Chapter 2
Background and Problem Definition

The previous chapter motivated and gave an intuitive description of the problem of mapping multiple applications onto heterogeneous MPSoCs. Three sub-problems were identified, corresponding to three different application types and tool flows: sequential, parallel and SDR. Each of these problems is broad enough to allow for different formulations, interpretations and solutions. To avoid ambiguity, this chapter presents a formal description of the main problem and its sub-problems. The notation and the definitions introduced in this chapter are used throughout this book.

This chapter is organized as follows. Section 2.1 presents a motivational design that serves as running example in this book. Basic terminology and notation are introduced in Sect. 2.2. The multi-application mapping problem and its sub-problems are treated in Sects. 2.3–2.6. The chapter ends with a summary in Sect. 2.7.

2.1 Motivational Example

Consider the problem setup illustrated in Fig. 2.1, in which an embedded software engineer is asked to implement several applications on a heterogeneous MPSoC. Together with these applications, the input specification also provides a list of *use cases* or scenarios. Each use case consist of a list of applications that may run simultaneously, e.g., JPEG and LP-AF in the figure. The designer must then ensure that every use case executes on the target platform without violating any of the constraints.

The applications in Fig. 2.1 are real-life applications, which are used later in the case studies of this book. They are specified in different formats and have different constraints. The following four applications are used for test purposes:

Low-Pass Audio Filter (LP-AF): This is a C implementation of a low pass stereo audio filter in the frequency domain. It reads an input audio signal, transforms it into the frequency domain and applies a digital *Finite Impulse Response* (FIR) filter. The resulting signal is transformed back to the time domain and is forwarded to the

J. Castrillón Mazo and R. Leupers, *Programming Heterogeneous MPSoCs*, DOI: 10.1007/978-3-319-00675-8_2,

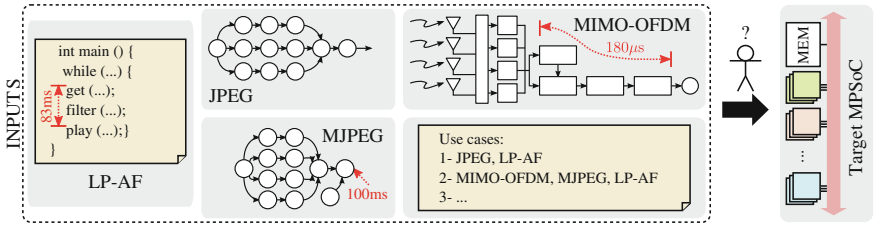

Fig. 2.1 Motivational example

loudspeakers. The application must produce filtered samples every 83 ms for a rate of 192 kbit/s.

JPEG Image Encoding/Decoding: This is a best-effort KPN implementation of the *Joint Photographic Experts Group* (JPEG) encoding and decoding standard [277].

MJPEG Video Decoder: This is a KPN implementation of *Motion JPEG* (MJPEG), which requires a minimum frame-rate of 10 frames-per-second.

MIMO-OFDM Transceiver: This is a generic algorithmic specification of a *Multiple-Input Multiple-Output* (MIMO) *Orthogonal Frequency-Division Multiplexing* (OFDM) digital wireless transceiver [188]. The specification includes a path latency constraint of 180 μs.

Today, provided with this input specification, a designer would follow a manual solution approach, due to the lack of automatic tools. He would start by implementing every single application individually and tuning its performance. Only after studying the behavior of every single application, a designer can start making multi-application design decisions. The following items describe a typical design process.

1. LP-AF (C application): A designer would first look for the best-suited processor to run the application on, which may include code optimizations. If the constraints are not met, the code must be parallelized and mapped to the platform. Several iterations may be required to finally meet the constraints.
2. JPEG (KPN application): For an already parallel application, the designer would start by analyzing and profiling the processes to identify bottlenecks. Typically, after obtaining an initial mapping of the application, several iterations are needed to maximize application performance.
3. MJPEG (KPN application): For this real-time application, instead of maximizing performance, different mappings must be explored until the constraints are met.
4. MIMO-OFDM (algorithmic application): An algorithmic description, e.g., in MATLAB, must be first understood in order to identify potential hardware acceleration or DSP routines. Algorithmic blocks that are not directly supported by the platform must be then implemented, e.g., in assembly or C. Once an implementation is available, the design is similar to that of MJPEG.
5. Multiple applications: With available single-application implementations, every use case must be analyzed individually. If the constraints are violated, the designer must *understand* the reasons and modify the mapping of some of the applications

accordingly. Usually, the configuration of less critical applications is modified first. For example, the best performing mapping of JPEG may block resources that are needed by the LP-AF application.

The steps above represent time-consuming *trial-and-error* tasks, e.g., application understanding, re-coding, debugging, profiling, simulation and testing. The steps change depending on the application type, which motivated the introduction of different tool flows. These intuitive steps underpin the problem definitions given in this chapter and the semi-automatic solutions presented in this book.

2.2 Preliminaries

Before introducing the formal problem definition, this section introduces the fundamentals of mapping and scheduling, basic concepts of source code performance estimation and the notation used throughout this book.

2.2.1 Mapping and Scheduling

Mapping and scheduling are recurring terms in this book. It is therefore important to first intuitively understand the differences between these two processes.

Consider an application composed of several tasks to be executed on an MPSoC. *Scheduling* refers to the process of deciding at which time instant a given task is to be run on a given PE. In *static scheduling*, the time instants are decided at design time, whereas in *dynamic scheduling*, they are decided at runtime. The latter is implemented by *scheduling policies*, which are used during program execution to arbitrate resource sharing. Examples of such policies include First-Come First-Served (FCFS) and priority-based scheduling. In the latter, a numerical value is assigned to every computation. At runtime, the scheduler selects the computation that is ready to execute and has the highest value. In FCFS, in turn, the scheduler executes the computations in the order they were issued.

Mapping refers to the process of assigning a PE to a task. In static or *fixed* mapping the allocation of PEs to tasks is performed at design time, whereas in *dynamic* mapping, tasks are dispatched to PEs at runtime according to a *mapping policy*. Examples of mapping policies include Round-Robin (RR) and Earliest Finishing Time (EFT). In a RR mapper, PEs are selected in a circular fashion, usually for the purposes of load balancing. In EFT, the dispatcher assigns the PE to the task that would execute it the earliest. The mapping and scheduling processes can be either coupled or decoupled. A decoupled mapper dispatches tasks to local schedulers, which then decide when to execute the task. A fixed mapping strategy is an example of such an approach. In a coupled process, both the *where* (mapping) and the *when* (scheduling)

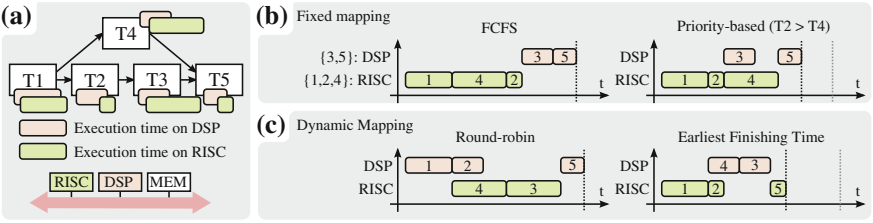

Fig. 2.2 Mapping and scheduling for the DAG problem. **a** Sample application. **b** Fixed mapping. **c** Dynamic mapping

are decided jointly. The EFT mapper is an example of a coupled process. Therefore, if not expressed explicitly, the term mapping refers to mapping *and* scheduling.

Figure 2.2 shows an example to illustrate the mapping and scheduling concepts introduced here. For the sake of clarity, a simple model is used in which an application is represented as a *Directed Acyclic Graph* (DAG), shown in Fig. 2.2a. In the graph, nodes represent tasks and arrows represent precedence constraints, i.e., tasks T2 and T4 can only execute after T1 has finished. The colored bars attached to the nodes represent the execution time of each task on each processor type (RISC and DSP). In this example, the communication cost is ignored. Figures 2.2b and c show the results of the mapping and scheduling process for different configurations in a *Gantt Chart*, where the time is plotted on the horizontal axis and the PEs on the vertical axis. Dotted lines are used to mark the total application execution time, often denoted *makespan*. Figure 2.2b shows possible results of a fixed mapping approach for the two different scheduling policies mentioned earlier. In both configurations, tasks T1, T2 and T4 are mapped to the RISC and tasks T3 and T5 to the DSP. It is assumed that after task T1 finishes, the FCFS scheduler selects task T4 before T2. The priority-based scheduler, in turn, selects T2 due to its priority. The results of a dynamic mapping approach are shown in Fig. 2.2c. In the RR example, it is assumed that the mapper assigns tasks to PEs in a strict order, even if they are busy. It is further assumed that the first assignment is T1 to DSP. Following the same order used for FCFS, the next assignments are T4 to RISC, T2 to DSP, T3 to RISC and T5 to DSP. The EFT mapper, instead, assigns a task to the PE that reports the earliest finishing time. As a result, T4 is assigned to the DSP and T2 to the RISC. This example shows how important the *runtime configuration* is for reducing the application makespan.

In general the problem of determining a static schedule for a DAG that minimizes the makespan on limited resources is NP-Complete [270]. Several heuristics have been therefore proposed to solve it, with a comprehensive survey in [157]. The parallel code problem introduced in Sect. 2.5 is in general more complex than the DAG problem.

There are other flavors of the mapping and scheduling processes which lie in between static and dynamic. For example, in *Quasi Static Scheduling* (QSS) several static schedules are computed for a single application. At runtime, different static scheduling configurations are switched depending on some execution conditions

or *scenarios* (see for example [95]). More generally, in *Quasi Dynamic Mapping* (QDM), several configurations are computed at design time which are switched dynamically at runtime. Each configuration may describe either a static or a dynamic mapping and scheduling. This is the approach followed in the multi-application problem.

2.2.2 Sequential Code Performance Estimation

In the DAG example in Fig. 2.2, the timing information of the tasks was assumed to be known. It is represented by colored bars with a length that expresses the sequential execution time of each one of the tasks on each of the two different processor types. It is clear that without this information it would be impossible to obtain the Gantt Charts shown in Fig. 2.2. In the multi-application problem, such information is not assumed to be known, but has to be determined by means of sequential code performance estimation.

Performance estimation is orthogonal to the problems addressed in this book. It is however very relevant both for the quality of the results and the applicability of the framework. While inaccurate estimates may invalidate the results, very accurate ones may be an overkill in early design phases. In such phases, many iterations are needed, hence fast, slightly inaccurate estimates may be preferred. In order to account for different use cases, the following methods for source code performance estimation can be used:

Annotation-based: This is an approach in which the programmer manually specifies the time of a given task on every given processor type. This simple technique is applicable for coarse-grained estimation, in cases where the programmer knows both the application and the target architecture. Within this book, this technique is only used for describing the timing of low-level assembly routines and hardware accelerators (see Chap. 7). For arbitrary source code, this technique could otherwise prove too inaccurate.

Table-based: Table-based performance estimation is a profiling technique based on source instrumentation, similar to the well-known GNU tool *gprof* [101]. For cross-platform estimation, the gprof approach is too coarse-grained since it estimates at the level of a line of code in the original program. In fine-grained approaches, the source code is lowered to a level of elementary operations, e.g., arithmetic and memory load/store. The cost for each of these operations is then read from a table, e.g., μ-Profiler [140]. This approach is applicable for determining application hot spots and for getting an early idea of the application runtime. It is however not very accurate for modeling non-scalar architectures, such as DSPs and VLIWs. It cannot model complex operations such as multiply-accumulate or *Single Instruction Multiple Data* (SIMD). Additionally, it ignores several of the performance variations introduced by

the target compiler, e.g., target specific optimizations and register spilling.

Emulation-based: Emulation-based approaches mimic the effects of the target compiler. This allows to account for target-specific optimizations and for the effects of the compiler backend phases: code selection, scheduling and register allocation. The more details of the target architecture and its compiler are emulated, the more accurate the estimation is. At the same time, the more details are emulated the slower the emulation becomes. This approach, in contrast to table-based, can estimate time for irregular architectures with special instructions.

In this book, the *Totalprof* [88] emulation-based profiler is used. TotalProf [88] is a retargetable source code profiler that mimics the effects of compiler backends and several dynamic effects of modern architectures (e.g., cache misses and branch mis-prediction). Compiler effects are modeled by a so-called *virtual backend*. This backend does not produce target binary code, but a host-based low-level code that mimics the target ISA. The timing reported by the backend scheduler is annotated back to this low-level code. As such, the timing of full portions of code is estimated. The granularity can vary from a set of operations, a loop iteration or an entire function, as opposed to single operations as it is done in the table-based approach.

Simulation-based: This approach uses *Instruction Set Simulators* (ISSs) to estimate time for different PEs. This method does not require compiler emulation since the ISSs directly run target binary code, hence the inaccuracies introduced by Totalprof's virtual backend disappear. In the last years, simulation speed has seen a steep increase, from *interpretive* simulation (e.g., Simple Scalar [34]), to *Just-In-Time Cached Compiled* (e.g., in [203]) and *Dynamic Binary Translation* (e.g., in [133]). These advances made it possible to apply simulation-based approaches within programming flows. This book uses the ISSs from Synopsys PA [250] and generated with Synopsys PD [252]. Despite the increase in simulation speed, target binary simulation is still usually slower than executing a host binary with timing annotations.

Measurement-based: This method estimates performance by using measurements from the target MPSoC. This method is potentially the most accurate, but requires the actual board which may not be available in early design phases or may not be accessible to all software developers. Due to its intrusive nature, measurement-based estimates may deviate from real application timing.

