo`) and it has a `ProcessNetwork` class, in which every part of the network is initialised and connected. A `Process` may have instances of the `Port` class as attributes, through which it can be connected to `Fifo`s. Every `Process` has its own thread of execution. It has a `main` function, which normally contains an infinite loop to signify the continuous nature of Kahn processes, and to let the process run on unbounded streams of data. In this loop, the functional code is specified; computations and `read`s and `write`s to ports can be done in any order. A `Process` blocks on `read`s when the incoming `Fifo` is empty, until there is new data available. More details of YAPI can be found in [12].

5.2.2 Actor semantics, FIFO capacity & timing

Since SDF, which is used as a model of computation for HAPI, is a subset of KPN, most of YAPI could be reused. Some extra functionality has been implemented to enable `Process`es to have SDF actor semantics. Actors do not run continuously like Kahn processes, but use firings. A firing can start as soon as there is enough input data to complete it (i.e. when the actor is enabled). Once it starts, the firing is not interrupted anymore: it is said to be atomic. To simulate this behaviour, it needs to be possible to check FIFOs for availability of tokens, to see if an actor is enabled. The actor should block, as long as it is not firing and not yet enabled.

In HAPI, a new function, called `peek_wait`, has been introduced. This function can be called inside a `Process`, using a `Port` as an argument. When it is executed, it blocks the `Process`, until data is available in the `Fifo` that is connected to the `Port`. HAPI also uses infinite loops in the `main` functions, so actors can process unbounded streams of data tokens. A firing is one full cycle of this loop. The `peek_wait` function has to be called on every input `Port` in the beginning of the `main` function's loop. The programmer is responsible for doing this in accordance with the rates at the connected arcs. Furthermore, the number of `read` tokens per `Port` should match the `peek_wait` calls.

In an SDF-graph, FIFOs with a bounded capacity can be modelled by inserting an extra arc back from the consuming actor to the producing actor; the number of initial tokens on the cycle that is now created, signifies the capacity of the FIFO (See Section 3.4). This mechanism makes sure that the producing actor is only enabled when there is enough space in the FIFO to which it needs to write its token. To ease modelling, it is not necessary in HAPI to add this extra arc. The capacity of each FIFO can be set in the `ProcessNetwork`

object. Furthermore, the `peek_wait` function is also defined for outgoing `Port`s to check for space: when called, the function only returns when there is sufficient free space in the FIFO. Also, the number of tokens that are written per `Port` should match the `peek_wait` calls.

A Kahn network, and therefore also YAPI, is timeless. As the simulation of SDF is done to acquire timing information, a notion of time needed to be introduced in HAPI. For this purpose, the SystemC simulation time is used as a global clock. This clock can be referenced anywhere in HAPI by a call to `sc_simulation_time`. The time can be advanced by calls to the function `wait`, using the delay as an argument.

HAPI also contains a function that can detect the steady state, the periodic phase of the execution. This function can be used to stop the simulation when the periodic phase is reached. However, as this functionality is based on token arrival times, it only works when static actor response times are used. As conservative simulations are desired, response time upper-bounds are used. If this is done, HAPI can derive the MCM of the graph. When data-dependent processing times and dynamic constructions are used and the focus is on statistical information for varying input streams, this steady state function is no longer useful.

5.2.3 Actor outline

The definition of an actor in HAPI starts with the declaration of local variables, including the definition of a constant `sc_time` for an upper-bound on the response time (`RTUB`); the response time has not been made data-dependent yet in this version of HAPI. This is followed by the infite loop. A HAPI actor's loop should consist of the following: first the `peek_wait`s are issued, then the simulation time is advanced with the response time upper-bound of the actor, followed by the (timeless) functional part that consists of computations, `read`s and `write`s. The `read`s in the functional part can never block, as the use of `peek_wait` calls ensures that enough data is available in the incoming FIFOs. Similarly, also the `write`s can never block, as the `peek_wait`s ensure that there is enough space available in the output FIFO. Thus, the `main` function of every actor should look like the example actor X in Listing 5.1, which has one input and one output. The rates on the arcs that are connected to an actor are implicitly modelled by the `peek_wait`, `read` and `write` calls.

5.2.4 Network channels

As said in Section 3.4, a special SDF model has been introduced in [19], to mimic the behaviour of network channels. Any FIFO can be turned into a complete network channel by simply replacing it by the channel sub-graph. In HAPI this is done by calling the function `table_mapping_fifos_to_channels.Add` in the constructor of the `ProcessNetwork` object, having a `Fifo` name and parameters that define the characteristics of the channel as arguments. When HAPI is executed, it will behave as if the channel sub-graph is there, when reading and writing tokens to the "FIFO". The `peek_wait` function is also defined for this channel, both to check for availability of tokens and for space in the FIFOs.

5.3 Extensions

The previous section described the first version of HAPI, as introduced in [19]. While the notion of time was already present in this version, no facilities were available to obtain detailed

Listing 5.1: Actor outline

```
1  void X::main (void)
2  {
3      int a = 0;
4
5      // set the response time upper-bound (10 ns)
6      sc_time RTUB = sc_time (1000, SC_NS);
7
8      while (true)
9      {
10         // wait until the actor is enabled:
11         // peek_waits on every in/output ports
12         peek_wait (in1);
13         peek_wait (out1);
14
15         // advance simulation time
16         wait (RTUB);
17
18         // functional part
19         // computations, reads and writes
20         read (in1, a);
21         a++;
22         write (out1, a);
23     }
24 }
```

information about when actors fire exactly, to check if deadlines are met or not and to derive the throughput during the whole simulation. Especially when data-dependencies are incorporated and when looking at soft-real time constraints, this type of infomation is needed.

Moreover, the original HAPI did not have a notion of actor-to-processor mappings. The processor assignment was hidden in the worst-case actor response times, which should include waiting and interruption times to signify arbitration.

Therefore, HAPI was extended with functions to acquire statistical timing information and features for explicit processor assignment. The timing results can be visualised in a web browser through an XML file that is generated by HAPI. The following sub-sections descibe the details of these new elements.

5.3.1 Timing statistics

To separate the extensions from the existing HAPI code, a new class was derived from the `Process` class, in which the new functionality was included. This class is called `Actor`. The class contains new member variables and functions, which can record timing statistics. For every actor that is derived from `Actor` instead of `Process`, useful timing information becomes available.

The first new member of the `Actor` class is the processing time upper-bound $P\pi(a)$, as defined in Section 3.4. The processing time is dependent on the type of processor the actor will be running on, but for now it is assumed that all processors are equal, so the subscript π is not used. The $P(a)$ is specified as a `sc_time` object that can be set through the `Actor` member function `SetTiming`. This is done centralised in the `ProcessNetwork` object's constructor for all `Actors`. The member function `WaitExecution` can be called in the `Actors` `main` function to advance the simulation time with this processing time. This mechanism, together with the introduction of scheduling in Section 5.3.2, replaces the use of the response time upper-bound constant `RTUB`.

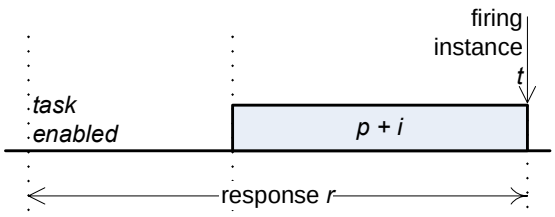


Figure 5.1: Simulation measures

The `Actor` class uses three timing measures on which its statistics are based: the *response time* of the actor, its *processing time* and the *firing instance*. The measures are illustrated in Figure 5.1 (similar to Figure 3.5). The response time $r(a, i)$ of an actor a is defined as the time it takes from when the actor is enabled until it is finished executing (similar to Definition 3.6 on page 27, in which the parameter π is dropped, as it is assumed for now that all processors are equal). The data-dependency of the response time is in this case expressed in the firing iterator i: $r(a, 0)$ is the response time of the first firing, etc. The response times that are measured in the simulation are denoted by $\tilde{r}(a, i)$. Likewise, the processing time $p(a, i)$ of an actor a is the net time the actor is doing its processing in firing i: the response time minus waiting and interruption times. The processing time in the simulation is called

$\tilde{p}(a,i)$. The firing instance $t(a,i)$ of actor a denotes the point in time at which the actor finished its execution; the index i is again the firing iterator. At this instance, the actor must have produced its data. Firing instances in the simulation are denoted by $\tilde{t}(a,i)$.

