Chapter 2
Application Modeling and Scheduling

Multimedia applications are becoming increasingly more complex and computation hungry to match consumer demands. If we take video, for example, televisions from leading companies are available with high-definition (HD) video resolution of 1080 × 1920 i.e. more than 2 million pixels (Sony 2009; Samsung 2009; Philips 2009) for consumers and even higher resolutions up to quad HD are showcased in electronic shows (CES 2009). Producing images for such a high resolution is already taxing for even high-end MPSoC platforms. The problem is compounded by the extra dimension of multiple applications sharing the same resources. *Good modeling* is essential for two main reasons: (1) to predict the behaviour of applications on a given hardware without actually synthesizing the system, and (2) to synthesize the system after a feasible solution has been identified from the analysis. In this chapter we will see in detail the model requirements we have for designing and analyzing multimedia systems. We discuss various models of computation, in particular so called dataflow models, and choose one that meets our design-requirements.

Another factor that plays an important role in multi-application analysis is determining when and where a part of application is to be executed, also known as *scheduling*. Heuristics and algorithms for scheduling are called *schedulers*. Studying schedulers is essential for good system design and analysis. In this chapter, we discuss the various types of schedulers for dataflow models. When considering multiple applications executing on multi-processor platforms, three main things need to be taken care of: (1) *assignment* – deciding which task of application has to be executed on which processor, (2) *ordering* – determining the order of task-execution, and (3) *timing* – determining the precise time of task-execution. (Some people also define only ordering and timing as scheduling, and assignment as *binding* or *mapping*.) Each of these three tasks can be done at either compile-time or run-time. In this chapter, we classify the schedulers on this criteria and highlight two of them most suited for use in multiprocessor multimedia platforms. We highlight the issue of *composability,* i.e. mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, as far as possible. This limits computational complexity and allows high dynamism in the system.

This chapter is organized as follows. The first section motivates the need of modeling applications and the requirements for such a model. Section 2 gives an intro-

A. Kumar et al., *Multimedia Multiprocessor Systems*, Embedded Systems,
DOI 10.1007/978-94-007-0083-3_2,

duction to the synchronous dataflow (SDF) graphs that we use in our analysis. Some properties that are relevant for this book are also explained in the same section. Section 3 discusses several other models of computation (MoCs) that are available, and motivates the choice of SDF graphs as the MoC for our applications. Section 4 discusses state-of-the-art techniques used for estimating performance of applications modeled as SDF graphs. Section 5 provides background on the scheduling techniques used for dataflow graphs in general. Section 6 extends the performance analysis techniques to include hardware constraints as well. Section 7 discusses the issue of Composability in depth. Section 8 provides a comparison between static and dynamic ordering schedulers, and Sect. 9 concludes this chapter.

1 Application Model and Specification

Multimedia applications are often also referred to as **streaming applications** owing to their repetitive nature of execution. Most applications execute for a very long time in a fixed execution pattern. When watching television for example, the video decoding process potentially goes on decoding for hours – an hour is equivalent to 180,000 video frames at a modest rate of 50 frames per second (fps). High-end televisions often provide a refresh rate of even 100 fps, and the trend indicates further increase of this rate. The same goes for an audio stream that usually accompanies the video. The platform has to process continuously to get this output to the user.

In order to ensure that this high performance can be met by the platform, the designer has to be able to model the application requirements. In the absence of a good model, it is very difficult to know in advance whether the application performance can be met at all times, and extensive simulation and testing is needed. Even now, companies report a large effort being spent on verifying the timing requirements of the applications. With multiple applications executing on multiple processors, the potential number of use-cases increases rapidly, and so does the cost of verification.

We start by defining a use-case.

Definition 1 (Use-case) Given a set of n applications $A_0, A_1, \ldots, A_{n-1}$, a use-case U is defined as a vector of n elements $(x_0, x_1, \ldots x_{n-1})$ where $x_i \in \{0, 1\}\ \forall i = 0, 1, \ldots n-1$, such that $x_i = 1$ implies application A_i is active, and $x_i = 0$ implies application A_i is inactive.

In other words, a use-case represents a collection of multiple applications that are active simultaneously. It is often impossible to test a system with all potential input cases in advance. Modern multimedia platforms (high-end mobile phones, for example) allow users to download applications at run-time. Testing for those applications at design-time is simply not possible. A good model of an application can allow for such analysis at run-time.

One of the major challenges that arise when mapping an application to an MPSoC platform is dividing the application load over multiple processors. Two ways are available to parallelize the application and divide the load over more than one

processor, namely task-level parallelism and data-level parallelism. In the former, each processor gets a different part of an application to process, while in the latter, processors operate on the same functionality of application, but different data. For example, in case of JPEG image decoding, inverse discrete cosine transform (IDCT) and colour conversion (CC), among other tasks, need to be performed for all parts (macro-blocks) of an image. Splitting the task of IDCT and CC on different processors is an example of task-level parallelism. Splitting the data, in this case macro-blocks, to different processors is an example of data-level parallelism. To an extent, these approaches are orthogonal and can be applied in isolation or in combination. In this book, we shall focus primarily on task-level parallelism since it is more commonly done for heterogeneous platforms. Data-level parallelism is more suited to homogeneous systems.

Parallelizing an application to make it suitable for execution on a multi-processor platform can be a very difficult task. Whether an application is written from start in a manner that is suitable for a model of computation, or whether the model is extracted from the existing (sequential) application, in either case we need to know how long the execution of each program segment will take, how much data and program memory will be needed for it, and when communication program segments are mapped on different processors, how much communication buffer capacity do we need. Further, we also want to know what is the maximum performance that the application can achieve on a given platform, especially when sharing the platform with other applications. For this, we have to also be able to model and analyze scheduling decisions.

To summarize, following are our requirements from an application model that allow mapping and analysis on a multiprocessor platform:

- *Analyze computational requirements:* When designing an application for MPSoC platform, it is important to know how much computational resource an application needs. This allows the designers to dimension the hardware appropriately. Further, this is also needed to compute the performance estimates of the application as a whole. While sometimes, average case analysis of requirements may suffice, often we also need the worst case estimates, for example in case of real-time embedded systems.
- *Analyze memory requirements:* This constraint becomes increasingly more important as the memory cost on a chip goes high. A model that allows accurate analysis of memory needed for the program execution can allow a designer to distribute the memory across processors appropriately and also determine proper mapping on the hardware.
- *Analyze communication requirements:* The buffer capacity between the communicating tasks (potentially) affects the overall application performance. A model that allows computing these buffer-throughput trade-offs can let the designer allocate appropriate memory for the channel and predict throughput.
- *Model and analyze scheduling:* When we have multiple applications sharing processors, scheduling becomes one of the major challenges. A model that allows us to analyze the effect of scheduling on performance of application(s) is needed.

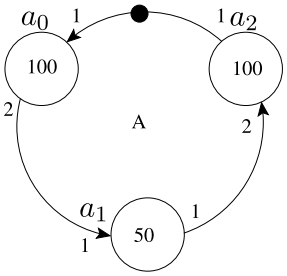


Fig. 2.1 Example of an SDF Graph

- *Design the system:* Once the performance of system is considered satisfactory, the system has to be synthesized such that the properties analyzed are still valid.

Dataflow models of computation fit rather well with the above requirements. They provide a model for describing signal processing systems where infinite streams of data are incrementally transformed by processes executing in sequence or parallel. In a dataflow model, processes communicate via unbounded FIFO channels. Processes read and write atomic data elements or tokens from and to channels. Writing to a channel is non-blocking, i.e. it always succeeds and does not stall the process, while reading from a channel is blocking, i.e. a process that reads from an empty channel will stall and can only continue when the channel contains sufficient tokens. In this book, we use synchronous dataflow (SDF) graphs to model applications and the next section explains them in more detail.

2 Introduction to SDF Graphs

Synchronous Data Flow Graphs (SDFGs, see (Lee and Messerschmitt 1987)) are often used for modeling modern DSP applications (Sriram and Bhattacharyya 2000) and for designing concurrent multimedia applications implemented on multiprocessor systems-on-chip. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. Tasks are modeled by the vertices of an SDFG, which are called *actors*. The communication between actors is represented by *edges*. Edges represent *channels* for communication in a real system.

The time that the actor takes to execute on a processor is indicated by the number inside the actor. It should be noted that the time an actor takes to execute may vary with the processor. For sake of simplicity, we shall omit the detail as to which processor it is mapped, and just define the time (or clock cycles) needed on a RISC processor (Patterson and Ditzel 1980), unless otherwise mentioned. This is also sometimes referred to as *Timed SDF* in literature (Stuijk 2007). Further, when we refer to the time needed to execute a particular actor, we refer to the worst-case execution-time (WCET). The average execution time may be lower.