Analytical: Analytical or static performance estimation tries to find theoretical bounds on the *Worst-Case Execution Time* (WCET), without actually executing the code.[1] Using compiler techniques, all possible control paths are analyzed and bounds are computed by using an abstract model of the architecture. This task is particularly difficult in the presence of caches and other architectural features that lead to non-deterministic timing behavior. For such architectures, the WCET might be too

[1] This method is not used in this book, but is listed for the sake of completeness.

pessimistic and thus induce bad decisions (e.g., wrong schedules). A good survey of static techniques can be foundin [289].

2.2.3 Notation

Throughout this book, the following conventions are used:

- Scalars are represented by lower case letters in normal type (a, b, c).
- Sets, tuples and sequences are represented by capital letters in normal type (A, B, C). Sometimes, more than one letter is used so that it is easier to recognize the concept that is referred to, e.g., PE could represent a processing element.
- Families of sets are denoted by capital letters in calligraphic type. The same notation is also sometimes used for sets of tuples, in order to distinguish the set name from its elements' names. For example, the set of all processing elements is denoted $\mathscr{PE}$. The power set of set A is represented by $\wp(A)$.
- Functions are represented by Greek letters (α, β, γ). For the sake of readability, some functions are given full names. This eases reading through the pseudocode of some algorithms. $\mathscr{I}(\alpha)$ denotes the image of a function α.
- Usually the context in which an entity (e.g., variable, set and tuple) is contained is given in superscript. Subscripts are used to represent count or to identify an element within a collection. As example, if SOC denotes a given hardware platform, the tuple PE_i^{SOC} represents the i-th PE of the platform SOC. This convention is sometimes ignored if the context is obvious.

2.3 Multi-Application Problem

This section provides a definition of the multi-application problem which is more precise than the intuitive description given in Chap. 1. Before formally stating the problem in Sect. 2.3.2, several fundamental definitions are given in Sect. 2.3.1.

2.3.1 Definitions

The multi-application problem in Fig. 1.5 involves an *architecture*, a set of *applications* and a set of *constraints*. Its solution is a *runtime configuration* data base that specifies how applications are to be run depending on the execution scenario. These concepts are formalized in the following sections.

2.3.1.1 Architecture

A heterogeneous MPSoC is composed of processing elements of different types, storage elements, peripherals and interconnect. This section defines the elements which are relevant to the multi-application mapping problem and introduces their models.

Definition 1 A **processor type** (PT) stands for a processor architecture that is instantiated, possibly multiple times, in the MPSoC. It is not only distinguished by its *Instruction Set Architecture* (ISA), but also by its timing and power characteristics. It can be associated with the reference name of the processor core, e.g., ARM926EJ-S or TI C64x+.

A processor type is modeled as a triple $PT = (\mathscr{CM}^{PT}, X^{PT}, V^{PT})$. $\mathscr{CM}^{PT}$ is an abstract *cost model* that can be seen as a set of functions that associate a numerical value with an application. Examples of these values are execution time and energy consumption. X^{PT} is a set of attributes that give extra information about the core and its supporting runtime system. The attributes model application independent costs, e.g., penalties for a cache misses or boot-up time. The attribute $x^{PT}_{\text{cs}} \in X^{PT}$ is the amount of time required to perform a context switch. The attribute $x^{PT}_{\text{tasks}} \in X^{PT}$ models the maximum amount of tasks that can share this type of resource. V^{PT} is a set of variables $V^{PT} = \{v_1, \ldots, v_n\}$, where v_i is defined over a domain $D^{PT}_{v_i}$. Processor variables as well as other variables defined in this chapter are used to model free parameters that are to be set by the mapping process. For example, the variable $v^{PT}_{\text{SP}} \in V^{PT}$ represents the scheduling policy used by the processor. The domain of this variable describes the policies supported by the processor type, e.g., priority-based or round-robin. The set of all processor types in the MPSoC is denoted $\mathscr{PT}$.

Definition 2 A **processing element** (PE) is an instance of a given processor type and thus inherits its cost model, attributes and variables. The set of all PEs in the MPSoC is denoted $\mathscr{PE} = \{PE_1, PE_2, \ldots, PE_{n_{PE}}\}$. For convenience, let PE^v_i be the i-th PE of type v, $PE^v_i \in \mathscr{PE}$, $v \in \mathscr{PT}$. Additionally, let $\mathscr{PE}^v$ denote the set of all PEs of type v. Consequently, $\mathscr{PE} = \sqcup_{v \in \mathscr{PT}} \mathscr{PE}^v$, with $\sqcup$ being the disjoint union.

Definition 3 A **communication resource** (CR) refers to hardware modules that can be used to implement communication among application tasks. Shared and local memories as well as hardware communication queues are examples of CRs. A CR is modeled as a pair of attributes that describe hardware properties, $CR = (x^{CR}_{\text{MEM}}, x^{CR}_{\text{CH}})$. The amount of memory of the CR is modeled by the attribute x^{CR}_{MEM}, whereas x^{CR}_{CH} models the amount of logical communication channels that can be mapped to the CR.

Definition 4 A **communication primitive** (CP) represents software APIs that can be used to communicate among application tasks in the target platform. It is modeled as a 4-tuple $CP = (PE_i, PE_j, S \subseteq \mathscr{CR}, \mathscr{CM}^{CP})$ that expresses how a task in PE_i can communicate with a task in PE_j, where $i = j$ is allowed. It further

expresses that the actual communication uses a subset S of the resources of the platform. This allows to represent a wider range of software communication means, e.g., *Inter-Processor Communication* (IPC) over more than one resource. It is possible to distinguish communication means implemented differently over the same resource, e.g., lock-less queues, semaphore-protected queues and queues with packaging over a single shared memory. $\mathscr{CM}^{CP}$ is a cost model of the communication primitive that consists of functions that associate communication volume with a numerical value. An example of such a function, denoted $\zeta^{CP}: \mathbb{N} \rightarrow \mathbb{N}$, associates a cost in terms of time with any amount of bytes to be transmitted. The set of all CPs in an MPSoC is denoted by $\mathscr{CP}$.

Definition 5 A **platform graph** (SOC) is a multigraph $SOC = (\mathscr{PE}, \mathscr{E})$ that represents the target platform. $\mathscr{E}$ is a multiset over $\mathscr{PE} \times \mathscr{PE}$ that contains an element $e_{ij} = (PE_i, PE_j)$ for every $CP \in \mathscr{CP}$. For convenience, let CP_{ij} denote the communication primitive associated with edge e_{ij}.

2.3.1.2 Applications

Applications are the second main input to the flow in Fig. 1.5. Different application types are modeled differently as discussed later in Sects. 2.4–2.6. This section introduces an abstract model that allows to formulate the multi-application problem in a general way.

Definition 6 An **application** (A) is a triple $A = (\mathscr{M}^A, V^A, \mathscr{K}^A)$. $\mathscr{M}^A$ is an abstract model of the application. Depending on the model, three application *types* can be distinguished: sequential, parallel and SDR. V^A is a set of application variables that are given a value within a domain after the mapping process. $\mathscr{K}^A$ is a set of constraints defined on V^A. Depending on the constraints, three application *classes* can be distinguished: hard real-time, soft real-time and non real-time (or best-effort). Hard real-time applications must fulfill every single deadline, whereas an average analysis suffices for soft real-time applications. The distinction between hard and soft real-time serves to configure the runtime system, so that more critical applications are given preference. The set of all applications in the multi-application problem is denoted by $\mathscr{A}$.

Application variables are used to define constraints and measure the quality of the mapping results. The exact semantics of the variables varies with the application type. During the analysis phases, models are extended by adding variables that help to steer the mapping process. Variables can express more than timing characteristics. They can be used to enforce mapping constraints among other constraints as discussed in Sect. 2.3.1.4.

In this book, hard and soft real-time applications are defined as in the wider context of real-time *systems* [150], where the compiler is expected to *guarantee* at design time that the *system* will react within the specified time, provided a load characterization (or *load hypothesis*). As with traditional C programming, it is up to

the user to ensure that both the application specification and the input stimuli lead to a high coverage. Additionally, with no assumptions on the target hardware, the tool flows presented in this book cannot claim strong guarantees due to the various sources of timing unpredictability [262]. Finally, this book is not concerned with other aspects that are typically associated with real-time systems, such as failure models and error recovery.

Application variables allow to model the single-application problem as a general *Constraint Satisfaction Problem* (CSP) [267]. Consider an application $A = (\mathscr{M}^A, V^A, \mathscr{K}^A)$ with application variables $V^A = \{v_1, \ldots, v_n\}$, where each variable v_j is defined over a domain $D^A_{v_j}$. A constraint $K^A_i \in \mathscr{K}^A$ is a pair $(S_i \subseteq V^A, R_i)$, with $S_i = \{v_1, \ldots, v_m\}$ a subset of application variables and R_i an m-ary relation $(m = |S_i|)$ on $\times_{v \in S_i} D^A_v$. The result of a mapping process can be seen as an *evaluation* of the application variables $\varepsilon : V^A \rightarrow \bigcup_{l=1}^{n} D^A_{v_l}$, with $\forall v_l \in V^A, \varepsilon(v_l) \in D^A_{v_l}$. That is, it assigns a value to every variable in its respective domain. A result satisfies a constraint $K^A_i = (S_i = \{v_1, \ldots, v_m\}, R_i) \in \mathscr{K}^A$ if the variable evaluation belongs to the region defined by R_i, i.e., $(\varepsilon(v_1), \ldots, \varepsilon(v_m)) \in R_i$. Consider as an example the five-task application model in Fig. 2.2. Suppose that the variable set consists of the time stamps at which each task finishes its execution $V^A = \{t_1, \ldots, t_5\}$, with $D^A_{t_i} = \mathbb{N}$ for every variable. A timing constraint k^t_i for each task is then represented as (V^A, R), with $R = \{(v_1, \ldots, v_5) \in \mathbb{N}^5, v_i \leq k^t_i, i \in \{1, \ldots, 5\}\}$.

The CSP described above is accompanied by a single-objective optimization problem that serves to finally select a solution among all feasible ones. Depending on the application class, a different objective is minimized: the application makespan for best-effort applications and the resource usage for real-time applications.

Application models partition the set $\mathscr{A}$ into a sequential subset ($\mathscr{A}^{\text{seq}}$), a parallel subset ($\mathscr{A}^{\text{kpn}}$) and an SDR subset ($\mathscr{A}^{\text{sdr}}$), so that $\mathscr{A} = \mathscr{A}^{\text{seq}} \sqcup \mathscr{A}^{\text{kpn}} \sqcup \mathscr{A}^{\text{sdr}}$. Similarly, application constraints partition the set into subsets of hard real-time applications ($\mathscr{A}^{\text{hrt}}$), soft real-time applications ($\mathscr{A}^{\text{srt}}$) and non real-time applications ($\mathscr{A}^{\text{nrt}}$).

How applications are expected to interact during runtime is specified in the form of a *multi-app* description as shown in Fig. 1.7. This description can be specified in form of a graph or a set of *use cases*, as follows.

Definition 7 An **application concurrency graph** (ACG) is an edge-weighted graph $ACG = (\mathscr{A}, E^{ACG}, W^{ACG})$. An edge $e_{ij} = (A_i, A_j) \in E^{ACG} \subseteq \mathscr{A} \times \mathscr{A}$ indicates that applications A_i and A_j may run simultaneously. The weights on the edge, $W^{ACG}(e)$, serve to mark that certain combinations of applications are more important than others.

Definition 8 A **use case** (UC) represents a set of applications that can run concurrently. It is modeled as a pair $UC = (S^{UC}, p^{UC})$ where $S^{UC} \subset \mathscr{A}$ is the set of applications and p^{UC} is a weighting factor. As with edge weights in the ACG, this factor is used to mark which use cases are more important for the user. It can be thought as the probability that the applications in S^{UC} actually run simultaneously. For convenience, the notation $A \in UC$ is used to represent that $A \in S^{UC}$, $UC = (S^{UC}, p^{UC})$.

Note that any fully connected subgraph of the ACG, i.e., any clique, determines a use case. Worst-case multi-application scenarios are determined by the maximal cliques of the graph. Let $UCG = (S^{UC} \subset \mathscr{A}, E^{UC} \subseteq E^{ACG}, W^{UC} \subseteq W^{ACG})$ be a subgraph of the ACG, so that S^{UC} is a clique. In this case, the weight of the use case is defined as

$$p^{UC} = \prod_{e \in E^{UC}} W^{UC}(e) \tag{2.1}$$

Equation 2.1 is motivated by the computation of the joint probability of independent events. Assume that the weights on the ACG edges represent the probability of two applications running concurrently, i.e., $\forall e_{ij} = (A_i, A_j) \in E^{ACG}$, $W^{ACG}(e_{ij}) = P(A_i \wedge A_j)$. Assuming statistical independence, the probability that all applications in a use case run simultaneously is given by

$$P\left(\bigcap_{A_i \in UC} A_i\right) = \prod_{A_i \in UC} P(A_i) = \left(\prod_{e \in E^{UC}} W^{UC}(e)\right)^{\frac{1}{|S^{UC}|-1}} \tag{2.2}$$

where the term in Eq. 2.1 can be identify on the right-hand side of Eq. 2.2.

Definition 9 An **application concurrency set** (ACS) provides an alternative representation to the ACG, where all use cases are explicitly defined in a set $ACS = \{UC_1, \ldots, UC_n\}$.

A graph representation provides a compact way to represent applications' interaction. However, a graph representation does not offer enough control to represent a given list of multi-application scenarios. Consider the case of three applications, where any combination of two applications is allowed. A graph representation would inevitably include a spurious use case with the three applications in it, since the graph is fully connected itself.