All measures have to be conservative in the simulation. This means the following for a simulation with $I(a)$ firings of actor a:

$$\forall_{0 \leq i < I(a)} \begin{cases} r(a,i) \leq \tilde{r}(a,i) \\ p(a,i) \leq \tilde{p}(a,i) \\ t(a,i) \leq \tilde{t}(a,i) \end{cases} \tag{5.1}$$

During simulation, the measures can be recorded for every firing of any `Actor` that one is interested in. For the statistics, also the total number of firings ($I(a)$) is recorded, as well as the cumulative sum of the response times of all firings and the cumulative sum of the squares of all response times. This data can be used to calculate averages and standard deviations. Next to that, also the minimum and maximum response times are stored. The same information is also stored for the processing times, as well as for the differences in the firing instances between successive firings: $\Delta\tilde{t}(a,i) = \tilde{t}(a,i) - \tilde{t}(a,i-1)$. This information leads to the calculation of the throughput of the job that is simulated. Following Definition 2.1 on page 7, the throughput $\tilde{\tau}$ of the simulated job is equal to the reciprocal of the average $\Delta\tilde{t}(o,i)$ over all firings, times the number of tokens $O_{env}(o)$ that are produced to the environment per firing by a pre-selected output actor o (Equation 5.2). This throughput result from simulation is conservative compared to the throughput τ of executions of the real implementation, meaning $\tau \geq \tilde{\tau}$ always holds.

$$\tilde{\tau} = \frac{I(o) - 1}{\sum_{i=0}^{I(o)-1} \Delta\tilde{t}(o,i)} \cdot O_{env}(o) \tag{5.2}$$

To make use of this functionality, two lines of code have to be added to the `main` function of the actor: the function `StatsBegin` has to be called right after the calls to `peek_wait` (so when the actor becomes enabled), and `StatsEnd` has to be called after advancing the response time (see Listing 5.2, in which also the response time advance is changed, as is explained below).

At the end of the simulation, HAPI writes all $\tilde{r}(a,i)$, $\tilde{p}(a,i)$ and $\tilde{t}(a,i)$ and derived results from all actors to an XML file called "output.xml". This file is linked to an XSL stylesheet file ("style.xsl"), which can be used in any modern web browser (like Internet Explorer 6) to display the results in the XML file as a web page (see Figure 5.2). This results web page contains a time line, containing time shapes for every actor. It also contains a table with the statistical results per actor.

5.3.2 Mappings

The previous sub-section shows how timing information can be obtained by using functions in the new `Actor` class. These timing results are obviously dependent on the response times of the actors, which include waiting and interruption times as a result of mapping decisions. To make these mapping decisions explicit in HAPI, a new class – `Processor` – is introduced. Instances of `Processor` should be included in the `ProcessNetwork` object for each processor in the platform. `Actors` can be associated with `Processors` to fix a certain actor-to-processor

Listing 5.2: `Actor` *usage*

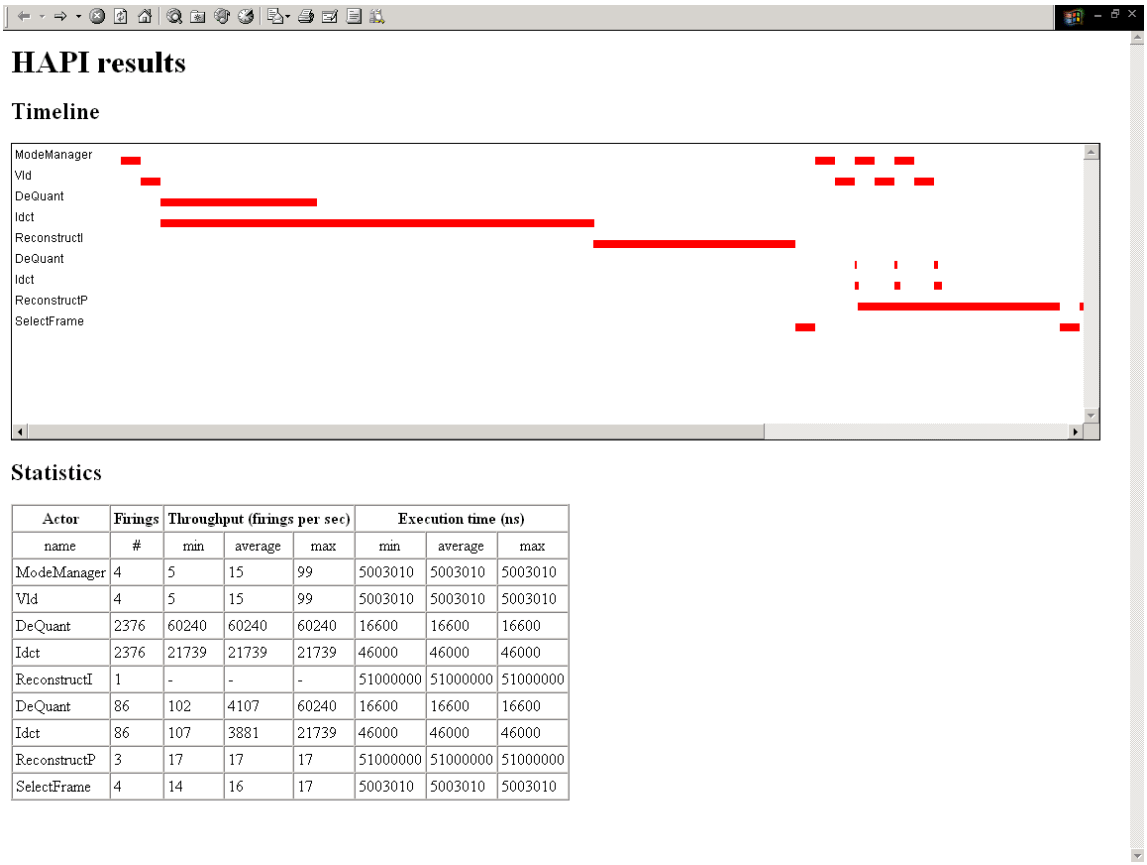
```
// wait until the actor is enabled
peek_wait (in1);
...

// actor enabled; begin stats
StatsBegin ();

// advance simulation time with response time
WaitScheduling ();
WaitExecution ();

// actor finished; end stats
StatsEnd ();

// functional part
...
```



HAPI results

Timeline

Statistics

Actor	Firings	Throughput (firings per sec)			Execution time (ns)		
name	#	min	average	max	min	average	max
ModeManager	4	5	15	99	5003010	5003010	5003010
Vld	4	5	15	99	5003010	5003010	5003010
DeQuant	2376	60240	60240	60240	16600	16600	16600
Idct	2376	21739	21739	21739	46000	46000	46000
ReconstructI	1	-	-	-	51000000	51000000	51000000
DeQuant	86	102	4107	60240	16600	16600	16600
Idct	86	107	3881	21739	46000	46000	46000
ReconstructP	3	17	17	17	51000000	51000000	51000000
SelectFrame	4	14	16	17	5003010	5003010	5003010

Figure 5.2: Hapi output

assignment. The most important property of a `Processor` is the type of scheduling. Support is implemented for the static assignment arbitration schemes Round Robin, TDMA and combined TDMA/Round Robin (see Section 3.5.3).

The `Processors` are initialised in the initialisation list of the `ProcessNetwork` by specifying an ID and the type of arbitration. The following options are possible for an example `Processor` called `proc1` with ID "0":

- `proc1 (0, RR)`
  to select Round Robin arbitration
- `proc1 (0, TDMA, sc_time(100, SC_NS))`
  to select TDMA arbitration with a period of 100ns
- `proc1 (0, TDMARR, sc_time(100, SC_NS), .5)`
  to select the TDMA/Round Robin combination with a TDMA period of 100ns and 50% of the period reserved for this job

An `Actor` is mapped by calling the `Actor`'s member function `SetProcessor` with a reference to a `Processor`. This is done in the constructor of the `ProcessNetwork` object, where also the `SetTiming` calls are located, so the whole processor assignment can be specified centrally. If the processor uses TDMA arbitration, the time slot fraction for the `Actor` can be added as second argument to `SetTiming`. An example for an `Actor` called `actor1` that is mapped to `proc1` is given in Listing 5.3. This `actor1` gets a time slot of 25% of the TDMA period, if TDMA is selected as the arbitration method on `proc1`.

Listing 5.3: `Actor` *mapping*

```
1 // map actors
2 actor1.SetProcessor (&proc1);
3 actor1.SetTiming (sc_time (150, SC_NS), .25);
```

The `Actor` class has a member function `WaitScheduling` that determines the waiting and interruption times for the `Actor` on selected `Processor` and advances the simulation time accordingly. This function is supposed to be called before the call to `WaitExecution` (see Listing 5.2).

5.4 Dynamic behaviour

Now that the simulator has been extended with functionality to include mapping decisions and to extract the timing results, it is time to add dynamic behaviour. The previous functionallity is based on SDF graphs that are annotated with fixed (worst-case) processing times. The following sub-section first extends this by introducing data-dependent processing times. However, the support for data-dependent processing times is still very incomplete and only possible in special cases. The two subsequent subsections respectively focus on how to use the PDDF conditionals and data-dependent iterations.

5.4.1 Data-dependent processing times

A call to the `Actor` function `WaitExecution` internally calls the function `GetE`, which is supposed to determine the processing time for the current firing. When it has the processing time, it advances the simulation time by this duration. The function `GetE` is defined as a virtual function of the `Actor` class, which by default returns the worst-case processing time that should be set in the `ProcessNetwork` object. If this is not desired, this function can be overridden in the derived `Actor` class, so it can return an appropriate processing time that is dependent on the current data in the `Actor`.

The simulation is supposed to be conservative and the precise timing of `read`s and `write`s is generally unknown. Being conservative implies that these `read`s and `write`s have to take place in the simulation as late as they can possibly occur, so at the end of the firing. Therefore, the whole – possibly data-dependent – response time has to be advanced, before `read`s and `write`s are carried out. This raises a problem, when the processing time is dependent on the data that is read during a firing: some `read`s should then be done before advancing the processing time. This may lead to free space in the input `Fifo`s, which may enable other actors, earlier than in the implementation. This calls for a peek function that can see the value of tokens in the input `Fifo`s, without actually removing them from the buffer. Unfortunately, this function has not yet been implemented, so these cases should be avoided by using the processing time upper-bound for this actor, instead of the data-dependent version.