Figure 2.1 shows an example SDF graph. There are three actors in this graph. As in a typical data flow graph, a directed edge represents the dependency between actors. Actors need some input data (or control information) before they can start,

and usually also produce some output data; such information is referred to as *tokens*. The number of tokens produced or consumed in one execution of an actor is called *rate*. In the example, a_0 has an input rate of 1 and output rate of 2. Further, its execution time is 100 clock cycles. Actor execution is also called *firing*. An actor is called *ready* when it has sufficient input tokens on all its input edges and sufficient buffer space on all its output channels; an actor can only fire when it is ready.

The edges may also contain *initial tokens*, indicated by bullets on the edges, as seen on the edge from actor a_2 to a_0 in Fig. 2.1. In the above example, only a_0 can start execution from the initial state, since the required number of tokens are present on its single incoming edge. Once a_0 has finished execution, it will produce 2 tokens on the edge to a_1. a_1 can then proceed, as it has enough tokens, and upon completion produce 1 token on the edge to a_2. However, a_2 has to wait before two executions of a_1 are completed, since it needs two input tokens.

Formally, an SDF graph is defined as follows. We assume a set *Ports* of ports, and with each port $p \in Ports$ we associate a finite rate $Rate(p) \in \mathbb{N} \setminus \{0\}$.

Definition 2 (Actor) An actor a is a tuple (In, Out, τ) consisting of a set $In \subseteq Ports$ of input ports (denoted by $In(a)$), a set $Out \subseteq Ports$ of output ports with $In \cap Out = \phi$ and $\tau \in \mathbb{N} \setminus \{0\}$ representing the execution time of a $(\tau(a))$.

Definition 3 (SDF Graph) An SDF graph is a tuple (A, C) consisting of a finite set A of actors and a finite set $C \subseteq Ports^2$ of channels. The channel source is an output port of some actor, the destination is an input port of some actor. All ports of all actors are connected to precisely one channel, and all channels are connected to ports of some actor. For every actor $a = (I, O, \tau) \in A$, we denote the set of all channels that are connected to the ports in I (O) by $InC(a)$ $(OutC(a))$.

When an actor a starts its firing, it removes $Rate(q)$ tokens from all $(p, q) \in InC(a)$. The execution continues for $\tau(a)$ time units and when it ends, it produces $Rate(p)$ tokens on every $(p, q) \in OutC(a)$.

A number of properties of an application can be analyzed from its SDF model. We can calculate the maximum performance possible of an application. We can identify whether the application or a particular schedule will result in a deadlock. We can also analyze other performance properties, e.g. latency of an application, buffer requirements. Below we give some properties of SDF graphs that allow modeling of hardware constraints that are relevant to this book.

Modeling Auto-concurrency

The example in Fig. 2.1 brings a very interesting fact to notice. According to the model, since a_1 requires only one token on the edge from a_0 to fire, as soon as a_0 has finished executing and produced two tokens, two executions of a_1 can start si-

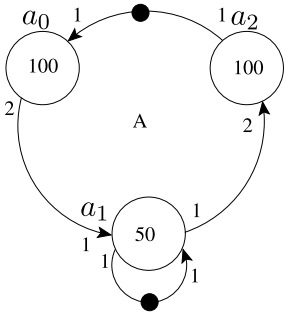


Fig. 2.2 SDF Graph after modeling auto-concurrency of 1 for the actor a_1

multaneously. However, this is only possible if a_1 is mapped and allowed to execute on multiple processors or hardware components simultaneously. In a typical system, a_1 will be mapped on a processor. Once the processor starts executing, it will not be available to start the second execution of a_1 until it has at least finished the first execution of a_1. If there are other actors mapped on it, the second execution of a_1 may even be delayed further.

Fortunately, there is a way to model this particular resource conflict in SDF. Figure 2.2 shows the same example, now updated with the constraint that only one execution of a_1 can be active at any point in time. In this figure, a *self-edge* has been added to actor a_1 with one initial token. For a self-edge, the source and destination actor is the same. This initial token is consumed in the first firing of a_1 and produced after a_1 has finished the first execution. Interestingly enough, by varying the number of initial tokens on this self-edge, we can regulate the number of simultaneous executions of a particular actor. This property is called **auto-concurrency**.

Definition 4 (Auto-concurrency) The **auto-concurrency** of an actor is defined as the maximum number of simultaneous executions of that actor.

In Fig. 2.2, the auto-concurrency of a_1 is 1, while for a_0 and a_2 it is infinite. In other words, the resource conflict for actors a_0 and a_2 is not modeled. In fact, the single initial token on the edge from a_2 to a_0 limits the auto-concurrency of these two actors to one; a self-edge in this case would be superfluous.

Modeling Buffer Sizes

One of the very useful properties of SDF graphs is its ability to model available buffers easily. Buffer-sizes may be modeled as a back-edge with initial tokens. In such cases, the number of tokens on that edge indicates the buffer-size available. When an actor writes data on a channel, the available size reduces; when the receiving actor consumes this data, the available buffer increases, modeled by an increase in the number of tokens.

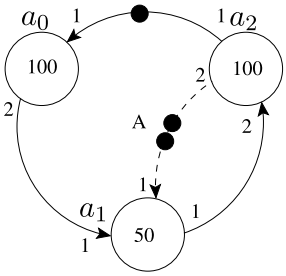


Fig. 2.3 SDF Graph after modeling buffer-size of 2 on the edge from actor a_2 to a_1

Figure 2.3 shows such an example, where the buffer size of the channel from a_1 to a_2 is shown as two. Before a_1 can be executed, it has to check if enough buffer space is available. This is modeled by requiring tokens from the back-edge to be consumed. Since it produces one token per firing, one token from the back-edge is consumed, indicating reservation of one buffer space on the output edge. On the consumption side, when a_2 is executed, it frees two buffer spaces, indicated by a release of two tokens on the back-edge. In the model, the output buffer space is claimed at the start of execution, and the input token space is released only at the end of firing. This ensures atomic execution of the actor.

3 Comparison of Dataflow Models

While SDF graphs allow analysis of many properties and are well-suited for multimedia applications, they do have some restrictions. For example, conditional and data-dependent behaviour cannot be expressed in these models. In this section, we provide an overview of the other models of computation (MoC). In (Stuijk 2007), Stuijk has summarized and compared many models on the basis of their expressiveness and succinctness, efficiency of implementation, and analyzability. Expressiveness determines to what extent real-applications can be represented in a particular model. Models that are static in nature (e.g. SDF) cannot accurately capture behaviour of highly dynamic applications (e.g. object segmentation from an input sequence) accurately. Succinctness (or compactness) determines how compact that representation is. Efficiency of implementation determines how easily the application model can be implemented in terms of its schedule length. Analyzability determines to what extent the model can be analyzed and performance properties of applications determined. As mentioned in the earlier section, this is one of the most important considerations for us. In general, a model that can be easily analyzed at design-time is also more efficient for implementation, since most scheduling and resource assignment decisions can be made at design-time. This implies a lower run-time implementation overhead. Figure 2.4 shows how different models are placed on these three axes.

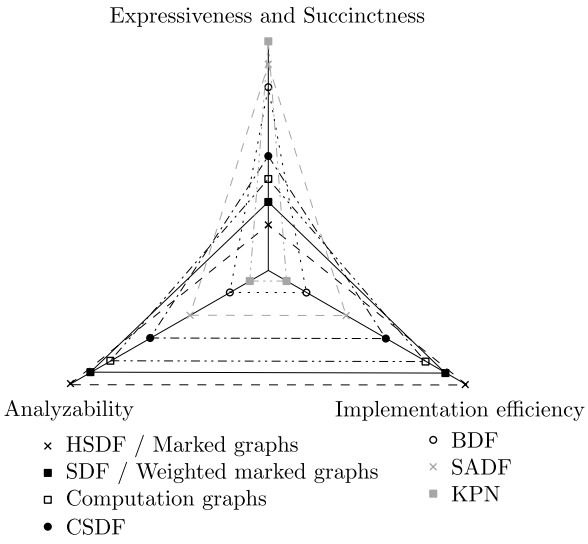


Fig. 2.4 Comparison of different models of computation (Stuijk 2007)

Kahn Process Network

Kahn process network (KPN) was proposed by Kahn in 1974 (Kahn 1974). The amount of data read from an edge may be data-dependent. This allows modeling of any continuous function from the inputs of the KPN to the outputs of the KPN with an arbitrarily small number of processes. KPN is sufficiently expressive to capture precisely all data dependent dataflow transformations. However, this also implies that in order to analyze properties like the throughput or buffer requirements of a KPN all possible input values have to be considered.