2.3.1.3 Performance Estimation Functions

As mentioned in Sect. 2.2.2, performance estimation plays an important role in the mapping process. Timing information for both source code and for data communication is needed. The timing estimators are modeled as functions within the cost model of processor types and communication primitives. In this book, time is measured in terms of cycles of the platform's main clock. Therefore, the cost models of resources that use a different clock have to be scaled accordingly. In this way, the costs are independent of the specific frequency used in the platform, which may be variable.

Consider an application $A = (\mathscr{M}^A, V^A, \mathscr{K}^A)$ to be executed on an $SOC = (\mathscr{PE}, \mathscr{E})$, with processor types $\mathscr{PT} = \{PT_1, \ldots, PT_n\}$, $PT_i = (\mathscr{CM}^{PT_i}, X^{PT_i}, V^{PT_i})$ and communication primitives $\mathscr{CP} = \{CP_{ij}, CP_{ij} = (PE_i, PE_j, S, \mathscr{CM}^{CP_{ij}})\}$.

Definition 10 Communication cost estimation is the process of estimating the amount of time needed to transfer data by using a given primitive CP_{ij}. The cost is modeled by a function $\zeta^{CP_{ij}} \in \mathscr{CM}^{CP_{ij}}$, $\zeta^{CP_{ij}} : \mathbb{N} \to \mathbb{N}$. The function is parameterized by three constants: *offset*, *start* and *stair*. The *offset* is a constant overhead in terms of clock cycles that models time spent in synchronization and initialization. Usually these values do not depend on the amount of bytes to be transferred. The *start* parameter models bytes that can be transferred during the initialization phase. The *stair* parameter is a pair of values that determines how the time increases as a function of the transferred bytes. For example, the pair (4, 1) tells that every 4 bytes the communication cost is increased by 1 cycle. Let the stair be described as (s_1, s_2), the communication cost for b bytes is defined by:

$$\zeta^{CP_{ij}}(b) = \begin{cases} offset & \text{if } b < start \\ offset + s_2 \cdot \lceil (b - start + 1)/s_1 \rceil & \text{otherwise} \end{cases} \tag{2.3}$$

This simple model allows to represent communication primitives based on different underlying physical communication channels, such as dedicated hardware queues and buses.

Definition 11 Source code performance estimation is the process of estimating the amount of time needed for a given computation executing on a given processor type PT_i. This is also modeled by functions $\{\zeta_1^{PT_i}, \ldots, \zeta_l^{PT_i}\} \subseteq \mathscr{CM}^{PT_i}$, where each function is defined as $\zeta_j^{PT_i} : S \subseteq \mathscr{M}^A \to \mathbb{N}$. As an example, let $S = \{T1, \ldots, T5\}$ for the application in Fig. 2.2a. The execution time of the i-th task for the two different processors can then be expressed as $\zeta^{PT_1}(T_i)$ and $\zeta^{PT_2}(T_i)$.

Compared to communication cost estimation, source code performance estimation is a more intricate process. Any of the techniques described in Sect. 2.2.2 can be used for this purpose. The exact definition of the functions $\zeta^{PT_{ij}}$ depends on the application type (sequential, parallel or SDR).

2.3.1.4 Constraints

Consider an application $A = (\mathscr{M}^A, V^A, \mathscr{K}^A)$ to be executed on an $SOC = (\mathscr{PE}, \mathscr{E})$. Recall that $K_i^A = (S_i \subseteq V^A, R_i)$, with variable $v \in S_i$ defined over a domain D_v^A. Several constraint types are allowed, defined by the nature of v and D_v^A.

Definition 12 Timing constraints are defined over application variables that relate to timing. Depending on the application model, these variables could represent application makespan, task finishing times, etc. Since time is measured in terms of clock cycles, these variables are defined over the natural numbers, i.e., $D_v^A \subset \mathbb{N}$.

Definition 13 Processing constraints are defined over the platform computing resources. They allow to constrain the mapping process, so that tasks are only mapped

to a given subset of the PEs, i.e., $D_v^A \subset \mathscr{PE}$, or a given processor type u, i.e., $D_v^A = \mathscr{PE}^u$.

Definition 14 **Memory constraints** are defined over the platform communication resources. They allow to constrain the whole memory requirements of the application and the individual memory requirements of communication channels. For these constraints, $D_v^A \subset \mathbb{N}$.

2.3.1.5 Runtime Configuration

For an application $A = (\mathscr{M}^A, V^A, \mathscr{K}^A)$ and platform $SOC = (\mathscr{PE}, \mathscr{E})$ a configuration describes the results of the mapping process. More formally, let $S_{\text{proc}} \subset \mathscr{M}^A$ denote the model elements that relate to processing (e.g., tasks in the example in Fig. 2.2a). Similarly, let $S_{\text{comm}} \subset \mathscr{M}^A$ denote the elements that relate to communication (e.g., edges in Fig. 2.2a).

Definition 15 A **runtime configuration** (RC^A) is a triple $RC^A = (\mu_p, \mu_c, \mu_a)$ of functions. The mapping of processing to PEs is specified by $\mu_p \colon S_{\text{proc}} \subset \mathscr{M}^A \rightarrow \mathscr{PE}$. The mapping of application communication channels is defined by $\mu_c \colon S_{\text{comm}} \subset \mathscr{M}^A \rightarrow \mathscr{E}$. Finally, the function μ_a is a mapping of platform variables and application variables to their corresponding domains, i.e., $\forall PE_i^{PT} \in \mathscr{I}(\mu_p), \forall v \in V^{PT}, \mu_a(v) \in D_v^{PT}$, similarly, $\forall v \in V^A, \mu_a(v) \in D_v^A$. In other words, μ_a fixes the runtime configuration for all processing elements (e.g., scheduling policy) and remaining free variables of the application (e.g., tasks' finishing time and memory consumption). A set of runtime configurations for a given application A is denoted $\mathscr{RC}^A$.

Definition 16 A runtime configuration is **valid** if the following conditions are met:

1. Application constraint satisfaction: As mentioned before, a mapping can be seen as an evaluation of the application variables, $\varepsilon \colon V^A \rightarrow \bigcup_{l=1}^{n} D_{vl}^A$. The application constraints are satisfied if

$$\forall K_i^A = (S_i = \{v_1, \ldots, v_m\}, R_i) \in \mathscr{K}^A, (\varepsilon(v_1), \ldots, \varepsilon(v_m)) \in R_i \tag{2.4}$$

2. Platform constraint satisfaction: The assignment must not violate the attributes provided by the platform resources. For example, memory allocation on communication primitives must not exceed the maximum memory available in the CR, nor the amount of logical channels (x_{MEM}^{CR}, x_{CH}^{CR} in Definition 3).
3. Logical validity: For every application communication channel $C \in S_{\text{comm}}$, with producer task $\text{src}(C) \in S_{\text{proc}}$ and consumer $\text{dst}(C) \in S_{\text{proc}}$, it must hold:

$$\text{src}(\mu_c(C_i)) = \mu_p(\text{src}(C)) \wedge \text{dst}(\mu_c(C_i)) = \mu_p(\text{dst}(C)) \tag{2.5}$$

The last condition ensures that tasks that communicate with each other run on processors that can communicate. As mentioned in Sect. 2.3.1.2, from all valid configurations the single application flows would select the one that minimizes the makespan (for best-effort) or the resource usage (for real-time).

Definition 17 A **use case runtime configuration** ($\mathscr{RC}^{UC}$) is a set of runtime configurations. Given a use case $UC = (S^{UC} = \{A_1, \ldots, A_n\}, p^{UC})$, $\mathscr{RC}^{UC} = \{RC^{A_1}, \ldots, RC^{A_n}\}$. A $\mathscr{RC}^{UC}$ is **valid** if each $RC^{A_i} \in \mathscr{RC}^{UC}$ is valid and if they are *jointly* valid, i.e., the validity is not violated by mapping all applications in the UC to the platform.

2.3.2 Problem Statement

With the previous definitions it is now possible to state the multi-application problem.

Definition 18 **Multi-application problem:** Given an application set $\mathscr{A}$, a graph $ACG = (\mathscr{A}, E^{ACG}, W^{ACG})$ (or a set ACS) and a target platform model $SOC = (\mathscr{PE}, \mathscr{E})$, find a valid use case runtime configuration $\mathscr{RC}_i^{UC}$ for every use case $UC_i \in ACS$.

As can be inferred from this definition, solving the multi-application problem is a challenging task. It is therefore divided into sub-problems that first address single application problems. A logical division arises from the different application models. The sequential problem for $A \in \mathscr{A}^{\text{seq}}$ is stated in Sect. 2.4, the parallel problem for $A \in \mathscr{A}^{\text{kpn}}$ in Sect. 2.5 and the SDR problem for $A \in \mathscr{A}^{\text{sdr}}$ in Sect. 2.6.

2.4 Sequential Code Problem

As mentioned in Sect. 2.1, a possible approach for mapping a sequential application ($A \in \mathscr{A}^{\text{seq}}$) consist in selecting the PE on which the application meets the constraints. An automatic flow following this approach would have two disadvantages. First, it would fail if no PE can run the application within its time constraint. Second, it would not profit from the parallel computing power of the target MPSoC.

The sequential problem is therefore addressed as a parallelism extraction problem from a sequential C program. Once a parallel specification is obtained, it becomes a parallel mapping problem. Before formally stating the problem, the next section gives background knowledge on compiler technology and parallelism extraction.

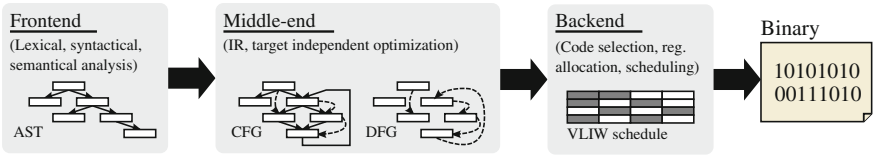

Fig. 2.3 Phases in a traditional uni-processor compilation flow

2.4.1 Compilers and Parallelism Extraction

A compiler is typically divided into three phases: The *frontend*, the *middle-end* and the *backend* [3, 193] (see Fig. 2.3). The frontend checks for the lexical, syntactic and semantic correctness of the application. The main data structure delivered by the frontend is the so-called *Abstract Syntax Tree* (AST). An AST is a structured representation of the application according to the syntax of the underlying programming language. It is used for program analysis and for code selection in the backend phase of the compiler. ASTs provide a language-independent abstraction of the application known as *Intermediate Representation* (IR). The middle-end performs different analyses on the IR that enable target-independent optimizations that mainly aim at improving the performance of the generated code. Two data structures are key for the optimizations in the middle-end, namely the *Control Flow Graph* (CFG) and the *Data Flow Graph* (DFG). Finally, the backend produces target binary code. The backend itself is also divided into phases. Typical backend phases include *code selection*, *register allocation* and *instruction scheduling*. These phases are machine dependent and therefore require a model of the target architecture.

For the purpose of parallelism extraction, the compiler phases perform conceptually the same tasks but at a different, *coarser* level. While a traditional compiler optimizes for *Instruction Level Parallelism* (ILP), a coarser parallelism is sought when targeting a multi-processor platform. Therefore, this section introduces some compiler concepts that are needed to obtain an application model ($\mathscr{M}^A$) that is suitable for parallelism extraction. A more comprehensive treatment of compilers can be found in [3, 193]. This section also introduces basic concepts of parallelism.

2.4.1.1 Intuitive Presentation

Consider the sample application presented in Fig. 2.4a. Suppose that the call to function `f2` does not modify the contents of array `A`, so that the first and the second loops are independent. Notice also that in the second loop, the variable `s` is only updated every four iterations. Based on these observations it is possible to represent the application as the abstract graph shown in Fig. 2.4b, where some details were omitted for the sake of clarity. In the graph, solid lines represent *control flow* induced by the sequential specification, and dashed lines represent *data dependencies*. Control edges are annotated with the amount of times each node follows its predecessor

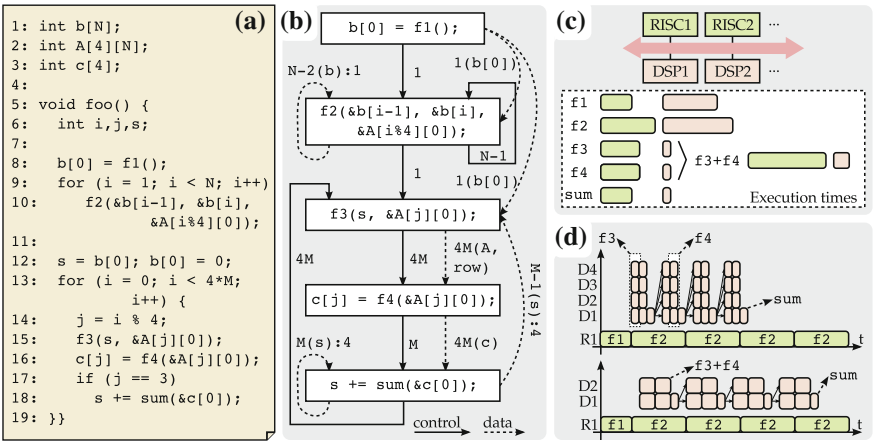

Fig. 2.4 Sequential code problem. **a** Original C code. **b** Control and data flow representation. **c** Timing information. **d** Two possible mappings

in an execution of the application. Besides the edge count, data edges are annotated with the variable that produced the dependency. In the case of loops, an additional *dependency distance* is annotated after the colon. The distance represents the amount of iterations between a write to and a read from a variable. From this graph, it follows that after `f1` returns, both loops can execute in parallel. Furthermore, every four iterations of the functions `f3` and `f4` can be run in parallel as well. This graph now exposes the parallelism that was originally hidden in the application. Now assume a platform with two different processor types and the execution times represented by colored bars in Fig. 2.4c. With this platform information, two possible schedules are shown in Fig. 2.4d. In the schedules it is assumed that `N = 4` and `M = 4`. The upper schedule exploits all the parallelism exposed by the application. However, it is not necessarily better than the second schedule, which produces the same application makespan while only using three PEs.