5.4.2 Conditionals

The next step of including data-dependent behaviour in the simulation is creating a conditional construct, as defined in Section 4.3. For this construct, the two actors SWITCH and SELECT have to be created. It is quite straightforward to create HAPI `Actor`s that incorporate this behaviour. The SWITCH and SELECT can be implemented as separate `Actor`s, but they will often be integrated in other `Actor`s in the graph.

Example `main` functions for the SWITCH and SELECT `Actor`s are given in Listing 5.4 and Listing 5.5. The SWITCH has the input `Port`s `inData` (the data input) and `inCtrl` (the control input), and two Boolean outputs `outT` and `outF`. The SELECT has a control input `inCtrl`, two Boolean inputs `inT` and `inF` and one data output `dataOut` These `Actor`s are based on the SWITCH and SELECT models of Section 4.3 that have bounded FIFO input and outputs and use two internal actors to correctly represent the timing behaviour. Both two-actor models can still be implemented as a single HAPI `Actor`, because the two actors always execute sequentially, in a fixed order.

Listing 5.4: SWITCH `Actor`

```
bool c;
int d;

while (true)
{
    // ** first actor:
    // get control & data tokens

    // wait until actor is enabled
    peek_wait (inData);
    peek_wait (inCtrl);

    // functional part
    read (inData, d);
    read (inCtrl, c);

    // ** second actor:
    // transfer data

    // wait until actor is enabled
    if (c == true)
        peek_wait (outT);
    else
        peek_wait (outF);

    // advance simulation time
    WaitScheduling ();
    WaitExecution ();

    if (c == true)
        write (outT, d);
    else
        write (outF, d);
}
```

Listing 5.5: SELECT `Actor`

```
bool c;
int d;

while (true)
{
    // ** first actor:
    // get control token

    peek_wait (inCtrl);
    read (inCtrl, c);

    // ** second actor:
    // transfer data

    // wait until actor is enabled
    if (c == true)
        peek_wait (inT);
    else
        peek_wait (inF);
    peek_wait (outData);

    // advance simulation time
    WaitScheduling ();
    WaitExecution ();

    // functional part
    if (c == true)
        read (inT, d);
    else
        read (inF, d);

    write (outData, d);
}
```

5.4.3 Data-dependent iterations

The last type of dynamic behaviour that can be modelled with a new, predictable construction, is the data-dependent iteration (DDI, see Section 4.4). For the DDI construct, two special actors are needed: the DDI-open and DDI-close actors. Both can be implemented in HAPI in a straightforward manner. Just like is the case for conditionals, these actors can be implemented as separate `Actors` in HAPI, or incorporated as parts of other `Actors`. In Listing 5.6 and Listing 5.7 examples of both actors as standalone `Actors` are given. The DDI-open has one data input `inData`, a data output `outData` and a iterator output `outI`. Similarly, the DDI-

close actor has the following Ports: inData, inI and outData.

Listing 5.6: DDI-open `Actor` *example*

```
int I;
double a;

while (true)
{
    // ** first actor

    peek_wait (inData);
    read (inData, I);

    // ** second actor: loop

    // wait until actor is enabled
    peek_wait (outI);
    peek_wait (outData, I);

    // advance simulation time
    WaitScheduling ();
    WaitExecution ();

    // functional part
    write (outI, I);
    a = 1;
    for (int i = 0; i < I; i++)
    {
        write (outData, a);
        a *= .5;
    }
}
```

Listing 5.7: DDI-close `Actor` *example*

```
int I;
double a, b;

while (true)
{
    // ** first actor

    peek_wait (inI);
    read (inI, I);

    // ** second actor: loop

    // wait until actor is enabled
    peek_wait (inData, I);
    peek_wait (outData);

    // advance simulation time
    WaitScheduling ();
    WaitExecution ();

    // functional part
    b = 0;
    for (int i = 0; i < I; i++)
    {
        read (inData, a);
        b += a;
    }
    write (outI, b);
}
```

5.5 Conclusions

When dealing with jobs that have soft real-time constraints and when information about the transient phase of the execution is needed, timing analysis of a job does not provide the necessary data. In these cases, simulations can serve as a source of information.

This chapter introduces the HAPI simulator, as it was developed in Philips Research. Furthermore, it describes useful extensions and indicates how dynamic behaviour, as introduced in the PDDF model of computation in Chapter 4, can be incorporated.

Contributions of this chapter are:

- Extensions to the existing HAPI simulator for:

  - aquisition of detailed timing information, to derive a job's throughput and the number of deadline misses.
  - explicit actor-to-processor assignment and scheduling.

- Introduction of dynamic behaviour in HAPI (Goal 5):
  - data-dependent processing times.
  - conditionals.
  - data-dependent iterations.

Chapter 6

Case study: H.263 decoder

6.1 Introduction

The eventual goal of the design approach that is discussed in this thesis, is to arrive at efficient implementations of streaming multimedia jobs. To illustrate this, the theory has been applied to a practical test case: an H.263 video decoder. This chapter gives an overview of the decoder and shows how a decoder job can be modelled. The goal of this case study is to compare the results of the SDF and PDDF based approaches. Initially, an SDF model is used to show the limitations of SDF and to serve as a reference. Then, the new PDDF techniques are used to arrive at a model that better reflects the dynamic behaviour of the job. The end result is a mapping of the H.263 job to a multiprocessor platform that is much more efficient in terms of processing resources when PDDF is used, than in the SDF case.

Section 6.2 first briefly explains the basics of the H.263 decoder. Subsequently, the structure of the decoder job is explained and SDF and PDDF specification models are derived. The next step in the design flow (see Section 3.2) is to extend the job model with mapping details. Two (not necessarily optimal) mappings are suggested and incorporated in the SDF and PDDF graphs. The next design step depends on the type of real-time contraints the job has to deal with: timing analysis is carried out for jobs with hard real-time contraints, while simulation is done for soft real-time jobs. Both options are worked out for the H.263 decoder, respectively in Sections 6.3 and 6.4. The results are summarised in the last section of this chapter.

6.2 The H.263 model

6.2.1 H.263 overview

H.263 is a video codec that was developed by the telecommunication standardisation organisation ITU-T, mainly for video conferencing. For this case study, a fully functional H.263 decoder implementation in YAPI [12], coded at the TU/e, was used as a starting point. This decoder can process H.263 encoded video streams, having a so-called CIF resolution (352 $\times$ 288 pixels). The video stream consists of *frames*. In a frame a picture is encoded, which consists of 396 *macro blocks* of 16 $\times$ 16 pixels, which in turn contain six *blocks* with colour information each. The decoder supports two types of video frames: intra frames (I-frames), in which a complete picture is coded (2376 blocks), and predicted frames (P-frames),

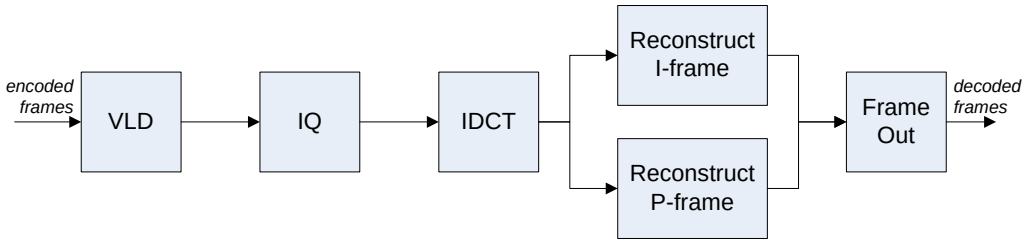


Figure 6.1: H.263 decoder – block diagram

which contain only the differences between the current and the previous frame. A P-frame typically contains much fewer blocks than an I-frame; the number of blocks can vary per frame.

For the purpose of this case study, it is not important to know the exact details of H.263 decoding. Globally, the decoding steps are expressed in six main tasks (see Figure 6.1). Every frame in a video stream first has to be decoded by a Variable Length Decoder (VLD). After this is done, the type of the frame (I or P) and the number of blocks it contains is known. Every block inside the frame has to pass an Inverse Quantiser (IQ) and an Inverse Discrete Cosine Transformer (IDCT). For I-frames, the next step is simply putting the blocks into place to form the picture (the Reconstruct I-frame task), as all blocks are now available. P-frames, however, need more computation: motion estimation is used to put blocks from the previously decoded frame to a position in the current frame and several blocks have to be updated according to the differences that are encoded in the P-frame's blocks. This is done in the Reconstruct P-frame task. The decoded frames are gathered from the Reconstruct tasks by a task called FrameOut that outputs the frames to the environment in the correct order. Each of these tasks has been implemented in C.