Similar to SDF graph, in a KPN, processes communicate via unbounded FIFO channels. While reading from a channel is blocking, i.e. if there is not sufficient data on any of its inputs, the process stalls, writing to an output channel is always non-blocking due to unbounded capacity on the output channels. Processes are not allowed to test an input channel for existence of tokens without consuming them.

Scenario Aware Dataflow

Scenario aware dataflow (SADF) was first introduced by Theelen in 2006 (Theelen et al. 2006). This is also a model for design-time analysis of dynamic streaming and signal processing applications. This model allows for data-dependent behaviour in processes. Each different execution pattern is defined as a *scenario*. Such scenarios denote different modes of operations in which resource requirements can differ considerably. The scenario concept enables to coherently capture the variations in behaviour of different processes in a streaming application. A key novelty of SADF is the use of a stochastic approach to capture the scenario occurrences as well as the

occurrence of different execution times within a scenario in an abstract way. While some properties of these graphs like deadlock and throughput are possible to analyze at design time, in practice this analysis can be quite slow. This model is less compact than KPN, since all scenarios have to be explicitly specified in the model, and known at design time. This also makes it less expressive since not all kinds of systems can be expressed accurately in SADF.

Boolean Dataflow

The last model of computation that we discuss having data-dependent behaviour is boolean dataflow (BDF) model (Lee 1991; Buck and Lee 1993). In this model, each process has a number of inputs and outputs to choose from. Depending on the value of *control tokens* data is read from one of the input channels, and written to one of the output channels. This model is less expressive than the earlier two models discussed, since the control freedom in modeling processes is limited to either true or false. Similar to the earlier two models discussed, the analyzability is limited.

Cyclo Static Dataflow

Now we move on to the class of more deterministic data flow models of computation. In a cyclo-static dataflow (CSDF) model (Lauwereins et al. 1994; Bilsen et al. 1996), the rates of data consumed and produced may change between subsequent firings. However, the pattern of this change is pre-determined and cyclic, making it more analyzable at design time. These graphs may be converted to SDF graphs, and are therefore as expressive as SDF graphs. However, the freedom to change the rates of data makes them more compact than SDF in representing some applications. They are also as analyzable, but slower if we consider the same number of actors, since the resulting schedule is generally a little more complex.

Recently, special channels have been introduced for CSDF graphs (Denolf et al. 2007). Often applications share buffers between multiple consumers. This cannot be directly described in CSDF. The authors show how such implementation specific aspects can still be modeled in CSDF without the need of extensions. Thus, the analyzability of the graph is maintained, and appropriate buffer-sizes can be computed from the application model.

Computation Graphs

Computation graphs were first introduced by Karp and Miller in 1966 (Karp and Miller 1966). In these graphs there is a threshold set for each edge specifying the

minimum number of tokens that should be present on that edge before an actor can fire. However, the number of tokens produced and consumed for each edge is still fixed. These models are less expressive than CSDF, but more analyzable. A synchronous data flow graph is a subset of these computation graphs.

Synchronous Dataflow

Synchronous dataflow (SDF) graphs were first proposed by Lee and Messerschmitt in 1987 (Lee and Messerschmitt 1987). However, as has been earlier claimed (Stuijk 2007), these correspond to subclass weighted marked graph (Teruel et al. 1992) of Petri nets, which is a general purpose model of computation with a number of applications (Petri 1962; Murata 1989). SDF graphs have a constant input and output rate that does not change with input or across different firings. They also don't support execution in different scenarios as may be specified by data. Therefore, their expressivity is rather limited. However, this also makes them a lot easier to analyze. Many performance parameters can be analyzed as explained in Sect. 4.

Homogeneous Synchronous Dataflow

Homogeneous Synchronous dataflow (HSDF) graphs are a subset of SDF graphs. In HSDF graph model, the rates of all input and output edges is one. This implies that only one token is read and written in any firing of an actor. This limits the expressiveness even more, but makes the analysis somewhat easier. HSDF graphs can be converted into SDF and vice-versa. However, in practice the size of an HSDF for an equivalent SDFG may be very large as shown by examples in (Stuijk 2007). Lately, analysis techniques have been developed that work almost as fast directly on an SDF graph as on an HSDF graph (for the same number of nodes) (Ghamarian 2008). Therefore, the added advantage of using an HSDF graph is lost.

After considering all the alternatives, we decided in favour of SDF graphs since their ability to analyze applications in terms of other performance requirements, such as throughput and buffer, was one of our key requirements. Further, a number of analysis tools for SDF graph were available (and more in development) when this research was started (SDF3 2009). Further, a CSDF model of an application can easily be represented in an SDF model. However, since SDF graphs are not able to express dynamism in some real applications accurately, we do have to pay a little overhead in estimating performance. For example, the execution time is assumed to be the worst-case execution-time. Thus, in some cases, the performance estimates may be pessimistic.

4 Performance Modeling

In this section, we define the major terminology that is relevant for this book.

Definition 5 (Actor Execution Time) Actor execution time, $\tau(a)$ is defined as the time needed to complete execution of actor a on a specified node. In cases where the required time is not constant but varying, this indicates the maximum time for actor execution.

$\tau(a_0) = 100$, for example, in Fig. 2.3.

Definition 6 (Iteration) An **iteration** of an SDF graph is defined as the minimum non-zero execution (i.e. at least one actor has executed) such that the state of the graph before that execution is obtained.

In Fig. 2.3, one iteration of graph A is completed when a_0, a_1 and a_2 have each completed one, two and one execution(s) respectively.

Definition 7 (Repetition Vector) Repetition Vector q of an SDF graph A is defined as the vector specifying the number of times each actor in A has to be executed to complete one iteration of A.

For example, in Fig. 2.3, $q[a0\ a1\ a2] = [1\ 2\ 1]$.

Definition 8 (Application Period) Application Period $Per(A)$ is defined as the time SDFG A takes to complete one iteration.

$Per(A) = 300$ in Fig. 2.3, assuming it has sufficient resources and no contention, and all the actors fire as soon as they are ready. (Note that actor a_1 has to execute twice.)

Definition 9 (Application Throughput) Application Throughput, Thr_A is defined as the number of iterations of an SDF graph A in one second.

This is simply the inverse of period, $Per(A)$, when period is defined in seconds. For example, an application with a throughput of 50 Hz takes 20 ms to complete one iteration. When the graph in Fig. 2.3 is executing on a single processor of 300 MHz, the throughput of A is 1 MHz since the period is 1 micro-second.

Throughput is one of the most interesting properties of SDF graphs relevant to the design of any multimedia system. Designers and consumers both want to know the sustained throughput the system can deliver. This parameter often directly relates to the consumer. For example, throughput of an H264 decoder may define how many frames can be decoded per second. A higher throughput in this case directly improves the consumer experience.

Steady-State vs Transient Behaviour

Often some actors of an applications may execute a few times before the periodic behaviour of the application starts. For example, consider the application graph as

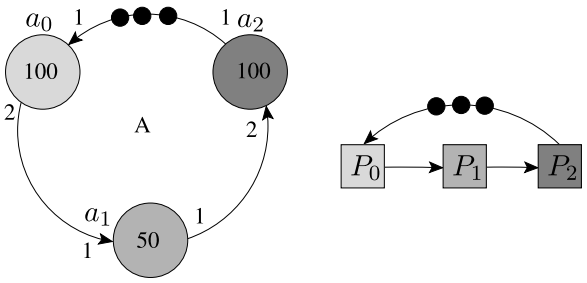


Fig. 2.5 SDF Graph and the multi-processor architecture on which it is mapped

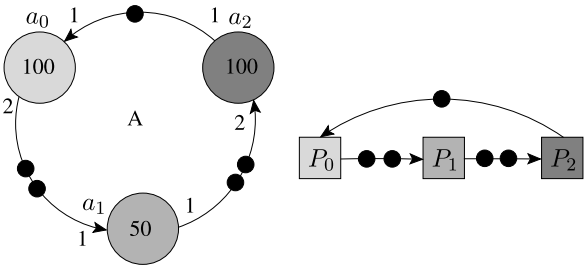


Fig. 2.6 Steady-state is achieved after two executions of a_0 and one of a_1

shown earlier in Fig. 2.1, but now with three initial tokens on the edge from a_2 to a_0. Consider further, that each of the three actors is mapped on a multi-processor system with three processors, P_0, P_1 and P_2, such that actor a_i is mapped on P_i for $i = 0, 1, 2$. Let us assume that the processors are connected to each other with a point-to-point connection with infinite bandwidth with directions similar to channels in the application graph. Figure 2.5 shows the updated graph and the three processor system with the appropriate mapping.