This example shows the main steps needed to address the sequential problem. The steps will be further discussed in the following sections. (1) Obtain a graph representation of the application that displays all kinds of dependencies (see Sects. 2.4.1.2–2.4.1.4). (2) Discover hidden parallelism in the graph (see Sect. 2.4.1.5). (3) Finally, select a parallel specification that better suits the target platform (see Sect. 2.4.1.6).

2.4.1.2 Control Flow Analysis

The example in Fig. 2.4 intuitively introduced the concept of the control flow of a program. The following definitions formalize the concepts.

Definition 19 An **intermediate representation** (IR^A) of an application is a pair $IR^A = (S^A_{\text{stmt}}, S^A_{\text{f}})$, where $S^A_{\text{stmt}} = \{s^A_1, \ldots, s^A_n\}$ is the set of all IR-statements. IR-

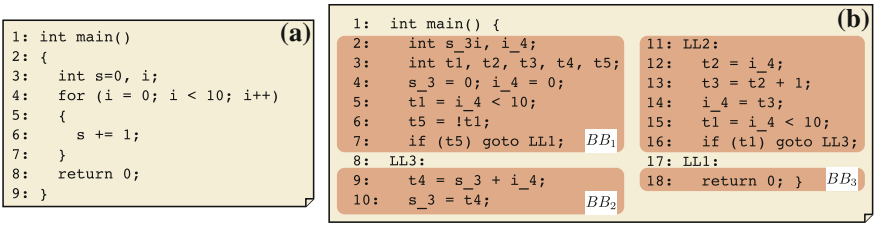

Fig. 2.5 Example of three-address code representation. **a** Original C code. **b** 3AC representation

statements are a *Three-Address Code* (3AC) representation of the original source code statements. $S_{\mathrm{f}}^{A} = \{f_1^A, \ldots, f_m^A\}$ is the set of all functions defined in the application. Each function $f \in S_{\mathrm{f}}^{A}$ maps to a subset of statement $S_{\mathrm{stmt}}^{f^A} \subset S_{\mathrm{stmt}}^{A}$.

An example of a 3AC representation of a simple application is shown in Fig. 2.5. The example shows how C constructs are lowered into simpler statements, labels and `goto` statements. The for loop in Lines `4`–`7` in Fig. 2.5a is implemented in Lines `7`–`16` in Fig. 2.5b. The shaded regions in Fig. 2.5b are *Basic Blocks* (BBs) which are basic analysis elements in compiler theory.

Definition 20 A **basic block** (BB^A) of an application A is a maximal sequence of consecutive IR-statements $BB^A = (s_1^{BB^A}, \ldots, s_n^{BB^A})$ in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [3]. The first statement, $s_1^{BB^A}$, is called the leader of the basic block. The set of all basic blocks in a function f^A is denoted by $\mathscr{BB}^{f^A}$ and in an application A by $\mathscr{BB}^A$.

In the example in Fig. 2.5b, three basic blocks are defined by the statements in Lines `2`–`7`, Lines `9`–`16` and Line `18`, represented by shaded areas. Note that label `LL2` does not define a new basic block, since this label is never jumped to. Which basic blocks are executed is controlled by conditions (see for example Line `7` in Fig. 2.5b). This is formally modeled as *control dependencies*.

Definition 21 **Control dependence** (δ^c) is a relation between two statements. A statement s_2^A is control dependent on a statement s_1^A, denoted $s_1^A \; \delta^c \; s_2^A$, if its execution depends on the execution of s_1^A.

In the example in Fig. 2.5b, $s_7 \; \delta^c \; s_9$ and $s_7 \; \delta^c \; s_{18}$, where statements are identified by the lines they appear on. The control dependence relation can be extended to basic blocks and ignored for statements inside a basic block, since they are always executed if the first statement is executed. For example, $BB_1 \; \delta^c \; BB_2$ and $BB_1 \; \delta^c \; BB_3$. Dependencies at the level of basic blocks are easily visualized by means of a CFG.

Definition 22 A **control flow graph** (CFG^{f^A}) of a function $f^A \in S_{\mathrm{f}}^{A}$ is a directed graph $CFG^{f^A} = (\mathscr{BB}^{f^A}, E_c^{f^A})$ in which there is a control edge $e_{ij} =$

$(BB_i^A, BB_j^A) \in E_c^{f^A} = \mathscr{BB}^{f^A} \times \mathscr{BB}^{f^A}$ if $BB_i^A \; \delta^c \; BB_j^A$. For a node $BB^A \in \mathscr{BB}^{f^A}$, $\text{Succ}(BB^A)$ and $\text{Pred}(BB^A)$ are the sets of direct successors and predecessors in the graph, i.e., $\text{Succ}(BB^A) = \{v \in \mathscr{BB}^{f^A}, (BB^A, v) \in E_c^{f^A}\}$ and $\text{Pred}(BB^A) = \{v \in \mathscr{BB}^{f^A}, (v, BB^A) \in E_c^{f^A}\}$.

In order to discover loops and find regions in a CFG, the concepts of *dominance* and *postdominance* are important. Given two nodes $n_1, n_2 \in \mathscr{BB}^{f^A}$, it is said that n_1 dominates n_2, denoted n_1 dom n_2, if every control path that reaches n_2 from the *start* node must go through n_1. Similarly, the node n_2 postdominates n_1, denoted n_2 pdom n_1, if every control path starting from n_1 and ending in the *end* node must go through n_2. The start and end nodes are special nodes that are usually added to the CFG of a function to ease analysis. This is simply done by adding edges from the start node to every node without predecessors, and from every node without successors to the end node.

2.4.1.3 Data Flow Analysis

Data Flow Analysis (DFA) serves to gather information at different program points, e.g., about available defined variables (*reaching definitions*) or about variables that will be used later in the control flow (*liveness analysis*). For the purpose of parallelism extraction, it is crucial to know where data is produced and used. Only with this information it is possible to insert correct communication statements if a portion of the application is offloaded to a different PE. It also serves to know if it is beneficial at all to parallelize a computation. In the example in Fig. 2.4b the backward edge to function `f2` prevents the first loop of the program to be parallelized (see the Gantt Charts in Fig. 2.4c). The following definitions serve to obtain a graph representation of a function, similar to the one in Fig. 2.4b.

Definition 23 A **data dependence** $(\delta^f, \delta^o, \delta^a)$ between two statements s_1^A, s_2^A indicates that s_2^A cannot be executed before s_1^A. There are three different kinds of dependencies:

- *Read After Write* (RAW, also *true/flow dependence*): Statements s_1^A and s_2^A are RAW-dependent, denoted $s_1^A \; \delta^f \; s_2^A$, if s_1^A writes a resource that s_2^A reads thereafter.
- *Write After Write* (WAW, also *output dependence*): Statements s_1^A and s_2^A are WAW-dependent ($s_1^A \; \delta^o \; s_2^A$) if s_2^A modifies a resource that was previously modified by s_1^A.
- *Write After Read* (WAR, also *anti-dependence*): Statements s_1^A and s_2^A are WAR-dependent ($s_1^A \; \delta^a \; s_2^A$) if s_2^A modifies a resource that was previously read by s_1^A.

Definition 24 A **data flow graph** (DFG^{f^A}) of a function $f^A \in S_\text{f}^A$ is a directed multigraph $DFG^{f^A} = (S_\text{stmt}^{f^A}, E_d^{f^A})$ where $S_\text{stmt}^{f^A}$ is the set of statements of the

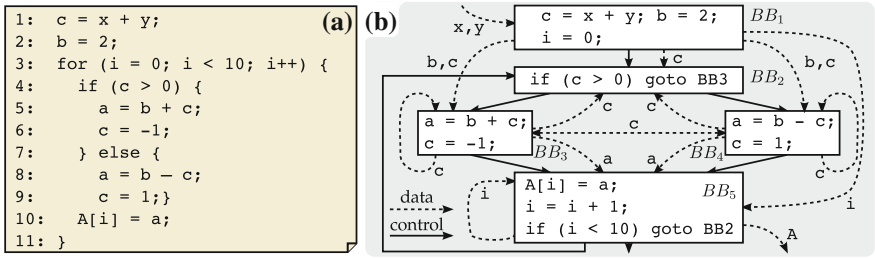

Fig. 2.6 CDFG Example. **a** Original C code. **b** Control and data flow representation

function and $E_d^{f^A}$ is a multiset on $S_{\text{stmt}}^{f^A} \times S_{\text{stmt}}^{f^A}$. There is a data edge $e_{ij} = (s_i^A, s_j^A) \in E_d^{f^A}$ if $s_i^A \; \delta^{f,o,a} s_j^A$. Edges may be labeled by the kind of dependency they represent.

Definition 25 A **control-data flow graph** ($CDFG^{f^A}$) of a function $f^A \in S_{\text{f}}^A$ is an annotated directed multigraph $CDFG^{f^A} = (\mathscr{BB}^{f^A}, E_c^{f^A} \cup E_d^{f^A*}, \text{var}_{\text{size}})$ where $E_d^{f^A*}$ is the set of data flow edges $E_d^{f^A}$ *summarized* at the basic block level. $\text{var}_{\text{size}}: E_d^{f^A*} \to \mathbb{N}$ is a function that returns the size in bytes of the data type associated with the variable that caused the data flow edge.[2]

For parallelism extraction, every function in a sequential application can be modeled as a CDFG (or similar graph representations). An example of a CDFG is shown in Fig. 2.6b for the code snippet in Fig. 2.6a. For clarity, the data edges in the CDFG are annotated with the variable that originates the dependency instead of the variable size. Notice that the value of variable `c` in Line `4` could have been defined in Line `1`, Line `6` or Line `9`. That is the reason for the three incoming edges to BB_2 labeled with variable `c`. A similar analysis can be performed for the rest of the edges in the figure.

For the C language, the problem of determining all data dependencies statically at compile time is NP-complete and in some cases undecidable. This is due to the use of pointers [111] and indexes to data structures that can only be resolved at runtime. For this reason, dependencies are sometimes tracked at runtime to get less conservative data dependency edges, allowing a more aggressive parallelism extraction. This kind of analysis is known as dynamic DFA and is employed in the MAPS framework.

2.4.1.4 Inter-procedural Analysis and Profiling

Until now, there is a fairly compact representation of functions within an application. However, parallelism extraction requires a wider analysis scope, often regarded as *Whole Program Analysis* (WPA). WPA is usually enabled by means of a *Call Graph* (CG) that expresses how functions are called to make up an application. In practice,

[2] In this book, it is assumed that the size of data types does not depend on the processor type the code is compiled to run on. In reality, the actual size depends on the individual target compilers.

WPA is sometimes restricted due to code visibility for the compiler, i.e., library functions or functions defined in different compilation units. Here, it is assumed that the whole program is visible to the compiler.

In addition to widening the scope of the analysis, parallelism extraction requires profiling information. Note that the total time spent in a basic block depends not only on its IR-statements, but also on the amount of times the basic block was executed. Similarly, the total amount of data that has to be transmitted due to a data edge, depends not only on the data type, but also on the amount of times the data was actually produced and used. As an example, consider the information annotated to the edges in the example in Fig. 2.4b. The CG and the profiling concepts are formalized in the following.

Definition 26 A **call graph** (CG^A) of an application $A \in \mathscr{A}^{\text{seq}}$ is a directed multigraph $CG^A = (S^A_{\text{f}}, E^A_{\text{cg}}, S^A_{\text{stmt}}, \sigma^A)$. E^A_{cg} is a multiset of edges over $S^A_{\text{f}} \times S^A_{\text{f}}$, where $e_{ij} = (f_i^A, f_j^A) \in E^A_{\text{cg}}$ expresses that function f_i^A calls f_j^A. S^A_{stmt} is the set of all statements in the application (recall Definition 19). Finally, the function σ^A associates every edge with a *call site*, i.e., the exact statement that calls the function. It is therefore defined as $\sigma^A \colon E^A_{\text{cg}} \to S^A_{\text{stmt}}$. Naturally, $\sigma^A(e_{ij}) \in S^{f_i^A}_{\text{stmt}}$.

Definition 27 A **sequential application element set** ($\mathscr{SE}^A$) is a set that contains all *elements* of a sequential application. An application element can be a statement, a basic block, a control edge, a data edge or a function. Let $E^A_c = \bigcup_{f^A \in S^A_{\text{f}}} E_c^{f^A}$ and $E^A_d = \bigcup_{f^A \in S^A_{\text{f}}} E_d^{f^A *}$ be the sets of all control and data edges in the application. The set of all elements is defined as $\mathscr{SE}^A = S^A_{\text{stmt}} \cup \mathscr{BB}^A \cup E^A_c \cup E^A_d \cup S^A_{\text{f}} \cup E^A_{\text{cg}}$.

Definition 28 A **sequential profile** (π^A) of an application $A \in \mathscr{A}^{\text{seq}}$ is a function $\pi^A \colon \mathscr{SE}^A \to \mathbb{N}$ that returns the amount of times a given application element was executed during a profiling run.

Recall the general definition of an application $A = (\mathscr{M}^A, V^A, \mathscr{K}^A)$ in Definition 6. With the definitions presented in this section, it is now possible to formally define the application model used in the sequential code problem.

Definition 29 A **sequential application model** ($\mathscr{M}^A$) for an application $A \in \mathscr{A}^{\text{seq}}$ is a triple $\mathscr{M}^A = (\mathscr{SE}^A, CG^A, \pi^A)$. Let the notation $e \in \mathscr{M}^A$ refer to $e \in \mathscr{SE}^A$.