Table 6.1: H.263 task processing time upper-bounds

Task t	$P_t(blks)$ (i686 processor cycles)
VLD	$13025 + 302534 \cdot blks$
IQ	830
IDCT	2300
Reconstruct I-frame	$9.62 \cdot 10^6$
Reconstruct P-frame	$19.63 \cdot 10^6 + 4354 \cdot blks$
FrameOut	0

Upper-bounds for the processing times of each task, in terms of processor cycles on the i686 instruction set architecture (e.g. Pentium II), were obtained through a new tool that is being developed at TU/e for this purpose. These results are given in Table 6.1. The processing times of IQ, IDCT and Reconstruct I-frame are not data-dependent and thus constant for every execution. The processing time of FrameOut is neglected, as this task is only used for synchronisation. The VLD and Reconstruct P-frame tasks, however, have execution times that depend, among others, on the number of blocks that are processed. Therefore, the processing times for these tasks are expressed as linear functions of the number of blocks; these timings are still upper-bounds for that number of blocks. There is one important remark about the processing time upper-bound of the VLD. In the input stream, symbols

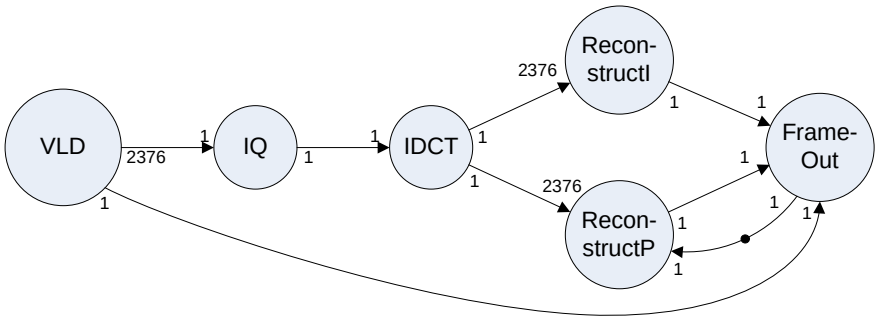


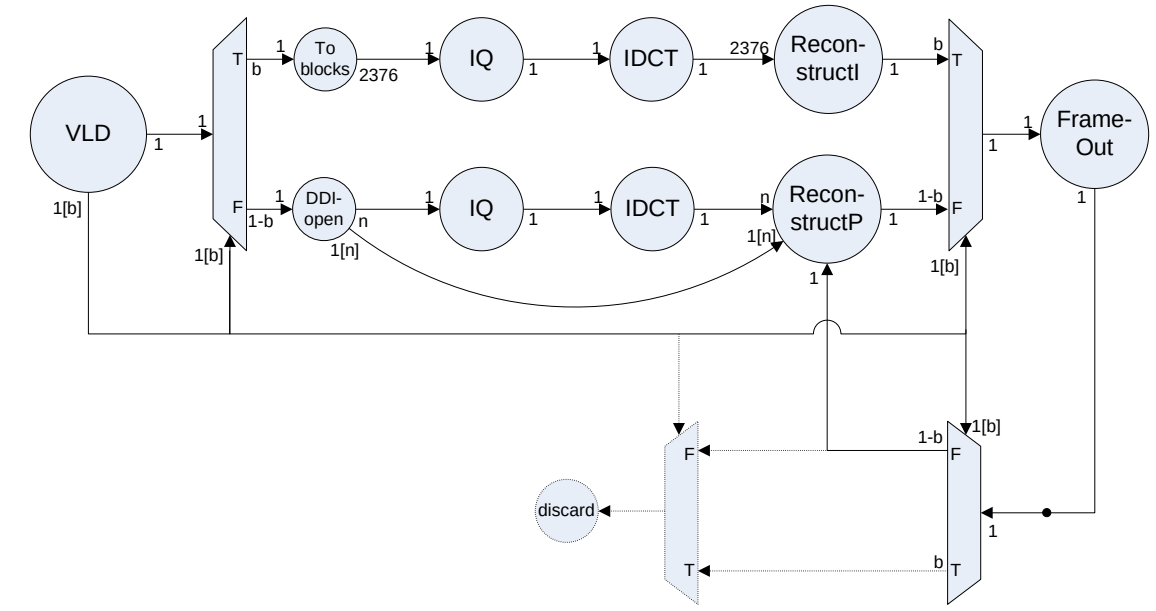
Figure 6.2: H.263 decoder – SDF

with a certain fixed length (number of bits) are encoded with a variable length. Depending on how often the symbol occurs in the stream it is given a length shorter or longer than the original length, so that the total stream will be shorter, up to a factor of more than a hundred. The processing time to decode a symbol is roughly proportional to its length. To bound the processing time, it was assumed that every symbol in a frame is encoded with the maximum length, which will lead to a huge over-estimation of the real frame length and so also of the processing time of the VLD task. This makes the VLD actor by far the most computationally intensive and a large bottleneck in this H.263 job. However, the purpose of this case study is to show the advantages of PDDF over SDF. The VLD is not important here, as it will not appear in a PDDF construct. Therefore, in the simulations, in which "real" processing times are supposed to be used, an average symbol length was assumed for VLD input streams, to ensure that the VLD does not have such a large and unrealistic influence. Note that the simulations in this case are no longer always conservative, but the situation is much more realistic and the example is more illustrative for the comparison of SDF and PDDF.

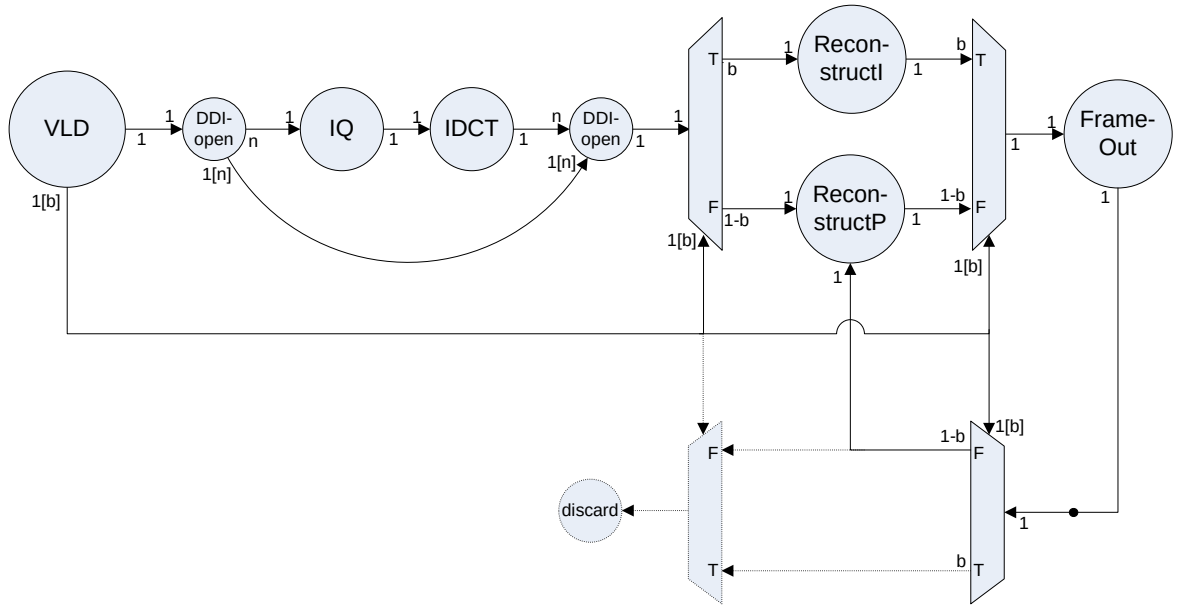
In the next sub-section, the decoder job is modelled with SDF and PDDF.

6.2.2 SDF and PDDF graphs for specification

The SDF specification of the decoder job follows quite straightforwardly from the block diagram. Figure 6.2 shows the resulting SDF graph. Note that in this model the job's inputs and outputs are not explicit: it is assumed that the VLD and FrameOut actors handle the interaction with the environment, which is never constraining the job. This implies that input data is always provided at a sufficiently high rate and output data can at any time be offered to the environment. The VLD, ReconstructI, ReconstructP and FrameOut actors all process one complete frame per firing. The IQ and IDCT actors work at block level, and therefore they fire 2376 times for each frame. As SDF does not support data-dependent firing of actors, the distinction between I- and P-frames cannot be made. Consequently, the ReconstructI and ReconstructP actors have to be fired *both* for each frame, regardless the frame type. Also, as the rates throughout the whole SDF graph have to be fixed, every frame has to be considered to contain the maximum 2376 blocks, even though P-frames generally contain a smaller number. Furthermore, it is always needed to send the previously decoded frame back to the ReconstructP actor, as it is needed for the decoding of P-frames (and any frame may be a P-frame). This lack of expressiveness of the SDF model of computation makes it quite counter-intuitive to use it for the H.263 decoder job and, as will become clear, leads to less efficient implementations.