In this example, if we look at the time taken for one single iteration, we get a period of 300 cycles. However, since each actor has its own dedicated processor, soon we get the token distribution as shown in Fig. 2.6. From this point onwards, all the actors can continue firing indefinitely since all actors have sufficient tokens and dedicated resources. Thus, every 100 cycles, an iteration of application A is completed. This final state is called *steady-state*. The initial execution of the graph leading to this state is called *transient phase*.

For the graph as shown in Fig. 2.5, the maximal throughput for 300 MHz processors is 3 MHz or three million iterations per second. In this book when we refer to the throughput of a graph, we generally refer to the maximal achievable throughput of a graph, unless otherwise mentioned. This only refers to the steady-state throughput. When we use the term *achieved throughput* of a graph, we shall refer to the long-term average throughput achieved for a given application. This also includes the transient phase of an application. Please note that for *infinitely long* execution, the long-term average throughput is the same as the steady-state throughput.

Another way to define throughput is the rate of execution of an *output* actor divided by its repetition vector entry. If we consider actor a_2 as the output actor

of application A, we see that the throughput of the application is the same as the execution rate of a_2, since its repetition vector entry is 1.

Throughput Analysis of (H)SDF Graphs

A number of analysis techniques are available to compute the throughput of SDF graphs (Ghamarian 2008; Stuijk 2007; Sriram and Bhattacharyya 2000; Dasdan 2004; Bambha et al. 2002). Many of these techniques first convert an SDF graph into a homogeneous SDF (HSDF) graph. HSDF is a special class of SDF in which the number of tokens consumed and produced is always equal to one. Techniques are available to convert an SDF into HSDF and the other way around (Sriram and Bhattacharyya 2000). After conversion to HSDF, throughput is computed as the inverse of the *maximal cycle mean (MCM)* of the HSDF graph (Dasdan 2004; Karp and Miller 1966). MCM in turn is the maximum of all *cycle-means*. A cycle-mean is computed as the weighted average of total delay in a cycle in the HSDF graph divided by the number of tokens in it.

The conversion to HSDF from an SDF graph may result in an explosion in the number of nodes (Pino and Lee 1995). The number of nodes in the corresponding HSDF graph for an SDF graph is determined by its repetition vector. There are examples of real-applications (H263 in this case), where an SDF model requires only 4 nodes and an HSDF model of the same application has 4754 nodes (Stuijk 2007). This makes the above approach very infeasible for many multimedia applications. Lately, techniques have been presented that operate on SDF graphs directly (Ghamarian et al. 2006; Ghamarian 2008). These techniques essentially simulate the SDF execution and identify when a steady-state is reached by comparing previous states with the current state. Even though the theoretical bound is high, the experimental results show that these techniques can compute the throughput of many multimedia applications within milliseconds.

A tool called SDF^3 has been written and is available on-line for use by the research community (Stuijk et al. 2006a; SDF3 2009). Beside being able to generate random SDF graphs with specified properties, it can also compute throughput of SDF graphs easily. It allows to visualize these graphs as well, and compute other performance properties. The same tool was used for throughput computation and graph generation in many experiments conducted in this book.

However, the above techniques only work for fixed execution times of actors. If there is any change in the actor execution time, the entire analysis has to be repeated. A technique has been proposed in (Ghamarian et al. 2008) that allows variable execution time. This technique computes equations that limit the application period, for a given range of actor execution times. When the exact actor execution time is known, these equations can be evaluated to compute the actual period of the application. This idea is used in Chap. 3 to compute throughput of applications.

It should be mentioned that the techniques mentioned here do not take into account contention for resources like processor and memory, and essentially assume that infinite resources are available, except SDF^3. SDF^3 also takes resource

contention into account but is limited to preemptive systems. Before we see how throughput can be computed when considering limited computation resources, we review the basic techniques used for scheduling dataflow graphs.

5 Scheduling Techniques for Dataflow Graphs

One of the key aspects in designing multiprocessor systems from any MoC is scheduling. Scheduling, as defined earlier, is the process of determining when and where tasks are executed. Compile-time scheduling promises near-optimal performance at low cost for final system, but is only suitable for static applications. Run-time scheduling can address a wider variety of applications, at greater system cost. Scheduling techniques can be classified in a number of categories based on which decisions are made at compile time (also known as design-time) and which decisions are made at run-time (Lee and Ha 1989; Sriram and Bhattacharyya 2000). There are three main decisions when scheduling tasks (or actors) on a processor: (1) which tasks to **assign** to a given processor, (2) what is the **order** of these tasks on it and (3) what is the precise**timing** of these tasks. (When considering mapping of channels on the network, there are many more categories.) We consider four different types of schedulers.

(1) The first one is **fully static**, where everything is decided at compile time and the processor has to simply execute the tasks. This approach has traditionally been used extensively for DSP applications due to their repetitive and constant resource requirement. This is also good for systems where guarantees are more important and (potential) speed-up from earlier execution is not desired. Further, the run-time scheduler becomes very simple since it does not need to check for availability of data and simply executes the scheduled actors at respective times. However, this mechanism is completely static, and cannot handle any dynamism in the system like run-time addition of applications, or any unexpectedly higher execution time for a particular iteration.

(2) The second type is **self-timed**, where the assignment and ordering is already done at compile time. However, the exact time for firing of actors is determined at run-time, depending on the availability of data. Self-timed scheduling is more suitable for cases when the execution time of tasks may be data-dependent, but the variation is not very large. This can often result in speed-up of applications as compared to analysis at design-time, provided the worst case execution time estimates are used for analyzing the application performance. Since earlier arrival of data cannot result in later production of data – a property defined as *monotonicity*, the performance bounds computed at compile-time are preserved. However, this also implies that the schedule may become non-work-conserving, i.e. that a task may be waiting on a processor, while the processor is sitting idle waiting for the task in order.

(3) The third type is **static assignment**, where the mapping is already fixed at compile time, but the ordering and timing is done at run-time by the scheduler. This

Table 2.1 The time which the scheduling activities "assignment", "ordering", and "timing" are performed is shown for four classes of schedulers. The scheduling activities are listed on top and the strategies on the left (Lee and Ha 1989)

Scheduler	Assignment	Ordering	Timing	Work-conserving
Fully static	compile	compile	compile	no
Self timed	compile	compile	run	no
Static assignment	compile	run	run	processor-level
Fully dynamic	run	run	run	yes

allows the schedule to become work-conserving and perhaps achieve a higher overall throughput in the system. However, it might also result in a lower overall throughput since the bounds cannot be computed at compile-time (or are not preserved in this scheduler). This scheduling is most applicable for systems where applications have a large variation in execution time. While for a single application, the order is still imposed by the data-dependency among tasks and makes self-timed more suitable, for multiple applications the high variation in execution time, makes it infeasible to enforce the static-order.

(4) The last one is called **fully dynamic**, where mapping, ordering and timing are all done at run-time. This gives full freedom to the scheduler, and the tasks are assigned to an idle processor as soon as they are ready. This scheduler also allows for task migration. This may result in a yet higher throughput, since this tries to maximize the resource utilization and minimize the idle time, albeit at the cost of performance guarantee. It should be noted that run-time assignment also involves a (potentially) higher overhead in data movement. When the assignment is fixed at compile time, the task knows the processor to which the receiving actor is mapped *a priori*.

These four scheduling mechanisms are summarized in Table 2.1. As we move from fully-static to fully-dynamic scheduler, the run-time scheduling activity (and correspondingly overhead) increases. However, this also makes it more robust for handling dynamism in the system. The last column shows the work-conserving nature of the schedulers. A fully-static scheduler is non-conserving since the exact time and order of firing (task execution) is fixed. The self-timed schedule is work-conserving only when at most one task is mapped on one processor, while the static-assignment is also work-conserving for multiple applications. However, in static assignment if we consider the whole system (i.e. multiple processors), it may not be work-conserving, since tasks may be waiting to execute on a particular processor, while other processors may be idle.

In a homogeneous system, there is naturally more freedom to choose which task to assign to a particular processor instance, since all processors are identical. On the contrary, in a heterogeneous system this freedom is limited by which processors can be used for executing a particular task. When only one processor is available for a particular task, the mapping is inherently dictated by this limitation. For a complete heterogeneous platform, a scheduler is generally not fully dynamic, unless a task is allowed to be mapped on different types of processors. However, even

in those cases, assignment is usually fixed (or chosen) at compile time. Further, the execution time for multimedia applications is generally highly variant making a fully-static scheduler often infeasible. Designers therefore have to make a choice between a self-timed or a static-assignment schedule. The only choice left is essentially regarding the ordering of tasks on a processor.