Recall the source code performance estimation functions in Definition 11, $\zeta_j^{PT_i} \colon S \subseteq \mathscr{M}^A \to \mathbb{N}$ for processor type PT_i. It is now possible to refine this general definition into $\zeta_j^{PT_i} \colon S^A_{\text{stmt}} \cup \mathscr{BB}^A \cup S^A_{\text{f}} \to \mathbb{N}$, where the index j refers to the different means for obtaining the estimate (see Sect. 2.2.2). In particular, the table-based performance estimation can be represented by a function $\zeta_{\text{tb}}^{PT_i} \colon S^A_{\text{stmt}} \to \mathbb{N}$. The *static cost* of a basic block BB^A can be then computed as $\zeta_{\text{tb}}^{PT_i,\text{st}}(BB^A) = \sum_{s^A \in BB^A} \zeta_{\text{tb}}^{PT_i}(s^A)$. By using the profiling information, it is also possible to compute the *dynamic cost* of a basic block BB^A as $\zeta_{\text{tb}}^{PT_i,\text{dy}}(BB^A) = \pi^A(BB^A) \cdot$

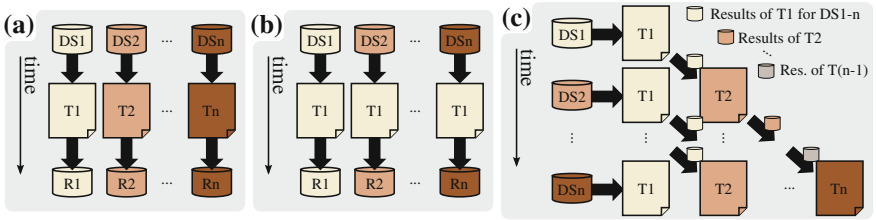

Fig. 2.7 Forms of parallelism. Different tasks (T1, T2,…Tn), their corresponding input data set (DS) and output results (R) are represented with different colors. **a** TLP. **b** DLP. **c** PLP

$\zeta_{\text{tb}}^{PT_i,\text{st}}(BB^A)$. Similarly, the estimation of the total time spent in a function f^A, can be computed as $\zeta_{\text{tb}}^{PT_i,\text{dy}}(f^A) = \sum_{BB^A \in \mathscr{B}\mathscr{B}^{f^A}} \zeta_{\text{tb}}^{PT_i,\text{dy}}(BB^A)$. The average cost, per function call, can be also easily computed as $\zeta_{\text{tb}}^{PT_i,\text{dy}}(f^A)/\pi^A(f^A)$. With this simple model, it is possible to obtain the execution times that were assumed given in the example in Fig. 2.4c.

2.4.1.5 Forms of Parallelism

The sequential application model provides the information that is needed to extract parallelism. There are different kinds of parallelism that can be hidden in an application. While a traditional compiler focuses on exploiting *Instruction Level Parallelism* (ILP), parallelism extraction for MPSoCs happens at a coarser level. The most prominent types of coarse parallelism are illustrated in Fig. 2.7 and are described in the following.

- *Task (or functional) Level Parallelism* (TLP): In TLP, a computation is divided into multiple tasks that operate in parallel on different *data sets*, as illustrated in Fig. 2.7a. Tasks may have dependencies to each other, but once a task has its data ready, it can execute in parallel with the already running tasks in the system. The DAG example in Fig. 2.2 exposes TLP. This kind of parallelism is a form of *Multiple Instruction Multiple Data* (MIMD) [87].
- *Data Level Parallelism* (DLP): In DLP, a computation is replicated into several equal tasks that operate in parallel on different data sets, as illustrated in Fig. 2.7b. This kind of parallelism can be seen as a generalization of SIMD [87]. DLP can be found in multimedia applications, where a decoding task performs the same operations on different portions of an image or video (e.g., *macro blocks* in H.264).
- *Pipeline Level Parallelism* (PLP): In PLP, a computation is broken into a sequence of tasks which are repeated for different data sets as shown in Fig. 2.7c. This kind of parallelism can be seen as a generalization of software pipelining [159]. PLP can be seen as a mixture of DLP and TLP. Every diagonal sequence of tasks in Fig. 2.7c perform the same computation (i.e., T1, ... Tn) on different data sets

(DLP). Additionally, at a given time (horizontal cut in Fig. 2.7c), different tasks operate on different data sets coming from the previous *pipeline stage* (TLP).

A parallelism extractor should be aware of the different forms of coarse-grained parallelism. It should decide which parallelism form or which combination of parallel patterns to exploit. Consider the motivational example in Fig. 2.4. The sequential application can be divided into two tasks, the first one containing the functions `f1` and `f2`, and the second one the functions `f3`, `f4` and `sum`. These two tasks are an example of a TLP partition of the original application. The second task can be partitioned as well, with two possibilities shown in Fig. 2.4d, being both an example of DLP.

2.4.1.6 Parallelism Extraction

Given a sequential application model $\mathscr{M}^A = (\mathscr{SE}^A, CG^A, \pi^A)$ where functions are modeled as graphs (e.g., $CDFG^{f^A}$), parallelism extraction is usually implemented by means of *graph clustering* or *graph partitioning*. A good introduction to the graph clustering terminology and a survey of graph clustering methods can be found in [230]. Two good examples of graph clustering algorithms are *METIS* [158] and *MCL* [69]. Fundamental data clustering techniques are surveyed in [127]. This section only focuses on the definitions needed for the problem statement.

Definition 30 A **clustering of a graph** $G = (V, E)$ is a collection of non-empty sets $\mathscr{C}^G = \{C_1, \ldots, C_k\}$ such that $\bigcup_{i=1}^{k} C_i = V$. An element $C_i \in \mathscr{C}^G$ is called a cluster. Each subset $C_i \in \mathscr{C}^G$ has a corresponding *induced subgraph* $G_i = (C_i, E_i)$, with $E_i = \{e = (u, v) \in E, u, v \in C_i\}$.

Definition 31 A **partition of a graph** $G = (V, E)$ is a clustering $\mathscr{C}^G = \{C_1, \ldots, C_k\}$ where the sets $C_i \in \mathscr{C}^G$ are pairwise disjoint, i.e., $\forall i, j, i \neq j, (C_i \cap C_j = \emptyset)$.

Definition 32 A **hierarchical clustering/partition of a graph** $G = (V, E)$ is a finite ordered list of clusterings/partitions $(\mathscr{C}_1^G, \ldots, \mathscr{C}_l^G)$ where each $\mathscr{C}_{i+1}^G, 1 \leq i < l$ can be obtained by joining elements of $\mathscr{C}_i^G$.

Definition 33 A **partitioned graph** of a graph $G = (V, E)$ and partition $\mathscr{C}^G$ is a graph $G' = (V', E')$. There is a node $v_i' \in V'$ for every set $C_i \in \mathscr{C}^G$, and there is an edge $e_{ij}' = (v_i', v_j') \in E'$ if there was an edge $e_{ij} = (v_i, v_j) \in E$, with $v_i \in C_i$ and $v_j \in C_j$. A partitioned graph is also regarded as the induced graph of a partition.

A partitioned graph explicitly exposes TLP within a function. Notice however, that other forms of parallelism are not explicit. In order to expose PLP and DLP, the graph has to include additional information about the nature of the cluster. For example, how many versions of the cluster can be started in parallel in the case of DLP. Similarly, a sequence of clusters may expose PLP, which has to be additionally annotated. These parallelism patterns are added to the graph as annotations.

Definition 34 A **parallel annotation** (PA^C) of a cluster C in a hierarchical, partitioned graph contains information about the different forms of parallelism that can be exploited. It is generally defined as a triple $PA^C = (\{\text{DLP, PLP}\}, X^{PA^C}, V^{PA^C})$. The first element distinguishes between DLP and PLP. Note that TLP is not annotated, since it is expressed by default. X^{PA^C} is a set of attributes associated with the annotation. V^{PA^C} is a set of variables $\{v_1, \ldots, v_k\}$ associated with the annotation, where a variable v_i can take a value within a domain $D_{v_i}^{PA^C}$.

The definition of a parallel annotation becomes clearer if the two cases (DLP, PLP) are analyzed separately. In the case of DLP, the attribute set is empty, while the variable set contains a single variable that specifies how many copies of the task are to be generated. The domain of this variable is $\{1, 2, \ldots, m\} \subset \mathbb{N}$, where m is determined by the maximum amount of data parallel tasks. This can be bounded by the maximum *trip count* of the loop that originated the task, i.e., the maximum number of iterations of the loop.

In the case of PLP, the attribute set contains a collection of clusters from the previous partition in the hierarchy, $X^{PA^C} = \{X_1\}$. As an example, consider a cluster from a graph partition $\mathscr{C}_{i+1}^G$ that consist of m clusters of the previous partition hierarchy $\{C_1, \ldots, C_m\} \subseteq \mathscr{C}_i^G$. The PLP attribute would be the set $X_1 = \{C_1, \ldots, C_m\}$, which indicates the partitions C_i the pipeline can be composed from. Apart from the attribute, two variables are defined for a PLP annotation $V^{PA^C} = \{v_1, v_2\}$. The first one determines how many pipeline stages are to be implemented. This variable can take a value that is bounded by the amount of partitions, i.e., $D_{v_1}^{PA^C} = \{1, \ldots, m\} \subset \mathbb{N}$, with $m = |X_1|$. The second variable is a function that determines which partitions are to be mapped to which pipeline stages, i.e., $v_2 \colon X_1 \to D_{v_1}^{PA^C}$. The domain of this variable is the set of all possible assignments of clusters to pipeline stages, i.e., $D_{v_2}^{PA^C} = X_1 \times D_{v_1}^{PA^C}$ for a given value of v_1.

Definition 35 A **parallel-annotated graph** is a hierarchical, partitioned graph $G' = (V', E', \mathscr{PA}^{V'})$, where $\mathscr{PA}^{V'}$ is a set of parallel annotations associated with the nodes in V'. Since a data parallel loop can be sometimes implemented as a pipeline, there may be more than one annotation per node.

Definition 36 Parallel-annotated application model: ($\mathscr{M}_{\text{par}}^A$) Consider a sequential application model $\mathscr{M}^A = (\mathscr{SE}^A, CG^A, \pi^A)$ (in Definition 29) in which functions are modeled as graphs (e.g., $CDFG^{f^A}$). A parallel-annotated application model contains a parallel-annotated graph for every function in CG^A. It is represented as $\mathscr{M}_{\text{par}}^A = (\mathscr{SE}^A, CG_{\text{par}}^A, \pi^A)$.

The model with parallel annotations ($\mathscr{M}_{\text{par}}^A$) explicitly exposes all kinds of parallelism available in the application. If the application is to be run on a parallel platform, the model has to be refined. That is, the parallelism settings must be fixed. For example, if a cluster exposes DLP, the amount of data parallel tasks must be set.

In the example in Fig. 2.4d two possible DLP configurations are shown, for 2 and 4 tasks to execute `f3` and `f4`. This is captured by a *parallel implementation option*.

Definition 37 A **parallel implementation option** ($\mathscr{PI}^A$) of a parallel-annotated application model $\mathscr{M}^A_{\text{par}} = (\mathscr{SE}^A, CG^A_{\text{par}}, \pi^A)$ is the result of (1) selecting a single clustering result from the hierarchical clustering ($\mathscr{C}_i^{CDFG^{f^A}}$ for all $f^A \in S^A_{\text{f}}$), (2) selecting only one parallel annotation for nodes containing more than one and (3) assigning values to the variables of the selected parallel annotations. It is represented as $\mathscr{PI}^A = (\mathscr{SE}^A, CG^A_{\text{pi}}, \pi^A)$.

Definition 38 A **suitable parallel implementation** for a target platform model $SOC = (\mathscr{PE}, \mathscr{E})$ is a parallel implementation that exposes all *relevant* parallelism for a later mapping phase, i.e., it only contains parallel tasks that produce a considerable speedup without wasting resources.

As an example consider the application analysis in Fig. 2.4. Intuitively, it is clear that the parallelism provided by the upper configuration in Fig. 2.4d is not *relevant*, since the whole execution is already lower-bounded by the runtime of the functions `f1` and `f2`. How to define *relevant* parallelism to obtain a *suitable* parallel implementation is not further formalized in this chapter. Initial works on parallelism extraction focused on optimizing a single graph property, e.g., the *ratio cut* [278] or a given similarity measure [48, 230]. However, a single optimization criterion is not general enough to produce good results in such a complex problem. The subjective criteria used to determine a suitable implementation are built in the heuristics described in Chap. 5.

2.4.2 Problem Statement

The definitions from the previous sections now allow to define the sequential problem as:

Definition 39 Sequential code problem: Given a sequential application $A \in \mathscr{A}^{\text{seq}}$ specified using the C language and a target platform model $SOC = (\mathscr{PE}, \mathscr{E})$, find a suitable parallel implementation $\mathscr{PI}^A = (\mathscr{SE}^A, CG^A_{\text{pi}}, \pi^A)$.

The problem is solved in three steps. (1) First, the sequential application model $\mathscr{M}^A = (\mathscr{SE}^A, CG^A, \pi^A)$ is built. (2) Then, it is analyzed to obtain the parallel-annotated model $\mathscr{M}^A_{\text{par}} = (\mathscr{SE}^A, CG^A_{\text{par}}, \pi^A)$. (3) Finally, from all possible options allowed in $\mathscr{M}^A_{\text{par}}$ the most suitable parallel implementation is selected.