(a) option 1



(b) option 2

Figure 6.3: H.263 decoder – PDDF

Two forms of dynamism are apparent in the H.263 decoder. There are two conditional paths: one for I-frames and one for P-frames. Also, for P-frames, the number of encoded blocks may vary. PDDF constructs can be used to deal with these data-dependencies. The conditional construct can be used to explicitly create separate branches for I-frames and P-frames. Moreover, the data-dependent iteration (DDI) construct is capable of dealing with the varying number of blocks inside the P-frames. Figure 6.3 shows two possible ways to construct the PDDF graph for this job. The first one separates the decoding of I- and P-frames altogether, by duplicating the IQ and IDCT actors to divide them over the two branches. A DDI construct is used only inside the P-branch in this case. The other option shares the IQ and IDCT for both frame types and embeds them into a DDI construct, while only the Reconstruct actors are put into conditional branches. It depends on the mapping choices which option is the best. Both options need a special way to lead the decoded frame back to the ReconstructP actor, when the next frame is a P-frame: as ReconstructP is now nested inside a conditional construct, no incident arcs from outside that particular conditional branch are allowed. The only exception is an arc from another conditional that is controlled by the *same* Boolean stream. In Figure 6.3 another conditional is drawn, which either leads the frame back to ReconstructP, or discards it. The Boolean streams to both conditionals are identical, as they all come directly from the VLD, without any delay. The dotted parts of the conditional are actually not needed; they are only drawn to prove the correctness of the models. As the SWITCH and SELECT actors only transfer one data token per firing, an extra actor (called "To blocks") has to be introduced in the I-branch of the first option, to convert the frame tokens into 2376 block tokens. In the SDF case, this is done directly by the VLD and in the second PDDF case the DDI actors perform this task.

To complete the specification, in both cases (SDF and PDDF) self-edges have to be introduced around every actor, to ensure that multiple instances of an actor cannot be fired concurrently. The H.263 job is now ready to be mapped to the multiprocessor platform.

6.2.3 Mapping-extended graph

It is assumed that three processors are available for this job on the target multiprocessor platform: π_1, π_2 and π_3. All processors use the i686 instruction set and run at a clock speed of 1 GHz. It is further assumed that there is always sufficient local memory in every processor tile and sufficient communication bandwidth between every processor available. When multiple actors are mapped to the same processor, static assignment scheduling is applied in the way that is described in Section 2.5: TDMA is used to fix a processing budget for actors of a certain job, and within the job's time slot, Round Robin arbitration is used to schedule these actors. When the decoder has been mapped to the platform, it needs to be capable of producing decoded frames at a rate of 25 frames per second. The goal is now to reserve time slots of minimum size for the job on the three processors, while this throughput constraint is still met.

In a first attempt to distribute the workload evenly over the processors, the VLD actor is mapped to π_1, the IQ and IDCT actors are mapped to π_2 and the Reconstruct and FrameOut actors are mapped to π_3. The dataflow models can now be extended with mapping details, according to the steps in Section 3.4.2. The first two steps – the creation of the initial dataflow graph and the introduction of self-edges around every actor – have already been done. An additional step for the PDDF case, is to substitute the more detailed versions of the special dynamic actors (like SWITCH and SELECT), as explained in Section 4.3, and extra

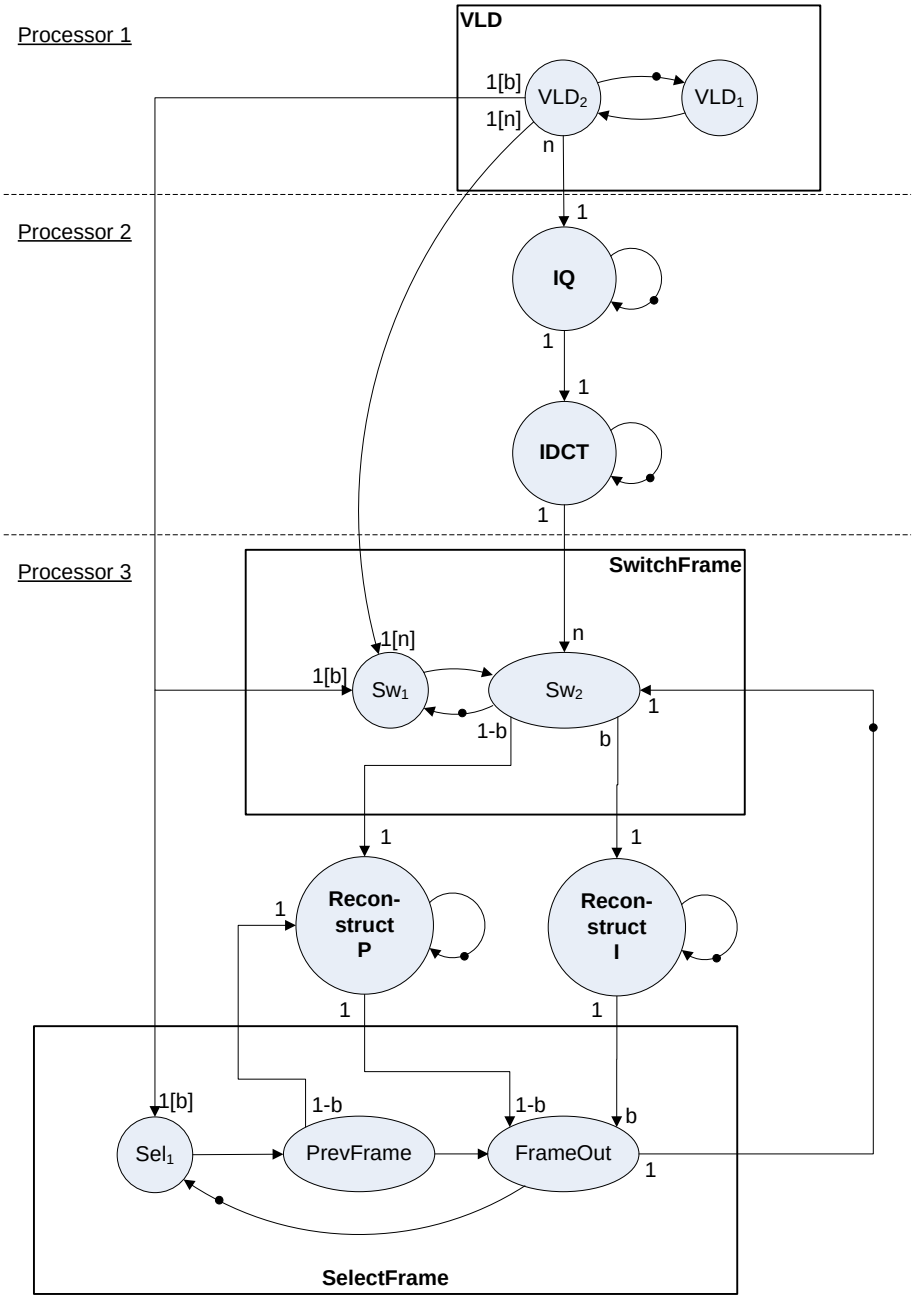


Figure 6.4: H.263 decoder – PDDF

feedback arcs may be introduced to make the actors in a conditional mutually exclusive. In Appendix A the full PDDF graphs for both options are given. In Figure 6.4 the second PDDF graph is presented again, but in this picture only the parts that are implemented are drawn (the dotted parts are left out) and several PDDF actors are combined to make the graph more compact. Actors inside a rectangle are always statically ordered (they are in a loop with one initial token), to indicate that they share a common state. The behaviour of this graph is still equal to the behaviour of the graph in Appendix A and this is the model that will be used for analysis and simulation. The first option's PDDF graph is not drawn again; as it will appear later, the second version has advantages, so that one will be used. When comparing this PDDF graph to the SDF graph (Figure 6.2), it appears that two extra SWITCH actors are introduced (Sw_1 and Sw_2), that the VLD is split up into two actors (Vld_1 and Vld_2) and FrameOut is extended with two additional actors (Sel_1 and PrevFrame), forming a loop called SelectFrame.

The next step is the annotation of the actors with response times. The response time of an actor does not only depend on its processing time, but also on the waiting and interruption times that are introduced by the scheduling method that is used. How to calculate the response times for the combined TDMA/Round Robin scheduling, is derived in Section 3.5.3. For conditional PDDF constructs, special rules have to be taken into account, to reduce the waiting times (see Section 4.3). All extra introduced actors in the PDDF graph are assumed to have a processing time of zero. Now the sizes of the time slots for the job on each processor have to be chosen. First, the job is given the full processors (the TDMA fraction $f_\pi = 1$ for all π), to start with a least constraining option for this particular actor-to-processor mapping. To arrive at minimum processing budgets, the time slots are gradually decreased until the point that the throughput is just met. As the actors on a processor inside the job's time slot are Round Robin scheduled, a certain order has to be chosen. Knowledge about this order may help to derive tighter bounds on the response times of actors. For the SDF example the following is chosen:

$$\begin{cases} \pi_1 : \text{VLD} \\ \pi_2 : \text{IQ} \rightarrow \text{IDCT} \\ \pi_3 : \text{ReconstructI} \rightarrow \text{ReconstructP} \rightarrow \text{FrameOut} \end{cases} \tag{6.1}$$

According to Section 3.5.3, an upper-bound on this response time is equal to the processing time of the actor, plus a waiting time comprising of the processing time upper-bounds of the other actors on the same processor. However, for some actors a much sharper upper-bound on the waiting time can be derived. An overview of the waiting times for the H.263 actors is given in Table 6.2. For the VLD, the waiting time is easy to derive: as it has no other actors of the decoder job on its processor (π_1), no scheduling is needed and its waiting time is equal to zero. For processor π_2 the story is a little more complicated. The waiting time of IQ is equal to the processing time upper-bound of IDCT, as usual. But, due to the arc from IQ to IDCT, IDCT is enabled always immediately after IQ completes a firing. Also, the Round Robin mechanism ensures that IDCT gets the chance to fire immediately after IQ, and so it will. Thus, the waiting time for IDCT is always zero. A similar story holds for processor π_3. The waiting times for ReconstructI cannot be further tightened: as it is solely enabled by an actor that runs on another processor, this can happen at any point in time. When ReconstructP becomes enabled, it never has to wait for FrameOut, as both actors are in a loop with one initial token on it, so they can never be active at the same time. Thus,

the waiting time for ReconstructP is only the processing time of ReconstructI. FrameOut becomes enabled after a firing of either ReconstructI or ReconstructP. By inspection of the Round Robin list of this processor, it becomes clear that, when FrameOut becomes enabled after a firing of ReconstructP, it can fire right away. When it becomes enabled after a firing of ReconstructI, it may also fire immediately, because ReconstructP can never be active at the same time. Hence, the waiting time of FrameOut is equal to zero.