In the next section, we shall see how to analyze performance of multiple applications executing on a multiprocessor platform for both self-timed and static-assignment scheduler. Since the only difference in these two schedulers is the time at which ordering of actors is done, we shall refer to self-timed and static-assignment scheduler as **static-ordering** and **dynamic-ordering** scheduler respectively for easy differentiation.

6 Analyzing Application Performance on Hardware

In Sect. 4, we assumed we have infinite computing and communication resources. Clearly, this is not a valid assumption. Often processors are shared, not just among tasks of one application, but also with other applications. We first see how we can model this resource contention for a single application, and later for multiple applications.

We start with considering an HSDF graph with constant execution times to illustrate that even for HSDF graphs it is already complicated. In (Bambha et al. 2002), the authors propose to analyze performance of a *single application* modeled as an HSDF graph mapped on a multi-processor system by modeling dependencies of resources by adding extra edges to the graph. Adding these extra edges enforces a strict order among the actors mapped on the same processor. Since the processor dependency is now modeled in the graph itself, we can simply compute the maximum throughput possible of the graph, and that corresponds to the maximum performance the application can achieve on the multiprocessor platform.

Unfortunately, this approach does not scale well when we move on to the SDF model of an application. Converting an SDF model to an HSDF model can potentially result in a large number of actors in the corresponding HSDF graph. Further, adding such resource dependency edges essentially enforces a static-order among actors mapped on a particular processor. While in some cases, only one order is possible (due to natural data dependency among those actors), in other cases the number of different orders is also very high. Further, different orders may result in different overall throughput of the application. The situation becomes even worse when we consider multiple applications. This is shown by means of an example in the following sub-section.

Static Order Analysis

In this section, we look at how application performance can be computed using static-order scheduler, where both processor assignment and ordering is done at

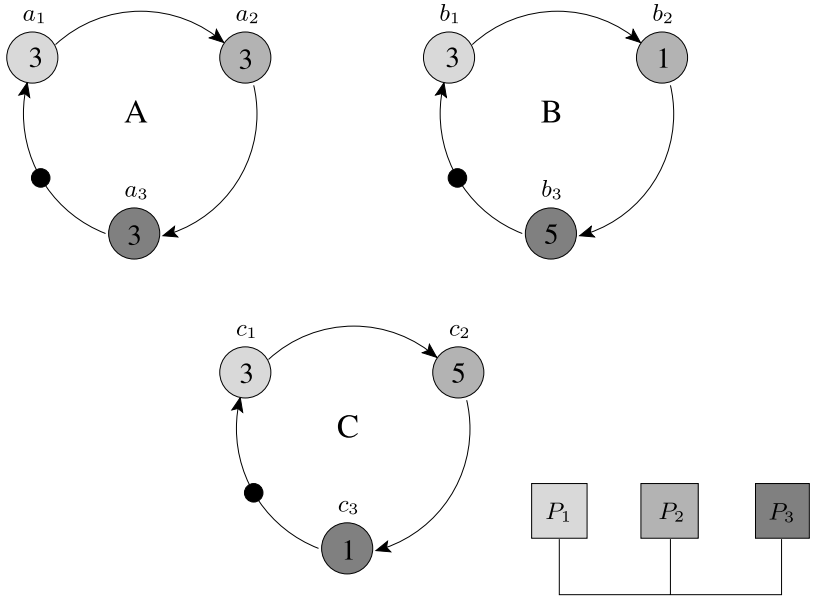


Fig. 2.7 Example of a system with 3 different applications mapped on a 3-processor platform

compile-time. We show that when multiple applications are mapped on multiple processors sharing them, it is (1) difficult to make a static schedule, (2) time-consuming to analyze application performance given a schedule, and (3) infeasible to explore the entire scheduling space to find one that gives the best performance for all applications.

Three application graphs – A, B and C, are shown in Fig. 2.7. Each is an HSDF with three actors. Let us assume actors T_i are mapped on processing node P_i where T_i refers to a_i, b_i and c_i for $i = 1, 2, 3$. This contention for resources is shown by the dotted arrows in Fig. 2.8. Clearly, by putting these dotted arrows, we have fixed the actor-order for each processor node. Figure 2.8 shows just one such possibility when the dotted arrows are used to combine the three task graphs. Extra tokens have been inserted in these dotted edges to indicate the initial state of arbiters on each processor. The tokens indicating processor contention are shown in grey, while the regular data tokens are shown in black. Clearly, this is only possible if each task is required to be run an equal number of times. If the rates of each task are not the same, we need to introduce multiple copies of actors to achieve the required ratio, thereby increasing analysis complexity.

When throughput analysis is done for this complete graph, we obtain a mean cycle count of 11. The bold arrows represent the edges that limit the throughput. The corresponding schedule is also shown. One actor of each application is ready to fire at instant t_0. However, only a_1 can execute since it is the only one with a token on all its incoming edges. We find that the graph soon settles into the periodic schedule of 11 clock cycles. This period is denoted in the schedule diagram of Fig. 2.8 between the time instant t_1 and t_2.

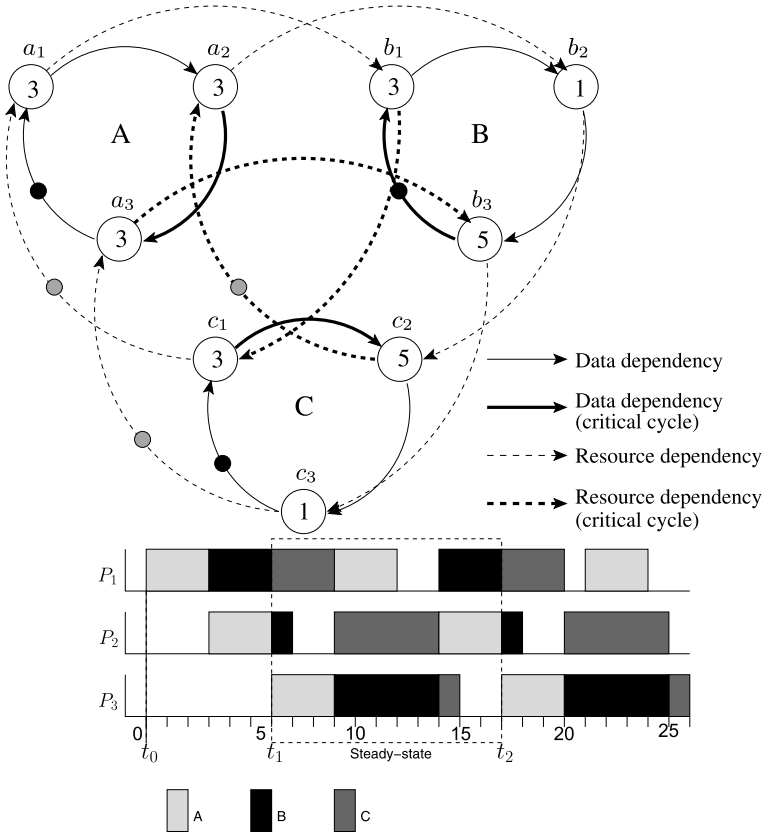


Fig. 2.8 Graph with clockwise schedule (static) gives MCM of 11 cycles. The critical cycle is shown in bold

Figure 2.9 shows just another of the many possibilities for ordering the actors of the complete HSDF (we reversed the ordering of resources between the actors). Interestingly, the mean cycle count for this graph is 10, as indicated by the bold arrows. In this case, the schedule starts repeating after time t_1, and the steady state length is 20 clock cycles, as indicated by the difference in time instants t_1 and t_2. However, since two iterations for each application are completed, the average period is only 10 clock cycles.

From arbitration point of view, if application graphs are analyzed in isolation, there seems to be no reason to prefer actor b_1 or c_1 after a_1 has finished executing on P_1. There is at least a delay of 6 clock cycles before a_1 needs P_1 again. Also, since b_1 and c_1 take only 3 clock cycles each, 6 clock cycles are enough to finish their execution. Further both are ready to be fired, and will not cause any delay. Thus, the local information about an application and the actors that need a processor resource does not easily dictate preference of one task over another. However, as we see in this example, executing c_1 is indeed better for the overall performance.