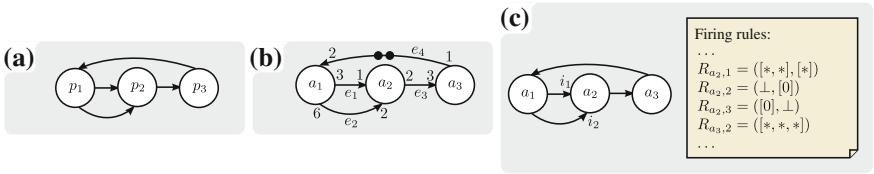

Fig. 2.8 Example of three different concurrent MoCs. **a** KPN model. **b** SDF model. **c** DDF model

2.5 Parallel Code Problem

The parallel code problem deals with finding a valid runtime configuration (see Definition 16) for a parallel application ($A \in \mathscr{A}^{\text{kpn}}$) within the multi-application specification or coming from a parallelism extraction process. There are many different ways of specifying parallel applications. MAPS uses the CPN language which allows to represent applications as a mixture of MoCs. Therefore, this section first provides an overview of process networks and dataflow models in Sect. 2.5.1. Thereafter, the CPN language is briefly introduced in Sect. 2.5.2. This book focuses on mapping techniques for applications that follow the KPN model. New definitions and refinements of the definitions in Sect. 2.3.1 are presented in Sect. 2.5.3. Finally, the parallel code problem is stated in Sect. 2.5.4.

2.5.1 Process Networks and Dataflow Models

Parallel programming models based on concurrent MoCs have been gaining momentum in the embedded domain, especially for describing signal processing applications. The MoC theory originated from theoretical computer science, with the goal of formally describing a computing system. It was initially used to compute bounds on complexity (the "big O" notation —$O(\cdot)$). In the early 80s, MoCs were used to model VLSI circuits. Later in the 90s, they started to be applied for modeling parallel applications. The formalism behind some MoCs makes it possible to analyze application properties (timing, memory consumption). This made such programming models so attractive for embedded software.

A big collection of concurrent MoCs have been proposed for embedded programming. The most prominent ones are *Synchronous Dataflow* (SDF) [166], *Cyclo-Static Dataflow* (CSDF) [28], *Boolean Dataflow* (BDF) [164], *Dynamic Dataflow* (DDF) [33], *Process Networks* (PN) and *Kahn Process Networks* (KPN) [135]. These programming models have in common a directed graph representation of the application, in which nodes represent computation and edges represent logical communication channels. An example of such a graph is shown in Fig. 2.8a.

Apart from explicitly exposing parallelism, MoC-based programming models became attractive for two reasons. On the one hand, they are well suited for graphical

programming, a common specification paradigm for signal processing algorithms. On the other hand, some of the properties of the underlying MoC enable tools to perform analysis and compile the specification into both software and hardware.

Models differ from each other in their execution semantics, i.e., in the way the computation is triggered and how channels are accessed. In the following, some basic MoCs are defined and some of their characteristics are introduced. A more comprehensive treatment can be found in [26, 128, 163, 239].

2.5.1.1 Synchronous Dataflow (SDF)

Definition 40 An **SDF graph** is an annotated multigraph $G = (V, E, W)$, where nodes represent computation and edges represent queues of data items or *tokens*. Nodes are denoted $a \in V$ and are called *actors*. W is a set of annotations, $W = \{w_1, \ldots, w_{|E|}\} \subset \mathbb{N}^3$. An annotation is a triple of integers for every edge, $w_e = (p_e, c_e, d_e)$. Given a channel $e = (a_1, a_2) \in E$, p_e represents the number of tokens produced by every execution of actor a_1, c_e represents the number of tokens consumed in every execution of actor a_2 and d_e represents the number of initial tokens present on edge e. Initial tokens are also called *delays* and represent data dependencies over different iterations of the graph. SDF is also regarded as *Multi-Rate Dataflow* (MRDF).

Definition 41 A **Homogeneous SDF** (HSDF) is an SDF graph $G = (V, E, W)$ with unit token production and consumption rates, i.e., $\forall e \in E, w_e = (1, 1, d_e)$. Every SDF can be turned into an HSDF [239]. HSDFs are also regarded as *Single-Rate Dataflow* (SRDF).

The execution of an SDF is controlled by data in its edges. Every actor is executed or *fired* once it has enough tokens in its input channels. The amount of tokens needed for an actor to fire is determined by the second component of the edge annotation (c_e).

It has been shown that it is possible to compute a static schedule for an SDF graph [166]. Two major scheduling approaches can be identified: *blocked* and *non-blocked*. In the former, a schedule for one cycle is computed and repeated without overlapping, whereas in the latter, the executions of different iterations of the graph are allowed to overlap. For computing a blocked schedule, a *complete cycle* in the SDF has to be determined. A complete cycle is a sequence of actor firings that brings the SDF to its initial state. Finding a complete cycle requires that (1) enough initial tokens are provided in the edges and (2) there is a non trivial solution for the system of equations $\Gamma \cdot \mathbf{r} = 0$. Γ is called the *topology matrix* of the SDF and is defined as $[\Gamma_{ij}] = p_{ij} - c_{ij}$, where p_{ij} and c_{ij} are the number of tokens that actor j produces to and consumes from channel i respectively. The vector $\mathbf{r}$ is called *repetition vector*.

An example of an SDF graph is shown in Fig. 2.8b with delays represented as dots on the edges (see edge e_4). According to the SDF execution model, once there are 2 tokens on e_4, a_1 will fire, producing 3 tokens to e_1 and 6 to e_2. For the SDF in the example, $W = \{(3, 1, 0), (6, 2, 0), (2, 3, 0), (1, 2, 2)\}$, its topology matrix is:

$$\Gamma = \begin{pmatrix} 3 & -1 & 0 \\ 6 & -2 & 0 \\ 0 & 2 & -3 \\ -2 & 0 & 1 \end{pmatrix}$$

and a repetition vector is $\mathbf{r} = [1\ 3\ 2]^T$, meaning that after 1 execution of actor a_1, 3 executions of actor a_2 and 2 executions of actor a_3 the SDF graph returns to its original state. By *unfolding* the SDF according to its repetition vector and removing the feedback edges (those with delay tokens) one obtains a DAG [239]. Using this procedure, the problem of mapping an SDF is turned into a DAG problem (see Sect. 2.2.1).

2.5.1.2 Dynamic Dataflow

Definition 42 A **DDF graph** is a directed multigraph $G = (V, E, \mathscr{R})$, with $\mathscr{R} = \{R_{a_1}, \ldots, R_{a_{|V|}}\}$ a family of sets, one set for every node $a \in V$. Edges have the same semantics as in the SDF model. Actors, in turn, have a more complex firing semantics determined by a set of *firing rules* in $\mathscr{R}$. Every actor $a \in V$ has a set of firing rules $R_a \in \mathscr{R}$, $R_a = \{R_{a,1}, \ldots, R_{a,r}\}$. A firing rule for an actor a with p inputs is a p-tuple $R_{a,i} = (c_1, \ldots, c_p)$ of conditions, which describe a sequence of tokens that has to be available at the given input queue. Parks introduced a notation for such conditions in [210]. The condition $[X_1, X_2, \ldots, X_n]$ requires n tokens with values $X_1, X_2, \ldots, X_n$ to be available at the top of the input queue. The conditions $[*]$, $[*, *]$, $[*_{(1)}, \ldots, *_{(m)}]$ require at least 1, 2 and m tokens respectively with arbitrary values to be available at the input. The symbol $\perp$ represents any input sequence, including an empty queue. For an actor a to be in the ready state at least one of its firing rules need to be satisfied.

An example of a DDF graph is shown in Fig. 2.8c. In this example, the actor a_2 has 3 different firing rules. This actor is ready if there are at least two tokens in input i_1 and at least 1 token in input i_2 ($R_{a_2,1}$), or if the next token on input i_2 or i_1 has value 0 ($R_{a_2,2}$, $R_{a_2,3}$). Notice that more than one firing rule can be activated, in this case the dataflow graph is said to be non-determinate.

Note that an SDF can be seen as a simplification of a DDF model,[3] in which an actor with p inputs has only one firing rule of the form $R_{a,1} = (n_1, \ldots, n_p)$ with $n \in \mathbb{N}$. Additionally, the amount of tokens produced by one execution of an actor on every output is also fixed.

[3] Being more closely related to the so-called *Computation Graphs* [139].

2.5.1.3 Kahn Process Networks

Definition 43 A **KPN** is a directed multigraph $G = (V, E)$. Computational nodes in V are called *processes* and edges represent infinite token queues or First-In First-Out (FIFO) buffers. Processes do not feature the firing semantics of actors in dataflow graphs. Instead, they are allowed to be in one of two states: ready or blocked. The blocked state can only be reached by reading from *only one* empty input channel, commonly denoted as *blocking reads* semantics. A KPN is said to be determinate, i.e., the history of tokens produced on the communication channels does not depend on the scheduling [135]. Formally, a process is modeled as a functional mapping from input *streams* to output *streams*, where streams are possibly infinite sequences of data elements [135, 210].

An example of a KPN is shown in Fig. 2.8a. Note that the graph does not describe how processes access channels, i.e., how processes consume and produce tokens to the channels like SDF graphs do. Each process has its own control flow and may produce and consume data with arbitrary patterns that may be controlled by incoming data. This makes KPNs flexible enough to represent a wider range of applications, but at the same time, makes it difficult to analyze them. In fact, for general KPNs it is not possible to compute a static schedule, so that they are scheduled dynamically.

There are two major ways of dynamically scheduling a KPN: *data* and *demand* driven. In data-driven scheduling, every process in the KPN is moved to the ready state once sufficient data is available at its inputs. A dynamic scheduler then decides which process gets executed on which processor at runtime. A demand-driven scheduler first schedules processes with no output channels. This processes execute until a read blocks in one of the input channels. The scheduler then triggers only the processes from which data has been requested (*demanded*). This process continues recursively.

Data-driven scheduling exploits the available parallelism in the KPN, but can lead to unbounded accumulation of tokens in the channels. This is the case for processes with no input channels which would be always in the ready sate. Unbounded token accumulation can be avoided by setting an upper limit on the FIFO channels and implementing *blocking writes* semantics, i.e., a process would not only block on a read from an empty channel, but also on a write to a full channel. The process of determining the sizes of the FIFO queues in a KPN is known as *buffer sizing*. Introducing blocking writes to the KPN model can lead to *artificial deadlocks*, i.e., deadlocks that would not occur in the case of unbounded channels. Demand-driven scheduling, in turn, produces only the tokens that are needed. As a consequence, there is no unbounded accumulation of tokens in the channels. However, due to the way processes are activated, demand-driven produces a sequential execution of the network and introduces a high context switching overhead. In practice, hybrid data and demand-driven schedulers are used to schedule KPNs.

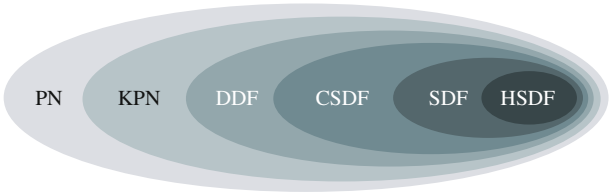

Fig. 2.9 Graphical representation of the expressiveness of different MoCs. A graph in an inner model can also be expressed by a graph in an outer model

2.5.1.4 Comparison of MoCs

When selecting an underlying MoC for an application specification, an implicit trade-off between *expressiveness* and *analyzability* is made. Static models (e.g., SDF) are more amenable to design-time analysis. For example, the questions of *termination* and *boundedness*, i.e., whether an application runs forever with bounded memory consumption, are decidable for SDFs but undecidable for BDFs, DDFs and KPNs [210]. Static models are however not general enough to represent applications with data-dependent communication patterns. With dynamic models (e.g., DDF or KPN) this kind of behavior can be modeled. As a consequence of this higher expressiveness, it is more difficult to reason at design time about possible runtime configurations (mapping, scheduling and buffer sizes).

Besides analyzability and expressiveness, the complexity of specifying an application using a MoC varies. This can be measured by the amount of information a programmer has to provide in the specification. In the examples in Fig. 2.8 it is clear that the KPN MoC displays the lowest specification effort and DDF the highest.

The expressiveness of some MoCs is shown graphically in Fig. 2.9. In addition to the models introduced in the previous section, the figure includes CSDF and PN. CSDF is an extension to SDF that allows to model several, predefined, static, cyclic behaviors [28]. PNs are KPNs without blocking reads semantics and are therefore more general. The graphical representation in Fig. 2.9 shows that an application represented in an inner model can also be represented in an outer model. Every HSDF is also an SDF, every SDF is also a CSDF and so forth. It is arguable whether or not DDF applications are a subset of KPN applications, since they can be non-determinate as discussed in Sect. 2.5.1.2. The figure refers to determinate DDFs.

The work presented in this book deals with parallel applications represented as KPNs. The motivation for addressing KPNs is twofold. (1) A simpler specification increases the acceptability of the language. At the same time, the software productivity is improved by a simpler language. (2) A more expressive model allows to represent modern, dynamic applications, thereby making the solutions applicable to a wider range of problems.