Table 6.2: SDF actor waiting times

Actor a	$W(a)$
VLD	0
IQ	$P(\text{IDCT}) = 2300$
IDCT	0
ReconstructI	$P(\text{ReconstructP}) + P(\text{FrameOut}) = 29.97 \cdot 10^6$
ReconstructP	$P(\text{ReconstructI}) = 9.62 \cdot 10^6$
FrameOut	0

In the PDDF case, the waiting time upper-bounds can be even much smaller than is the case for SDF, because of the use of conditionals. To exploit this fact, the two branches in a conditional construct need to be made mutually exclusive by adding an arc with a single initial token from the actor directly after the SELECT back to be actor directly before the SWITCH. In PDDF option 1, this would mean an extra act from FrameOut back to VLD. However, this will prohibit pipelining over frames, as the three processors cannot run concurrently anymore. The second PDDF option is much better, as the conditional construct is entirely located on a single processor. The feedback loop around this single processor is no real restriction, as actors on a processor can anyway never run in parallel. The loop only statically orders the execution of the actors. Therefore, only the second PDDF option, the graph as in Figure 6.4, is worked out in this exercise. For the VLD, IQ and IDCT actors the waiting times are equal to those in the SDF case. On processor π_3 it is different, because of the conditional and the loop: Sw_2, the Reconstructs and FrameOut are now all mutually exclusive. Consequently, their waiting times could only depend on the other actors on π_3, but these are all zero. Therefore, the waiting times of all actors in this loop are zero. Also for the actors in the loop ($\text{Sel}_1 \rightarrow$ PrevFrame $\rightarrow$ FrameOut) it can be seen that they can only be enabled when actors with zero processing time are active, which means that also these actors have a no waiting time. Only the Sw_1 has a waiting time: it may have to wait for either ReconstructI or ReconstructP, which is at maximum $29.97 \cdot 10^6$. Note that even though the waiting times in the PDDF case are lower than in the SDF case, this does not necessarily mean that the minimum throughput will be higher as well: an extra cycle was introduced in the PDDF graph and the throughput depends on which cycles are critical. Analysis or simulation is needed to derive the minimum throughput that can be guaranteed.

The fourth step in the creation of mapping-extended graphs is the introduction of bounded FIFOs by doubling every arc that exists in the graph, reversing these copies and adding a number of initial tokens to every new arc, which represents the buffer size. These arcs are not drawn in the graphs in this chapter to keep the pictures simple. The FIFO sizes that are used in this case study are given in Appendix A.

The last step would be to add the network channel model for inter-processor communication, but this is left out for simplicity in this case study. This also implies that all FIFO

latencies are assumed to be zero. These resulting graphs can be analysed or simulated, to see what will be the minimum throughput of the job for a particular mapping. This is done in the following sections.

6.3 Timing Analysis

The timing analysis of SDF graphs is done with an application, called Heracles, that is being developed at Philips Research. Heracles can transform any consistent and deadlock-free SDF graph into an HSDF representive and performs MCM analysis (see Section 3.3.2). This is done for the mapping-extended SDF graph that was derived in the previous section. The MCM, which is the result of the analysis, is equal to the property λ of the HSDF graph. The average number of firings per period of an actor in the HSDF graph, during the periodic phase of its self-timed execution, is equal to $\frac{1}{\lambda}$. In the H.263 case, FrameOut is the actor that produces frames to the output of the decoder, so the throughput of this actor determines the throughput of the job as a whole. FrameOut has one corresponding actor in the HSDF graph (its repetition factor equals one) and, per firing, it produces one token to the environment. Therefore, the throughput of FrameOut, and thus the throughput of the H.263 decoder, is equal to $\frac{1}{\lambda} = \frac{1}{\text{MCM}}$. It is now assumed that the H.263 decoder is a job with hard real-time constraints, so for any possible input stream the required output rate of 25 frames per second has to be realised.

It appears that the critical cycle in the graph (the cycle that is responsible for the worst-case throughput), is the cycle containing the VLD only. This was expected, because the worst-case processing time of the VLD was extremely large. Even though it is very unlikely that this worst case will ever occur (not even close), it has to be taken into account for a strictly hard real-time job. Given this fact, the guaranteed throughput for the H.263 job using this mapping is only 1.4 frames per second.

However, as the purpose of this case study is to show the advantages of the dynamic PDDF constructs, it is assumed that the VLD can be taken care of in some way (for example by putting it on a hardware accelerator), so it will no longer be present in a critical cycle. In this case, the self-loops around ReconstructI and ReconstructP become the critical cycles. They both have a cycle mean of $39.59 \cdot 10^6$ clock cycles, which leads to a guaranteed throughput of 25.3 frames per second. The critical cycles are completely contained on processor π_3. As the constraint is just met, it makes no sense to try a smaller time slot on π_3. However, it may be possible to reduce the time slot on processor π_2, while maintaining the same throughput (the VLD's processor π_1 is not taken into consideration). When reducing π_2's time slot down to 25%, the MCM still appears to be below $40 \cdot 10^6$. Consequently, for this hard-real time H.263 case, the full processor π_3 and 25% of π_2 are needed, plus extra hardware for the VLD.

Also the mapping-extended PDDF graph is analysed with Heracles. For a conservative analysis, it is necessary to replace all data-dependent rates in the graph by upper-bounds. This means that every rate b or $1 - b$ is replace by 1 and every n is replaced by $N = 2376$. Now, analysis can be done on the PDDF graph, as if it were an SDF graph.

As the part of the PDDF graph that is mapped to the processors π_1 and π_2 is basically the same as in the SDF case, the same issues arise here. It is again assumed that an accelerator is used for the VLD, so this becomes no bottleneck. Again, 25% of π_2 appears to be needed. This cannot be further reduced, even though there is a data-dependent iteration construct on

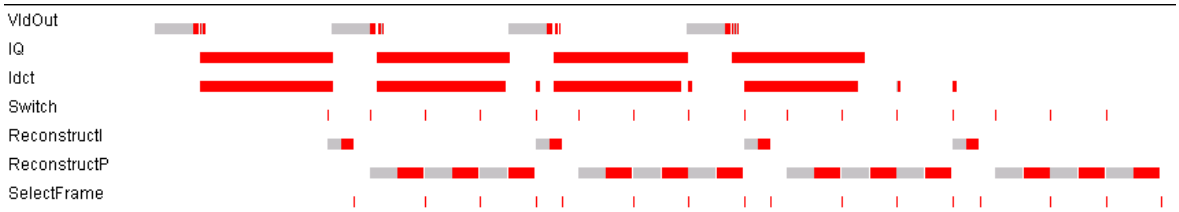


Figure 6.5: H.263 timeline

π_2; in hard real-time cases like this, the DDI construct offers no gain in resources. However, π_3 contains the conditional construct that improves the maximum cycle mean compared to the SDF case. The maximum cycle mean is now $30.0 \cdot 10^6$, caused by the ReconstructP actor. This implies that there is some room to save processing time on π_3. When a smaller time slot is chosen, the response times of ReconstructI and ReconstructP have to be updated accordingly. To meet the throughput constraint, the critical cycle is allowed to have a cycle mean of $40.0 \cdot 10^6$. When a TDMA period of 10000 cycles is used and a fraction of 75% is appointed to the H.263 job, the response time of ReconstructP is exactly $40.0 \cdot 10^6$. Thus, while saving 25% on processor π_3, the constraint can still be met in the PDDF case, while this was not possible in the SDF case.

6.4 Simulation

In video jobs, like H.263, it is typically not a problem when occasionally a deadline is missed. When a frame is too late, for example, the previous frame could be displayed again. When this does not happen too often, the user will not notice it. Because of this, it makes sense to define the timing constraints for such jobs as soft real-time rather than hard real-time. Simulations with a representative set of input streams can be done on the SDF or PDDF model of the job to derive a suitable mapping.

Simulations have also been done for the H.263 decoder job. The job was modelled in the HAPI simulation environment, as described in Chapter 5. Only a single type of input stream is used in the simulations. This is hardly a representative set, but since this case study is only meant to show the advantages of PDDF over SDF, it is sufficient. The input stream contains four times the following frame sequence: I-P-P-P. The P-frames contain respectively 16, 28 and 42 blocks. This movie was tested on the mapping-extended SDF and PDDF graphs as described in Section 6.2. The same actor-to-processor mapping was used as for the analysis of the previous section. For the PDDF graph, where actors are drawn in a box in Figure 6.4, they are combined in one HAPI `Actor`-derived class. Again, the goal was to decrease the time slots on the processors as much as possible. The throughput of the test sequence was attempted to be about 25 frames per second. The throughput is again defined as the average number of FrameOut firings per second, measure from the first until the last firing.