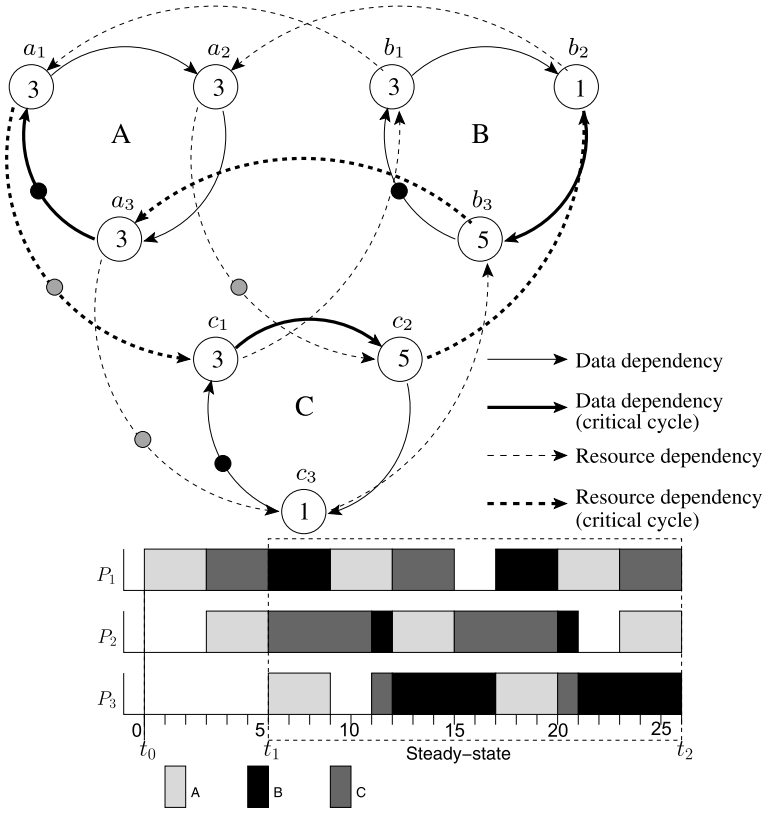


Fig. 2.9 Graph with anti-clockwise schedule (static) gives MCM of 10 cycles. The critical cycle is shown in bold. Here two iterations are carried out in one steady-state iteration

Computing a static order relies on the global information and produces the optimal performance. This becomes a serious problem when considering MPSoC platforms, since constructing the overall HSDF graph and then computing its throughput is very compute intensive. Further, this is not suitable for dynamic applications. A small change in execution time may change the optimal schedule.

The number of possibilities for constructing the HSDF from individual graphs is very large. In fact, if one tries to combine g graphs of say a actors, scheduled in total on a processors, there are $((g-1)!)^a$ unique combinations, each with a different actor ordering. (Each processor has g actors to schedule, and therefore $(g-1)!$ unique orderings on a single processor. This leads to $((g-1)!)^a$ unique combinations, since ordering on each processor is independent of ordering on another.) To get an idea of vastness of this number, if there are 5 graphs with 10 actors each we get 24^{10} or close to $6.34 \cdot 10^{13}$ possible combinations. If each computation would take only 1 ms to compute, 2009 years are needed to evaluate all possibilities. This is only considering the cases with equal rates for each application, and only for HSDF graphs. A typical SDF graph with different execution rates would only make the

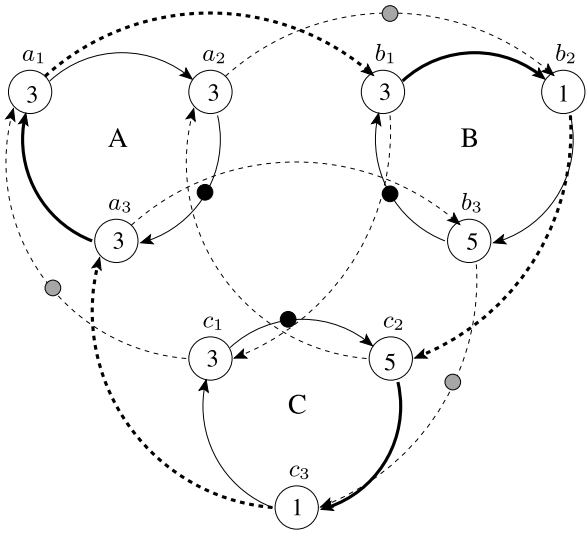


Fig. 2.10 Deadlock situation when a new job, C arrives in the system. A cycle a_1, b_1, b_2, c_2, c_3, a_3, a_1 is created without any token in it

problem even more infeasible, since the transformation to HSDF may yield many actor copies. An exhaustive search through all the graphs to compute optimal static order is simply not feasible.

We can therefore conclude that computing a static order for multiple applications is very compute intensive and infeasible. Further, the performance we obtain may not be optimal. However, the advantage of this approach is that we are guaranteed to achieve the performance that is analyzed for *any* static order at design-time *provided the worst-case execution time estimates are correct*.

Deadlock Analysis

Deadlock avoidance and detection is an important concern when applications may be activated dynamically. Applications modeled as (H)SDF graphs can be analyzed for deadlock occurrence within each (single) application. However, deadlock detection and avoidance between multiple applications is not so easy. When static order is being used, every new use-case requires a new schedule to be loaded into the platform at run-time. A naive reconfiguration strategy can easily send the system into deadlock. This is demonstrated with an example in Fig. 2.10.

Say actors a_2 and b_3 are running in the system on P_2 and P_3 respectively. Further assume that a static order for each processor currently is $A \rightarrow B$ when only these two applications are active, and with a third application C, $A \rightarrow B \rightarrow C$ for each node. When application C is activated, it gets P_1 since it is idle. Let us see what happens to P_2: a_2 is executing on it and it is then assigned to b_2. P_3 is assigned to c_3 after b_3 is done. Thus, after each actor is finished executing on its currently assigned processor, we get a_3 waiting for P_3 that is assigned to task c_3, b_1 waiting for P_1 which is assigned to a_1, and c_2 waiting for P_2, which is assigned to b_2.

Looking at Fig. 2.10, it is easy to understand why the system goes into a deadlock. The figure shows the state when each actor is waiting for a resource and not able to execute. The tokens in the individual sub-graph show which actor is ready to fire, and the token on the dotted edge represents which resource is available to the application. In order for an actor to fire, a token should be present on all its incoming edges – in this case both on the incoming dotted edge and the solid edge. It can be further noted that a cycle is formed without any token in it. This is clearly a situation of deadlock (Karp and Miller 1966) since the actors on this cycle will never be enabled. This cycle is drawn in Fig. 2.10 with bold edges. It is possible to take special measures to check and prevent the system from going into such a deadlock. This, however, implies extra overhead at both compile-time and run-time. The application may also have to be delayed before it can be admitted into the system.

Dynamic Order Analysis

In this section, we look at static-assignment scheduling, where only processor assignment is done at compile-time and the ordering is done at run-time. First-come-first-serve (FCFS) falls under this category. Another arbiter that we propose here in this category is round-robin-with-skipping (RRWS). In RRWS, a recommended order is specified, but the actors can be skipped over if they are not ready when the processor becomes idle. This is similar to the fairness arbiter proposed by Gao in 1983 (Gao 1983). However, in that scheduler, all actors have equal weight. In RRWS, multiple instances of an actor can be scheduled in one cycle to provide an easy rate control mechanism.

The price a system-designer has to pay when using dynamic scheduling is the difficulty in determining application performance. Analyzing application performance when multiple applications are sharing a multiprocessor platform is not easy. An approach that models resource contention by computing **worst case response time** for TDMA scheduling (requires preemption) has been analyzed in (Bekooij et al. 2005). This analysis also requires limited information from the other SDFGs, but gives a very conservative bound that may be too pessimistic. As the number of applications increases, the minimum performance bound decreases much more than the average case performance. Further, this approach assumes a preemptive system. A similar worst-case analysis approach for round-robin is presented in (Hoes 2004), which also works for non-preemptive systems, but suffers from the same problem of lack of scalability.