2.5.2 The C for Process Networks Language

The MAPS CPN language is an extension to the C programming language that consists of a small set of keywords that allow to describe KPN processes, SDF actors and FIFO communication channels. Listing 2.1 shows an example of a process with one input channel and one output channel that decodes a *Run-Length Encoded* (RLE) sequence of integers (see Lines 1–9). Channels are declared with the `__PNchannel` keyword and can be of any type, including user-defined structures (see Line 10). The actual definition of the process (in Line 1) is preceded by `__PNkpn` and followed by the declaration of input and output channels. The keywords `__PNin` and `__PNout` within a process are used to mark portions of code in which channels are accessed. Within these scopes, every access to the channel variable will read/write the same position in the FIFO buffer. In the example, the accesses to channel `A` in Line 4 will read values of different positions. Instead, all accesses to channel `A` in Line 8 within an execution of the for loop will return the same value. Given an input stream $\{2, 1, 3, 5, \ldots\}$, the output of the process would be $\{1, 1, 5, 5, 5 \ldots\}$. Finally, the code in Lines 11–12 shows how to instantiate processes and connect them with channels.

```
1  __PNkpn rle_dec __PNin(int A) __PNout(int B) {
2   int cnt, i;
3   while (1) {
4    __PNin(A) { cnt = A; }
5
6    __PNin(A) {
7     for (i = 0; i < cnt; ++i)
8      __PNout(B) { B = A; }
9  }}}
10 __PNchannel int src, dec;
11 __PNprocess src = Source __PNout(src); // Defined elsewhere
12 __PNprocess rle_dec = rle_dec __PNin(src) __PNout(dec);
```

Listing 2.1 Sample code for RLE decoding

2.5.3 KPN Mapping and Scheduling

As mentioned above, computing a mapping of a KPN application is a challenging task. The semantics of the KPN graph representation are much more involved than those of the initial DAG example in Fig. 2.2. Note that the DAG problem is equivalent to the problem of mapping a single iteration of an acyclic HSDF. For both problems, only the timing information is relevant for the mapping process (if the communication cost is ignored). In a KPN, instead, more information about the processes is required. A process cannot be considered a *black-box* like in the case of HSDF, but a model of its internal behavior is needed (e.g., its internal CFG).

Consider the sample KPN application shown in Fig. 2.10a together with the CFGs of its processes. Read and write accesses to channels are represented by the functions `read(...)` and `write(...)`. The channel that is accessed is represented by variables `c1`–`c4` (for channels C_1–C_4). The actual data containers are omitted from the function signature for the sake of clarity. Actual processing is represented by

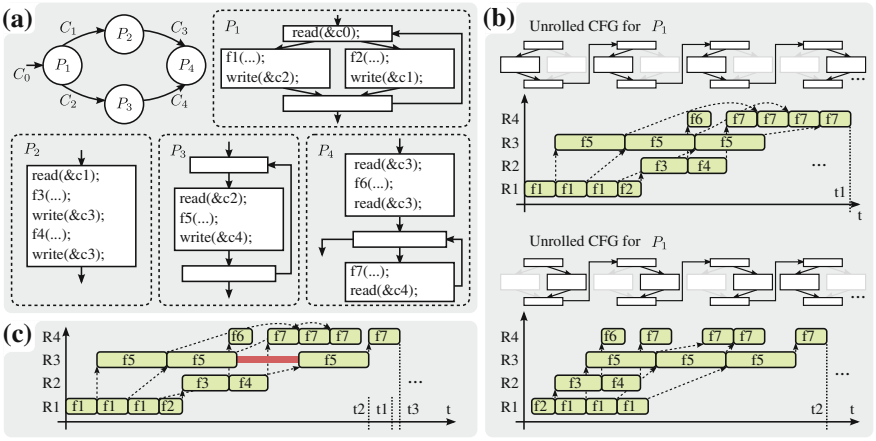

Fig. 2.10 KPN application mapping. **a** KPN application and CFGs for each process. **b** Possible schedules for hypothetical runs of P_1. **c** Example of the effect of buffer sizing for channel $C4$

functions `f1()`–`f7()`. Figure 2.10b shows two possible schedules for two different hypothetical control paths for process P_1 (on top of each Gantt Chart). For simplicity, the example supposes a single processor type, negligible communication costs and a one-to-one mapping of processes to PEs (P_1 mapped to PE R1, P_2 to R2 and so on).

In a real implementation, the theoretically infinite FIFO queues have to be bounded. As a result, processes feature blocking writes semantics in addition to the blocking read semantics, as discussed in Sect. 2.5.1.3. The effect of buffer sizing for this example is shown in Fig. 2.10c for the upper CFG of process P_1 from Fig. 2.10b. All buffer sizes are assumed to be set to one. As a result, process P_3 blocks when writing for the second time to channel C_4 since process P_4 has not yet read the first token. This introduces a blocking time in the third PE (see the red, thinner bar). The Gantt Chart in Fig. 2.10c contains the makespan for the three configurations for comparison purposes. Also note, that this buffer sizing scheme would have no effect in the lower configuration in Fig. 2.10b.

The example in Fig. 2.10 demonstrates that even with a simplified setup, the control paths followed by a process in a KPN greatly influence the application makespan. It also shows that buffer sizing can influence the makespan as well. The remainder of this section formalizes the KPN model for the parallel code problem and refines some of the definitions presented in Sect. 2.3.1.

Note that a process is itself a sequential problem, thus the sequential application model in Definition 29 can be used to describe it.

Definition 44 A **process** (P^A) in a KPN application A is a triple $P^A = (\mathcal{SE}^{P^A}, CG^{P^A}, \pi^{P^A})$. As in Definition 29, $\mathcal{SE}^{P^A}$ is the set of all sequential elements of the process (e.g., basic blocks and functions), CG^{P^A} is the call graph of the process and π^{P^A} is the profiling function $\pi^{P^A}: \mathcal{SE}^{P^A} \rightarrow \mathbb{N}$. The set of all processes of an application is denoted $\mathcal{P}^A$.

The KPN model is an *untimed* MoC [128], which means that there is no explicit dependency among events (read or write) that happen on separate channels in the network. This makes it difficult to reason about timing constraints. To enable timing constraints in a KPN application, the concept of *time checkpoints* is added to processes.

Definition 45 A **time checkpoint** of a process P^A is a program point in its graph model (CG^{P^A}) that tells when to perform controlling actions due to timing constraints.

A time checkpoint can be seen as a function call in the source code, similar to the functions provided by Real-Time OSs (RTOSs), e.g., `waitForNextPeriod()` in real-time java [129]. A process may have several static calls to the time checkpoint function. The time elapsed between dynamic calls to this function during application execution is used to check timing constraints (Definition 12). With time checkpoints enabling a real-time application specification, it is now possible to refine the definition of the KPN graph used in this book.

Definition 46 A **KPN application model** (KPN^A) is an annotated KPN graph $KPN^A = (\mathscr{P}^A, \mathscr{C}^A, \text{var}_{\text{size}})$. $\mathscr{P}^A$ is the set of processes (see Definition 44), some of which may be extended with time checkpoints. $\mathscr{C}^A$ is a multiset of FIFO channels over $\mathscr{P}^A \times \mathscr{P}^A$. Similarly to Definition 25 for a CDFG, $\text{var}_{\text{size}}: \mathscr{C}^A \rightarrow \mathbb{N}$ is a function that returns the size in bytes of the data token associated with a FIFO channel.

Definition 47 A **parallel application element set** ($\mathscr{PAE}^A$) is a set that contains all the *KPN elements* of a parallel application. A KPN element can be a process or a channel, i.e., $\mathscr{PAE}^A = \mathscr{P}^A \cup \mathscr{C}^A$.

Recall the general definition of an application $A = (\mathscr{M}^A, V^A, \mathscr{K}^A)$ in Definition 6. With the definitions presented in this section, it is now possible to formalize the application model used in the parallel code problem.

Definition 48 A **parallel application model** ($\mathscr{M}^A$) for an application $A \in \mathscr{A}^{\text{kpn}}$ is a pair $\mathscr{M}^A = (\mathscr{PAE}^A, KPN^A = (\mathscr{P}^A, \mathscr{C}^A, \text{var}_{\text{size}}))$. A particular set of application variables in V^A determine the amount of memory allocated for each buffer. The variable for the size of buffer $C^A \in \mathscr{C}^A$ is denoted $b^A_{C^A}$, with $\forall C^A \in \mathscr{C}^A$, $b^A_{C^A} \in V^A$. Let the set of all channel size variables be $V^A_{\text{size}} \subset V^A$. Let also the notation $e \in \mathscr{M}^A$ refer to $e \in \mathscr{PAE}^A$.

2.5.4 Problem Statement

With the new definitions the parallel problem can be stated as:

Definition 49 **Parallel code problem:** Given a parallel application $A \in \mathscr{A}^{\text{kpn}}$, $A = (\mathscr{M}^A, V^A, \mathscr{K}^A)$ with its behavior specified using the CPN language with underlying

KPN model $\mathscr{M}^A = (\mathscr{PAE}^A, KPN^A = (\mathscr{P}^A, \mathscr{C}^A, \text{var}_{\text{size}}))$ and target platform $SOC = (\mathscr{PE}, \mathscr{E})$, find an optimal valid runtime configuration $RC^A = (\mu_p, \mu_c, \mu_a)$ (see Definition 16).

Recall the general Definition 15, $RC^A = (\mu_p, \mu_c, \mu_a)$ for a $SOC = (\mathscr{PE}, \mathscr{E})$ with communication primitives $\mathscr{CP}$. For a KPN application, this reduces to $\mu_p: \mathscr{P}^A \subset \mathscr{M}^A \rightarrow \mathscr{PE}$ and $\mu_c: \mathscr{C}^A \subset \mathscr{M}^A \rightarrow \mathscr{CP}$. The function μ_a represents the mapping of platform and application variables, as mentioned in Definition 15. The result of the buffer sizing process can be therefore seen as part of the function μ_a. Let $\beta: V^A_{\text{size}} \rightarrow \mathbb{N}$ represent the result of the buffer sizing process, i.e., $\beta(b^A_{C^A})$ is the amount of tokens allocated to channel C^A. For convenience, the alternative representation $\beta: \mathscr{C}^A \rightarrow \mathbb{N}$ is also used.

2.6 Software Defined Radio Problem

The SDR problem deals with finding a valid runtime configuration for an SDR application ($A \in \mathscr{A}^{\text{sdr}}$) within the multi-application description. SDR applications are *waveforms* or transceiver specifications of a given radio standard. They typically feature more stringent constraints than the applications addressed by the parallel code problem. As a consequence, hardware acceleration and specialized library routines are commonly used in their implementations, as discussed in Sect. 1.1.1.

This book addresses SDR applications that describe the two bottom layers of wireless communication standards, namely the *physical* (PHY) and the *Medium Access Control* (MAC) layers [126]. They are modeled as KPN applications using the CPN language. Note however, that the pure software CPN specification does not directly support hardware acceleration. From a C specification of a process it is impossible to infer whether or not a process might be run by a hardware accelerator or might be already implemented as an assembly routine in one of the platform's PEs. SDR applications are therefore treated in a slightly different way than plain CPN applications, as will be discussed in this section.

The SDR problem can be seen as an extension of the parallel code problem, with two new main features. (1) Processes can be *marked* as special computational kernels, with a high-level algorithmic description that allows to seek for support in the target MPSoC. (2) The model of the MPSoC is extended to express which computational kernels are supported by hardware acceleration or by specialized library functions. These extensions were introduced and implemented within the context of the *Nucleus Project*, which is introduced in Sect. 2.6.1. Thereafter, the extensions are formalized in Sect. 2.6.2 and the SDR problem is finally stated in Sect. 2.6.3.

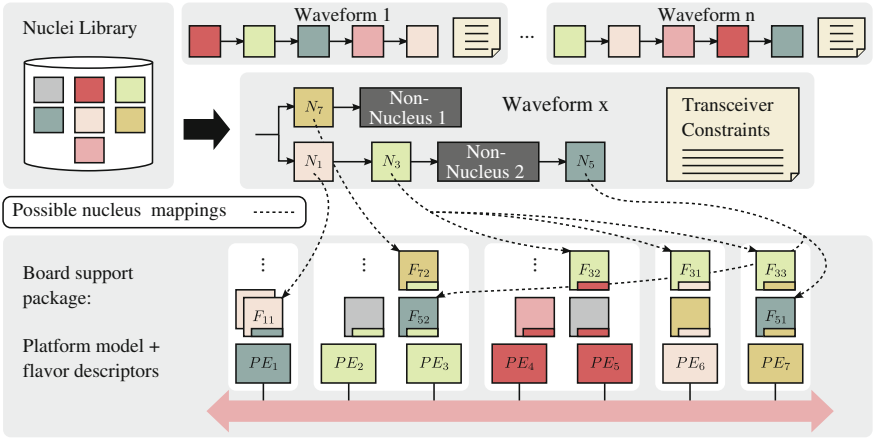

Fig. 2.11 The Nucleus waveform development concept (adapted from [222]). Different waveforms can be composed by using the nuclei in the library. Processor types are represented by different colors

2.6.1 The Nucleus Project

The Nucleus Project is a large scale project of the UMIC Excellence Cluster [271] that involves several chairs from different universities in a common effort to develop novel wireless communication algorithms (e.g., in [175, 299]), novel architectures (e.g., in [31, 292]) and novel tools and methodologies. The solution to the SDR problem presented in this book forms part of the Nucleus methodology.

The main concept behind the SDR tool flow was introduced in [222] and is illustrated in Fig. 2.11. It consists of a component-based methodology in which waveforms are composed out of library blocks called *nuclei*, which are platform-independent algorithmic entities that represent demanding computational kernels common to different wireless communication standards. Examples of such algorithms include, among others, matrix-vector operations, matrix-matrix operations, matrix factorizations, the Fast Fourier Transform (FFT), search algorithms on lists and trees. Nuclei are characterized by variables that serve to parameterize their underlying algorithm, e.g., size of data types and vector lengths. A nucleus can be implemented on different platforms in several ways. Each of these implementations is called a *flavor*. The specification of a flavor contains platform-dependent implementation details, e.g., the PE on which the flavor may run, its communication interface and its data representation. As an example, consider the platform-dependent linear algebra libraries used in HPC, e.g., the *Basic Linear Algebra Subprograms* (BLAS) [68] and the *Linear Algebra PACKage* (LAPACK) [6].