Because no hard real-time guarantees have to be given anymore, the VLD problem that occurred during the analysis, was tackled by simply using average-case processing times for the VLD. The processing times were also obtained from the VLD C-code by the software that is being developed at TU/e. This makes the simulations much more realistic.

Firstly, the SDF model was simulated to serve as a reference. It appeared that, when fully using the three processors, the throughput becomes about 25 frames per second, as

was the case for analysis as well. Again, the critical cycles seem to reside fully on π_3, so it may be possible to choose smaller time slots on the other two processors. After trying some configurations, it seems that the TDMA fractions of the job on processors π_1 and π_2 can both be reduced to 25%, while the average throughput is still about 25 frames per second. In total, only 1.5 processing resources are needed to reach the goal.

As the H.263 decoder inherently has a lot of dynamism, it is expected that the PDDF model leads to much better results compared to the SDF model. When simulating this model with three full processors, it appeared that the throughput was about 56 frames per second. As this is much more than the goal, the TDMA time slots on the processors were gradually lowered, until the throughput was 25 frames per second. In this configuration, the time slots for the processors π_1, π_2 and π_3 respectively were 15%, 10% and 49%. In fact all actors now fit on a single processor: the total amount of needed processing resources is 0.74. Compared to the SDF case, this is a gain in processing time of more than 50%. The time line that is produced by HAPI is shown in Figure 6.5. There are horizontal bars for each HAPI `Actor` that represent the time shapes. The bars are dark when the `Actor` is processing; the waiting and interruption times are combined in a grey bar in front of a dark bar.

6.5 Conclusions

An implementation of the video decoder H.263 was used as a practical case study in this chapter, to show the benefits of the dynamic constructs of the PDDF model, compared to SDF. The design methods that were introduced in Chapter 3 were applied on the H.263 decoder and the analysis and simulation methods were used to arrive at an efficient implementation in terms of processor usage. It is shown how the PDDF constructs can be used to deal with the various data-dependencies that are inherent in the H.263 decoder. Both the hard real-time and the soft real-time cases were investigated. It appeared that in the hard real-time case, the usage of PDDF leads to a reduction of 25% of the needed processor time on one of the processors. But especially in the soft real-time case the advantages of PDDF become clear: the total gain over all processing resources was slightly over 50%.

Contributions of this chapter are:

- SDF and PDDF models of the H.263 decoder for analysis and simulation.
- Quantitative support for the statement that PDDF leads to more efficient implementations compared to SDF, for both hard and soft real-time, based on a practical test case. This corresponds to Goal 6.

Chapter 7

Conclusions & recommendations

Throughout this document it became clear what the issues are in the area of predictable job design. It appeared that a more precise job model that explicitly uses knowledge about certain types of dynamic behaviour in the job, can lead to more efficient implementations. Below is a summary of the most important results. The numbers correspond to the Goals that were stated in Section 1.2.

1. Three important types of dynamism that can be present in a job are considered: data-dependent processing times, conditionals and data-dependent iterations.

2. An overview of a model-driven job design flow around the SDF model of computation is presented in Chapter 3. Some points in the design flow have been refined: the specification of mapping-extended SDF graphs has been precisely defined and better methods to include scheduling of actors on processors have been derived.

3. Chapter 4 introduces the Predictable Dynamic Dataflow (PDDF) model of computation, which extends SDF with construction rules to include conditionals and data-dependent iterations. In SDF, only upper-bounds on dynamic constructions can be modelled, which is very inaccurate in many cases. The new PDDF constructs are fully modular, so they can be nested in any way. Furthermore, their input and output rates are fixed, so they can be plugged into any existing SDF graph. Just like for SDF, the correct construction of a PDDF graph guarantees that the timing and the memory usage of the resulting job is bounded and these bounds can be obtained by analysis or simulation. PDDF can be used in exactly the same design flow as SDF, as mentioned in the previous point.

4. Also in Chapter 4 it is shown how the use of PDDF models can lead to implementations that need less resources compared to implementations that are obtained through the SDF-based approach. For hard real-time jobs, the total gain can be up to 50% (when a job has two completely different modes of equal size); for soft real-time jobs, this could be even more, depending on the amount of dynamicity that is present in the job.

5. An existing simulator for SDF models, called HAPI, was extended with the possibility to explicitly select an actor-to-processor mapping and to obtain timing results for each actor. It is shown in Chapter 5 how this works, and also how the new PDDF constructs can be used in HAPI.

6. As a practical test case, a fully functional model of an H.263 video decoder has been studied. It appeared that, when using the PDDF model of computation to model the H.263 job, a lot of resources can be saved: considering the job as hard real-time, the gain in processing resources was about 25%. In the soft real-time case, using one specific input stream, the savings were even over 50%.

With these results, the research question that was formulated in Section 1.2 is considered to be sufficiently answered.

Although dynamic behaviour in a single job can be treated well now, there are still a lot of issues that deserve further research. These include, in no particular order:

- Models for soft real-time fall-back mechanisms. How can fall-back mechanisms be taken into account in timing analysis and simulations?
- Refinement of the design flow, especially the mapping step.
- More precise timing of token production/consumption in simulations to arrive at sharper results. Currently, it is assumed that these events happen at the end of an actor's firing, to be sure that the simulation is conservative.
- A better characterisation of input streams: this extra knowledge could be used to arrive at sharper results.
- The estimation of waiting times for static assignment schedules: a more structured method is desirable.
- A model for predictable run-time reconfigurations of a multiprocessor platform.
- Inter-job communication: how can jobs communicate with each other, while both remain independent and predictable?

Bibliography

[1] N Bambha, V. Kianzad, M. Khandelia, and S.S. Bhattacharyya, *Intermediate representations for design automation of multiprocessor DSP systems*, Design Automation for Embedded Systems **7** (2002), 307–323.

[2] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen, *Predictable embedded multiprocessor system design*, SCOPES 2004, 8th International Workshop on Software and Compilers for Embedded Systems. Amsterdam, The Netherlands, September 2004.

[3] M. Bekooij and J. van Meerbergen, *Timing analysis of data driven hard-RT multiprocessor systems*, submitted to DATE 2005.

[4] B. Bhattacharya and S.S. Bhattacharyya, *Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems*, 11th International Workshop on Rapid System Prototyping, June 2000, pp. 21–23.

[5] J.T. Buck, *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*, Ph.D. thesis, University of California, Berkeley, 1993, p. 154.

[6] A. Dasdan and R.K. Gupta, *Faster maximum and minimum cycle algorithms for system-performance analysis*, IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems **17** (1998), no. 10, 889–899.

[7] K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Rădulescu, E. Rijpkema, E. Waterlander, and P. Wielage, *Guaranteeing the quality of services in networks on chip*, Networks on Chip (Axel Jantsch and Hannu Tenhunen, eds.), Kluwer, 2003, pp. 61–82.

[8] S. Ha, *Compile-time scheduling of dataflow program graphs with dynamic constructs*, Ph.D. thesis, University of California, Berkeley, 1992.

[9] S. Ha and E.A. Lee, *Compile-time scheduling of dynamic constructs in dataflow program graphs*, IEEE Trans. Comput. **46** (1997), no. 7, 768–778.

[10] M. Jersak, R. Henia, and R. Ernst, *Context-aware performance analysis for efficient embedded system design*, Proceedings of the conference on Design, automation and test in Europe, IEEE Computer Society, 2004, p. 21046.

[11] G. Kahn, *The semantics of a simple language for parallel programming*, IFIP Congress, North-Holland Publishing Co, 1974, pp. 471–475.

[12] E. de Kock and G. Essink, *Y-chart application programmer's interface, version 1.1*, Philips Research Eindhoven, 2000.

[13] R. Lauwereins, M. Engels, M. Ade, and J.A. Peperstraete, *Grape-II: A system-level prototyping environment for DSP applications*, Computer **28** (1995), no. 2, 35–43.

[14] E.A. Lee, *Consistency in dataflow graphs*, IEEE Trans. Parallel Distrib. Syst. **2** (1991), no. 2, 223–235.

[15] E.A. Lee and S. Ha., *Scheduling strategies for multiprocessor DSP*, IEEE Global Telecommunications Conference and Exhibition, Dallas, TX, vol. 2, IEEE Computer Society, November 1989, pp. 1279–1283.

[16] E.A. Lee and D.G. Messerschmitt, *Static scheduling of synchronous data flow programs for digital signal processing*, IEEE Trans. Comput. **36** (1987), no. 1, 24–35.

[17] E.A. Lee and T.M. Parks, *Dataflow process networks*, Proceedings of the IEEE (1995), 773–799.

[18] *http://www.omg.com/mda*.

[19] A. Moonen, *Modeling and simulation of guaranteed throughput channels of a hard real-time multiprocessor system*, Master's thesis, Eindhoven University of Technology, January 2004.

[20] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman, *Task-level timing models for guaranteed performance in multiprocessor networks-on-chip*, Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems, ACM Press, 2003, pp. 63–72.

[21] P. Poplavko, M. Pastrnak, T. Basten, J. van Meerbergen, and P.H.N. de With, *Mapping of an MPEG-4 shape-texture decoder onto an on-chip multiprocessor*, PRORISC 2003, 14th Workshop on Circuits, Systems and Signal Processing, Veldhoven, The Netherlands, November 2003, pp. 26–27.