Let us revisit the example in Fig. 2.7. Since 3 actors are mapped on each processor, an actor may need to wait when it is ready to be executed at a processor. The maximum waiting time for a particular actor can be computed by considering the *critical instant* as defined by Liu and Layland (Liu and Layland 1973). The **critical instant** for an actor is defined as an instant at which a request for that actor has the largest response time. The **response time** is defined as the sum of an actor's waiting time and its execution time. If we take worst case response time, this can

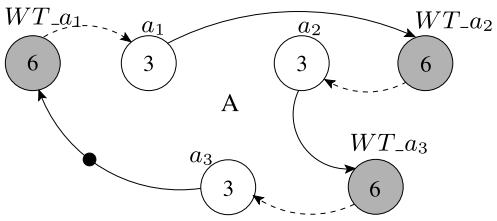


Fig. 2.11 Modeling worst case waiting time for application *A* in Fig. 2.7

Table 2.2 Table showing the deadlock condition in Fig. 2.10

Node	Assigned to	Task waiting and reassigned in RRWS
P1	A	B
P2	B	C
P3	C	A

be translated as the instant at which we have the largest waiting time. For dynamic scheduling mechanisms, it occurs when an actor becomes ready just after all the other actors, and therefore has to wait for all the other actors. Thus, the total waiting time is equal to the sum of worst-case execution times of all the other actors on that particular node and given by the following equation.

$$t_{wait}(T_{ij}) = \sum_{k=1,\, k\neq i}^{m} t_{exec}(T_{kj}) \tag{2.1}$$

Here $t_{exec}(T_{ij})$ denotes the worst case execution time of actor T_{ij}, i.e. actor of task T_i mapped on processor j. This leads to a waiting time of 6 time units as shown in Fig. 2.11. An extra node has been added for each 'real' node to depict the waiting time (WT_a_i). This suggests that each application will take 27 time units in the worst case to finish execution. This is the maximum period that can be obtained for applications in the system, and is therefore guaranteed. However, as we have seen in the earlier analysis, the applications will probably settle for a period of 10 or 11 cycles depending on the arbitration decisions made by the scheduler. Thus, the bound provided by this analysis is about two to three times higher than real performance.

The deadlock situation shown in Fig. 2.10 can be avoided quite easily by using dynamic-order scheduling. Clearly, for FCFS, it is not an issue since resources are never blocked for non-ready actors. For RRWS, when the system enters into a deadlock, the arbiter would simply skip to the actor that is ready to execute. Thus, processors 1, 2 and 3 are reassigned to B, C and A as shown in Table 2.2. Further, an application can be activated at any point in time without worrying about deadlock. In dynamic scheduling, there can never be a deadlock due to dependency on processing resources for atomic non-preemptive systems.

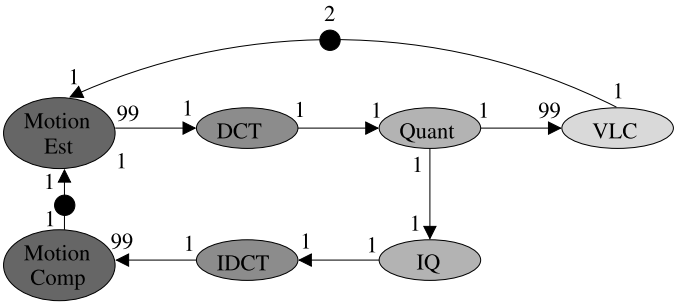
7 Composability

As highlighted in Chap. 1, one of the key challenges when designing multimedia systems is dealing with multiple applications. For example, a mobile phone supports various applications that can be active at the same time, such as listening to mp3 music, typing an sms and downloading some java application in the background. Evaluating resource requirements for each of these cases can be quite a challenge even at design time, let alone at run time. When designing a system, it is quite useful to be able to estimate resource requirements early in the design phase. Design managers often have to negotiate with the product divisions for the overall resources needed for the system. These estimates are mostly on a higher level, and the managers usually like to adopt a *spread-sheet* approach for computing it. As we see in this section, it is often not possible to use this view.

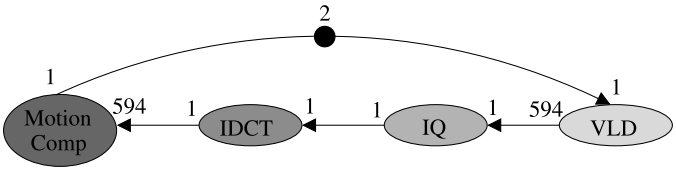
We define *composability* as mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation, i.e. requiring limited information from other applications. Note that this is different from what has been defined in literature by Kopetz (Kopetz and Obermaisser 2002; Kopetz and Suri 2003). Composability as defined by Kopetz is *integration of a whole system from well-specified and pre-tested sub-systems without unintended side-effects*. The key difference between this definition and our definition is that composability as defined by Kopetz is a property of a system such that the performance of applications in isolation and running concurrently with other applications is the same. For example, say we have a system with 10 applications, each with only one task and all mapped on the same processor. Let us further assume that all tasks take 100 time units to execute in isolation. According to the definition of Kopetz, it will also take 100 time units when running with the other tasks. This can only be achieved in two ways.

(1) The platform supports complete *virtualization* of resources, and each application gets one-tenth of processor resources. This implies that we only use one-tenth of the resources even when only one application is active. Further, to achieve complete virtualization, the processor has to be preempted and its context has to be switched every single cycle. (This could be relaxed a bit, depending on the observability of the system.)
(2) We consider a worst-case schedule in which all applications are scheduled, and the total execution time of all applications is 100 time units. Thus, if a particular application is not active, the processor simply waits for that many time units as it is scheduled for. This again leads to under-utilization of the processor resources. Besides, if any application takes more time, then the system may collapse.

Clearly, this implies that we cannot harness the full processing capability. In a typical system, we would want to use this compute power to deliver a better quality-of-service for an application when possible. We want to let the system execute as many applications as possible with the current resource availability, and let applications achieve their best behaviour possible in the given use-case. Thus, in the example with 10 applications, if each application can run in 10 time units in isolation, it might take 100 time units when running concurrently with all the other applications.



(a) SDF model of H263 encoder



(b) SDF model of H263 decoder

Fig. 2.12 SDF graphs of H263 encoder and decoder

We would like to predict the application properties given the application mix of the system, with as little information from other applications as possible.

Some of the things we would like to analyze are for example, deadlock occurrence, and application performance. Clearly, since there is more than one application mapped on a multi-processor system, there will be contention for the resources. Due to this contention, the throughput analyzed for an application in isolation is not always achievable when the application runs together with other applications. We see how different levels of information from other applications affect analysis results in the next sub-section.

Performance Estimation

Let us consider a scenario of video-conferencing in a hand-held device. Figure 2.12 shows SDF graphs for both H263 encoding and decoding applications. The encoder model is based on the SDF graph presented in (Oh and Ha 2004), and the decoder model is based on (Stuijk 2007). (The self-edges are removed for simplicity.) The video-stream assumed for the example is of QCIF resolution that has 99 macro-blocks to process, as indicated by the rates on the edges. Both encoding and decoding have an actor that works on variable length (VLC and VLD respectively), quantization (Quant and IQ respectively), and discrete cosine transform (DCT and IDCT respectively). Since we are considering a heterogeneous system, the processor responsible for an actor in encoding process is usually responsible for the corresponding decoding actor. E.g. when the encoding and decoding are done concurrently, the

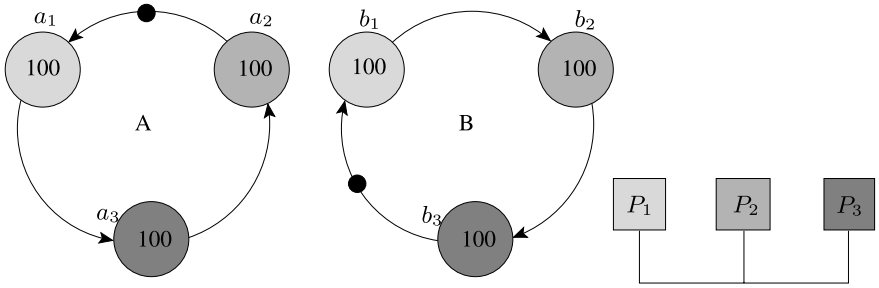


Fig. 2.13 Two applications running on same platform and sharing resources

DCT and IDCT are likely to be executed on the same processor, since that processor is probably more suited for cosine transforms. This resource dependency in the encoder and decoder models is shown by shading in Fig. 2.12. Since decoding works in opposite order as compared to encoding, the resource dependency in encoding and decoding is exactly reversed. A similar situation happens during decoding and encoding of an audio stream as well.

The above resource dependencies may cause surprising performance results, as shown by the example in Fig. 2.13. The figure shows an example of two application graphs A and B with three actors each, mapped on a 3-processor system. Actors a_1 and b_1 are mapped on p_1, a_2 and b_2 are mapped on p_2, and a_3 and b_3 are mapped on p_3. Each actor takes 100 clock cycles to execute. While both applications A and B might look similar, the dependency in A is anti-clockwise and in B clockwise to highlight the situation in the above example of simultaneous H263 encoding and decoding.