The SDR application mapping is computed by using the algorithmic information of the waveform contained in the nucleus specifications and the available flavors contained in the *Board Support Package* (BSP). This SDR mapping process, sketched

by dashed arrows in Fig. 2.11, is different from the mapping process considered so far. Instead of mapping processes to PEs, nuclei are mapped to flavors, where a single PE may contain different flavors for a single nucleus. In Fig. 2.11, nucleus N_1 has two implementations on PE_1 (F_{11} and F_{12}) and nucleus N_3 has three implementations on different PEs (F_{31} on PE_6, F_{32} on $PE_{4,5}$ and F_{33} on PE_7). As shown in the figure, some of the blocks in the application may not correspond to a nucleus (*non-nuclei* blocks). These blocks are treated by a traditional compilation approach.

In addition to determining which flavor to use, the SDR mapping process must consider several new constraints. For example, the hardware interfaces of two flavors that communicate with each other must be configured so that they match. The configuration may include adapting the data sizes and the synchronization approach.

The nucleus methodology, sketched in Fig. 2.11, is not far from that of commercial visual programming languages, e.g., Simulink [184] or LabView [200]. The novelty of the approach lies on a characterization of the flavors with both algorithmic and hardware knowledge, that allows to obtain good application performance while using a high-level transceiver description language. In this way, the final implementation does not suffer from the performance gap observed between C implementations of computationally intensive kernels and their corresponding optimized versions (as assembly routines for parallel architectures or as HW accelerators). Not seldom, this gap extends over orders of magnitude (see for example [143]), which makes traditional compilation approaches ill-suited.

2.6.2 Extensions to the Parallel Problem

This section formally defines the concepts that were intuitively presented in the previous section (nuclei and flavors) and adapt some of the previous definitions to match the new problem elements. Platform-independent computational kernels in the application are modeled as nuclei, defined as follows.

Definition 50 A **nucleus** (N^A) of an SDR application A is modeled as a 5-tuple $N^A = (P^{N^A}, V^{N^A}, \mathscr{K}^{N^A}, IN^{N^A}, OUT^{N^A})$. P^{N^A} is a process in the sense of Definition 44. This functional specification is used in case there is no suitable flavor in the target platform. V^{N^A} is a set of variables $V^{N^A} = \{v_1, \ldots, v_m\}$ associated with the underlying algorithm. Each variable $v \in V^{N^A}$ is defined over a domain $D_v^{N^A}$. $\mathscr{K}^{N^A}$ is a set of nucleus specific constraints imposed by the programmer. These constraints are defined on the nucleus variables, i.e., $K_i^{N^A} \in \mathscr{K}^{N^A}$, $K_i^{N^A} = (S_i \subseteq V^{N^A}, R_i)$. Finally, IN^{N^A} and OUT^{N^A} are sets of input and output ports. The set of all nuclei defined in an application is denoted $\mathscr{N}^A$, and the set of all nuclei defined in the library $\mathscr{N}$.

Nuclei variables are used to model parameterizable algorithms. As an example, an FFT algorithm could be parameterized by the number of points (v_1) and the number of bits used for the data representation (v_2). Once a nucleus is instantiated

in a waveform, the programmer may fix the value of a variable (e.g., $v_1 = 1024$ for a 1024-point FFT) or may restrict the domain of a variable (e.g., $v_2 \in \{8, 16\} \subset D_{v_2}^{N^A}$). The setting of variables is modeled by the nucleus-dependent constraint set $\mathscr{K}^{N^A}$. The sets of input and output ports were omitted in the process definition. For a nucleus, they are explicitly modeled (IN^{N^A} and OUT^{N^A}). This enables hardware interface considerations during the mapping process, since hardware interfaces are generally less flexible than software interfaces.

A flavor is a platform-dependent implementation of a nucleus. More formally,

Definition 51 A **flavor** (F^{SOC}) in a target platform SOC is modeled as a 7-tuple $F^{SOC} = (N, V^{F^{SOC}}, \mathscr{K}^{F^{SOC}}, IN^{F^{SOC}}, OUT^{F^{SOC}}, \mathscr{CM}^{F^{SOC}}, \mathscr{PE}^{F^{SOC}})$. The first component is the nucleus $N \in \mathscr{N}$ the flavor implements. $V^{F^{SOC}}$ is a set of flavor variables which contains all variables defined in the nucleus and additional implementation-dependent variables. Each variable $v \in V^{F^{SOC}}$ is defined over a domain $D_v^{F^{SOC}}$ on which constraints ($\mathscr{K}^{F^{SOC}}$) are defined. $IN^{F^{SOC}}$ and $OUT^{F^{SOC}}$ are the sets of input and output ports of the implementation. For every port in the associated nucleus, there has to be a port in the flavor. $\mathscr{CM}^{F^{SOC}}$ is the cost model of the flavor, represented as a set of functions on the flavor variables. Finally, $\mathscr{PE}^{F^{SOC}}$ denotes the set of PEs in the platform that contain this flavor. Naturally, $\mathscr{PE}^{F^{SOC}} \subseteq \mathscr{PE}$, with $SOC = (\mathscr{PE}, \mathscr{E})$. The set of all flavors in the target platform is denoted $\mathscr{F}^{SOC}$.

Implementation-dependent flavor variables allow to model additional algorithmic parameterization and hardware features. Additional algorithmic parameterization may include the parallelization degree or the memory stride. Hardware-related variables describe the way the flavor is interfaced with the rest of the system. This may include sizes of internal buffers, address ranges visible to the ports of the hardware accelerator, and the synchronization strategy used by the flavor.

The cost model of a flavor serves to compute different metrics of the implementation, such as area (in case of reconfigurable architectures), power and timing (latency and throughput). In this book the analysis is restricted to latency, i.e., the cost model replaces the software performance estimation functions from Definition 11. The latency cost function is defined as $\zeta^{F^{SOC}} : \times_{v \in V^{F^{SOC}}} D_v^{F^{SOC}} \rightarrow \mathbb{N}$. As an example, consider the implementation of an FFT, with variables that model the number of points (v_1), the number of bits used for the data representation (v_2) and the parallelism degree (v_3). Its latency is then modeled by an arbitrary function, e.g., $\zeta^{F^{SOC}}(v_1, v_2, v_3) = c \cdot v_1 \cdot v_2 / v_3$, with $c \in \mathbb{R}$.

Recall the parallel application model in Definition 48, $\mathscr{M}^A = (\mathscr{PAE}^A, KPN^A = (\mathscr{P}^A, \mathscr{C}^A, \text{var}_{\text{size}}))$. By adding nuclei to the parallel specification, it is now possible to formalize the application model used in the SDR problem.

Definition 52 An **SDR application model** ($\mathscr{M}^A$) for an application $A \in \mathscr{A}^{\text{sdr}}$ is a pair $\mathscr{M}^A = (\mathscr{PAE}^A, KPN^A = (\mathscr{P}^A \cup \mathscr{N}^A, \mathscr{C}^A, \text{var}_{\text{size}}))$. The underlying KPN

specification has two different types of nodes, and consequently, $\mathscr{C}^A = (\mathscr{P}^A \cup \mathscr{N}^A) \times (\mathscr{P}^A \cup \mathscr{N}^A)$. The set of parallel elements is also extended from Definition 48 and is now defined as $\mathscr{PAE}^A = \mathscr{P}^A \cup \mathscr{N}^A \cup \mathscr{C}^A$.

In order to map the SDR problem to the parallel code problem, a mapping from nuclei to flavors has to be performed. This includes an assignment of flavor variables to their respective domains. Only with fixed variable values it is possible to lower the specification into an executable description.

The following definitions hold for an SDR application $A \in \mathscr{A}^{\text{sdr}}$ with underlying model $KPN^A = (\mathscr{P}^A \cup \mathscr{N}^A, \mathscr{C}^A, \text{var}_{\text{size}})$ on a platform with flavor set $\mathscr{F}^{SOC}$.

Definition 53 A **nucleus mapping** (NC^A) is a pair of functions $NC^A = (\mu_n, \mu_f)$. μ_n is a partial function on $\mathscr{N}^A$, that assigns a flavor to a subset of the application's nuclei $\mu_n \colon S \subseteq \mathscr{N}^A \to \mathscr{F}^{SOC}$. It is a partial function since some nuclei may have no optimized implementation in the target platform. For such nuclei, its functional specification ($P_i^{N^A}$) is used. μ_f is a mapping of flavor variables to their corresponding domains, i.e., $\forall F^{SOC} \in \mathscr{I}(\mu_n), \forall v \in F^{SOC}, \mu_f(v) \in D_v^{F^{SOC}}$.

Not every mapping would result in an implementable specification. Flavors that communicate with each other must have a *matching* configuration. As an example, consider two flavors F_1 and F_2 that communicate over a single channel. Suppose that the variables v and v' determine the locations in system memory that both flavors can use to communicate, with domains $D_v^{F_1}$ and $D_{v'}^{F_2}$. It is clear that the nucleus mapping results for these variables must be equal and contained in both variable domains, i.e., $\mu_f(v) = \mu_f(v') \in D_v^{F_1} \cap D_{v'}^{F_2}$. The same reasoning extends to more complex interfaces exposed by hardware accelerators in the target platform. More formally,

Definition 54 An **interface matching** ($\equiv_{\text{IF}}$) of two connected flavors is a relation between the flavor variables that describe the ports that connect the flavors. Let F_i and F_j be two connected flavors after a nucleus mapping $NC^A = (\mu_n, \mu_f)$. Let the flavors be connected due to a channel $c \in \mathscr{C}^A$ and let $V_c^{F_i} \subset V^{F_i}$ and $V_c^{F_j} \subset V^{F_j}$ be the set of variables that define the interfacing configuration over connection c for each of the flavors. The port interfaces match, denoted $V_c^{F_i} \equiv_{\text{IF}} V_c^{F_j}$, if $\forall v \in V_c^{F_i}, \mu_f(v) = \mu_f(v') \in D_v^{F_i} \cap D_{v'}^{F_j}$, where $v' \in V_c^{F_j}$ is the variable that corresponds to $v \in V_c^{F_i}$.

A nucleus mapping returns an implementable specification only if all flavors can actually communicate. Formally,

Definition 55 A **matching nucleus mapping** is a nucleus mapping $NC^A = (\mu_n, \mu_f)$ in which all interconnected flavor ports have matching interfaces, i.e., $\forall c_{ij} \in \{(n_i, n_j) \in (\mathscr{N}^A \times \mathscr{N}^A) \cap \mathscr{C}^A, \mu_n(n_i) = F_i, \mu_n(n_j) = F_j\}, V_{c_{ij}}^{F_i} \equiv_{\text{IF}} V_{c_{ij}}^{F_j}$.

By selecting flavors for some nuclei in the SDR application and fixing their configuration parameters so that they match, the SDR application model is lowered to an equivalent plain KPN model.

Definition 56 An **SDR implementation model** ($\mathcal{SI}^A$) for an application $A \in \mathcal{A}^{\text{sdr}}$ is an implementable KPN model $\mathcal{SI}^A = KPN^A = (\mathcal{P}^A, \mathcal{C}^A, \text{var}_{\text{size}})$ (see Definition 46) resulting from a matching nucleus mapping $NC^A = (\mu_n, \mu_f)$.

2.6.3 Problem Statement

With the definitions from the previous sections, it is now possible to define the SDR problem, as the problem of selecting the best matching flavors for an abstract waveform description so that the final implementation meets the constraints.

Definition 57 SDR problem: Given an SDR application $A = (\mathcal{M}^A, V^A, \mathcal{K}^A)$, $A \in \mathcal{A}^{\text{sdr}}$, with SDR application model $\mathcal{M}^A = (\mathcal{PAE}^A, KPN^A = (\mathcal{P}^A \cup \mathcal{N}^A, \mathcal{C}^A, \text{var}_{\text{size}}))$ and target platform $SOC = (\mathcal{PE}, \mathcal{E})$ with flavors $\mathcal{F}^{SOC}$, find a matching nucleus mapping $NC^A = (\mu_n, \mu_f)$ so that there is a valid runtime configuration $RC^A = (\mu_p, \mu_c, \mu_a)$ for the resulting SDR implementation model $\mathcal{SI}^A = KPN^A = (\mathcal{P}^A, \mathcal{C}^A, \text{var}_{\text{size}})$.

Note that after finding a matching nucleus mapping, the problem of finding an optimal valid runtime configuration reduces to the parallel problem in Definition 49. A simple solution approach could therefore be to iteratively build SDR implementation models and seek for a valid runtime configuration until one is found.

2.7 Synopsis

This chapter introduced several concepts relevant to this book, (1) a general overview of the mapping and scheduling terminology in Sect. 2.2.1, (2) a brief survey of performance estimation methods in Sect. 2.2.2, (3) basic compiler technology and parallelism extraction concepts required for the sequential tool flow in Sect. 2.4.1, (4) background knowledge on process networks needed for the parallel tool flow in Sect. 2.5.1 and (5) the context and basic terminology of the SDR tool flow in Sect. 2.6.1. Additionally, the chapter introduced a formal framework that allowed to represent the multi-application problem and its sub-problems as a general constrained optimization problem. The abstract constructs of the different problem statements, mainly variables and constraints, are further refined when presenting the corresponding tool flows in Chaps. 5–8.

This chapter included intuitive, simplified examples to illustrate the basic setup and the goal of the different problems, (1) the fundamental DAG mapping and scheduling problem in Fig. 2.2, (2) the parallelism extraction problem in Fig. 2.4, (3) the KPN mapping problem in Fig. 2.10 and (4) a motivational example for the multi-application problem in Sect. 2.1.

Before delving into the details of the solutions to the problems stated here, the next chapter presents an overview of solutions to similar problems.

http://www.springer.com/978-3-319-00674-1

Programming Heterogeneous MPSoCs
Tool Flows to Close the Software Productivity Gap
Castrillón Mazo, J.; Leupers, R.
2014, XV, 232 p. 73 illus., 65 illus. in color., Hardcover
ISBN: 978-3-319-00674-1