[22] *http://www.real-time.org*.

[23] S. Sriram and S.S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*, Marcel Dekker, Inc., 2000.

[24] P. Wauters, M. Engels, R. Lauwereins, and J.A. Peperstraete, *Cyclo-dynamic dataflow*, Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96), IEEE Computer Society, 1996, p. 319.

List of Definitions

Appendix A

H.263 decoder – case study

A.1 Expanded PDDF graphs

See pages 88 and 89 for two examples of expanded PDDF graphs for the H.263 decoder.

A.2 FIFO buffer sizes

Table A.1: H.263 FIFO sizes: SDF graph

Actor a	**Size (#tokens)**
VLD → IQ	2376
IQ → IDCT	2376
IDCT → ReconstructI	2376
IDCT → ReconstructP	2376
ReconstructI → FrameOut	1
ReconstructP → FrameOut	1
FrameOut → ReconstructP	1

Table A.2: H.263 FIFO sizes: PDDF graph

Actor a	**Size (#tokens)**
VLD_2 → IQ	2376
VLD_2 → Sw_1	10
VLD_2 → Sel_1	10
IQ → IDCT	2376
IDCT → Sw_2	2376
Sw_2 → ReconstructI	1
Sw_2 → ReconstructP	1
ReconstructI → FrameOut	1
ReconstructP → FrameOut	1
PrevFrame → ReconstructP	1

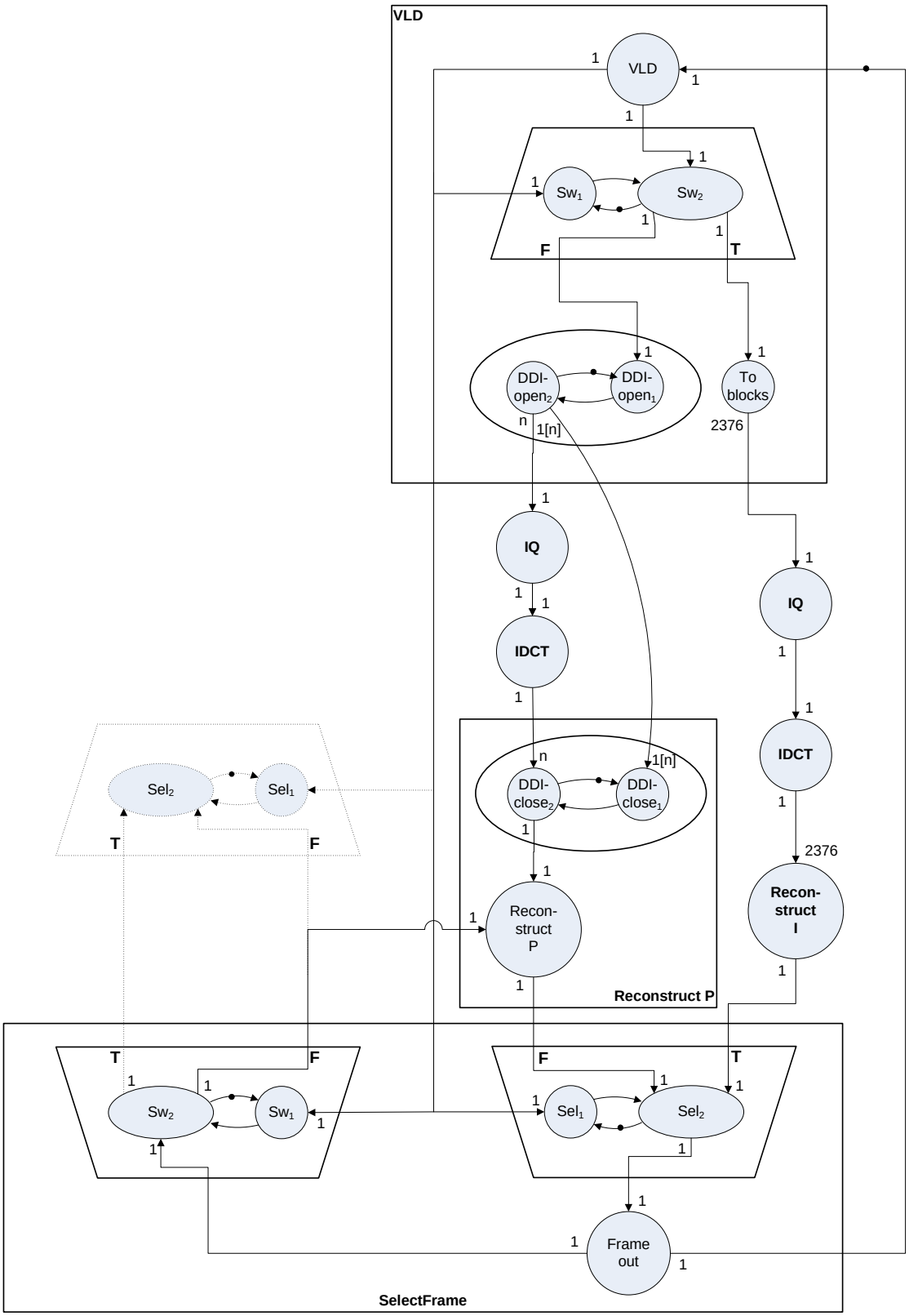


Figure A.1: H.263 decoder – mapping option 1

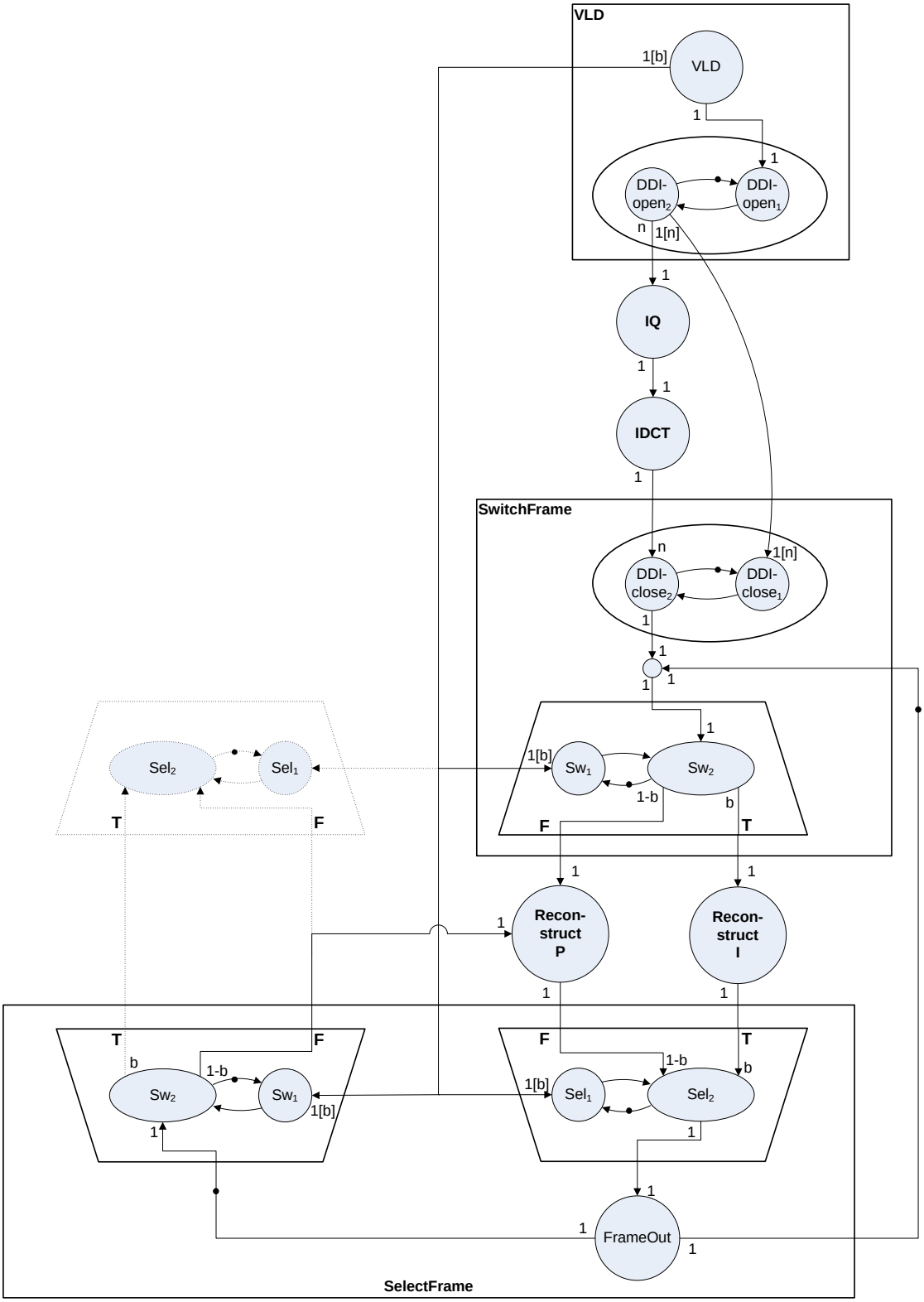


Figure A.2: H.263 decoder – mapping option 2