Let us try to *add* the resource requirement of actors and applications, and try to reason about their behaviour when they are executing concurrently. Each processor has two actors mapped; each actor requires 100 time units. If we limit the information to only actor-level, we can conclude that one iteration of each a_1 and b_1 can be done in a total of 200 time units on processor P_1, and the same holds for processors P_2 and P_3. Thus, if we consider a total of 3 million time units, each application should finish 15,000 iterations, leading to 30,000 iterations in total. If we now consider the graph-level local information only, then we quickly realize that since there is only one initial token, the minimum period of the applications is 300. Thus, each application can finish 10,000 iterations in 3 million time units. As it turns out, none of these two estimates are achievable.

Let us now increase the information we use to analyze the application performance. We consider the worst-case response time as defined in Eq. 2.1 for each actor. This gives us an upper bound of 200 time units for each actor. If we now use this to compute our application period, we obtain 600 time units for each application. This translates to 5,000 iterations per application in 3 million time units. This is the guaranteed lower bound of performance. If we go one stage further, and try to analyze the full schedule of this two-application system by making a static schedule, we obtain a schedule with a steady-state of 400 time units in which each application

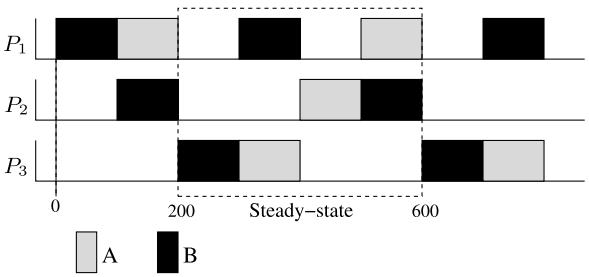


Fig. 2.14 Static-order schedule of applications in Fig. 2.13 executing concurrently

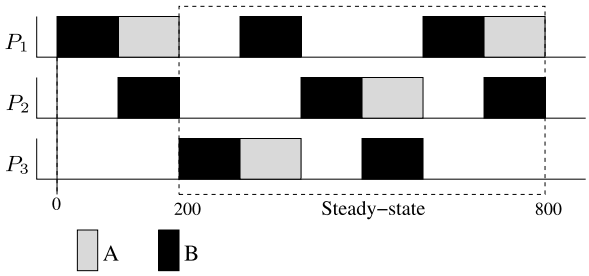


Fig. 2.15 Schedule of applications in Fig. 2.13 executing concurrently when B has priority

completes one iteration. The corresponding schedule is shown in Fig. 2.14. Unlike the earlier predictions, this performance is indeed what the applications achieve. They will both complete one iteration every 400 time units. If we consider dynamic ordering and let the applications run on their own, we might obtain the same order as in Fig. 2.14, or we might get the order as shown in Fig. 2.15. When the exact execution order is not specified, depending on the scheduling policy the performance may vary. If we consider a first-come-first-serve approach, it is hard to predict the exact performance since the actors have equal execution time and they arrive at the same time. If we assume for some reason, application A is checked first, then application A will execute twice as often as B, and vice-versa. The schedule in Fig. 2.15 assumes application B has preference when both are ready at the exact same time. The same behaviour is obtained if we consider round-robin approach with skipping. Interestingly, the number of combined application iterations are still 15,000 – the same as when static order is used.

Table 2.3 shows how different estimating strategies can lead to different results. Some of the methods give a false indication of processing power, and are not achievable. For example, in the second column only the actor execution time is considered. This is a very naive approach and would be the easiest to estimate. It assumes that all the processing power that is available for each node is shared between the two actors equally. As we vary the information that is used to make the prediction, the performance prediction also varies. This example shows why composability needs to be examined. Individually each application takes 300 time units to complete an iteration and requires only a third of the available processor resources. However, when another application enters in the system, it is not possible to schedule both of them with their lowest period of 300 time units, even though the total request for a node is only two-third. Even when preemption is considered, only one application

Table 2.3 Estimating performance: iteration-count for each application in 3,000,000 time units

Appl.	Only actors	Only graph	WC analysis (both graphs)	Static	RRWS/FCFS	
					A pref	B pref
A	15,000	10,000	5,000	7,500	10,000	5,000
B	15,000	10,000	5,000	7,500	5,000	10,000
Total	30,000	20,000	10,000	15,000	15,000	15,000
Proc Util	1.00	0.67	0.33	0.50	0.50	0.50

Table 2.4 Properties of scheduling strategies

Property	Static order	Dynamic order
Design time overhead		
Calculating Schedules	−−	++
Run-time overhead		
Memory requirement	−	++
Scheduling overhead	++	+
Predictability		
Throughput	++	−−
Resource Utilization	+	−
New job admission		
Admission criteria	++	−−
Deadlock-free guarantee	−	++
Reconfiguration overhead	−	+
Dynamism		
Variable Execution time	−	+
Handling new use-case	−−	++

can achieve the period of 300 time units while the other of 600. The performance of the two applications in this case corresponds to the last two columns in Table 2.3. Thus, predicting application performance when executing concurrently with other applications is not very easy.

8 Static vs Dynamic Ordering

Table 2.4 shows a summary of various performance parameters that we have considered, and how static-order and dynamic-order scheduling strategy performs considering these performance parameters. The static-order scheduling clearly has a higher design-time overhead of computing the static order for each use-case. The run-time scheduler needed for both static-order and dynamic-order schedulers is quite simple, since only a simple check is needed to see when the actor is active and ready to

fire. The memory requirement for static scheduling is however, higher than that for a dynamic mechanism. As the number of applications increases, the total number of potential use-cases rises exponentially. For a system with 10 applications in which up to 4 can be active at the same time, there are approximately 400 possible combinations – and it grows exponentially as we increase the number of concurrently active applications. If static ordering is used, besides computing the schedule for all the use-cases at compile-time, one also has to be aware that they need to be stored at run-time. The scalability of using static scheduling for multiple applications is therefore limited.

Dynamic ordering is more scalable in this context. Clearly in FCFS, there is no such overhead as no schedule is computed beforehand. In RRWS, the easiest approach would be to store all actors for a processor in a schedule; when an application is not active, its actors are simply skipped, without causing any trouble for the scheduler. It should also be mentioned here that if an actor is required to be executed multiple times, one can simply add more copies of that actor in this list. In this way, RRWS can provide an easy rate-control mechanism.

The static order approach certainly scores better than a dynamic one when it comes to predictability of throughput and resource utilization. Static-order approach is also better when it comes to admitting a new application in the system since the resource requirements prior and after admitting the application are known at design time. Therefore, a decision whether to accept it or not is easier to make. However, extra measures are needed to reconfigure the system properly so that the system does not go into deadlock as mentioned earlier.

A dynamic approach is able to handle dynamism better than static order since orders are computed based on the worst-case execution time. When the execution-time varies significantly, a static order is not able to benefit from early termination of a process. The biggest disadvantage of static order, however, lies in the fact that any change in the design, e.g. adding a use-case to the system or a new application, cannot be accommodated at run-time. The dynamic ordering is, therefore, more suitable for designing multimedia systems. In the following chapter, we show techniques to predict performance of multiple applications executing concurrently.

9 Conclusions

In this chapter, we began with motivating the need of having an application model. We discussed several models of computation that are available and generally used. Given our application requirements and strengths of the models, we chose the synchronous dataflow (SDF) graphs to model application. We provided a short introduction to SDF graphs and explained some important concepts relevant for this book, namely modeling auto-concurrency and modeling buffer-sizes on channels. We explained how various performance characteristics of an SDF graph can be derived.

The scheduling techniques used for dataflow analysis were discussed and classified depending on which of the three things – assignment, ordering, and timing – are done at compile-time and which at run-time. We highlighted two arbiter classes –

static and dynamic ordering, which are more commonly used, and discussed how application performance can be analyzed considering hardware constraints for each of these arbiters.

We then highlighted the issue of *composability* – mapping and analysis of performance of multiple applications on a multiprocessor platform in isolation. We demonstrated with a small, but realistic example, how predicting performance can be difficult when even small applications are considered. We also saw how arbitration plays a significant role in determining the application performance. We summarized the properties that are important for an arbiter in a multimedia system, and decided that considering the high dynamism in multimedia applications, the dynamic-ordering is more suitable.

http://www.springer.com/978-94-007-0082-6

Multimedia Multiprocessor Systems
Analysis, Design and Management
Kumar, A.; Corporaal, H.; Mesman, B.; Ha, Y.
2010, XVI, 164 p., Hardcover
ISBN: 978-94-007-0082-6